ObfSec: Measuring the Security of Obfuscations from a Testing Perspective

Héctor D. Menéndez[1], Guillermo Suárez-Tangil

King's College London, IMDEA Networks Institute

Abstract

Code obfuscation protects the intellectual property of software. However, systematically altering the control- and data-flow of a program can deteriorate the security of the resulting program. There are a wide-range of obfuscation methods available that alter the layout of the program in different ways. These modifications can introduce bugs in the program or modify the nature and the severity of an existing ones.

We propose a novel strategy, called *ObfSec* (Obfuscation Security), to understand the implications behind obfuscating software. ObfSec starts by detecting errors on software and exposes how the obfuscation can change the nature of those errors, looking in particular at transformations that turn software bugs into a exploitable vulnerable program. Our results, on a corpus of around 70,000 programs and obfuscations, show that obfuscation can deteriorate the security of a program.

Keywords: Obfuscations, Security, Testing

1. Introduction

Code obfuscation is a software protection technique that aims at frustrating automatic or manual code analysis using deliberately made transformations to the control- and data-flow of a program (Garg et al., 2016). Millions of different software applications leverage obfuscation to conceal the intellectual property of their code, and protect themselves from different reverse engineering and analysis strategies (Wang et al., 2018a). However, although obfuscation is helpful to protect intellectual property, it also affects the quality of the software (Chen et al., 2016). There are two reasons to this. First, the applied transformations can be faulty and thus bugs are introduced into the program. Second, the original program might contain bugs that are carried to the obfuscated program. Although the original bugs might not necessarily be exploitable, the transformations that are made by the obfuscation framework can expose those bugs differently — turning some of them into exploitable bugs. One can also find a combination of the aforementioned reasons. Consequently, the obfuscation process generates unreliable control- and data-flow sequences that poses an additional threat to the reliability of the program.

[*]Corresponding author

Email addresses: `hector.menendez@kcl.ac.uk` (Héctor D. Menéndez), `guillermo.suarez-tangil@imdea.org` (Guillermo Suárez-Tangil)

Related works have been evaluating the security of obfuscation engines as a way to measure their strength against reconstruction (i.e., de-obfuscation) (Schrittwieser et al., 2016). Despite the obfuscation process may *aggressively* change a program (Hammad et al., 2018), little efforts have been placed on understanding the reliability of the obfuscation life-cycle from a software quality perspective. Approaches such as symbolic execution (Banescu et al., 2016a) have tried to deal with obfuscation. However, these techniques have issues to scale to bigger and complex programs because of path exploitation (Banescu et al., 2016b). Furthermore, symbolic execution and other similar techniques underperform with obfuscation (Ma et al., 2014). Instead, recent works suggest that fuzz testing can deal with obfuscated programs (Jung et al., 2019; Güler et al., 2019) (both published in Usenix'19). While authors in (Jung et al., 2019) show that obfuscation does not introduce a 'resistance' to the fuzzer, it is not clear to what extend it can be used to improve our knowledge about the software quality. Also, for the best of our knowledge, the literature lacks on a systematic study assessing whether (and how) the — implementation of an — obfuscation algorithm degrades the resulting quality of the software.

In this paper, we introduce the first study looking at the life-cycle of the obfuscation process from a testing perspective. We measure the security of a program before and after it is obfuscated, which in turn is used to model to what extent the obfuscation process can lead to the identification of unknown vulnerabilities. For this, we design four novel measures that integrate into a processing pipeline that we call *ObfSec*. These measures quantify: i) the resistance, ii) the exploitability iii) the stability, and iv) the complementarity of an obfuscation. First, the resistance measures the effort required to find crashes. Second, the exploitability shows the likelihood that the implementation of an obfuscation will introduce a vulnerability into a program. Third, the stability measures how transferable are the observations made in an obfuscated program to the original program. Finally, the complementarity shows how much we learn, in terms of unmasking crashes and exploits, from fuzzing an obfuscated program on top of what we can infer from the original program. Altogether, these four measures describe how reliable the obfuscation process. Challenging the reliability of the obfuscation process is particularly relevant in the presence of adversarial conditions. In particular, we assume that malicious adversaries may try to exploit existing vulnerabilities in obfuscated programs by providing especially crafted inputs unexpected to the program.

In summary, we presents the following contributions:

- We propose a new methodology based to evaluate the security of an obfuscation engine. We study 20 state-of-the-art obfuscations from a range of open and commercial tools. We identify that the implementation of the AntiBr obfuscation is fundamentally flawed.
- We propose a number of novel measures to profile the life-cycle of errors during the obfuscation process. We apply our measures to analyze 646,854 crashes in 70,137 programs. We show that the obfuscation does not make the fuzzing process harder. To the contrary, we unexpectedly discover that

the obfuscation process can effectively be used as a testability transformation.

- We introduce a triage system to prioritize the analysis of crashes. Our triage system leverage on a number of heuristics that quantify the severity of the crash. In total, we study 198,468 critical crashes and track these cases to the obfuscation engines. We show that the combination of certain errors and specific obfuscations is more prone to generate exploitable crashes.

The rest of the paper is organized as follows. Section 2 motivates the problem of studying the bug life-cycle. Then, Section 3 presents ObfSec. For the evaluation of our methodology, shown in Section 5, we select a corpus of programs and obfuscations explained in Section 4. Our results are discussed in Section 6. Finally, review the literature and present our conclusions.

2. Motivating Examples

This section introduces the background that motivates our work, together with a motivating example that shows how the use intellectual property protections, in the form of obfuscation, affects the quality of the program. In this example, the obfuscation masks an error that would be exposed in the original program.
Background. Obfuscation has generally been used in compiler testing to evaluate the compiler's behaviour. A common technique is metamorphic testing with differential analysis (Tao et al., 2010). The obfuscation process turns the program into a semantically-equivalent version, altering code and variables. Differential testing then compares the output of both versions (the original and the obfuscated program) to identify issues in the compiler. These issues correspond with *program crashes.* However, there are two underlying *assumptions* with this compiler-based testing: i) the original program is free of bugs, and ii) the obfuscation process does not introduce crashes. In order to understand the security of obfuscation, we explore what happens when either assumption is not met. Thus, we work with buggy programs and explore a wide range of obfuscations. On the contrary, we assume that the compiler is reliable and thus we work with a well tested version as detailed in Section 4.
Motivating example. We observe that the obfuscation process can mask errors. For example, we consider layout obfuscation — one of the simplest forms of obfuscation that changes variable names. Figure 1 shows two programs that are exactly the same program but for the names of the variables. The original program (in the upper part of the listing) is obfuscated with the `Num` obfuscation of the `cobfusc` tool.[1] Given a unit test that passes "13133 3" as input, we observe that the original program produces a segmentation fault in line 7. However, the equivalent in line 18 does not produce a crash and the program finishes normally. Both programs start initializing the loop boundary n and $q1$, respectively. The reminder variables are initialized to 0. After the program has been looping for a while, the positions of the arrays ar (lines 7) and $q13$ (line

[1]https://manned.org/cobfusc/3dfcb129

18) are both out of bounds. This means that both programs have the same bug despite that the same unit test produces different results. Due to the effect of the layout obfuscation, the compiler returns two runtime programs with the memory variables sorted in different order as shown in Figure 1. This happens for both GCC and Clang. After analysing the stack with GDB, we see that the issue lays on the way in which the *end-loop* variable n ($q1$ in the obfuscated version) is accessed. In particular, we see from the memory layout that the original program sets n before the array, while the obfuscated program sets $q1$ after. When the array is overflowed in the obfuscated version, $q1$ is rewritten to 0 and the program ends. In this example, an attacker could purposely craft an input that would overwrite $q1$ to loop for ever.

Through this motivating example, we see that obfuscating a program with a bug has an impact in the way the program can be further exploited.

Motivation. Even the most simple obfuscation can influence the way programs behave. This has an important impact on software testing, as the inputs in one domain can expose crashes that are otherwise masked or does not exist in the other domain. Thus, we highlight that: 1) it is paramount to choose the right algorithm when obfuscation is part of the development life-cycle; and 2) testing both the obfuscated program and the original one can provide a better overview of the security of the program.

In what follows, we show the effect of following these advices in a practical way, presenting a methodology called *ObfSec*. Our aim is to understand the relevance of crossing tests between a program and its obfuscations. This will show how semantic equivalent transformation can change the behaviour of crashes, just by focusing on the crashes that are produced either on the original or the obfuscated version.

3. ObfSec: The Security of Obfuscation

ObfSec quantifies the security an obfuscation. As a by-product, our methodology allows to identify when an obfuscation system is fundamentally flawed.

ObfSec seamlessly integrates four measures into a processing pipeline. On the one hand, these measures focus on independent properties of obfuscated programs, like: a) the effort required to find crashes (`fuzzibility`), and b) the likelihood that an obfuscation engine will introduce a vulnerability into a program (`exploitability`). Both relate to the reliability of an obfuscation engine, but they provide complementary angles. The fuzzibility is related to finding crashes while the exploitability measures their severity.

Then, ObfSec devises an additional set of measures comparing the obfuscation with the original program to quantify: c) how *transferable* are the observations made in an obfuscated program to the original program and vice versa (`stability`), and d) how much we learn, in terms of unmasking crashes and exploits, from fuzzing an obfuscated program on top of what we can infer from the original program (`complementarity`). Related work refers to *testability transformation* when modifications in a program help in the testing process (Harman et al., 2004).

```c
int i,n,p1,l,r,ar[2][101];
int main(int argc, char *argv[]){
    scanf("%d%d",&n,&p1);
    for (i=0; i<n; i++){
        scanf("%d%d",&l,&r);
        ar[0][i]=l;
        ar[1][i]=r;
    }
    return(0);
}
-------------
int q0,q1,q3,q8,q9,q13[2][101];
int main(int argc, char *argv[]){
    scanf("%d%d",&q1,&q3);
    for (q0=0; q0<q1; q0++){
        scanf("%d%d",&q8,&q9);
        q13[0][q0]=q8;
        q13[1][q0]=q9;
    }
    return(0);
}
```

```
&q13 = (int (*)[2][101]) 0x555555755060 <q13>
&(q13[1][101]) = (int *) 0x555555755388 <q1>
&(q1) = (int *) 0x555555755388 <q1>
-----
&ar = (int (*)[2][101]) 0x555555755080 <ar>
&(ar[1][101]) = (int *) 0x5555557553a8
&n = (int *) 0x555555755064 <n>
```

Figure 1: Example of a program and its obfuscation where the variable's order is changed in memory. The bottom shows the GBD execution of the example, showing the memory position of the main variables affecting the masking process.

The fuzzification and exploitability measures relate to the obfuscation itself and are independent of the original program. On the contrary, stability and complementarity establish the relationship between the obfuscation and the original program, measuring how much overlap there is terms on crashes and vulnerabilities observed.

Our processing pipeline has three steps: *generation*, *triage* and *cross-testing* as shown in Figure 2. Given a program under test, the first step is creating a number obfuscations using a wide range of algorithms. We consider the program and its protected version as Subjects Under Test (SUTs). Our system then generates inputs for all SUTs in the first step. Here, we use a fuzzing system generating a set of inputs, or test suite, for each subject, i.e., for each program and any other executable resulting from the obfuscation of the original program. When the program is vulnerable, certain inputs will result into a crash. Our triage process measures the severity of the crash, using heuristics to identify exploitable crashes. Our heuristics take into account the type of crash (e.g., segmentation faults, illegal instructions, or floating point exceptions) to quantify how exploitable is the program. These two steps allow us to measure fuzzification and exploitability of the engines. Inputs that resulted in a crash or exploit are then cross-tested with the original program and its obfuscation in the cross-testing phase. This step allows us to compute the stability and complementarity. As a result, developers can prioritize fixes depending on the specific copyright protection they require.

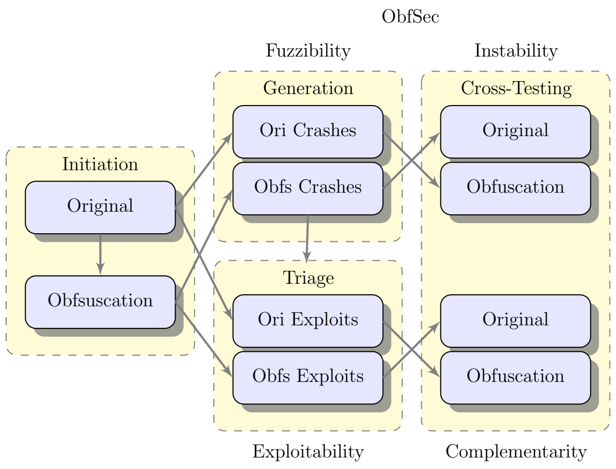


Figure 2: ObfSec process. It is divided in three phases: the generation phases that find crashes, the triage that checks their exploitability, and the cross-testing that determines their stability and complementarity.

3.1. Generation phase

Through a fuzzing system, the generation phase provides the initial set of crashes for a corpus of programs and their obfuscations. From this initial set, we measure the effort of the fuzzer to find crashes inside the code, namely `fuzzibility`.

Let P be a program and $\mathcal{O}(P)$ the obfuscation of program P with algorithm $\mathcal{O}$. The generation starts with a corpus of programs $\mathbb{C}_P$ and their obfuscations $\mathbb{C}_{\mathcal{O}(P)}$, also known as SUTs (Subjects Under Test). Then, applying fuzz testing, we create a test suite per program, called P_T, and an equivalent per obfuscation, called $\mathcal{O}(P)_T$. The fuzzer starts with the same set of seeds per program, i.e., initial inputs, and runs for the same amount of time in all versions of the program. Providing the same initial seeds instead of complementary ones is a common practice in the literature (Klees et al., 2018) and relates to our interest of using the corpus to measure how fuzzible the code is. The works of Jung et al. (Jung et al., 2019) and Guler et al. (Güler et al., 2019) motivate this measure. Jung et al. intuitively show that fuzzers are effective against LLVM-based obfuscations as well as different packing systems. Similarly, Guler et al. (Güler et al., 2019) also show this on Tigress obfuscations (Güler et al., 2019). However, the previous works do not provide a specific metric to measure the effectiveness, which is what we do next.

From P_T and $\mathcal{O}(P)_T$, we select only the inputs that produce crashes on P and $\mathcal{O}(P)$, called P_C and $\mathcal{O}(P)_C$, respectively. We primarily consider system crashes and divide them into the following categories:

1. **Segmentation fault**: the program writes in a memory section that is not available in its memory space.
2. **Illegal Instruction**: the program tries to execute a malformed, unknown or unauthorized instruction.
3. **Corruption**: the program is trying to manipulated memory associated with corrupted pointers.
4. **Floating Point Exception**: a test produces an invalid floating point operation, for instance, a division by zero.
5. **System Timeout**: a test execution exceeds the available execution time allocated to the program.
6. **Out of Memory**: a test execution exceeds the available memory allocated to the program.

We define our fuzzibility measure in terms of the total crashes detected during the fuzzing process over the size of corpus:

$$Fuzzibility(\mathbb{C}, \mathcal{O}) = \frac{\sum_{\mathcal{O}(P) \in \mathbb{C}_{\mathcal{O}(P)}} |\mathcal{O}(P)_C|}{|\mathbb{C}_{\mathcal{O}(P)}|}. \tag{1}$$

The fuzzibility can be normalized by applying Equation 1 to the original corpus with no obfuscation. Fuzzibility opposes fuzzification. When fuzzibility is low, fewer crashes can be found during the fuzzing process, therefore, the obfuscation is more resistant. On the contrary, it is high when the resistance to fuzz the non-obfuscated version of the program is similar to the resistance presented by the obfuscated program. Likewise, it is also high when fuzzing the obfuscated program displays novel crashes.

3.2. Triage phase

Our methodology relies on a triage that allows our system to prioritize the most critical crashes. For this, we consider a discrete function that returns whether a crash is exploitable or not. As this is an undecidable problem, we leverage a number of heuristics to make a prediction. Our heuristics relate to the most common type of exploits and are described as follows:

1. Destination Address Violation (**DestAv**): This heuristic looks at crashes that might have gain control of a 'write' address. This happens when the crash appears on an address that is known to belong of the *destination* operand of an instruction.
2. Branch Access Violation (**BranchAv**): This heuristic looks at crashes that might have overwritten *branch* instructions that could hijack the control flow of a program.
3. Return Address Violation (**ReturnAv**): This heuristic relates to crashes that might have been produced due to stack a corruption. This happens when the *return* instruction is overwritten as result of a crash.
4. Segmentation Fault on PC (**SegFaultOnPc**): This heuristic looks at violations of the address that stores the Program Counter (PC). This happens when a *branch* instruction is invoked with a crafted argument, or the issue causing the crash might have attempted to overwrite beyond the memory boundaries. This exploit gives an attacker control on the program flow.
5. **Stack Corruption**: This heuristic shows whenever a crash has corrupted the stack of the program. This is triggered when the return address in the stack does not match with the bounds of the process' address space. This is also triggered when the stack pointer addresses a location beyond the stack map. When there is an stack corruption after a crash, the program is generally exploitable.
6. **Heap Error**: This heuristic shows that a crash has corrupted the heap of the program. This is generally triggered when the program is executing a heap function over an address that is not in the right bounds. For instance, when calling to free() over a pointer that has been "smashed". This also relates to buffer overflow attacks.
7. **Bad Instruction**: This heuristic relates to generic cases where a crash relates to an instruction that is malformed and/or privileged. This might mean that attackers can control the program's flow if they know how to overwrite the malformed instruction.

Based on these heuristics, we then select from P_C and $\mathcal{O}(P)_C$ only crashes that are exploitable, P_V and $\mathcal{O}(P)_V$, respectively. This is used to extract a measure that estimates the `exploitability` of a corpus of programs $\mathbb{C}$ given $\mathcal{O}$:

$$Exploitability(\mathbb{C}, \mathcal{O}) = \frac{\sum_{\mathcal{O}(P) \in \mathbb{C}_{\mathcal{O}(P)}} |\mathcal{O}(P)_V|}{\sum_{\mathcal{O}(P) \in \mathbb{C}_{\mathcal{O}(P)}} |\mathcal{O}(P)_C|}. \tag{2}$$

This equation defines the percentage of crashes that are exploitable, therefore it is already normalized. Nevertheless, its comparison with the original program's exploitability relates to the crash triage as it appears after the obfuscation. This is used to provide a notion of how exploitable a program can be when it is obfuscated with a given algorithm. High values of this metric are considered more relevant. This means that the corpus of programs $\mathbb{C}$ tested with this metric are more prone to be exploited after using the obfuscation engine $\mathcal{O}$.

3.3. Cross-Testing phase

This phase studies the next step of the obfuscation life-cycle. Here, ObfSec checks whether the original and the obfuscated test suites are equivalent, opposites or complementary.

For this, we apply cross-testing: we take the crash test suite of P (P_C), and the equivalent of $\mathcal{O}(P)$ ($\mathcal{O}(P)_C$) and run them on the opposite program. Formally, we apply $P[[\mathcal{O}(P)_C]]$ and $\mathcal{O}(P)[[P_C]]$. The rationale behind cross-testing is as follows. The obfuscation process is supposed to generate a semantically-equivalent program. Thus, the same input should generate the same result in the original program as in any of its obfuscations. When this is not met, it means that either (1) the obfuscation process has introduced an error or that (2) an existing error has been masked as shown in Section 2. Alternatively, (3) one input can generate a change in the type of crash and this phases also explores this.

We introduce three ways to understand the ability of the program to continue to function after the obfuscation process, namely *stability* of the obfuscation. In particular, we look at how the crashes distribution changes with respect to the testability of programs. For this, we first look at the test suites created in the generation phase (Section 3.1). Considering the probability of program errors as a probability distribution, we have four distributions. Two distributions from the program error probability on the original program and the equivalent on the obfuscated program. These can in turn be produced either by the test suite created in the original or the obfuscated program. When combined together, we have the following distributions:

$$p = P[[P_C]], \quad q = \mathcal{O}(P)[[\mathcal{O}(P)_C]]$$

$$px = P[[\mathcal{O}(P)_C]], \quad qx = \mathcal{O}(P)[[P_C]]$$

where the weight of each distribution is the probability of a type of crash after applying the test suite. This is equivalent for exploitable crashes as given by our triage phase (Section 3.2).

Initially, ObfSec checks how stable is the original test suite in the obfuscated program (qx), especially considering that the obfuscated program is thought as final version of the software. This measure, called `instability`, evaluates how efficiently qx approximates the probability distribution of crashes on the original program (p). This is literally the Kullback-Leibler (KL) divergence between p and qx over the original test suite (P_C), and it is defined as:

$$\mathit{Instability}(\mathcal{O}) = KL(p||qx) = \sum_{i \in TC} p(i) \log \frac{p(i)}{qx(i)}, \tag{3}$$

where TC represents the types of crashes. When the divergence is 0 or close to 0, it means that the distribution of crashes in the original program and the obfuscated one is the same — therefore, the obfuscation is stable. As the divergence is always positive by definition, higher values capture uncertainty on the testability. This is, when the obfuscation can not provide information about the original program. The stability can either apply to crashes and vulnerabilities by restricting the test sets.

Similarly, we consider the opposite scenario: applying the test suite created for the corpus of obfuscated programs to the original program (px). This intends to understand those cases where errors have been masked (c.f., the motivating example in Section 2). Effectively, this concept checks how well the original program can approximate the distribution of errors in the obfuscation ($q||px$), and we call it `inverse instability`. Formally,

$$Inv.\ Inst.(\mathcal{O}) = KL(q||px) = \sum_{i \in TC} q(i) \log \frac{q(i)}{px(i)}. \tag{4}$$

Finally, the obfuscation process can guide fuzzers in the process of discovering errors that were not originally expose by the testing process, serving as testability transformations (Harman et al., 2004). In this case, the obfuscation mechanism complements the testing process when the obfuscation's test suite can detect specific types of errors that are otherwise resistant in the original testing. We also measure this complementary behaviour in terms of probability distributions, but, in this case, the distributions are related to both test suites applied to the original program alone. This way we can understand, giving the original test suite, how much the obfuscation test suite can complement it ($p||px$). This is similar to the instability, but in this case, the higher the Kullback-Leibler divergence between both, the better the guidance (i.e., the better the obfuscated test suite complements the original). We call this measure `complementarity`:

$$Complem.(\mathcal{O}) = KL(p||px) = \sum_{i \in TC} p(i) \log \frac{p(i)}{px(i)}. \tag{5}$$

These measures help developers to decide which software protection has a larger impact in securing their software. For all the divergences, when the denominator is 0, we add a perturbation of 10^{-10}. The following evaluates how our proposed measures help to understand the implications of using several well-known obfuscation systems. We also show to what extent software protections can be used as a testability transformations tool.

4. Experimental Setup

Central to the design of a suitable experimental setup is the definition of a fine-grained set of research questions. Thus, in this section we first provide a break down of our objectives into three block of questions (§4.1). Then we detail our experimental setup by describing the obfuscation strategies we leverage (§4.2), introducing our the fuzzer we use (§4.3) and our dataset (§4.4). Finally, we detail the error codes and type of exploits our implementation covers (§4.5). Validating ObfSec metrics requires a deep set of experiments that provide a complete and general perspective of the obfuscation effects on multiple security scenarios. To achieve this aim, we will cover an extended set of cases showing how ObfSec generalize to different obfuscation engines and compilers, and how the different transformation relate to crashes and exploits.

4.1. Research Questions

As ObfSec depends on the fuzzification abilities of an obfuscation, we start asking: **RQ1.1**: Is the obfuscation disrupting the testing process by being resistant to fuzzers? If that were the case, the fuzzer would not be able to detect new crashes on obfuscated programs, and ObfSec would be devoid. Answering to this question requires evaluating the difficulty underlying the fuzzing process, which includes understanding how efficient the test suits are in finding (exploitable) crashes. This immediately leads to ask, **RQ1.2**: are programs that are obfuscated with a given algorithm more prone to be exploited when compared to other algorithms?

This block of questions relate to the generation and triage phases as described in Section 3.1. We later leverage the fuzzibility and exploitability measures to answer these questions. The higher the fuzzibility, the lower the resistance to testing. Depending on the resistance we find, the fuzzer will detect errors in the original program or in the obfuscated program. We use our methodology to evaluate different obfuscations strategies over to a significant program corpus.

Once we verify that we can leverage the generation phase to learn about the fuzzibility of an obfuscation, we prompt the following questions: **RQ2.1**: How can an obfuscation mask crashes, i.e., how is the instability of the obfuscation? **RQ2.2**: How can an obfuscation be used to estimate or complement the testing process of a program? Once the test suites are generated, ObfSec cross-tests the original and the obfuscated programs to understand the instability and testability of obfuscations from the crashes. This also provides information about how a crash evolves under specific types of program transformations. We ask two similar questions regarding the triage phase: **RQ3.1**: How do obfuscations conceal exploits in terms of instability? **RQ3.2**: How well can obfuscations unmask exploits as testability transformations?

Finally, in order to understand which crashes and exploits require more attention after the triage and obfuscation, we need to understand how they can change their nature. Then, we ask: **RQ4**: Are the crashes and exploits following specific transformation patterns for different unstable obfuscations? Crashes or exploits whose transformation patterns are noisy and unpredictable will require more attention in the bug fixing process that those whose transitions are more static.

4.2. Obfuscation Engines and Strategies

For the evaluation of our ObfSec methodology we have selected four obfuscation engines. These engines have different properties and strategies, but some of them are common, such as *literal* modifications and arithmetic transformations. For our experiments, we consider four state-of-the-art tools for `C` commonly used in other works related to obfuscation (Chen et al., 2016): As our experiments focused on `C` code, we select the most established tools: i) Tigress, ii) LLVM-O, iii) a commercial tool called Stunnix (CXX), and iv) a system tool called CObfusc. Our experiments also compare across compilers, with Clang for the LLVM obfuscations and GCC for the reminder. We use stable releases of the compilers — 6.3.0 for GCC and 3.8.1 for Clang.

Overall we explore 20 different transformations from the four engines we leverage. The first engine that we consider is Tigress. This comprehensive engine has several different obfuscation strategies: 1) Injection of predicates that are never visited during the program execution (`InitOp`), and also 2) those using directly system variants (`UpOp`), 3) virtualization (`Virtu`), 4) implicit flow (`ImplFl`), 5) entropy manipulation (`InitEn`), 6) manipulation of branches (`InitBr`), 7) anti-branch analysis (`AntiBr`), 8) anti-Taint analysis (`AntiTa`), 9) anti-alias analysis (`AntiAl`), 10) encoding of literals (`EncLit`) and 11) encoding of arithmetic operations (`EncAri`). We also apply a system tool called C-OBFUSC that manipulates literals either with 12) numbers (`CobfsNum`) or 13) words (`CObfsWord`). We also test a known commercial tool called Stunnix, using the following obfuscations: 14) the first reduces the names to the shortest possible ones (`cxxobfShortest`), 15) the second uses encoding of strings with hexadecimal codes (`cxxhexchar`), 16) and the last one transform constants in a sum of three elements (`cxxobfSum3`). Finally, the last obfuscation engine that we apply is the LLVM-Obfuscation using four different transformations: 17) the first replaces arithmetic instructions (`llvmsub`), 18) the second flatters the controls flow graph (`llvmfla`), 19) the third splits basic blocks (`llvmsplit`), 20) and the last bogus the control flow adding basic flows before functions calls with opaque predicates (`llvmbcf`).

The rationale behind our choice of transformations is to cover a wide spectrum of transformations described by Collberg et al. in (Collberg et al., 1997), from a range of open and commercial tools.

4.3. American Fuzzy Lop

This work uses the American Fuzzy Lop (Zalewski, 2019) to generate the test. This tool starts with an initial test corpus or *seeds*. The tool instruments the program in compilation time adding instructions to measure coverage. It sets the seed into a queue, and run them to evaluate the initial coverage. Then, the fuzzer applies different mutations to a queue of inputs in order to identify new branches and improve coverage. Those tests that identify new branches are added to the queue. This process continues until the user stops it.

4.4. Dataset

For our ObfSec experiments we select a corpus of 7,436 real programs called *codeflaws*. These program have been designed for different programming competitions called Codeforces[2] (Tan et al., 2017). The dataset comes with an initial test suite per program that is provided for the competition goal, which it is then used as seed in the fuzzing process. This will help to stress different possible behaviors and detect

[2]`https://codeflaws.github.io` and `http://codeforces.com`

Error Code	Message Example	Meaning
0,1	Any in Stdout/Stderr	User-made
124	Any in Stdout/Stderr	Timeout
132	Illegal instruction	Unknown or unauthorized instructions
134	Double free or corruption	Free unattached memory
136	Floating point exception	Erroneous Arithmetic Operation
137	Execution killed	Out of memory
135,139	Segmentation Fault	Illegal memory access

Table 1: Common errors found in the original program

potential crashes faster. The average size of the programs is 75 LoC, which facilitates the generation of a wide range obfuscations for the programs and a comprehensive fuzzing in a timely manner. For the analysis, we focus on the programs that contain crashes, from the corpus, 3,739 programs. Overall we have 70,137 programs for analysis after applying the obfuscations described in Section 4.2. These programs have 646,854 crashes from which 198,468 are exploitable (Section 5.1). The average lines of code per program is 4,366.

4.5. Error Codes and Exploits

We study program bugs interpreting execution crashes. For this, we look at the signals raised by the operating system to terminate the program. These are very meaningful signals to understand the nature of the crash. Table 1 shows a map between the signals we study and the crashes described in Section 3.1.

The main signals that we consider are: **illegal instruction** or corrupted binary (SIGILL 132), double free exception or **corruption** (SIGABRT/SIGIOT 134), **bad memory access** (SIGBUS 135), **floating point exception** or integer overflow (SIGFPE 136), explicitly killed or **memory exceeded** (SIGKILL 137), and **segmentation fault** (SIGSEGV 139).

A crash may lead to the exploitation of a vulnerability in the program. In order to quantify the criticality of a crash, we use a triage process. Our triage process leverages crashwalk[3] (Wang et al., 2018b) to assess every crash based on the heuristics described in Section 3.2. The vulnerabilities identified by crashwalk are the ones defined in Section 3.1.

5. Evaluation

We leverage ObfSec to understand the security and reliability of obfuscations. The following shows our main outcomes.

[3]`https://github.com/bnagy/crashwalk` together with GDB exploitable

5.1. Fuzzing the Obfuscation is not Challenging

We first study the results of applying AFL during 5 minutes per program to the corpus of 70K original and obfuscated programs. Table 2 shows a summary of our results. Recall that the fuzzer starts applying the same set of inputs for both the original program and its obfuscations. We next look at how ObfSec performs when using GCC and LLVM and, later, we study its results.

GCC: We see that that 3,739 programs compiled with GCC have crashes, with a total of 44,950 unique crashes. From these, our triage system identifies 14,873 of them as exploitable. Applying different obfuscations to these programs, we expect some resistance during the fuzzing testing process. However, this resistance is only manifested with specific obfuscations, as in several cases the number of crashes is similar. The most relevant cases are discussed as follows. On the one hand, we see that `AntiAl` obfuscation significantly reduces the fuzzibility, also reducing the number of programs containing crashes. Contrary, `EncAri` does not significantly reduces the number of programs containing crashes. However, it reduces the fuzzibility to half of the original value. On the other hand, `InitEn` and `Virtu` reduce the number of programs with crashes to under 3,000. `Virtu` is especially interesting as it also reduces the fuzzibility to a third with respect to the original corpus. Finally, both `CObfusc` and `CXX` have similar fuzzibility results than the non-obfuscated version. This is because the obfuscation engines only transform the literals inside the programs. They also have the highest crash intersection with the original test suite. Only the `Shortest` obfuscation reduces the number of crashes found.

LLVM: We observe that changing the compiler strongly affects the capabilities of the fuzzer to detect crashes in programs. In particular, AFL only detects 17,758 crashes and reduces the number of programs with crashes to 3,485 with respect to the original corpus compiled with LLVM. Besides, the fuzzibility is smaller than a half of the original. On the contrary, LLVM obfuscations increases the number of crashes — especially for `BCF` and `Split` obfuscations. On another note, `AntiBr` generates a crash for every input. This relates to the error in the function that redirects the branches as discussed in Section 2. Thus, all crashes produce the same error, a segmentation fault. Nevertheless, as Table 5 shows, these crashes has different exploitability.

Fuzzibility: When looking at the resistance, results show that most of the obfuscations are not helpful to the *fuzzification* process or to *anti-fuzzing* systems (Jung et al., 2019; Güler et al., 2019). Only `Virtu` can be considered as a potential fuzzification. Nevertheless, this obfuscation only makes the process of detecting crashes 3 times harder, while state-of-the-art fuzzifications make this process, at least, 40 times harder (Jung et al., 2019).

In terms of exploitability of these crashes, `EncAri` finds less crashes. However, their exploitability is higher than the originals (42.02%). This suggests that the obfuscation reduces the security of the program. Similarly, `EncLit` is also incrementing the exploitability, although it is not significant. On the majority of the cases, `Tigress` reduces exploitability, especially on `AntiAl`, which can reduce it from 33.09% to 16.50%

Method	Prog. w/ Crashes	Total Crashes	Total Exploits	Crash Intersec.	Fuzzibility	Exploit.
Original GCC	3739	44950	14873	-	12.022	33.09%
Tigress AntiAl	**3108**	32635	**5384**	1897	10.500	**16.50**%
Tigress AntiBr	-	-	-	-	-	-
Tigress AntiTa	3369	38322	12199	2859	11.375	31.83%
Tigress EncAri	3358	**20464**	8599	**1054**	**6.094**	**42.02**%
Tigress EncLit	3364	40102	*13856*	2818	11.921	**34.55**%
Tigress ImplFl	3366	40478	13425	2956	*12.026*	*33.17*%
Tigress InitBr	3207	34873	7678	2269	10.874	22.02%
Tigress InitEn	**2950**	30580	7705	1805	10.366	25.20%
Tigress InitOp	3226	34894	7238	2427	10.816	**20.74**%
Tigress UpOp	3017	34884	7511	2199	11.562	**21.53**%
Tigress Virtu	**2619**	**11067**	**2410**	**491**	**4.226**	**21.78**%
CObfs Num	*3567*	40427	13288	**3589**	11.334	32.87%
CObfs Word	*3467*	*42257*	*13911*	**3520**	*12.189*	*32.92*%
CXXobf Shortest	3377	34554	10650	3318	10.232	30.82%
cxxobfHexchar	3661	39382	12573	**3790**	10.757	31.93%
cxxobfSum3	3674	39030	11944	**3809**	10.623	30.60%
Original LLVM	3488	17758	7200	-	5.096	40.55%
llvmsub	3485	17552	7334	1741	5.032	41.78%
llvmfla	3201	16024	6639	849	5.006	41.43%
llvmbcf	3429	18485	**6397**	1512	5.391	**34.61**%
llvmsplit	3465	18136	**7654**	2122	5.234	**42.20**%
Total	70,137	646,854	198,468	45,025	-	-

Table 2: Fuzzing results for the original programs and their different obfuscations. The columns shows the fuzzing results in terms of crashes found and intersection of their triggers.

and the number of exploits to a third of the originals. Other obfuscations in `Tigress` like `InitOp`, `UpOp` and `Virtu` also show robustness in terms of exploitability reducing both, total exploits and exploits per crash. `Virtu` significantly reduces the total number of exploits to a sixth of the non-obfuscated version. For `CObfs` and `CXX` results are similar in terms of crashes. In LLVM, `BCF` reaches the minimum on exploits and exploitability while `Split` increments these values over the threshold of the original program (454 more exploits and 1.65 points more of exploitability).

RQ1.1: *ObfSec is viable as different obfuscations do not disrupt the fuzzing process, even for different compilers. The fuzzibility shows that in average the fuzzer can find, at least, 4 crashes per program in a timely fashion.*

RQ1.2: *Obfuscations affect the exploitability of a program and they can either reduce it up to 50% of the original in the case `AntiAl`, or increase it up to 27% in the case of `EncAri`.*

5.2. On How the Obfuscation Changes the Type of Crash

This section analyses how the nature of the crash evolves. This relates to the cross-testing phase, where ObfSec passes the relevant test cases (i.e., those that generated crashes in the original program) to its obfuscation and viceversa. Table 3 shows the different type of errors we identify for the most relevant cases of cross-testing for p (Ori $\rightarrow$ Ori) and qx (Ori $\rightarrow$ Obf). Some of the inputs generate false positives as a consequence of the parameters or the instrumentation of the fuzzer. These are shown as 'No Error/User' error in the table. The reminder errors relate to the taxonomy of errors introduced in Section 3.1. We also show in the table the instability of the obfuscations as a distribution. A more complete picture of the the instability is shown in Figure 3, where we depict how this measure changes across obfuscation engines for all cross-tests.

GCC: Several of the crashes affecting the original program "disappear" in the obfuscated version. We refer to this cases in the example shown in Section 2. This is especially relevant for `Tigress` and, in particular, for `AntiAl`, `InitBr`, `InitOp` and `Virtu`, where the percentage of original crashes that become false positives grows from 14.18 (`Virtu`) to 24.67 points (`AntiAl`). Several of the crashes that disappear have originally produced a segmentation fault. Here, the overall number of segmentation faults is reduced 23.24 points. `Virtu` also reduces the segmentation faults significantly (19.14 points), but, at the same time, it increases the number of timeouts (4.33 points more).

When looking at other type of errors that disappear, we see that illegal instruction errors never occur with `AntiAl`. Similarly, we observe that this errors significantly decrease with `InitEn` and `Virtu`. Instead, `ImplFl` increases the number of illegal instruction errors. Furthermore, `AntiAl`, `InitBr`, `InitOp` and `UpOp` increase the number of out of memory errors from 0 up to 0.17%. In terms of instability, `Virtu`, `UpOp`, `InitOp`, `InitEn`, `InitBr` and, especially, `AntiAl` are the most unstable obfuscations. Theirs instability ranges from 3.74 up to 16.84. `AntiBr` is fully unstable as it leads every input to a crash.

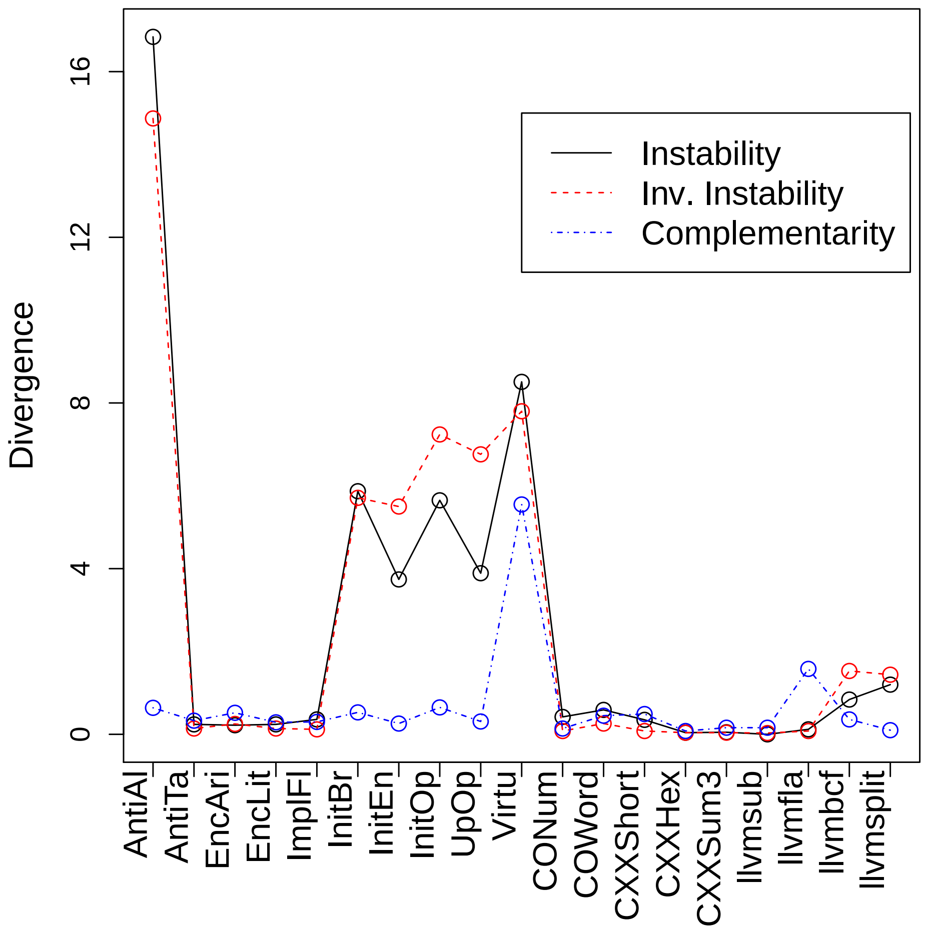


Figure 3: Instability, inverse instability and complementarity for every obfuscations in terms of crashes.

Method	No error	SF	Il. Inst	Corrup.	FPE	TOut	OoM	Others	Inst
GCC Ori → Ori	24.06%	68.74%	0.17%	0.60%	3.65%	2.77%	0.00%	0.00%	0.00
Ori → AntiAl	**48.73**%	**45.50**%	**0.00**%	0.62%	3.39%	1.61%	0.14%	0.00%	16.84
Ori → AntiBr	0.00%	**99.79**%	0.00%	0.00%	0.21%	0.00%	0.00%	0.00%	699.26
Ori → ImplFl	26.91%	66.73%	**0.22**%	0.86%	3.14%	2.15%	0.00%	0.00%	0.36
Ori → InitBr	**39.56**%	53.83%	0.09%	0.84%	3.69%	1.82%	0.16%	0.00%	5.87
Ori → InitEn	35.69%	56.91%	**0.01**%	0.86%	3.65%	2.87%	0.00%	0.01%	3.74
Ori → InitOp	**39.25**%	54.17%	0.10%	0.84%	3.62%	1.85%	**0.17**%	0.00%	5.65
Ori → Virtu	**38.93**%	**49.60**%	0.05%	0.89%	3.38%	**7.10**%	0.00%	0.06%	8.51
LLVM Ori → Ori	18.51%	71.43%	0.18%	0.63%	6.76%	2.45%	0.01%	0.03%	0.00
Ori → llvmsub	**18.40**%	**71.72**%	0.20%	0.62%	6.65%	2.37%	0.01%	0.03%	0.00
Ori → llvmsplit	24.65%	**65.98**%	0.14%	0.55%	6.49%	2.17%	0.00%	0.02%	1.20

Table 3: Relevant cases of the errors related to the cross-testing between the original program applied to itself and its obfuscated versions.

Finally, `CObfs` and `CXX` obfuscations show similar results than in the original program, and they are stable. However, it is important to highlight that these obfuscations are not performing complex transformations as is the case of Tigress. Recall that these transformations simply change the names of the literals in the case of `Cobfs`, or arithmetic operations and strings in the case of `CXX`.

LLVM: LLVM obfuscations increment the number of false positives with respect to the baseline (i.e., the original program compiled with LLVM). However, the `Sub` obfuscation reduces them 0.11 points and increments the number of segmentation faults 0.29 points. The reminder of the LLVM obfuscations overall reduce the number of crashes. The most representative case is `Split`, that reduces the number of segmentation faults 5.45 points incrementing the report of false positives 6.14 points.

Cross-testing: We measure the inverted instability and the complementarity in the cross-testing phase. Figure 3 shows how these measures change across transformations. Table 4 summarizes the most interesting cross-tests where we apply inputs from the obfuscated programs to the original one. As we observe in Section 5.1, `CXX` and `CObfs` have similar results than the original program, they are neither significantly unstable nor complementary. However, the behavior of Tigress and LLVM-Obfs diverge depending on the type of obfuscation strategy. First, `AntiAl`, `InitBr`, `InitEn`, `InipOp`, `UpOp` and `Virtu` change the nature of the identified errors (on the obfuscated program). This renders these obfuscation significantly unstable. All of them but `Virtu` produce a high number of false positives (between 37.95 and 45.11%) and several of these false positives turn to segmentation faults when they run on the original program. Specifically, segmentation faults increase from 13.66 to 22.41 points. Furthermore, `Virtu` finds less false positives than the other programs. This is probably a consequence on the timeouts reduction between `Virtu` and the original program.

For `AntiAl`, `InitBr`, `InitOp` and `UpOp`, the original program also increments the illegal instructions between 0.15 to 0.18 points, something uncommon on the other obfuscations. Cross-testing also shows timeout changes between these four obfuscations and the original program. Interestingly, they increase with respect to the obfuscation, between 1.23 and up to 1.73 points. Considering them as testability transformations, these obfuscations help to find timeouts in programs. Nevertheless, almost no obfuscation complements the testing process significantly — `Virtu` being an exception as shown in Figure 3. `Virtu` also increments floating point exception on the original program by 0.67 points. This suggests that `Virtu` is a good testability transformation to unmask floating point exceptions on real programs. For LLVM-Obfs, `BCF` and `Split` have similar results than the `Tigress`' obfuscations on false positives and segmentation faults, but only `Fla` has a good complementary ability.

RQ2.1: *ObfSec shows how the error changes, depending on the obfuscations, to different types of error. In several cases, the crash disappears, especially with `AntiAl` and `InitBr`. Nevertheless the majority of the obfuscations are stable in terms of errors found by the fuzzers.*

RQ2.2: *ObfSec shows that using the obfuscation as a testability transformation unmask several errors when the obfuscation is `Virtu` or LLVM-`Fla`, but not in general. `Virtu` discovers new FPE errors that are harder to unmask during the fuzzing process on the original program.*

5.3. On How the Obfuscation Changes Vulnerabilities

After studying the life-cycle of crashes, we then look at crashes that can be exploited. Our system triages exploits and associate them to the crashes (see Section 3.2). Figure 4 compares the divergence between the three measures we introduce to study the triage across different obfuscations. We show a summarized break down of some of the most relevant tests in Table 5. This summary shows the prevalence of the different type of exploits and how they evolve from the original program to the obfuscation. We refer to these cases as *changed crash* as we discuss in Section 2.

GCC: The first thing to note relates to `AntiBr`. Recall that this obfuscation results on segmentation fault in all our tests. However, there are a wide range of different exploits in our test suit affected by this obfuscation. In particular, 18.24% of the exploits are DestAv, 59.84% BranchAv and 21.93% Stack Corruptions. This suggests that, even thought different paths reach the error, there are several vectors to attack it. Disregarding `AntiBr`, the obfuscations that have more influence on the transitions of exploits are `AntiAl`, `InitBr`, `InitEn`, `InitOp`, `UpOp` and `Virtu`. Even though these obfuscations were already unstable in terms on crashes (Section 5.2), the instability in terms of exploits grows almost an order of magnitude for all but `Virtu`, which decreases. All these obfuscations turn ReturnAv vulnerabilities, exploited as an stack corruption of the return statement of a function, to different exploit.

Method	No error	SF	Il. Inst	Corrup.	FPE	TOut	OoM	Others	InvI.	Com
AntiAl → AntiAl	**45.11**%	**46.64**%	**0.01**%	0.54%	4.90%	**2.79**%	0.01%	0.01%		
AntiAl → Ori	**21.12**%	**69.05**%	**0.19**%	0.56%	4.92%	**4.15**%	0.00%	0.01%	14.87	0.64
InitBr → InitBr	**41.18**%	**50.84**%	**0.06**%	0.67%	4.46%	2.78%	0.01%	0.01%		
InitBr → Ori	**26.24**%	**64.58**%	**0.21**%	0.67%	4.13%	4.16%	0.00%	0.01%	5.71	0.53
InitEn → InitEn	**37.95**%	**54.31**%	0.01%	0.52%	4.12%	3.09%	0.00%	0.00%		
InitEn → Ori	**23.25**%	**67.97**%	0.09%	0.57%	4.31%	3.80%	0.00%	0.00%	5.50	0.26
InitOp → InitOp	**42.71**%	**49.21**%	**0.06**%	0.70%	4.53%	**2.78**%	0.01%	0.01%		
InitOp → Ori	**25.97**%	**64.45**%	**0.24**%	0.64%	4.18%	**4.51**%	0.00%	0.01%	7.24	0.65
UpOp → UpOp	**41.54**%	**50.72**%	**0.05**%	0.69%	4.25%	**2.73**%	0.01%	0.01%		
UpOp → Ori	**25.26**%	**65.89**%	**0.22**%	0.71%	3.96%	**3.96**%	0.00%	0.01%	6.76	0.31
Virtu → Virtu	**17.76**%	65.36%	0.00%	0.67%	7.80%	**8.22**%	0.00%	0.18%		
Virtu → Ori	**21.58**%	66.53%	0.00%	0.74%	8.47%	**2.68**%	0.00%	0.00%	7.80	5.55
llvmbcf → llvmbcf	**23.23**%	**67.37**%	**0.21**%	0.55%	6.10%	2.47%	0.01%	0.07%		
Ori → llvmbcf	**16.96**%	**73.62**%	**0.46**%	0.52%	6.12%	2.30%	0.00%	0.03%	1.53	0.36
llvmsplit → llvmsplit	**24.65**%	**66.32**%	0.14%	0.63%	6.09%	2.13%	0.00%	0.04%		
Ori → llvmsplit	**18.02**%	**73.00**%	0.23%	0.58%	6.02%	2.10%	0.01%	0.04%	1.44	0.10

Table 4: Most relevant obfuscations related to cross-testing from the obfuscation to the original program, separated by type of crashes.

Method	SFOnPc	DestAv	RetAv	HeapEr	BranchAv	BadInst	StackC	ExInst
Ori → Ori	6.82%	64.76%	19.86%	2.61%	2.90%	0.00%	3.04%	0
Ori → AntiAl	8.09%	**73.59%**	**2.17%**	**6.29%**	**7.70%**	**0.09%**	**2.07%**	30.61
Ori → AntiBr	0.00%	**18.24%**	0.00%	0.00%	**59.84%**	0.00%	**21.93%**	816.78
Ori → InitBr	3.86%	56.14%	**1.13%**	4.14%	**18.26%**	**0.07%**	**16.40%**	58.34
Ori → InitEn	2.68%	**83.37%**	**0.95%**	3.87%	4.54%	0.00%	4.58%	46.73
Ori → InitOp	3.87%	56.09%	**1.17%**	4.14%	**18.25%**	**0.07%**	**16.41%**	57.74
Ori → UpOp	2.92%	56.11%	**1.16%**	3.37%	**19.24%**	**0.07%**	**17.13%**	60.04
Ori → Virtu	**11.01%**	**52.82%**	17.08%	**5.88%**	4.67%	0.00%	8.53%	6.29
LLVM → LLVM	5.76%	66.57%	16.85%	1.85%	6.26%	0.00%	2.71%	0.00
Ori → LLVMBcf	**4.50%**	**72.84%**	**12.07%**	1.95%	6.84%	0.00%	**1.80%**	1.51
Ori → LLVMSplit	**3.33%**	**74.70%**	**10.90%**	1.99%	6.53%	**0.01%**	2.54%	2.29

Table 5: Relevant obfuscations during the cross-testing phase of the triage from the original program to the obfuscations.

On the one hand, `AntiAl` mainly changes to DestAv (8.83 points higher than the original) and, in some of cases to BrachAv (4.8 points higher), HeapError (3.68 points) and even a few to BadInstruction (0.09 points) — a kind of vulnerability that was not present in the original program. On the other hand, `InitBr`, `InitOp` and `UpOp` not only reduce the number of exploits shown as ReturnAv but also as DestAv, changing the vulnerabilities mainly to: i) BranchAv (between 15.35 and 16.34 points more than the original), ii) BadIntruction (0.07 points more), and iii) StackCorruption (between 13.36 and 14.09 points). Interestingly, Tigress increases StackCorruption vulnerabilities for all obfuscations but `AntiAl`, that reduces them by one point. Finally, `Virtu` is an exception as this obfuscation transforms DestAv and some ReturnAv into SegFaultOnPC (4.19 points more than the original) and Heap Error (3.27 points more).

LLVM: `CObfus` and `CXX` do not alter the original distribution of exploits significantly. In several cases there is a small increment on ReturnAv and a small reduction on DestAv vulnerabilities (around 1 or 2 points). Nevertheless, they are as stable as what we observe with crashes (Figure 4). For the LLVM-obfuscator, we see an opposite tendency than in the case of Tigress. First, the obfuscations reduce the Stack Corruption up to 0.91 points. Second, `BCF` and `Split` obfuscations transform ReturnAv mainly to DestAV (between 6.27 and 8.13 points more). Finally, we see that they are also more unstable for exploits than for crashes but not significantly (Figure 4).

Triage: Table 6 display the most relevant cases we see after applying the triage process to all crashes in Table 4. For each test, we study the type of exploit observed, the inverse instability and the complementarity. In some obfuscations, there is an obvious change on the type of exploit. This is shown in Figure 4, where

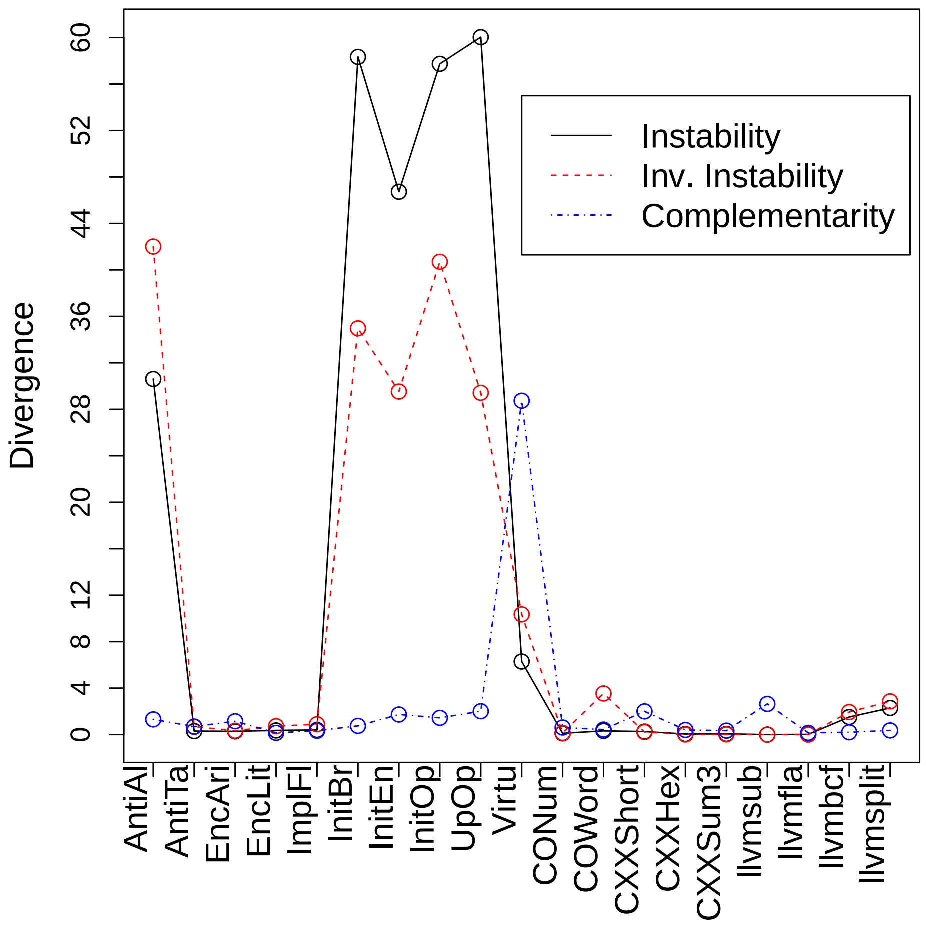


Figure 4: Instability, inverse instability and complementarity for every obfuscations in terms of triage.

Method	SFOnPc	DestAv	RetAv	HeapEr	BranchAv	BadInst	StackC	IvIns	Com
AntiAl → AntiAl	**22.14%**	**66.29%**	**0.78%**	3.96%	5.59%	0.02%	**1.23%**		
AntiAl → Ori	**5.94%**	**60.34%**	**22.95%**	3.12%	3.80%	0.00%	**3.85%**	42.01	1.31
EncAri → EncAri	4.88%	**68.97%**	**16.83%**	1.54%	4.81%	0.06%	2.91%		
EncAri → Ori	4.53%	**66.67%**	**19.85%**	1.55%	4.42%	0.02%	2.96%	0.33	1.15
InitBr → InitBr	**0.76%**	**87.33%**	**1.00%**	3.50%	**5.00%**	0.26%	2.15%		
InitBr → Ori	**5.88%**	**60.37%**	**24.19%**	2.44%	**3.63%**	0.00%	3.48%	34.97	0.75
InitEn → InitEn	**2.32%**	**87.54%**	**1.03%**	2.91%	4.36%	0.00%	1.84%		
InitEn → Ori	**4.71%**	**60.30%**	**27.02%**	1.93%	3.01%	0.00%	3.04%	29.53	1.73
InitOp → InitOp	**1.24%**	**87.75%**	**0.58%**	**3.98%**	**5.28%**	**0.26%**	0.91%		
InitOp → Ori	**6.38%**	**57.57%**	**26.10%**	**2.51%**	**3.85%**	0.00%	3.60%	40.70	1.44
UpOp → UpOp	**1.52%**	**86.85%**	**0.77%**	3.43%	6.10%	0.19%	**1.14%**		
UpOp → Ori	**6.43%**	**58.41%**	**21.76%**	2.40%	6.68%	**0.01%**	**4.31%**	29.42	2.01
Virtu → Virtu	**12.20%**	**48.34%**	**9.09%**	5.64%	**13.61%**	0.00%	**11.12%**		
Virtu → Ori	**6.06%**	**66.97%**	**2.88%**	4.01%	**11.71%**	0.00%	8.36%	10.35	28.74
CObfsWord → CObfsWord	5.69%	63.65%	20.74%	2.68%	2.78%	**0.15%**	4.31%		
CObfsWord → Ori	6.20%	61.39%	22.38%	2.95%	3.91%	0.00%	3.17%	**3.54**	0.43
LLVMFla → LLVMFla	3.80%	59.82%	25.25%	1.51%	6.11%	0.00%	3.51%		
LLVMFla → Ori	3.69%	59.78%	25.46%	1.51%	6.13%	0.00%	3.43%	0.00	2.64
LLVMSplit → LLVMSplit	**3.33%**	**74.70%**	**10.90%**	1.99%	6.53%	0.01%	2.54%		
LLVMSplit → Ori	**5.44%**	**68.43%**	**15.42%**	1.78%	5.60%	0.00%	3.33%	1.94	0.20
LLVMBcf → LLVMBcf	**4.16%**	**73.35%**	**11.27%**	1.76%	6.82%	0.00%	2.64%		
LLVMBcf → Ori	**6.84%**	**64.87%**	**18.07%**	1.61%	6.02%	0.17%	2.41%	2.86	0.36

Table 6: Most relevant cross-testing triage results from the obfuscated corpora to the original.

we represent the divergences for every obfuscation. When looking at obfuscations in `Tigress`, the crashes found over `AntiAl`, `InitBr`, `InitEn`, `InitOp`, `UpOp` and `Virtu` obfuscations significantly change the type of exploit when applied to the original program. They also increase their inverse instability almost an order of magnitude (Figure 4 and Figure 3). In particular, 22.14% of the exploits observed in `AntiAl` are related to SegFaultOnPc, which reduces to 5.94% with respect to the original. Instead, ReturnAv increases to 22.95% (22.17 points more than the obfuscation).

A common feature across all obfuscations in `Tigress` is that the prevalence of the DestAv exploit reduces and ReturnAv increases. When looking at these two type of exploits, we note that the following obfuscations stand out `InitBr`, `InitEn`, `InitOp`, `UpOp` and `Virtu`.

These obfuscations, as testability transformations, become more complementary for vulnerabilities than for crashes as shown when comparing Figure 4 and Figure 3. Their complementarity in several cases ranges from 1.31 to 28.74. Although all the cases are relevant from a testability transformation point of view, the case of `Virtu` is unique: with 28.74 of complementary, it discovers more BranchAv and StackCorruption exploits than other obfuscations. Furthermore, `Virtu` also affect to the original program in several cases. Moreover, `EncAri` and `UpOp` also stand — being the only obfuscations that produced BadInstruction exploits on the original program (0.02 and 0.01%).

On another note, `CObfs` and `CXX` do not change the nature of vulnerabilities significantly. Although, in the former keeps reduces DestAV vulnerabilities while incrementing ReturnAv when applied to the original program instead of to the obfuscated one. It is interesting to note the CObfsWord has a significant inverse instability, mainly because it generates BadInstruction errors that disappear on the original program. CXX keeps similar values to the original in all the cases. For LLVM obfuscations, the main differences in terms of exploits are `Split` and `BCF`. As testability transformations, these obfuscations find more SegFaultOnPc on the original program and, in the case of `BCF`, it finds 0.17% BadInstruction vulnerabilities that were not identified before. These obfuscations also reduces the number of DestAv exploits and increments ReturnAv.

RQ3.1: *ObfSec exposes how sensitive exploits are to the obfuscation process, especially in terms of the inverse instability which grows an order of magnitude with respect to the number of crashes. It shows that vulnerabilities can change, especially when using `Virtu`, `InitBr`, `InitOp` and `IpOp` obfuscations. It also shows that this is a cross-compiler phenomenon as also LLVM obfuscations (`BCF` or `Split`) change the nature of the exploits.*

RQ3.2: *ObfSec also shows how the quality offered by the obfuscation process to produce testability transformations also extends to detect novel exploits (in addition to crashes). Furthermore, ObfSec are significantly more complementary than crashes. The most relevant case is `Virtu`, whose complementary factor grows up to 28.74. It is also relevant the case of the BadInstruction vulnerability, unmasked in the GCC compilation by `UpOp` and on Clang by `BCF`.*

5.4. Analyzing Patterns in Error Changes

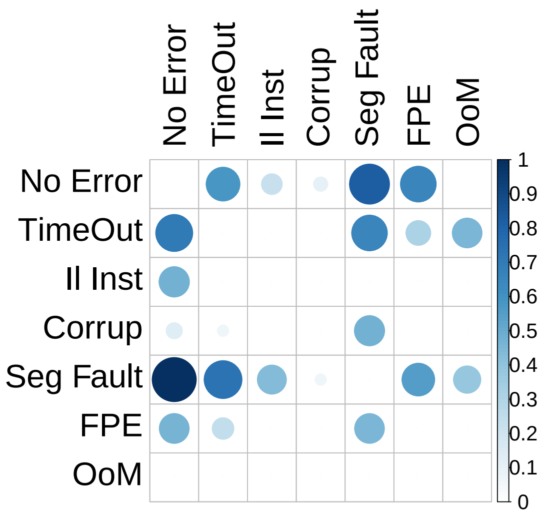
Variations between crashes and exploits follows specific patterns, i.e.: under different obfuscation methods, specific types of crashes or vulnerabilities tend to change to the same type of errors. We next study the patterns shown in the most unstable obfuscations. In particular, we look at the following six obfuscations: `AntiAl`, `InitBr`, `InitEn`, `InitOp`, `UpOp` and `Virtu`.

Crashes: Figure 5 shows the transitions between crashes during the cross-testing phase. Results are shown in normalized logarithmic scale. On Figure 5a, we see transitions of crashes after running our initial test suite: testing first on the original corpus and then on the obfuscations. Figure 5b shows the inverse process, which relates to the test suite generated on the obfuscated corpora. In both Figure 5a and 5b we observe a common trends. The fist thing to note is the significant amount of false positives that turn into segmentation faults and vice versa. This indicates that ObfSec is unmasking some of these crashes, which are difficult to identify otherwise. We also observe that segmentation faults can transit to almost every kind of crash, as it happens to some extent with to timeouts and no errors.

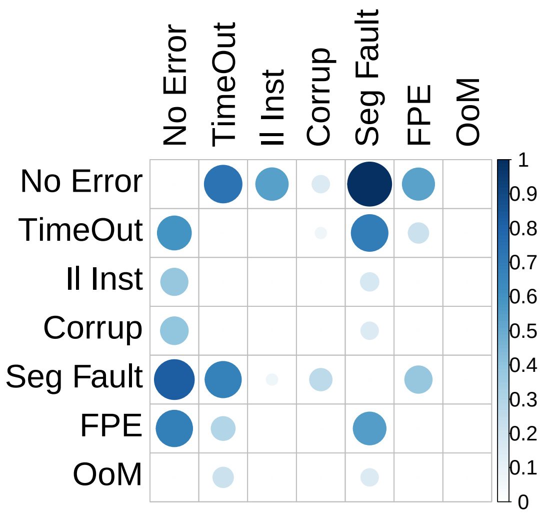
One of the most prevalent cases is the transition from segmentation fault to timeout, probably as a consequence of manipulating the boundaries of the different loops in a program. Another common transition from segmentation fault is to illegal instruction, which suggest that the former is prone to errors on the manipulation of the program instruction space. Interestingly, crashes such as as illegal instruction or corruption only change to either: no error, or — in the case of corruption — to segmentation faults. Similarly, FPE leads to these two types as well and rarely to timeouts. This suggests that there are no potential connections on the corpus logic between: corruptions, illegal instructions and floating point exceptions. Finally, several crashes transit to out of memory errors. This is a consequence of a memory abuse introduced by some obfuscations. In the inverse case, the results are similar: the transitions between corruption, illegal instruction and floating point exception are not connected either. Instead, segmentation faults and no errors are predominant.

Exploits: Figure 6 shows the corresponding transitions for the triage phase. In both Figure 6a and 6b we observe a similar trend, although the absolute number of transitions might differ. This is because exploits change more often, as the instability and inverse instability report (Section 5.3). There are three types of exploits that are predominately volatile: stack corruption, destination violation, and segmentation fault on PC. These three cases are related to the stack or they require modifications on the information stored in the stack. Thus, these transitions suggest that manipulations on the stack are sensitive to obfuscations. Effectively, these obfuscations are therefore increasing the attack surface of a program. This confirms the masking/unmasking properties of the obfuscation process discussed in Section 2.

Heap error and bad instruction are more stable exploits, as they do not transit to or receive transitions from obfuscations. There are two exceptions to this: DestAv and BadInstruction. In the inverse case, these results are similar but transpose, nevertheless, some pattern changes. For instance, BadInstruction transits to SegmentationFault and StackCorruption. This is a consequence of the complementary properties of some



(a) Original to Obfuscation



(b) Obfuscation to Original

Figure 5: Transitions between the types of crash during the cross-testing process. The left side shows transitions from the original test suite while the right side show the equivalent from the obfuscation.

obfuscations. They are finding connections between these exploits that were not exposed in the original test suite. Also, the transition proportions change for BranchAv and ReturnAv, leading to find more exploits related to BranchAv than in the original test suite.

These transitions between crashes and exploits suggest that unstable crashes, as well as exploits, require special attention during the bug fixing process. This is because they mask and expose vulnerabilities in ways that the developer can not control with standard try-catch clauses. Some of these crashes are segmentation faults, and some of the exploits are stack corruption, destination violation and segmentation fault on PC.

RQ4: *Obfuscations produce transitions between crashes an errors such that patterns emerge. The most severe crash is SegmentationFault that can evolve to almost any other type of crash. For vulnerabilities, there are three relevant vulnerabilities: DestAv, SegFaultOnPC and StackCorruption, all related to the way the stack is manipulated.*

6. Discussion

ObfSec shows that obfuscations are not fuzzification — or anti-fuzzing — techniques for a comprehensively large corpus of programs (70,137 programs). The obfuscation displaying the most significant resistance reduces the number of crashes detected to a third of the original, while anti-fuzzing techniques in the literature reduces it by 40 times (Jung et al., 2019). Our method can measure how some obfuscations are more

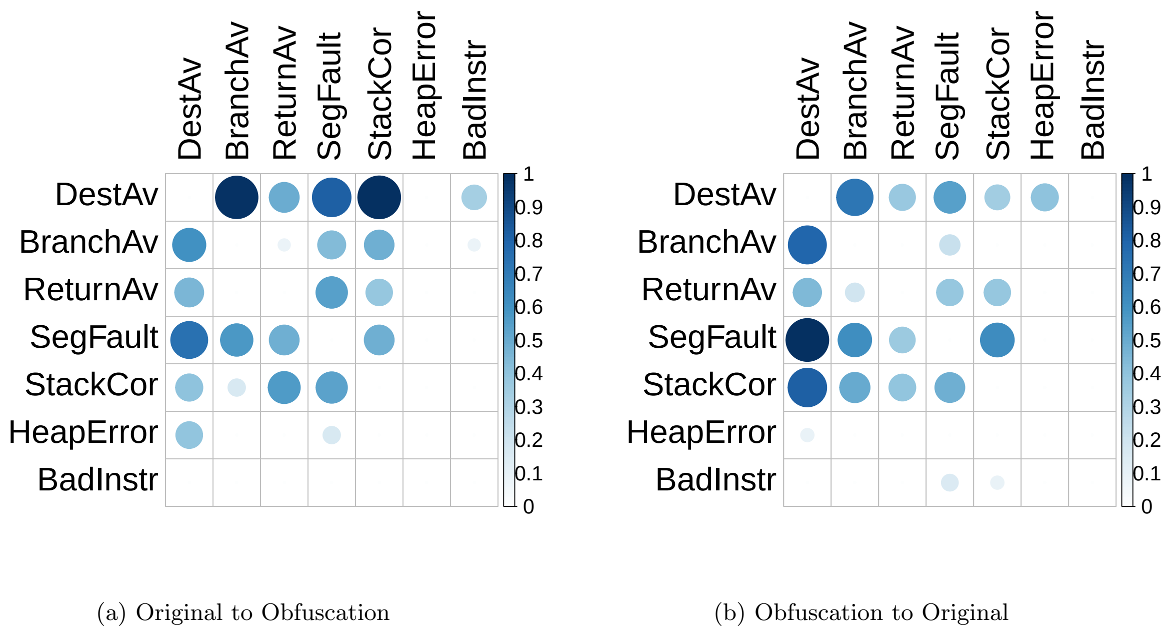


Figure 6: Transitions between the types of exploits during the cross-testing process. The left side shows transitions from the original test suite while the right side show the equivalent from the obfuscation.

prone to be exploited than others, and how some others can reduce the exposure.

Our cross-testing technique enable us to understand the life-cycle of a bug that goes through obfuscation. It shows that conservative obfuscations — those modifying literals and arithmetic operations — have less effect on the way bugs are manifested. However, obfuscations that modify the program's control flow graph alter the way the bug manifest. In particular, this affects specially to obfuscations that i) implement protections against taint or alias analysis, ii) add opaque predicates and, more importantly, iii) those based on visualization. Here, understanding how a bug turns into another (i.e., transitions) one is key to assess the severity of the exploitability of the crash. ObfSec shows that these transitions can be predicted as they follow very specific patterns. These patterns show that certain type of errors (i.e., illegal instructions, corruptions and floating point exceptions) are stable crashes. Instead, segmentation faults are completely unstable. In the case of exploits, those affecting the stack are also unstable. Using this information, our system can help developers on the daunting task of prioritizing the fix of the most critical errors in large software development projects.

There is a wealth of literature that explores how to triage identified errors (Wang et al., 2018b). Although our architecture admits any of such, our system currently relies on whether a crashed is deemed exploitable or not as given by the heuristics described in Section 3.1. We simply prioritize crashes that are exploitable. As a key novelty we contextualize this information with the type of obfuscation engine. As some errors are masked and certain obfuscations are more prone than others to unmask them, the triage process can not always be generalized to the type of algorithm as we show in Section 5.4. Thus, the triage phase is tailored to the program under test and the type of obfuscation under consideration. This process is paramount to guide developers on choosing the best obfuscation algorithm that matches their needs, especially in projects with limited budget, where there is a trade-off between finding and fixing all program crashes and the security of the obfuscation algorithm.

Finally, ObfSec also shows that obfuscating helps the fuzzer finding new crashes. Our findings call to rethink the software testing landscape to adopt the techniques used in obfuscation as effective testability transformation tools. This is, a metamorphic testing strategy where instead of mutating the input, we mutate the program itself. Finding the proper obfuscations enables fuzzers in the detection of novel bugs, working on semantically-equivalent representation of the program.

Threats to Validity. Our internal threats to validity relate to the corpus and obfuscation techniques selected. Our study focuses on programs with crashes, in this case the set of programs of CodeFlaws. These programs are individually small programs, nevertheless, we consider a sizable number altogether. Their size allows us to make particular studies of their behavior. As for the obfuscation techniques, we study the four most popular techniques currently available (Chen et al., 2016). Our external threats to validity relate to our choice of compilers. As we run our experiments on an LTS distribution of Debian, the compiler's version for GCC or Clang are not the last releases but stable ones. Thus, although the compiler can affect the crash generation process, the selection of the same compiler across the different experimental settings maintain the coherence.

7. Related Work

This section sets our contribution into the obfuscation arms race and discuss our novelty with respect of the testing techniques applied for obfuscation.

The obfuscation arms race. The obfuscation community has been evaluating the security of obfuscation engines as a way to measure their strength against adversarial analysts (Schrittwieser et al., 2016) – the *attacker*. While the obfuscation framework aims at concealing code –such as in the case of malware (Menéndez, 2021a), the attacker aims at understanding the hidden semantics to reconstruct its functionality (i.e., de-obfuscate). However, as Schrittwieser et al. show in (Schrittwieser et al., 2016), the attacker's resources directly affect the strength of the obfuscations. Code complexity also limits the analyst's resources, making simple obfuscations more effective. Schrittwieser et al. also show that software analysts do not focus on the functionality of the obfuscated program, which is part of our main goal. Our aim, instead, is to measure the reliability of the obfuscation process from a software quality perspective, while considering a malicious adversary trying to exploit vulnerabilities in the programs. Note that obfuscation is normally performed automatically, and it is applied at scale. This is, for instance, the case of Android apps (Wang et al., 2018a). Thus, it is paramount to understand how trustworthy current techniques are.

The goal of the analysts is to attack obfuscation with de-obfuscations techniques. These techniques aim to understand the software semantics, but they can also be used to measure the quality of the obfuscation (Ceccato et al., 2008, 2009). This measure of quality is refereed as "the security of the obfuscation" (Schrittwieser et al., 2016; Preda & Maggi, 2017) or its *potency* (Ceccato et al., 2008, 2009). Other measures extend information theoretic metrics to evaluate how good the security of an obfuscation is (Mohsen & Pinto, 2015).

The adversarial scenario aims to improve the quality of obfuscation techniques by understanding and strengthen them. A good example is the work of Rajendran et al. (Rajendran et al., 2012), who developed a system that was able to defeat the obfuscator EPIC (Roy et al., 2010), based on gate insertion on integrated circuits, in linear time, but it was used to strength this obfuscation and make it exponentially resistance. Some specific tools, like LOCO (Madou et al., 2006), interactively serve in both directions, they apply transformations to make the software easier or more complicated to read. Obfuscation can also be used as a mitigation strategy against trojans (Marcelli et al., 2018). Marcelli et al. proved that a software-based obfuscation can stop a Hardware Trojan by making semantic equivalent transformations on software.

Similarly to how compilers can introduce vulnerabilities on software in the presence of bugs (David, 2018), obfuscation can also create vulnerabilities when they are transforming buggy code. We aim to cover this usability gap by preventing bugs from spreading and exposing vulnerabilities through the obfuscation life-cycle. Recent works in the area (Xu et al., 2017; Mouha et al., 2018) have shown that the most prominent obfuscation techniques (e.g., (Xu et al., 2017; Gentry et al., 2015; Bitansky & Vaikuntanathan, 2018)) lack on *reliability*. This means that their performance and usability is too weak to generate practical obfuscation solutions, making it challenging for software security applications. This comes to show that related works can not find any *potent* and *reliable* approach, as a consequence of having no metrics that can lead to both directions, which is what we are introducing.

Different learning areas has also studied the impact that obfuscation has in programs, similarly to us. For instance, Berkovsky et al. studied the impact of data obfuscation for protecting users' identities in recommender systems (Berkovsky et al., 2012). In this scenario, obfuscation has also been used as an adversary to the recommender (Polatidis et al., 2017). On the other hand, obfuscation can protect users' identity, as Hazan et al shown in the case of the keystroke processes to guarantee that the users' identity is not stolen (Hazan et al., 2020).

Working at the code level, Sheneamer et al. studied how obfuscation affects to clone detection systems (Sheneamer et al., 2018). Yalcin et al. followed a similar line of research to validate plagiarism detection systems (Yalcin et al., 2022). Finally, from the perspective of malware analysis and security, obfuscation has always been a strong challenge to the detection problem as Hou et al. studies in web-based malware detection (Hou et al., 2010) and Fan et al. in Windows malware scenario (Fan et al., 2016).

Testing obfuscation. Approaches such as symbolic execution (Banescu et al., 2016a) have tried to deal with obfuscation. However, these techniques have issues to scale to bigger and complex programs because of path exploitation (Menéndez, 2021b). As obfuscation have proven to resists, not only symbolic execution, but other similar traditional testing techniques (Ma et al., 2014), we focused our efforts on using fuzz testing (Bounimova et al., 2013). The fuzzer creates inputs that traverse specific program areas. We use the same inputs to traverse the obfuscated version of the program in a controlled experiment. This allows us to map errors in the original code to errors in the obfuscated program for a given input. We also study how these errors are propagated. Jung et al. (Jung et al., 2019) and Guler et al. (Güler et al., 2019) provided intial results related to the abilities of fuzzers to some obfuscations as use cases. We are providing a large case study extending this information to stability, complementarity and exploitability of obfuscations.

Our idea is focused on commertial obfuscation, it is inspired by the work on indistinguishability obfuscation (IO), one of the most secure obfuscation techniques (Xu et al., 2017). IO use cryptographic methods to conceal the program making two obfuscated versions completly indistinguishable. When Xu et al. (Xu et al., 2017) collected different obfuscation techniques in order to identify strong obfuscations, their outcome pointed to the work of Garg et al. (Garg et al., 2016) on IO as the most secure. However, Garg et al. remarked that this approach lacks usability and it is a challenge to apply it to software. Their main outcome was that they could not find any secure and usable approach, as a consequence of having no metrics that can lead on those directions. Their aim is to motivate the communities to rethink this area, specially when the obfuscation implementation can contain bugs. Some authors have applied metamorphic testing to evaluate the usability of obfuscations (Chen et al., 2016), however, these works have mainly focused on the implementation quality, and not on the bug dissemination that the obfuscations can produced.

Although IO became one of the most promising areas of research, and some works have extended to multi-linear map reduction (Gentry et al., 2015) and functional encryption (Bitansky & Vaikuntanathan, 2018), it is still vulnerable to attacks (Coron et al., 2017). Besides, it is close to cryptography and, in the same way cryptography can lead to vulnerabilities (Mouha et al., 2018), IO, and other obfuscations, can lead to similar problems. Obfuscations are software transformations and as any software transformation, they can introduce vulnerabilities. As compilers can introduce vulnerabilities on software in the presence of bugs (David, 2018), obfuscation can also create vulnerabilities when they are transforming a bug.

8. Conclusions

We have introduced ObfSec, a methodology that leverages fuzz testing to measure the security of obfuscations from a testing perspective. ObfSec introduces four novel metrics and measures in three phases. The generation phase measures the resistance introduced by the obfuscations process into a fuzzing system (`fuzzibility`). The triage phase evaluates the severity of crashes that have gone through a obfuscation (`exploitability`). Finally, the cross-testing phase studies the testing divergence between the obfuscation and the original program (`instability`). We also measure to what extent the divergence can add value to the testability of the program (`complementarity`). Using these measures, on a corpus of 70,137 programs created with 20 different obfuscation algorithms, we were able to study how obfuscations affect the reliability of software in terms of testing, especially regarding the bug life-cycle.

As part of our future work, we will extend the application of our ObfSec system to different security problems. On the one hand, we will investigate how our metrics can guide the fuzzification (anti-fuzzing) process. On the other hand, we will use our system to improve the testability of the programs thorough a guided process that aims at selecting suitable testability transformations depending on the program context. This can also help to create new transformations. Finally, we will use our triage to generate rankings of potential problematic obfuscation and to systematically find fixes to those vulnerabilities.

Acknowledgements

This work has been supported by the "Ramon y Cajal" Fellowship RYC-2020-029401-I.

References

Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., & Pretschner, A. (2016a). Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (pp. 189–200). ACM.

Banescu, S., Collberg, C. S., Ganesh, V., Newsham, Z., & Pretschner, A. (2016b). Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016* (pp. 189–200). URL: `http://dl.acm.org/citation.cfm?id=2991114`.

Berkovsky, S., Kuflik, T., & Ricci, F. (2012). The impact of data obfuscation on the accuracy of collaborative filtering. *Expert Systems with Applications*, *39*, 5033–5042.

Bitansky, N., & Vaikuntanathan, V. (2018). Indistinguishability obfuscation from functional encryption. *J. ACM*, *65*, 39:1–39:37. URL: `http://doi.acm.org/10.1145/3234511`. doi:`10.1145/3234511`.

Bounimova, E., Godefroid, P., & Molnar, D. (2013). Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 122–131). IEEE Press.

Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., & Tonella, P. (2008). Towards experimental evaluation of code obfuscation techniques. In *Proceedings of the 4th ACM workshop on Quality of protection* (pp. 39–46). ACM.

Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., & Tonella, P. (2009). The effectiveness of source code obfuscation: An experimental assessment. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 178–187). IEEE.

Chen, T. Y., Kuo, F.-C., Ma, W., Susilo, W., Towey, D., Voas, J., & Zhou, Z. Q. (2016). Metamorphic testing for cybersecurity. *Computer*, *49*, 48–55.

Collberg, C., Thomborson, C., & Low, D. (1997). *A taxonomy of obfuscating transformations*. Technical Report Department of Computer Science, The University of Auckland, New Zealand.

Coron, J.-S., Lee, M. S., Lepoint, T., & Tibouchi, M. (2017). Zeroizing attacks on indistinguishability obfuscation over clt13. In S. Fehr (Ed.), *Public-Key Cryptography – PKC 2017* (pp. 41–58). Berlin, Heidelberg: Springer Berlin Heidelberg.

David, B. (2018). How a simple bug in ml compiler could be exploited for backdoors? *arXiv preprint arXiv:1811.10851*, .

Fan, Y., Ye, Y., & Chen, L. (2016). Malicious sequential pattern mining for automatic malware detection. *Expert Systems with Applications*, *52*, 16–25.

Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., & Waters, B. (2016). Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, *45*, 882–929.

Gentry, C., Lewko, A. B., Sahai, A., & Waters, B. (2015). Indistinguishability obfuscation from the multilinear subgroup elimination assumption. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)* (pp. 151–170). volume 00. URL: `doi.ieeecomputersociety.org/10.1109/FOCS.2015.19`. doi:`10.1109/FOCS.2015.19`.

Güler, E., Aschermann, C., Abbasi, A., & Holz, T. (2019). Antifuzz: Impeding fuzzing audits of binary executables. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (pp. 1931–1947).

Hammad, M., Garcia, J., & Malek, S. (2018). A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering* (pp. 421–431). ACM.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., & Roper, M. (2004). Testability transformation. *IEEE Transactions on Software Engineering*, *30*, 3–16.

Hazan, I., Margalit, O., & Rokach, L. (2020). Keystroke dynamics obfuscation using key grouping. *Expert Systems with Applications*, *143*, 113091.

Hou, Y.-T., Chang, Y., Chen, T., Laih, C.-S., & Chen, C.-M. (2010). Malicious web content detection by machine learning. *expert systems with applications*, *37*, 55–60.

Jung, J., Hu, H., Solodukhin, D., Pagan, D., Lee, K. H., & Kim, T. (2019). Fuzzification: Anti-fuzzing techniques. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (pp. 1913–1930).

Klees, G., Ruef, A., Cooper, B., Wei, S., & Hicks, M. (2018). Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2123–2138). ACM.

Ma, H., Ma, X., Liu, W., Huang, Z., Gao, D., & Jia, C. (2014). Control flow obfuscation using neural network to fight concolic testing. In *International Conference on Security and Privacy in Communication Systems* (pp. 287–304). Springer.

Madou, M., Van Put, L., & De Bosschere, K. (2006). Loco: An interactive code (de)obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* PEPM '06 (pp. 140–144). New York, NY, USA: ACM. URL: `http://doi.acm.org/10.1145/1111542.1111566`. doi:`10.1145/1111542.1111566`.

Marcelli, A., Sanchez, E., Squiller, G., Jamal, M. U., Imtiaz, A., Machetti, S., Mangani, F., Monti, P., Pola, D., Salvato, A., & Simili, M. (2018). Defeating hardware trojan in microprocessor cores through software obfuscation. In *2018 IEEE 19th Latin-American Test Symposium (LATS)* (pp. 1–6). doi:`10.1109/LATW.2018.8349680`.

Menéndez, H. D. (2021a). Malware: The never-ending arms race. *Open Journal of Cybersecurity*, *1*, 1–25.

Menéndez, H. D. (2021b). Software testing or the bugs nightmare. *Open Journal of Software Engineering*, *1*, 1–21.

Mohsen, R., & Pinto, A. M. (2015). Algorithmic information theory for obfuscation security. *IACR Cryptology ePrint Archive*, *2015*, 793.

Mouha, N., Raunak, M. S., Kuhn, D. R., & Kacker, R. (2018). Finding bugs in cryptographic hash function implementations. *IEEE Transactions on Reliability*, *67*, 870–884. doi:`10.1109/TR.2018.2847247`.

Polatidis, N., Georgiadis, C. K., Pimenidis, E., & Mouratidis, H. (2017). Privacy-preserving collaborative recommendations based on random perturbations. *Expert Systems with Applications*, *71*, 18–25.

Preda, M. D., & Maggi, F. (2017). Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, *13*, 209–232. URL: `https://doi.org/10.1007/s11416-016-0282-2`. doi:`10.1007/s11416-016-0282-2`.

Rajendran, J., Pino, Y., Sinanoglu, O., & Karri, R. (2012). Security analysis of logic obfuscation. In *Proceedings of the 49th*

Annual Design Automation Conference DAC '12 (pp. 83–89). New York, NY, USA: ACM. URL: `http://doi.acm.org/10.1145/2228360.2228377`. doi:`10.1145/2228360.2228377`.

Roy, J. A., Koushanfar, F., & Markov, I. L. (2010). Ending piracy of integrated circuits. *Computer*, *43*, 30–38.

Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, *49*, 4.

Sheneamer, A., Roy, S., & Kalita, J. (2018). A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications*, *97*, 405–420.

Tan, S. H., Yi, J., Yulis, Mechtaev, S., & Roychoudhury, A. (2017). Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion* ICSE-C '17 (pp. 180–182). Piscataway, NJ, USA: IEEE Press. URL: `https://doi.org/10.1109/ICSE-C.2017.76`. doi:`10.1109/ICSE-C.2017.76`.

Tao, Q., Wu, W., Zhao, C., & Shen, W. (2010). An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference* (pp. 270–279). IEEE.

Wang, P., Bao, Q., Wang, L., Wang, S., Chen, Z., Wei, T., & Wu, D. (2018a). Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*.

Wang, Y., Zhang, C., Xiang, X., Zhao, Z., Li, W., Gong, X., Liu, B., Chen, K., & Zou, W. (2018b). Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 1914–1927). ACM.

Xu, H., Zhou, Y., Kang, Y., & Lyu, M. R. (2017). On secure and usable program obfuscation: A survey. *arXiv preprint arXiv:1710.01139*, .

Yalcin, K., Cicekli, I., & Ercan, G. (2022). An external plagiarism detection system based on part-of-speech (pos) tag n-grams and word embedding. *Expert Systems with Applications*, *197*, 116677.

Zalewski, M. (2019). American fuzzy lop. URL: `http://lcamtuf.coredump.cx/afl/`.