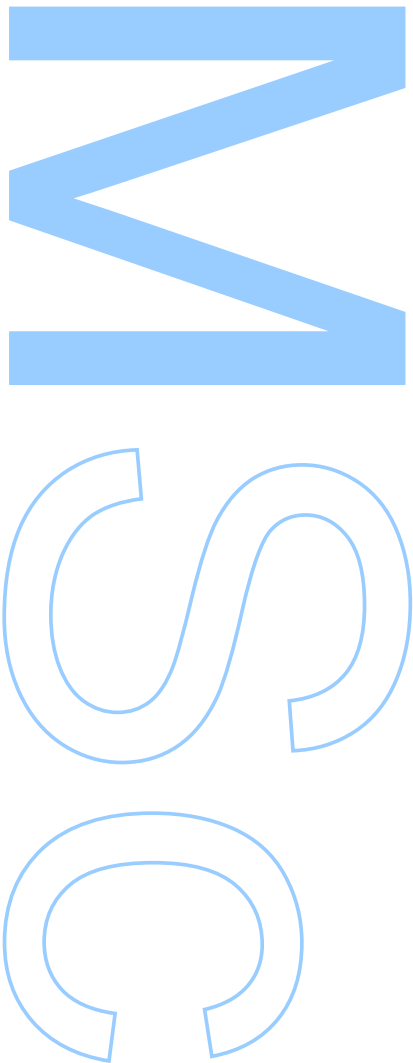
Resilience of code obfuscation to optimization and fuzzing

Frederico Lopes
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2022

Orientador
Prof. Dr. Eduardo Marques, Faculdade de Ciências da Universidade do Porto

Supervisor
Prof. André Baptista, Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Dedicated to my parents, friends and people who helped me achieve this goal.

Sworn Statement

I, Frederico Lopes, enrolled in the Master Degree of Information Security at the Faculty of Sciences of the University of Porto hereby declare, in accordance with the provisions of paragraph a) of Article 14 of the Code of Ethical Conduct of the University of Porto, that the content of this dissertation reflects perspectives, research work and my own interpretations at the time of its submission.

By submitting this dissertation, I also declare that it contains the results of my own research work and contributions that have not been previously submitted to this or any other institution.

I further declare that all references to other authors fully comply with the rules of attribution and are referenced in the text by citation and identified in the bibliographic references section. This dissertation does not include any content whose reproduction is protected by copyright laws.

I am aware that the practice of plagiarism and self-plagiarism constitute a form of academic offense.

Frederico Emanuel Almeida Lopes

30/09/2022

Acknowledgements

I would first like to thank my dissertation advisors, Professor Eduardo Marques and Professor André Baptista for their assistance and dedicated involvement in every step, throughout this long process. Their support was fundamental to reach this stage.

I also have to thank the persons that I encountered along my journey, including my colleagues at INESC TEC and most recently at Blaze Information Security. Their kindness, comprehension of my battles, and empathy was essential to be able to conciliate my studies and job.

I would also like to thank my parents and my friends for supporting me and providing a continuous encouragement throughout my years of study, with a lot of patience. This accomplishment would not have been possible without them.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

MSc. Information Security

Resilience of code obfuscation to optimization and fuzzing

by Frederico LOPES

Code obfuscation is being widely used not only to mask code in commercial products but also to obfuscate malware code. These obfuscations are made so that it is much harder to reverse engineer and analyse a given program, such that it becomes very hard or almost infeasible to do it. Although it is possible to manually reverse these obfuscation transformations, it is very time-consuming and impractical for much of the obfuscated software out there. Our goal is to analyse if state-of-the-art optimizers found on compilers and such can reverse these obfuscations, to what extent and what is their impact in automatic code analysis tools. For that, we built a framework for obfuscating, optimizing, analysing and extracting metrics from a set of programs ranging from very simple code, programs with several source-code files and functionalities, and programs specifically built to test automated analysis tools. These programs were obfuscated using the Tigress obfuscator of C programs, optimized using the Clang/LLVM compiler infrastructure and the Souper peephole optimizer, analysed with KLEE and AFL++, and had the metrics extracted from them, analysis and generated programs. To make this we implemented a set of scripts that automatically accomplish these steps, including building our own reproducible environment, and contributed to the update of an open source tool. From our results, we concluded that the optimizations could not systematically revert the obfuscations by improving the analysis of the tools, and in some cases they actually hindered the analysis. We could not find a variable that explains these results and the evidence shows that the behaviour of the analysis tools on the obfuscations and code optimizations is highly dependent on the original program. Another conclusion of this work is that the maximum time an analysis tool can run is the most important variable of all others to achieve a higher coverage and more reliable results.

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Ciência de Computadores

Mestrado em Segurança Informática

Resiliência de ofuscação de código a optimizações e fuzzing

por Frederico LOPES

A ofuscação de código está a ser amplamente utilizada não só para mascarar código em produtos comerciais, mas também para ofuscar malware. Estas ofuscações são feitas de tal forma que é muito mais difícil realizar engenharia reversa e analisar um determinado programa, tornando-a muito difícil ou quase inviável de fazê-la. Embora seja possível inverter manualmente estas transformações de ofuscação, é muito demorado e impraticável para grande parte do software ofuscado. O nosso objectivo é analisar se os optimizadores de última geração encontrados em compiladores e outros softwares podem inverter estas ofuscações, até que ponto, e qual é o seu impacto nas ferramentas de análise automática de código. Para isso, construímos uma framework para ofuscação, optimização, análise e extracção de métricas a partir de um conjunto de programas que vão desde códigos muito simples, programas com vários ficheiros de código-fonte e várias funcionalidades, e programas especificamente construídos para testar ferramentas de análise automática. Estes programas foram ofuscados utilizando o ofuscador Tigress de programas C, optimizados utilizando a infra-estrutura do compilador Clang/LLVM e o optimizador *peephole* Souper, analisados com as frameworks KLEE e AFL++, e tiveram as suas métricas extraídas, que sejam das análises ou programas gerados. Para tal, implementámos um conjunto de scripts que realizam automaticamente estas etapas, incluindo a construção do nosso próprio ambiente reprodutível, e contribuímos para a actualização de uma ferramenta de código-fonte aberto. A partir dos nossos resultados, concluímos que as optimizações não conseguiram reverter sistematicamente as ofuscações, melhorando a análise das ferramentas, e em alguns casos impediram a análise de obter melhores resultados. Não conseguimos encontrar uma variável que explicasse estes resultados e as provas mostram

que o comportamento das ferramentas de análise sobre as ofuscações e optimizações de código é altamente dependente do programa original. Outra conclusão deste trabalho é que o tempo máximo que uma ferramenta de análise pode correr é a variável mais importante de todas as outras para se conseguir uma cobertura mais elevada e reultados mais fidedignos.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code obfuscation is used to prevent an attacker from reverse engineering a software program. On commercial products, the end goal is to distribute software but prevent that someone reverse engineers it to ”steal” intellectual property (IP). On malware, the end goal is also to prevent reverse engineer and analysis of the given code, but the reason is so that it’s harder to understand its behaviour, what its functions are and how they are achieved. Although it does not prevent an attacker to reverse engineer the obfuscated program entirely, it makes the task very hard and impractical if the code is not changed to simplify its operations. To give an example, an obfuscation can be replacing classes and variable names with meaningless labels and adding unused or meaningless code to an application script. Software obfuscation is the technique of transforming the code of a program into another but considerably much harder to understand while retaining its semantics. This means that for each input, the modified program must output the same result as the original program (semantically equivalent). In essence, obfuscation is a protection against MATE (Man-At-The-End) attacks.

1.1 Motivation

Nowadays more and more applications are being analysed by automatic analysis tools in order to find bugs that the programmer and tester missed, and are starting to become part of the testing phase in Continuous Integration/Continuous Delivery (CI/CD) systems and in DevSecOps, both static and dynamic analysis tools. One of them is symbolic execution which analyses the program symbolically, meaning that an interpreter runs the

program where the inputs are symbolic, making the variables being symbolic values (formulas). When it encounters a branch, it divides the execution in two, one where the condition is true and another where it is false, and updates the constraints on those paths. The execution of the program resumes in both branches but separately. Because this technique can "test" all possible inputs, it is useful to detect memory bugs, discover which code in unreachable and with what inputs a certain part of the code is reachable. Another technique is fuzzing which essentially takes a set of inputs, mutates them, and executes the program with that new input to attempt to increase the coverage of the analysis on the target program.

1.2 Problem Statement

Although obfuscation is being widely used, in commercial and non-commercial reasons, commercial and academic state-of-the-art obfuscation techniques are vulnerable to a plethora of automated deobfuscation attacks, such as symbolic execution, optimization, taint analysis, or program synthesis. While there are several revised and enhanced obfuscation techniques, most of them can also be circumvented or removed entirely. This happens because these techniques only focus on a single attack vector, and attackers can use other more effective techniques to overcome them [1].

An open question is whether code optimisations supported by many compilers can aid or speedup dynamic analysis, given that in order to make code faster, optimization sometimes simplifies the code. Can they be seen as akin to "deobfuscation" in that sense? Some obfuscations can be reverted using code optimizations, but with which and to what extent? How to measure it? And how is their impact on automated analysis tools? Our main objective is to test if certain optimizations can revert obfuscation transformations and how automatic analysis tools such as symbolic execution and fuzzing are affected by it. There have been some studies that try to revert the transformations that obfuscation tools make, but no empirical study, tool or framework has been created to answer this question. Our assumption is that some optimizations can in fact revert some of the obfuscations, partially or event totally (e.g., encode arithmetic obfuscation), while others it cannot (e.g., virtualization obfuscation). Our aim is to develop a framework and workflow to standardize these types of tests so that we have an automated and analytical way to answer these questions.

1.3 Contributions

This thesis proposes a framework that can automatically obfuscate and optimize programs, analyse them, and extract metrics from them, including the coverage of the test cases generated by the analysis. The framework takes form as a set of scripts that automates these steps for each phase of the framework and outputs structured result files that can be used to analyse the results, a set of contributions to open source tools, and a study on how the analysis tools react when obfuscated code is optimized. Using the previously mentioned solutions, we studied the effect of optimizations on obfuscations using two analysis frameworks, KLEE and AFL++.

1.4 Outline

With the work described in this dissertation we made the following contributions:

- Chapter 2 presents and examines the state of the art of the field of obfuscations, optimizers, symbolic execution and fuzzing by giving an general overview of the techniques and tools used to better understand their functionality and how they work. It describes the features and limitations of each tool, and why we chose them to use on our framework.
- Chapter 3 states and describes our approach to using these tools to fulfill our requirements in all phases of the framework.
- Chapter 4 describes in detail our implementation. This includes the scripts created along with their functionality, the tools used and their options when called, and other software built to support the development and execution of the analysis.
- Chapter 5 described an evaluation of the proposed framework for a set of C programs, along with a detailed discussion of the associated results.
- Chapter 6 presents concluding remarks and discussed future work.

Chapter 2

State of the Art

In this chapter, we present the state-of-the-art on the topics contained in this thesis. We begin by giving an introduction to the topic of software obfuscation on Section 2.1, and on Section 2.2 by surveying related work about obfuscations, methods to evaluate obfuscations, symbolic execution and fuzzing tools including their limitations, ways to hinder symbolic execution, etc. After, on Section 2.3, we describe in detail what is obfuscation, which transformations exist, their purpose, and present an explanation of each one including the programs available that can apply them. On Section 2.4 we depict what is fuzzing, the advantages and disadvantages of the different types of fuzzing, and the tools available. On Section 2.5 we depict what is symbolic execution, its advantages and limitations, and the tools available. On Section 2.6 we present what optimization is used for in the current time, which optimizers exist and their advantages and disadvantages. Finally, on Section 2.7, we describe what is the cyclomatic complexity of a program and how this value is calculated.

2.1 Background

There is a good amount of theoretical research on obfuscating computation models but not many practical. While computation models are mathematical and their properties usually provable, real code is more complex with many edge cases and differences between programming languages and it is harder to solve their properties. Due to this complexity and differences in programming languages, the tools available focus on only one programming language and a set of obfuscations, in general.

The level of obfuscation of a program is defined by the transformations done to it and in what order. This is the concept of layered obfuscation [2]. When obfuscating a program, developers need to know the techniques available. Such knowledge is essential to choose the appropriate techniques and their order of application. Depending on the order of the obfuscations applied, the resulting program can have several orders of magnitude of obfuscation level. If a developer wishes to use virtualization as an transformation, then it should be applied first, so that the level of obfuscation of subsequent transformations is multiplied for the reason that the virtualization transformation adds a massive amount of code.

2.2 Related Work

Collberg et al. [3] proposed a general taxonomy for evaluating the quality of obfuscation transformations. This taxonomy states that code obfuscation should be scored according to these properties: *potency* against human-assisted (manual) attacks, *resilience* against automated attacks, speed and size cost (in terms of performance overhead) added by the obfuscation, and stealthiness, which measures the difficulty of identifying parts of obfuscated code in a given program. This article also proposed using several existing software features to evaluate potency, namely: program length, cyclomatic complexity, nesting complexity, data flow complexity, fan-in/out complexity, data structure complexity and object-oriented design metrics.

Banescu et al. [4] designed a system to predict the resilience of obfuscations against symbolic execution attacks using machine learning. To achieve it, it proposes a general framework for selecting the most significant software features to estimate the effort of an automated attack. It then uses the selected features to build regression models that can predict the resilience of different obfuscations against automated attacks (deobfuscation). To evaluate the obfuscations, they try to predict the time needed to deobfuscate a set of C programs, using an attack based on symbolic execution, or in other words, predict the effort needed by an automated deobfuscation attack. Before, Banescu et al. [5] also proposed using the effort needed to run a deobfuscation attack to measure software obfuscation resilience against automated attacks, but does not attempt to predict the effort needed for deobfuscation, which this paper tries to answer.

Banescu et al. [4] work is complementary to the Obfuscation Executive (OE) proposed by Heffner and Collberg [6]. The OE uses a framework to choose a sequence of obfuscations that should be applied to a program in order to increase its potency against manual attacks, while this work is focused on the resilience against automated attacks.

Yadegari et al. [7] addresses the problem that many of the deobfuscation approaches are obfuscation-specific, and although it is important and useful, such approaches fail to deliver correct results if the obfuscation is slightly different or uses another technique to achieve the same result. The approaches that exist to deobfuscate the virtualization transformation make strong assumptions about the structure and properties of the interpreter and dispatcher and this can cause the deobfuscator to not recognize the virtualization technique and hence not output the right result. They solve this issue by designing an automatic way to deobfuscate the two most biggest challenges in reverse engineering: emulation-based obfuscation (e.g., virtualization) and return-oriented programming. By using taint analysis, optimizations (code simplification, dead code elimination and control transfer simplification), concolic execution (concrete + symbolic execution), control dependency analysis (build the control-flow graph), they can effectively strip out the obfuscation and extract the logic of the original code.

Banescu and Collberg et al. [8] proposes a way to characterize the resilience of code obfuscation against automated symbolic execution attacks. What they found was that transformations that preserve program semantics can be easily deobfuscated by symbolic execution based deobfuscators, including virtualization. They also propose a new transformation that slightly changes program behavior, rendering symbolic execution based deobfuscation ineffective in practice.

Garba and Favaro created SATURN [9], a tool that can revert some obfuscations and recompile the obfuscated code based on the LLVM framework. More heavy obfuscations such has virtualization and self-modification are left out. They propose a way to lift executables back into the compiler intermediate language LLVM-IR and introduce a technique to recover the control flow graph of an obfuscated binary with an iterative control flow graph construction algorithm, based on compiler and external optimizations (Souper) and symbolic execution.

On the contrary of SATURN, Salwan et al. [10] focuses on defeating the virtualization

obfuscation. They present a generic and fully automatic approach to revert virtualization, using taint analysis, symbolic execution and code simplification. To show the result, they solved Tigress Challenge in a fully automated manner [11]. They also provide some potential defenses to these attacks.

Another approach is taken by Coogan et al. [12] that uses the programs interaction with the operating system though system calls to identify the instructions that affect these interactions. This is because the program must use this interface in order to make impact with his behaviour. The resulting set of instructions is an approximation of the original code, while the remaining are semantically uninteresting and are discarded.

Blazytko et al. developed Syntia [13], a generic automated code deobfuscation using program synthesis, guided by Monte Carlo Tree Search (MCTS). It works by simplifying execution traces by dividing them into distinct trace windows whose semantics are then ”learned” by the synthesis. They managed to synthesize the semantics of arithmetic instruction handlers in two of the best commercial virtualization-based obfuscators (VMProtect and Themida) with a success rate of more than 94%

Yadegari et al. [14] proposes three obfuscations that cause trouble to concolic analysis and leading them to imprecision and/or excessive resource usage. Two of these techniques can already be found in malware in the wild while the third is a simple variation on an existing technique. They show that with such obfuscations, existing symbolic execution based deobfuscators can’t revert the obfuscation or the time is increased in several orders of magnitude, and purposes methods to solve this issue using a combination of fine-grained bit-level taint analysis and architecture-aware constraint generation.

Branko Spasojević has done something similar to this thesis. He developed a plugin for the IDA Pro disassembler capable of rewriting binaries using optimization algorithms to facilitate the work of reverse engineers [15].

Suk et al. applies obfuscations and optimizations at source-level for C/C++ code [16]. They developed an optimization tool called SCORE (Source Code Optimization & REconstruction) that optimizes the obfuscated source-code. They claim that applying the optimizations at source-level is more effective because when using optimizers for Intermediate Languages (IR) it is hard to revert the IR code to source code (lifting), due to their

goal being to output optimized source-code, even when SCORE output source code is not optimized by the compiler optimization module.

Liang et al. combined trace analysis, symbolic execution and compiler optimization modules to revert the virtualization obfuscation [17]. They developed an symbolic execution based deobfuscator to automatically extract the semantic information of the virtual machine bytecode handlers (the execution of each bytecode corresponds to the invocation of its handler) and represent the handlers in high-level programming languages. This way they can process each bytecode and translate it into source code that represents the invocation of the corresponding functions, allowing to compile the code and in the process use the compiler's optimizer.

Canavese et al. [18] proposes a way to predict the value of the potency of an obfuscation by using Artificial Neural Networks (ANN). This solves the issue of only being able to compute the potency value after the obfuscations has been applied and can now estimate a priori it's value. They base their calculation on Collberg et al. research where he proposes a way to compute the potency by measuring changes in complexity metrics induced by the obfuscation transformations [3].

Unfortunately in this field there is a lack of maturity in the tools available. Only a small subset of them work as expected and there is always something missing or does not work quite well. We selected an obfuscator of C code, a symbolic execution program and a fuzzing program. These tools were selected by their maturity, features and documentation. Although obfuscation has been around for over 30 years, the tools available, commercially and open-source, are not well maintained, have limitations and have a lack of documentation, especially on errors.

Nowadays there are many tools that implement some techniques to obfuscate programs, and there are obfuscators for various programming languages. In this thesis, we will focus our efforts in the obfuscation, optimization and analysis of C programs due to the fact that obfuscation, optimization, symbolic execution and fuzzing tools are more mature in this language and compiled code.

2.3 Obfuscations

When referencing obfuscation, each tool implements its set of techniques and features of those transformations. There are many ways to achieve a certain transformation and they can vary in effectiveness, arduousness of reverse engineering by manual and automatic tools, code size, and speed of the resulting code.

The obfuscator we chose was Tigress [19]. Because we could not lift the code to an analysable format we had to opt for an source-code C obfuscator and the most complete one is Tigress. Tigress is an obfuscator for the C language that supports many defenses against both static and dynamic reverse engineering and de-virtualization attacks. Tigress as a source-to-source transformer, it takes a C source program as input and returns a new C program as output. It is written in OCaml (about 80,000 lines of code) on top of two libraries: CIL (a C front end) and MyJit (a runtime code generator). It is one of the state-of-the-art obfuscators due to it's numerous transformations, options and usage in research papers. Below we highlight a set of obfuscations performed using Tigress that we used on our tests.

2.3.1 Literals Encoding

This transformation replaces literal integers with opaque expressions and literal strings with calls to functions that generate them at runtime. This makes it harder to determine the strings in a program by static analysis. One example is to hide the IP address of the malware's control and command server.

Example:

```
1  void decodeStr(char *string){
2      ...
3      for(int i=0; string[i] != '\0'; i++)
4          string[i] = decodeChar(string[i]);
5      ...
6  }
7
8  int main(){
9      ...
10     char* string = "Hsc1Had2jHG";
11     decodeStr(string);
12     ...
```

```
}
```

LISTING 2.1: Literals Encoding string example

2.3.2 Arithmetic Encoding

Replace integer arithmetic with more complex expressions. This transformation essentially adds redundant operations to arithmetic expressions. It is used so that it is harder to determine the operands and results of an arithmetic operation.

Example:

```
x + y = x - ¬y - 1
      = (x ⊕ y) + 2*(x ∧ y)
      = (x ∨ y) + (x ∧ y)
      = 2*(x ∨ y) - (x ⊕ y)
```

LISTING 2.2: Arithmetic Encoding example

2.3.3 Data Encoding

This transformation behaves just like literals encoding but applies the technique to variables. This makes it harder to determine the result of a variable and its operands.

Example:

```
int main() {
        int arg1 = ...
        int arg2 = ...
        int a = arg1;
        int b = arg2;
        int x = a*b;
        printf("x=%i\n",x);
}
```

LISTING 2.3: Data Encoding example (Before)

```
int main() {
        int arg1 = ...
        int arg2 = ...
        int a = 1789355803 * arg1 + 1391591831;
        int b = 1789355803 * arg2 + 1391591831;;
```

```
6         int x = ((3537017619 * (a * b) - 3670706997 * a) - 3670706997 * b)
    + 3171898074;
7         printf("x=%i\n", -757949677 * x - 3670706997);
8 }
```

LISTING 2.4: Data Encoding example (After)

2.3.4 Branches Encoding

The goal of this transformation is to make it harder for automatic analysis tools (such as disassemblers) to determine the target of branches. It transforms branch jump operands to indirect jumps.

Example:

```
1 jmp L
```

LISTING 2.5: Branches Encoding example (Before)

```
1 void bf(){
2     ra <- L
3     ret
4 }
```

LISTING 2.6: Branches Encoding example (After)

2.3.5 Opaque Predicates

An opaque predicate is a predicate whose value is known to the obfuscator but is difficult to deduce. Essentially it is a branch where it is always true or false. Normally is used together with junk code so that it produces a cluttered control flow graph with paths that are infeasible and/or redundant. This makes analysis based on the control flow graph very hard. There are three types of opaque predicates. Invariant opaque predicates are built from well-known algebraic theorems like $x^2 < 0$, which is always false. Contextual opaque predicates are made using a value which is determined on runtime but only when executing a specific context. This means that the obfuscator knows the result of a predicate at some moment in the code but if the context is not satisfied the result is the inverse. The last is dynamic opaque predicates, being the most advanced one, which always evaluates the predicate to the same value, but the initial value is different is each run.

Example:

```
...
int a, b;
...
if (7*a*a-1 != b*b)
    //always true
    runCode();
else
    bogusCode();
```

LISTING 2.7: Opaque Predicates example

2.3.6 Control-Flow Flattening

Control-Flow Flattening puts every basic block at the same level. It employs a dispatcher and keeps a variable to determine the next basic block to execute. This is a dispatcher-based obfuscation that puts all basic blocks at the same level. The order of execution of the code blocks is determined at runtime, so it's is impossible to know the control flow of the program without executing the program. Tigress uses Chenxi Wang's algorithm to achieve this. The example below transforms a while loop into another form with switch-case.

Example:

```
int a = 1;
int b = 2;
while (a < 10) {
    b = a+b;
    if (b > 10)
        b--;
    a++;
}
printf("%d", b);
```

LISTING 2.8: Control-Flow Flattening example using switch (Before)

```
int swVar = 1;
switch (swVar) {
case 1:
    a = 1; b = 2;
    swVar = 2;
```

```
6       break;
7   case 2:
8       if (!(a<10))
9           swVar = 6;
10      else
11          swVar = 3;
12      break;
13  case 3:
14      b = b+a;
15      if (!(b>10))
16          swVar = 5;
17      else
18          swVar = 4;
19      break;
20  case 4:
21      b--;
22      swVar = 5;
23      break;
24  case 5:
25      a++;
26      swVar = 2;
27      break;
28  case 6:
29      printf("%d", b);
30      break;
31  }
```

LISTING 2.9: Control-Flow Flattening example using switch (After)

2.3.7 Anti Taint Analysis

The goal of this transformation is to disrupt analysis tools that make use of dynamic taint analysis. It works by transferring the value of the tainted variable indirectly to an untainted variable. Tigress counts up to the variable of the tainted variable and applies the value (of the count variable) to the untainted variable bit-by-bit, tested in an if statement.

```
1   int taintedVar, untaintedVar;
2   scanf("%d", taintedVar);
3   for (int i=INT_MIN; i < INT_MAX; i++)
4       if (i == taintedVar)
5           untaintedVar = i;
```

LISTING 2.10: Anti Taint Analysis example

2.3.8 Anti Alias Analysis

The goal of this transformation is to disrupt static analysis tools that make use of inter-procedural alias analysis. Alias Analysis is composed of techniques which attempt to determine whether or not two pointers can point to the same object in memory in a given moment of the code. There are many different algorithms for alias analysis and many different ways of classifying them but Tigress current implementation is simplistic: it simply replaces all direct function calls with indirect ones.

Example:

```
1 x = foo(n)
```

LISTING 2.11: Anti Alias Analysis example (Before)

```
1 void *arr[]  = {..., & foo, ...};
2
3 int expr = 42;
4
5 int x = ((int (*)(int n ))arr[expr])(n);
```

LISTING 2.12: Anti Alias Analysis example (After)

2.3.9 Virtualization

Virtualization is an emulation-based obfuscation and consists in interpreting the program instead of running it directly, just like Python is run. Virtualization-based obfuscators transform the instructions of the original code to instructions of a virtual instruction set, that is specialized for that specific interpreter. The generated interpreter consists of a virtual instruction set, specific of that interpreter, a bytecode array which contains the code in the virtual instruction set, a virtual program counter (VPC) and a virtual stack pointer (VSP) that act just like the normal program counter and stack pointer, a dispatch unit, and a list of instruction handlers, one for each virtual instruction. To make the program more difficult to analyse the transformation has been designed to induce as much diversity as possible, i.e. every decision made is dependent on a randomization seed.

```
enum ops {Locals = 116, Plus = 135, Load = 60, Goto = 231,
          Const  = 3, Store = 122, Return = 72};

unsigned char bytecode[41] = {
 Locals,24,0,0,0,Const,1,0,0,0,Locals,24,0,0,0,Load,Plus,Store,Locals,
 28,0,0,0,Const,0,0,0,0,Store,Goto,4,0,0,0,Locals,28,0,0,0,Load,Return};

int main() {
  while (1) {
    switch (*pc) {
    case Const: pc++; (sp+1)->_int = *((int *)pc); sp++; pc+=4; break;
    case Load: pc++; sp->_int = *((int *)sp->_vs); break;
    case Goto: pc++; pc+= *((int *)pc); break;
    case Plus: pc++; (sp+-1)->_int=sp->_int+(sp+-1)->_int; sp--; break;
    case Return: pc++; return (sp->_int); break;
    case Store: pc++; *((int *)(sp+-1)->_vs)=sp->_int; sp+=-2; break;
    case Locals: pc++; (sp+1)->_vs=(void *)(vars+*((int *)pc));
                 sp++; pc+=4; break;
}}
```

LISTING 2.13: Anti Taint Analysis example

2.4 Fuzzing

Fuzzing or fuzz testing is an automated software testing technique and the most relevant in vulnerability discovery. It is in essence a dynamic concrete execution technique, because the values of the variables are not symbolic but rather always concrete. Fuzzing consists in providing random inputs mutated from an initial set of user-provided inputs that can cause crashes, assertion failures or memory leaks, finding potential security vulnerabilities [20]. If the application crashed when given a specific input, the input was considered to have triggered a bug.

Fuzzing can be improved with several enhancements to help the fuzzer achieve a higher code coverage [21]. Bellow we highlight some of these improvements:

- Coverage-based fuzzing: coverage guided fuzzers seek to generate inputs that maximize the amount of code executed in the program for the reason that the more coverage an analysis has, the more likely it is to find a vulnerability [22]. American

Fuzzy Lop [23] is a state-of-the-art fuzzer that uses coverage as a metric for its guidance and has found several bugs in popular open-source software. Although the fuzzer will select the inputs that can help in reaching higher code coverage, the lack of semantic awareness of the program makes the fuzzer not knowing which parts of the input to mutate to cause a certain block of code to be executed.

- Taint-based fuzzing: taint-based fuzzers guide their exploration by analysing which inputs control which variables making the fuzzer aware of the parts an input should be mutated for it to drive execution down a given path but it is still unaware of how to mutate the input [24]. Depending on the taint technique employed it can deliver more complete results or performance improvements. The most complete taint analysis is a bit-level taint analysis due to it's precision but it requires more memory when used [25].

- Symbolic-assisted fuzzing: symbolic execution engines have very limited scalability due to the path explosion problem and one solution is to use faster analysis techniques such as fuzzing. This approach leverages the strength of both techniques, i.e., semantic awareness of the program that is used to mutate inputs that can increase code coverage and faster analysis time because of the fuzzing part. Symbolic execution is used to collect constraints for an input and negate them to generate new inputs because the more coverage an analysis has, the more likely is to find an vulnerability. Due to this some researchers are using the two techniques together in order to obtain better performance [26] [27].

In order to use fuzzers at their full potential, the target program needs to be instrumentalized with sanitizers. One example is to use the address sanitizer that causes the program to crash if an memory error happened, e.g., buffer overflow. By making the program crash when improper operations are done the fuzzer can know that a bug was found with that specific input. Normally this is done using the compilers embedded sanitizers.

Unfortunately, fuzzers don't function properly if initial test cases are not provided. Without carefully crafted test cases to mutate, a fuzzer has a hard time exploring most part of the code, covering only shallow functionality.

There are 4 types of fuzzers [20]:

- Black-box fuzzing: black-box fuzzing is the oldest and simplest form of fuzzing. These type of fuzzers are useful when dealing with unknown applications but because they lack knowledge of the internals of the target program, they do not deliver the best results because they do not know if the program reach a higher coverge or not.

- Gray-box fuzzing: gray-box fuzzers work without knowing the source code of the target program and gain the information about the internals of the program though program analysis.

- White-box fuzzing: white-box fuzzers build on heavyweight program analysis or constraint solving, and can this way collect more information about it though analysis on the code and how inputs affect the program state. Leverages techniques from program analysis just like gray-box fuzzing but use more effective methods such as symbolic execution. The advantage with white-box fuzzing is that every new test generate will in principle cover a new execution path in the program, increasing the coverage. This is the type of fuzzer that offers better results.

- Grammar-based fuzzing: in these type of fuzzers the user needs to provide a grammar that represents the input format of the target program so that the fuzzer can generate tests based on the rules the user has provided for the input. This search if very powerful because the user can guide the exploration to test specific patterns in the input and gives the fuzzer a way to generate inputs without the user needing to supply an initial set of inputs.

2.4.1 AFL++

There are lots of fuzzers but unfortunately very few are still actively maintained and have updated and efficient techniques. We chose a black-box fuzzer, AFL++ (AFLplusplus) [28], due to being the state-of-the-art tool in this category. It comes with it's own compiler to instrument the code with checks and information. Even though it needs access to the code to instrument the target program but can also instrument binary targets using QEMU [29], because it does not employ any heavyweight program analysis or constraint solving as it is not entirely a white-box fuzzer. In order for it to function, the final binary needs to be instrumented and a set of initial inputs must be given so that the fuzzer can mutate

them and try new inputs. We can also choose the number of cores and the maximum time that the fuzzer runs, which gives us the ability to do multi-core execution.

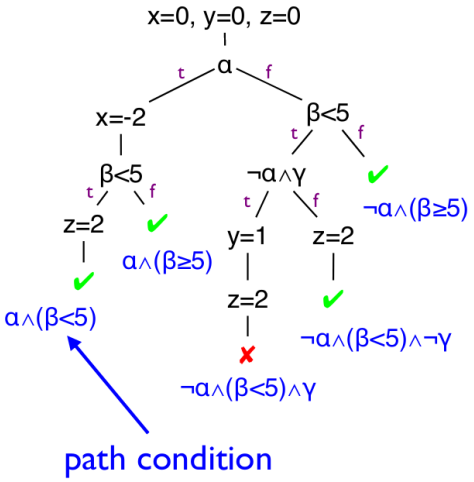
2.5 Symbolic Execution

Symbolic execution, a white-box fuzzing method, is a program analysis technique introduced in the mid-1970s to test whether certain properties of programs can be violated, i.e. the program does not behave correctly for all types of input. Some examples might be that no divisions by 0 can be done, no NULL pointers are dereferenced, there is no backdoor that can be used to bypass authentication, etc. Although, in general, there is no automatic way to determine a property (e.g., the target of an indirect jump), heuristics and approximate analysis have proved in practice to be useful in a variety of environments, including programs used in critical systems and in highly secure applications. This is useful in software testing where it is required that certain properties to be always true in all usage scenarios. One of the most known approaches is to test the program with different and possibly random inputs exhaustively but it has limitations such as a code block that only executes if the program is given a very specific input. Symbolic execution allows us to explore many possible execution paths at the same time and aggregate them by abstracting the variables into symbols and conditions (predicates) into equations and using constraint solvers to compute for which inputs a given code block is executed and to give an concrete input that satisfies those conditions. The state of the art in symbolic execution research and implementation is described by Baldoni et al. [30] and by Zhang et al. [31]. We will highlight some of the most essential information contained in this study, which is one of the best research papers into the state of this technique and it's advantages and disadvantages.

In 2013 DARPA announced its competition, the Cyber Grand Challenge (CGC) [32], lasting two years in order to boost the creation of automatic systems for detecting security flaws, exploit them and applying patches in near real-time. Some symbolic execution tools have been, since 2008, 24/7 analysing programs testing various software applications from Microsoft, finding for example close to 30% of all bugs discovered by file fuzzing during the development of Windows 7, which other analysis programs and blackbox testing failed to discover [33].

```
// a, b and c symbolic
void func(int a, int b, int c){
  int x = 0, y = 0, z = 0;
  if (a) {
    x = -2;
  }
  if (b < 5) {
    if (!a && c) {
      y = 1;
    }
    z = 2;
  }
  assert(x + y + z != 3);
}
```

(a)



(b)

FIGURE 2.1: Symbolic execution technique example

On the figure above 2.1 you can view the theory behind symbolic execution. Starting with the function arguments, because we don't know their values in concrete, these variables are declared symbolic (abstract) and at this moment in the execution of the code the only existing restrictions are the inferior and superior values of these variables, namely the limits of type `int` in C. After that, the variables `x`, `y` and `z` are concretely declared with the value 0. Following the code, we stop at the first predicate an divide the code exploration in two parts: one in that the variable `a` is zero and another in which it is different of zero, executing both paths independently. This way we can separate different execution paths and test all possible ways of execution the function. By dividing the execution in two, we also change the restrictions of the variable `a` and update the value of the variable `x` for each path. We follow the same steps for all the branches in the code along the execution. At the end of the function we have an assert, that tests if the sum of `x`, `y` and `z` is different than 3. If it is true, the program can return of the function and continue it's execution, else the execution is aborted and the values for which the program aborts are reported. Using this technique we can specify the values of the variables for which the function will have an incorrect behaviour, identifying more easily the why of that behaviour, facilitating the resolution of errors.

2.5.1 Symbolic Execution Limitations

In the last figure 2.1, symbolic execution can identify every input that will make the assert call fail. This is achieved by exhaustively explore all possible execution states of the program. From a theoretical point of view, exhaustive symbolic execution provides a sound and complete methodology. The meaning of soundness is preventing false negatives, i.e., all possible unsafe inputs are guaranteed to be found, while completeness is preventing false positives, i.e., all inputs values that are considered unsafe are in fact unsafe.

Unfortunately the plain version of symbolic execution has many drawbacks when dealing with other than small applications so there is a need to make a trade-off between soundness and performance if we want it to be scalable. The challenges of processing real-world applications are the following:

- Memory: one of the obstacles of symbolic execution is the way each tool handles pointers, arrays or other complex objects due to a fact that representing such memory in symbolic form is very demanding. One example is using pointers that manipulate jump instructions, making the addresses being described by symbolic expressions. There are different memory models and each one have their advantages and disadvantages [34] [35].

- State space explosion: symbolically executing every feasible path of the program does not scale in large programs. The number of possible paths grows exponentially with the increase of the program size an can even be infinite in the case of programs with infinite loops [36]. The solutions to this problem usually involve in using heuristics to find the paths that can increase the coverage of the program the most [37], reducing the execution time by making the execution of independent paths parallel [38], or by merging states with similar ones [39].

- Constraint solving: Satisfiability Modulo Theories (SMT) solvers can handle large amounts and complex combinations of constraints over a very large number of variables but constructs such as non-linear arithmetic (second and beyond order polynomials) is an obstacle to efficiency. There are also many ways to implement SMT solvers and there is also a competition that benchmarks these solvers in several areas [40].

- Environment interaction: normally programs need to make their behaviour have impact in the system where they are executing. The manner is to make calls to library code and use system calls. These calls cause side effects like creating files, opening a network socket, etc., and affect the execution of the program later and must be accounted for, although exploring every possible external interaction outcome may be unfeasible.

- Obfuscated code: obfuscated code uses various environment elements to decide if it should execute or not some block of code and many times the instruction pointer (IP) becomes symbolic due to the presence of some jump addresses being computed at runtime. One way to overcome this issue is to use taint analysis, by registering which variables have influence on the program, even in the computation of a jump target [34].

A symbolic execution engine can be augmented with fuzzing to reach deeper states in the exploration more quickly and efficiently [41] [42] because as we saw, symbolic execution has many scalability limitations.

Symbolic execution engines leverage the strength of SMT solvers to figure out if a certain path is actually possible, or if it should be discarded. Satisfiability modulo theories (SMT) is the problem of determining whether a mathematical formula is satisfiable or not. It generalizes the Boolean satisfiability problem (SAT), that uses boolean variables, to be able to solve more complex formulas containing all kinds of data like integers, floats, arrays, strings, etc. SMT solvers (e.g., Z3, STP, etc.) have been used as a building block for a wide range of tools across computer science, for example in program analysis such as symbolic execution [43].

2.5.2 KLEE

There are several open-source symbolic execution tools available, each one with their advantages and disadvantages. Some analyse binary files, e.g., S2E [44] [45] [46], Mayhem [47], Miasm [48], angr [27] [21] [49], Triton [50], while others need access to the source code, e.g., SymCC [51] or to intermediate representations (IR), e.g., KLEE [52]. The ones that analyse binaries first lift the code to an IR such as QEMU IR (e.g., S2E) or to their own IR (e.g., Miasm).

KLEE is an open source dynamic symbolic execution engine built on top of the LLVM compiler infrastructure and has been chosen as the symbolic execution engine for our tests for many reasons. It is one of the most complete symbolic execution engines and is widely supported. KLEE was designed with knowledge from a previous engine called EXE [53], which two of the three developers made KLEE, and employs a variety of constrain solving optimizations, represents program states compactly and uses search heuristics to achieve high code coverage [52]. The way that KLEE handles the external environment is to use models that understand the semantics of the desired action well enough to generate the required constraints. This allows KLEE to analyse bigger applications and system-intensive programs due to these performance optimizations. Zhang et al. [31] describes the current state of symbolic execution and makes an updated analysis of KLEE and its state in terms of improvements and obstacles.

KLEE analyses LLVM-bitcode files so it is needed to have the source code or use a lifter that can lift binaries to source code or LLVM Intermediate Representation (LLVM-IR).

2.6 Optimizations

Optimizers are in our everyday life when compiling code. They are as old as compilers and aid developers in focusing more on the code they write and less in small optimizations. Normally they are found embedded in compilers (GCC) and sometimes they are a separate module of a large infrastructure (Clang). Because they were created a long time ago and everyone that compiles human readable code to machine code uses it, they are very mature and offer very high effectiveness and correctness (semantically equivalent).

When talking about compiler optimizations, there are 4 main levels [54]:

- -O0: Means ”no optimization”: this level compiles the fastest and generates the most debuggable code.
- -O1: Somewhere between -O0 and -O2.
- -O2: Moderate level of optimization which enables most optimizations.
- -O3: Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

Each of these levels have their optimizations enabled and make a difference in the final program in terms of speed and size.

In the category of optimizers, the tools available are very mature and have plenty of features and documentation. The most commonly used optimizers are compiler optimizers due to the fact that is in everyone best interest to optimize their code to deliver faster computations. Additionally, there exists an optimizer which is not part of compilers but uses several techniques to optimize programs, called Souper, as we will see later.

2.6.1 Clang/LLVM

LLVM is a set of open-source modular and reusable compiler and toolchain technologies, which can be used to develop a compiler front end for any programming language and a back end for any instruction set architecture (ISA). LLVM is designed around a language-independent intermediate representation (IR), called LLVM IR, that serves as a portable, high-level assembly language that can be used in a variety of ways like optimization, translation, analysis, etc. LLVM provides the middle layers of a complete compiler system, that accepts intermediate representation (IR) code from a compiler (front end) as input and emits an optimized IR. After that, the optimized IR can then be compiled into architecture specific code or CPU specific code and linked to external libraries.

The Clang tool is a front end compiler that is used to compile programming languages such as C++, C, Objective C++ and Objective C into machine code and is a direct competitor of GCC. At its core, Clang uses the LLVM infrastructure as its back end and it has been included in the release of the LLVM since the LLVM 2.6. It has a wide support of architectures, CPU's, and language features (e.g., C++20, etc.). By using LLVM as it's backend, it means that it can compile any language that can be lowered to LLVM IR, meaning that creating compilers for new languages is much easier (Apple's Swift language uses LLVM as its compiler framework, and Rust uses LLVM as a core component of its tool chain).

The Clang Compiler has been designed to work just like any other compiler. Clang works in three different stages [55]:

- First stage: the frontend parses the source code. It checks the code for errors and builds a language-specific Abstract Syntax Tree (AST) to work as its input code.
- Second stage: optimize the AST that was generated by the frontend.

- Thirst stage: generate the final code (machine code) which can be dependent on the target system.

Clang has a design similar to LLVM because it uses a library-based architecture [56]. In this design, various parts of the front-end can be cleanly divided into separate libraries which can then be mixed up for different needs and uses. It also includes a static analyzer [57], sanitizers [58] and several code analysis tools.

We will use Clang to compile the code from our test set and use it's optimization levels to test their strength in removing obfuscations.

2.6.2 KLEE Optimize Flag

The KLEE optimize flag runs a built-in optimizer before the analysis and applies a subset of compiler optimizations to the code [59] [60]. This optimization only happens in the analysis phase due to being an optimization set chosen for the KLEE symbolic execution framework, a set with optimizations contained in clang, and does not output the resulting LLVM bitcode. This option is disabled by default and can be enabled with KLEE's `--optimize` command line option.

2.6.3 Souper

Souper is a peephole optimizer. Peephole optimization is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the peephole or window [61]. Peephole optimization involves changing the small set of instructions to an equivalent set that has better performance and is usually performed late in the compilation process after machine code has been generated. This is because peephole optimizations also use hardware specific instructions to optimize the code.

The more common techniques applied in peephole optimization are:

- Null sequences: Delete useless operations.
- Combine operations: Replace several operations with one equivalent.
- Arithmetic operations: Use arithmetic rules to simplify or reorder instructions.
- Special case instructions: Use special instructions that are only available in certain CPU's to optimize code.

- Address mode operations: Use memory access instructions of the targeted hardware.
- Redundant load and store elimination: Redundancy in memory operations is eliminated.
- Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time (e.g., instead of multiplying a number by 2, substitute the instruction with a shift left instruction.)
- Pattern matching: Transform code using patterns that are known to be equivalent but faster.
- Constant propagation: Process of substituting the values of known constants in expressions at compile time.
- Constant folding: Process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.

Souper uses an SMT solver to optimize the target program even further. Solvers are also a key enabler for program synthesis, which supports the discovery of new optimizations that are out of reach for naive search [62].

The input and output of souper is LLVM bitcode, but internally it uses it's own IR, Souper IR, that has 51 instructions, all derived from LLVM IR. This is very positive since KLEE analyzes LLVM bitcode files.

2.7 Cyclomatic Complexity

Metrics in software development are used to quantify and qualify software characteristics to measure it's performance, size, complexity, etc. These metrics can then be used to improve the software built or to make a decision on it's usage and implementation. One of the existing metrics is the cyclomatic complexity. Cyclomatic complexity is a software testing metric used to measure the complexity of a program. It is a quantitative measure of independent paths, paths that have at least one edge which has not been traversed before in any other paths, in the source code of a program and can be calculated using control flow graphs for functions, modules, methods or classes within a program. Programs

with lower Cyclomatic complexity are easier to understand and less risky to modify. This metric was developed by Thomas J. McCabe and it is based on the control flow representation of the program, which means that it is obtained through static analysis of the program code.

The example shown next describes the cyclomatic complexity metric. Below is an example program taken from *Tutorials Point* [63]:

```
1 A = 10
2 IF B > C THEN
3 A = B
4 ELSE
5 A = C
6 ENDIF
7 Print A
8 Print B
9 Print C
```

LISTING 2.14: Example program for cyclomatic complexity explanation

Knowing that the cyclomatic complexity is defined as $M = E - N + 2P$, where E equals the number of control flow graph edges, N is the number of nodes in the control flow graph and P standing for the number of interconnected components, we can compute the CFG for this program to calculate the cyclomatic complexity. Figure 2.2 depicts the CFG for the example program:

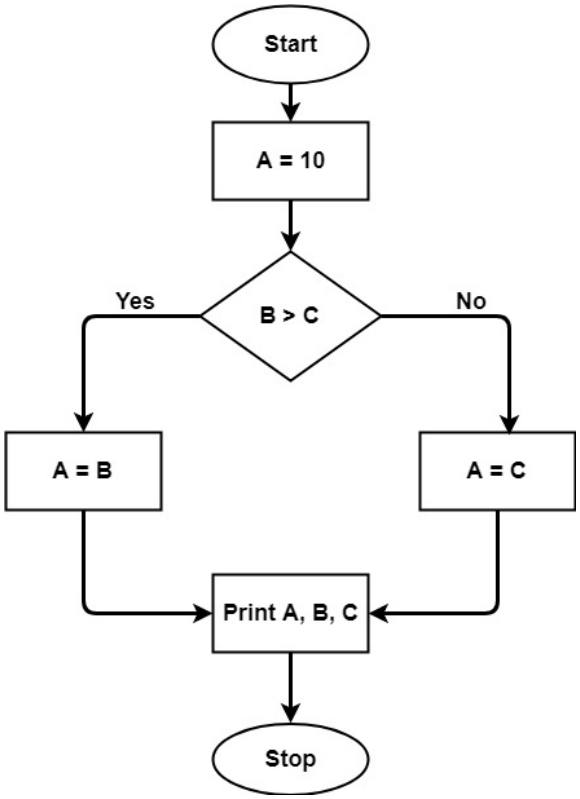


FIGURE 2.2: Control Flow Graph for the example program [63]

With this visual information we can easily calculate the metric. We can see there are seven nodes and seven edges in the graph, so the cyclomatic complexity is $7 - 7 + 2 = 2$.

In our case, we will use the cyclomatic complexity to evaluate the complexity generated by obfuscations and code optimizations on the final programs.

Chapter 3

Methodology

This chapter describes and details our framework to automatically transform, build, benchmark and extract metrics from the analysis made on the programs by the analysis tools.

We start by describing our framework in Section 3.1 by giving an abstract overview of the steps and tools used and how they are chained together to make an analysis from start to end. After that, on Section 3.2, we describe what metrics will be taken and their purpose, on Section 3.3 we specify the tools that are going to be used in order to achieve the proposed framework, and on section 3.4 we describe the virtual machine created that hosts all the tools needed to run the entire flow.

3.1 Framework

We developed a testing methodology for benchmarking the effect of optimizations on obfuscated programs in symbolic execution and fuzzing tools. Our approach to measure and define the level of revertness that optimizations have on the obfuscated program is to quantify how much the coverage of an analysis has been increased depending on the level of optimization.

Succinctly, our benchmarking approach is comprised of three phases: compilation of the target programs (obfuscation, optimization and compilation), analysis, and metric extraction. In the compilation phase, the programs are obfuscated using Tigress transformations and optimized. Then, these program are compiled to LLVM bitcode, for KLEE to analyse, and compiled to machine code, for AFL++ to analyse them. After we have the target

program obfuscated and optimized in both formats, we can go to the second phase. The second phase is where the programs that were compiled in the previous phase are analysed with KLEE and AFL++. These programs are run with a predefined timeout, memory and thread limits, and their analysis are saved on a separate directory for each combination of obfuscation/optimization. The third and last phase is to extract the metrics from the analysis, mainly the inputs that the analysis generated in order to compute their coverage against the original program. At the same time the coverage is computed, which is one of the metrics extracted, a CSV file is built containing the coverages of the analysis in order to import it to spreadsheets, make analysis, build plots, etc. Figure 3.1 depicts our workflow in a summarized and graphical manner.

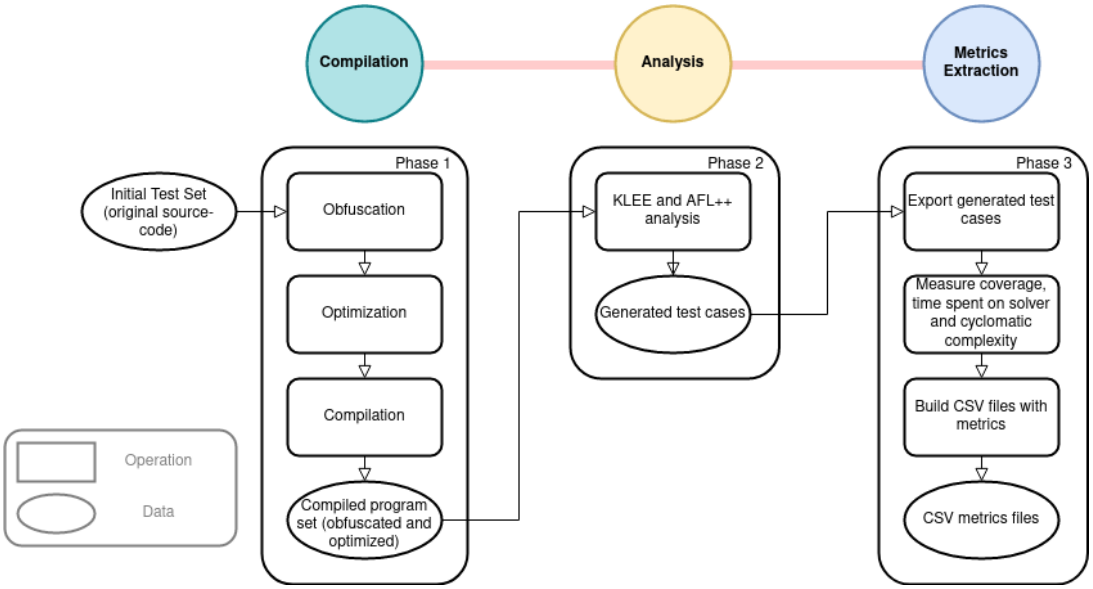


FIGURE 3.1: Framework workflow

The first step in our methodology is to take a program and individually obfuscate it with a transformation using Tigress. Because Tigress is a source-to-source obfuscator [19], this means that we can take the transformed source-code and use all tools that work on source-code, including compilers, etc. The transformations that we applied on our tests are 9 in total and are as follows: Literals Encoding, Arithmetic Encoding, Data Encoding, Branches Encoding, Opaque Predicates, Control-Flow Flattening, Anti Taint Analysis, Anti Alias Analysis and Virtualization. The obfuscator needs to know the function names and variables of the target program in order to use some obfuscations, and to make this framework generic, a file containing that information has to be created for each program by following the same structure.

Next we take all of the outputted programs from the former step and compile them by applying all optimizations for each one. These optimizations are the 4 levels of optimizations that Clang has [54] (-O0, -O1, -O2, and -O3), the Souper optimizer [62] and KLEE's optimize flag (only applied when executing KLEE). We use the Clang optimizations individually and combined with Souper, and the same step is applied to KLEE's optimize flag [60], which is applied in solo and in combination with Souper. This gives us 10 different test cases for each obfuscating transformation when analysing the program with KLEE and 8 when using AFL++ (due to AFL++ not having an optimize flag like KLEE does). There is also a version compiled without any obfuscations or optimizations but with special flags for measuring the coverage when the program is run with some input. This is used so that we can measure the coverage of the test cases generated from the analysis of KLEE and AFL++.

The compilation process is made in two steps: the first one compiles the program to LLVM bitcode using Clang, in order to analyse it with KLEE and to optimize it with Souper, and the second step compiles it to machine code using the custom Clang compiler that AFL++ has put together, called `afl-clang-lto` [64], which is one of the custom compilers that AFL++ has. This later compiler was tailor-made to aggregate all the instrumentations that AFL++ needs the target program to have, in one toolchain. It prepares the target to be fuzzed efficiently and instruments it with the necessary instructions to be able to catch crashes more precisely and to analyse the program more efficiently. We chose `afl-clang-lto` instead of `afl-clang-fast` and `afl-gcc-fast` because the documentation states that it is faster and gives better coverage [64]. Also, the names of the programs, their obfuscation, optimization level, and if they have been optimized with Souper or not, also follow a structured type. This way we can then filter the files by their transformation and optimization using their name.

After the previous steps are completed, we are left with 100 analysis benchmarks when using KLEE and 90 when using AFL++, for each program. The next step is to analyse these programs using the KLEE symbolic execution engine and the AFL++ fuzzer. For KLEE, we analyse all LLVM bitcode files, e.g., `.bc` extension, and for AFL++ we analyse all binary files. The test cases that KLEE generated for the target program without any obfuscations and optimizations (base program) are exported to AFL++ input set format

to use them as the initial input. This is because AFL++ is a fuzzer and fuzzers need an initial input set for them to mutate the input and reach new code blocks on its analysis.

After all analysis have been run, we export all the generated test cases for each analysis to individual inputs. Then, we run the target program compiled without any obfuscations, no optimizations, and with coverage options to see the coverage that the analysis has made with relation to the base program. With this coverage measurements of the analysis of KLEE and AFL++, we build a CSV file so that we can import it to programs and frameworks that analyse these type of files like spreadsheets or pandas [65], in order to build plots about the results of the analysis to visually analyse the results. This is done not only to the coverage but also to other metrics as we will specify later.

Here are the steps summarized:

1. Create a structured file in the root of the target program. This file must follow the rules and be semantically equal to the files used in other programs.

2. Obfuscate the target program with the transformations chosen using Tigress.

3. Compile the programs from step 2 to LLVM bitcode, using Clang, and to machine code (binaries), using `afl-clang-lto`, while applying the optimizations of the compiler (-O0, -O1, -O2, and -O3). Also compile a special binary without obfuscations or optimizations and with special compiler flags to measure the coverage when run.

4. Run Souper for the LLVM bitcode files from step 3 (except the special binary to measure the coverage), which also outputs LLVM bitcode, and compile the outputted programs with `afl-clang-lto`.

5. Run KLEE without the optimize flag on each LLVM bitcode program from the previous steps, except the special binary to measure the coverage.

6. Run KLEE with the optimize flag on the LLVM bitcode programs that have no optimization from the compiler, and that have no optimization from the compiler but have been optimized with Souper, from the previous steps, except the special binary to measure the coverage.

7. Convert an amount of test cases generated by KLEE's analysis on the target program that has no obfuscations or optimizations (base program) into text.

8. Run AFL++ on each binary program from the previous steps, except the special binary to measure the coverage, using the output of the previous step as the initial input.

9. Export the test cases generated by the analysis on all programs, compute the coverage for each analysis and build a CSV file with the results.

10. Do the previous step for the time spent on solver metric and compute the cyclomatic complexity of the programs generated.

3.2 Metrics

To be able to evaluate the analysis made, one of the metrics that will be used is the coverage the inputs generated achieve when run against the original program. An analysis which as a high coverage means that the inputs generated cover most of the program whilst an analysis with low coverage means that the inputs generated only cover a small part of the program, meaning that the analysis had difficulties in analysing most of the program. We will use this metric as our main assessment in how an obfuscation or optimization hampered or helped the analysis of the tools.

To try to explain the different results obtained, other metrics will be extracted such as the time spent on the solver in the case of the KLEE analysis and the amount of initial inputs in the case of AFL++, and if the analysis behaves better or worse if the time of the analysis or the input size is increased.

Another metric that will be computed is the cyclomatic complexity which is a metric that states the number of linearly independent pathways in a code segment and it's a software statistic that shows how complicated a program is. We will use this metric to see if there is any correlation of the obfuscations and optimizations with the cyclomatic complexity of the program.

3.3 Tools

Next, we present the tools we are going to use in order to build the proposed framework. Their selection and functionality will also be depicted. Likewise, a word on how the scripts should be developed is given.

3.3.1 Obfuscating, Compiling and Optimizing

The tools to build the examples (obfuscate, compile and optimize) should be aware if the target source-code has only one file or is made up of several files. The building of programs where the source-code is all in one source file is done by obfuscating the source-code with Tigress and compiling it to LLVM bitcode and binary formats with Clang and `afl-clang-lto`. To build programs with more than one source-code file, we will use the WLLVM (whole-program LLVM) [66] compiler that compiles the whole program using Clang or GCC (we use Clang), in which the LLVM bitcode of the whole program will be extracted from the outputted binary using the tool `extract-bc` [66], also contained in WLLVM. KLEE has a tutorial on how to test the GNU Coreutils [67] on KLEE in which they use WLLVM to compile the program and then extract the LLVM bitcode so that KLEE can analyse it [68].

3.3.2 Benchmarking with KLEE and AFL++

Similar to the design and organization of the scripts for the building stage, the analysis for the KLEE and AFL++ tools should also be separated in scripts accordingly to the obfuscation. Although the analysis with KLEE and AFL++ does not alter between obfuscations or optimizations like the building phase (because Tigress must be run with different options), we established that when invoking an analysis for an obfuscation it will first list all the files the tools in question can analyse, LLVM bitcode for KLEE and ELF binaries for AFL++, in the given directory but filter them by their obfuscation. For example, the script to benchmark the flatten transformation using KLEE should list all files in the directory given and filter the files that are LLVM bitcode and have the flatten text on their name. After that, it should call a generic script to analyse each file that was listed after the filtering. We decided to specify these additional scripts so that we can more easily analyse just the obfuscation we want. To analyse the files with KLEE's optimize flag we list all LLVM bitcode files without compiler optimizations and without compiler optimizations but optimized with Souper and run KLEE on those files.

3.3.3 Metrics Extraction

The last step is to export the metrics generated by KLEE and AFL++ analysis. One of the main metrics to be extracted is the coverage percentage the inputs of an analysis have when executed against the original program [69]. To do this we transform the test cases

generated from KLEE and AFL++ analysis to files that contain those inputs in plain text, and we will use the tool `ktest-tool` [70] to extract the inputs from the `.ktest` files. After we have the inputs exported to a directory, we execute the program compiled with the coverage compiler flags with all the inputs that were exported to the directory. To view the result of the coverage for all those inputs, we will use the `llvm-cov` tool [71] that keeps track of the coverage in a file contained in the source-code directory that has the name of the source-code file and the `.gcda` extension. After computing the coverage, we add the result to a CSV file that is incrementally built by repeating this stage to all of the benchmarks.

Other metric such as the time spent on the solver for the KLEE analysis will be extracted using the `klee-stats` tool [70] and the cyclomatic complexity will be extracted using the metrics tool [72]. The same procedure of the coverage metric applies to the rest of the metrics. In this case, the common procedure is to export all of these metrics by saving them to an CSV file that contains the obfuscations and optimizations and the corresponding metric.

What the common procedure allows us to do is to generate plots generically and automatically due to the fact that every plot is generated the same way, e.i., by giving the plot generator the CSV file which we want it to plot.

3.4 Virtual Machine

To make sure our test environment is portable and anyone can use it, a virtual machine will be developed that hosts all the tools needed, with the fixed versions and settings. Because we might want to update the versions of the tools or change some setting and redistribute that new VM, we will create a packer template to automate the building of these virtual machines [73]. Packer is an open source tool that enables the creation of identical virtual machine images or containers for multiple platforms from a single source configuration and is used to automate the creation of such machines. The steps to build the VM will be automated and they will be uploaded to Vagrant Cloud [74], where anyone can upload and download these virtual machines. The images outputted by packer are compatible with several virtualization software, e.g., VirtualBox [75].

Chapter 4

Implementation

This chapters depicts the concrete implementation that was done by following the abstract methodology from the previous chapter.

All of the steps mentioned above should be automated using generic scripts, meaning that one must only create the folder of the new target program, add the source code, add the structured file containing the function names, variables, etc., and update the benchmark script so that it does all the necessary steps to obfuscate, optimize, compile, analyse and extract the metrics of the analysis, but for the new program.

4.1 Obfuscation

To generate all the programs with the obfuscations applied to them, we developed some scripts to automatically obfuscate the programs which can be reused when a new program is tested. To keep track of the different variables, functions, and file names we have a special file that holds these variables, making the obfuscation scripts generic and reusable for each program. This file is also used when optimizing and compiling the generated programs.

The scripts to achieve such automation reside in the scripts directory, on the `build` and `build-cgc` directories, and they are separated by obfuscation type. Although the name might indicate that separate scripts are needed to build each program type, KLEE examples and CGC programs, the reality is that the first directory is for programs which only have one source code file and the second for programs with multiple source code files.

The scripts to obfuscate, optimize and compile the generated programs are the same and this is the first step of those scripts. They also accept arguments which change the optimization level of the generated program, and the directory containing the program and the special variables file.

The script that builds all programs with all obfuscations and with all optimizations is the `build-binaries.sh` script, which in turn will call the script `build-full.sh`, inside the `build` and `build-cgc` directories, for each optimization level of each program, which in turn will call each one of the obfuscation scripts. The execution tree in Listing 4.1 demonstrates the flow that each script makes.

```
build-binaries.sh
    build-full.sh -O0 ../Maze_Obfuscated/
        build-vanilla.sh -O0 ../Maze_Obfuscated/
        build-anti_alias_analysis.sh -O0 ../Maze_Obfuscated/
        build-anti_taint_analysis.sh -O0 ../Maze_Obfuscated/
        build-encode_arithmetic.sh -O0 ../Maze_Obfuscated/
        ...
    build-full.sh -O1 ../Maze_Obfuscated/
        build-vanilla.sh -O1 ../Maze_Obfuscated/
        build-anti_alias_analysis.sh -O1 ../Maze_Obfuscated/
        build-anti_taint_analysis.sh -O1 ../Maze_Obfuscated/
        build-encode_arithmetic.sh -O1 ../Maze_Obfuscated/
        ...
    build-full.sh -O2 ../Maze_Obfuscated/
        ...
    build-full.sh -O3 ../Maze_Obfuscated/
        ...
    build-coverage.sh -O0 ../Maze_Obfuscated/
    run_souper.sh ../Maze_Obfuscated/Transformations/

    build-full.sh -O0 ../Reg_expr_Obfuscated/
        build-vanilla.sh -O0 ../Reg_expr_Obfuscated/
        build-anti_alias_analysis.sh -O0 ../Reg_expr_Obfuscated/
        build-anti_taint_analysis.sh -O0 ../Reg_expr_Obfuscated/
        ...
```

LISTING 4.1: build-binaries.sh script execution tree

The script that builds the program without obfuscations is `build-vanilla.sh`, the one that builds the program to be used to compute the coverage is `build-coverage.sh`, `run_-souper.sh` runs souper on all the files in the specified directory, and the remaining scripts, `build-*.sh`, build the program with the obfuscations.

The output of this phase is the programs generated by it, which contains the original program and obfuscated ones, and the tool used to obfuscate the programs is Tigress.

4.2 Optimizations and Compilation

The next step in our testing flow is the optimization and compilation phase. This phase deals with the original C code and the one generated by Tigress, optimizes it by using the chosen optimizers, and compiles the programs into LLVM bitcode and binary formats.

The scripts to accomplish such task are the same as the obfuscation phase, and this is the next and final step of those scripts. In there, the optimizers and compilers are called to generate the programs which will then be fed into the analysis tools. The tools used in this phase are `clang`, one of the most used compilers, `wllvm` and `extract-bc`, which is a set of tools to compile whole programs into LLVM bitcode, `souper`, a peephole optimizer, and `afl-clang-lto`, a modified Clang compiler with built-in customizations to instrument the target program with sanitizers and other techniques, to be analyzed with AFL++.

If the program only contains one source code file, the scripts used are in the `build` directory and if it contains more than one source file, the scripts in the `build-cgc` directory are used. In the case of single source file programs, the first step in this phase is performed using `clang` and `afl-clang-lto` to optimize and compile the C source code into LLVM bitcode and binary formats, respectively. And in the case of multiple source files programs, the tools used are `wllvm` to optimize and compile programs, which then have their LLVM bitcode extracted by using the tool `extract-bc` that also comes with the wllvm package, and `afl-clang-lto` to instrument and compile to binary format.

With the usage of `clang`, may it be when using `clang` itself, `wllvm`, or `afl-clang-lto`, some flags were added to the compilation instructions, and they are described below:

- clang:

  - No optimization (-O0): -Wall, -g, -O0, -Xclang, -disable-O0-optnone [59], -fno-stack-protector, -std=c89, -emit-llvm, and -c
  - Optimized: -Wall, -g, -O*X*, -fno-stack-protector, -std=c89, -emit-llvm, and -c
  - Coverage: -Wall, -g, -O0, -Xclang, -disable-O0-optnone, -fno-stack-protector, -fprofile-arcs, and -ftest-coverage
- afl-clang-lto: -Wall, -g, -O*X*, -fno-stack-protector and -std=c89
- wllvm: -ldl, -fno-builtin, -fcommon, -w, -g, -O*X*, -fno-stack-protector, -ldl, -Wl, –no-as-needed and -std=gnu99

The scripts to accomplish this step are the same as the previous one. The output of this phase is the obfuscated programs but with the optimizations applied to them, and compiled into LLVM bitcode and binary formats. In total, there are 80 programs generated by this phase and they are ready to be analysed by KLEE and AFL++.

4.3 KLEE and AFL++ Analysis

This phase consists in running the analysis tools with the generated programs from the optimization and compilation phase. As stated before, the analysis tools chosen were KLEE, the symbolic execution framework, and AFL++, the fuzzer for binary programs.

To make this phase automated and generic, we developed scripts to analyse the given programs that change the analysis options accordingly to the options provided. These scripts reside on the `benchmark-klee` and `benchmark-aflpp` directories and work by calling the analysis tools and managing the output directories for each analysis. All of the scripts inside those directories will in the end call `benchmark.sh` present in the two directories, which is the script that executes the analysis tools with the options provided. In the case of KLEE, the script calls the tool `klee` and in the case of AFL++, `afl-fuzz` is the tool called.

For the KLEE analysis, we noted that some analysis were terminating in a matter of seconds and some strange warnings were being thrown. After investigating we found that the STP solver had a very big stack size when solving some formulas and it's limit was low, and the solution is to allow for an unlimited stack size before executing KLEE [76].

For the execution of the KLEE analysis, we use some flags which change the behaviour of the analysis. On Table 4.1 these options are specified and have their value used and functionality described for the KLEE framework and on Table 4.2 the flags for ALF++ are specified.

Flags	Value used	Description
--simplify-sym-indices	Set	Simplify symbolic accesses using equalities from other constraints (default=false).
--solver-backend	STP	Specifiy the core solver backend to use.
--solver-optimize-divides	Set	Optimize constant divides into add/shift/-multiplies before passing them to the core SMT solver (default=false).
--write-cov	Set	Write coverage information for each test case (default=false).
--write-kqueries	Set	Write .kquery files for each test case (default=false).
--write-paths	Set	Write .path files for each test case (default=false).
--write-sym-paths	Set	Write .sym.path files for each test case (default=false).
--write-test-info	Set	Write additional test case information (default=false).
--only-output-states-covering-new	Set	Only output test cases covering new code (default=false).
--external-calls	all	Specify the external call policy.
--posix-runtime	Set	Enable symbolic environment and link with POSIX runtime. Options that can be passed as arguments to the programs are: --sym-stdin <max-len>, --sym-args <min-argvs><max-argvs><max-len>+ file model options (default=false).
--libc	uclibc	Choose libc version (none by default).

--watchdog	Set	Use a watchdog process to enforce --max-time.
--output-dir	Depends on the analysis	Directory in which to write results (default=klee-out-<N>).
--optimize	Set/Not Set (depends on the analysis)	Optimize the code before execution (default=false).
--max-time	10min/30min (depends on the analysis)	Halt execution after the specified duration. Set to 0s to disable (default=0s).
--max-memory	6144MB	Refuse to fork when above this amount of memory (in MB) (see -max-memory-inhibit) and terminate states when additional 100MB allocated (default=2000).
--sym-stdin	Depends on the analysis	Provide a symbolic standard input and specify it's size in bytes.

Flags	Value used	Description
-i	Depends on the analysis	Input directory with test cases.
-o	Depends on the analysis	Output directory for fuzzer findings.
-D	Set	Enable deterministic fuzzing (once per queue entry).
-V	600/1800 seconds	Fuzz for a specified time then terminate.
-s	1234	Use a fixed seed for the RNG.
-M	fuzzer1	Distributed mode, name of the main fuzzer.
-S	fuzzer2/fuzzer3/ fuzzer4	Distributed mode, name of the secondary fuzzers.

4.4 Metrics Extraction

The last step in our workflow is to extract the metrics of the analysis made and use that to generate plots and subsequently evaluate each analysis.

To be able to extract these metrics, we extract all the inputs the analysis generated and test them against the original program to compute the coverage they achieve. The result is a CSV file, for each program, containing the coverage of the original program with the inputs of each analysis, separated by lines and columns. Each row is a different obfuscation and the columns represent the several optimizations done to the program. After that, we generate heatmaps to visualize the results of the analysis and to aid on our examination of the outcomes of obfuscations and optimizations. Apart from this measurements, we also generate heatmaps for the time spent on the solver by KLEE and the cyclomatic complexity of the programs generated.

To produce such files and plots we developed scripts to generically and automatically extract, compute, transform and plot the data and metrics generated for each analysis. These scripts reside in the main scripts directory and each has it's duty in this phase.

For the coverage metrics, first we export each input generated by the analysis by executing the script `coverage-csv.sh`, which calls the script `ktest-to-text.sh` in case of KLEE, which in turn executes the `ktest-tool` tool from LLVM, and `afl-to-text.sh` in case of AFL++, for each analysis. After that, we execute the script `compute-coverage.sh` to compute the coverage an analysis had, by executing the program with all the inputs generated and using the `llvm-cov` tool from LLVM to export the value. Now that we have a text file with the analysis names and the respective coverage, we execute the scripts `compute-coverage-klee-output-to-csv.sh` and `compute-coverage-afl-output-to-csv.sh` to transform that output into a CSV file that is then fed into the plot generator script, `csv-to-plot.py`, which is written in Python using the `matplotlib` [77], `pandas` [65], and `numpy` [78] frameworks. The output of this workflow is the heatmap plots. The execution tree in Listing 4.2 depicts this flow.

```
1 ./coverage-csv.sh
2     #KLEE analysis
3     for each analysis inside ../Maze_Obfuscated/Transformations/ do:
4         ./ktest-to-text.sh ../Maze_Obfuscated/Transformations/$analysis ../
    Maze_Obfuscated/inputs/$analysis
```

```
5       ./compute-coverage.sh ../Maze_Obfuscated/inputs/$analysis ../
    Maze_Obfuscated/maze.c ../Maze_Obfuscated/Transformations/maze-OO-
    coverage
6     ./compute-coverage-klee-output-to-csv.sh compute-coverage-output maze >
     compute-coverage-klee-maze.csv
7     #AFL++ analysis
8     for each analysis inside ../Maze_Obfuscated/Transformations/ do:
9       ./afl-to-text.sh ../Maze_Obfuscated/afl/$analysis ../
    Maze_Obfuscated/inputs/afl-$analysis
10      ./compute-coverage.sh ../Maze_Obfuscated/inputs/afl-$analysis ../
    Maze_Obfuscated/maze.c ../Maze_Obfuscated/Transformations/maze-OO-
    coverage
11    ./compute-coverage-afl-output-to-csv.sh compute-coverage-output maze >
    compute-coverage-afl-maze.csv
12
13    # Same steps for other programs
```

LISTING 4.2: Coverage metric scripts execution tree

The same workflow is used for the time spent on the solver and cyclomatic complexity metrics. In the case of the time spent on the solver, the scripts used are `klee-solver-time-csv.sh` which for each analysis executes the script `extract-solver-time.sh`, which internally calls the tool `klee-stats` to extract this metric from KLEE's analysis, and after completing the extraction for all analysis the script `klee-solver-output-to-csv.sh` is called to convert the output into a CSV format. For the cyclomatic complexity the main script is `cyclomatic-complexity-csv.sh` that for each generated program will call the tool `calc-mccabe` from the `metrics` package and that calls the script `compute-cyclocomp-output-to-csv.sh` to convert the output into CSV format.

One of the contributions of this thesis is the porting of the Metrics tool from LLVM 3.4 into LLVM 11 which was needed in order to compute the cyclomatic complexity metric for our programs.

4.5 Virtual Machine

Another contribution of this thesis is the set up of a virtual machine that hosts all of the tools needed to run the analysis with their correct versions. To be able to automatically

build a VM that is supported by several virtualization software's and with the exact software versions we need, we used packer to fulfill this requirement. By specifying the base image, in our case Ubuntu Focal, the bash scripts with the instructions to compile and install all the tools, and the virtualization software provider, packer automates this process. The output is a ready to run image, in our case for Virtual Box, that we upload to Vagrant Cloud to be able to distributed for anyone that whishes to use it.

Inside the `packer-template` directory resides the files that packer uses to build the VM. The configuration file used in this process is `vagrant.json` and the scripts which compile and install the tools needed are `klee.sh`, `souper.sh`, `aflplusplus.sh`, and `metrics.sh`. After this process is done, the image is uploaded to Vagrant Cloud.

4.6 Git Repository

All of the scripts and files specified on this chapter are available on a GitHub repository so that anyone can have access to it. The repository is available at FredyR4zox/tigress-obfuscations

Chapter 5

Results

The effectiveness of some obfuscations against fuzzing tools, including symbolic execution, has been studied by other peers in academia. During the evaluation of our testing framework we will focus in our objective which is to test the resilience of obfuscations against code optimization, while using fuzzing tools to benchmark this effectiveness. On Section 5.1 we detail the setup used on the tests, including the hardware specifications of the host system and operating system along with the tools used and their versions, the set of programs chosen to be apart on our tests, and the tests that were performed. Then, on Section 5.2, we present and discuss the results obtained by running our implementation with the specified specifications.

5.1 Testing Specifications

In this section we will discuss the specifications of the tests, ranging from the hardware specifications to the tests performed.

5.1.1 Software Setup

The tests were performed on a dedicated virtual machine hosted on OVH [79]. This virtual machine had dedicated cores, where no other software was running alongside the benchmarks. Another benefit is that the virtual machine hosted on OVH uses more efficient software to run the virtual machines in each server, increasing the computation power available to our tests, even if the processor is less powerful compared to the ones in our personal computers.

Specifications	
RAM	8GB
CPU	4 Cores 2.4GHz 16MB Intel Haswell (no TSX)
Disk	160GB NVMe SSD
Operating System	Ubuntu 20.04.4 LTS

TABLE 5.1: OVH virtual machine specifications

The technical specifications of the virtual machine hosted on OVH are depicted on Table 5.1.

The tools used were the ones discussed on the Methodology section. We used `tigress` as the obfuscator, `clang` and `afl-clang-lto` as the compilers to LLVM bitcode and executable format with specific instrumentation for the fuzzer, respectively, while also serving as optimizers, and `souper` as one of the optimizers of LLVM bitcode. In the compilers section, we also used `wllvm` and `extract-bc` as the tools to compile programs with more than one file and extract it's bitcode, e.g., for CGC programs, KLEE for the symbolic execution engine, and AFL++ for the fuzzer. To compute the metrics we used `ktest-tool` to extract the tests generated by KLEE and `klee-stats` to extract the metrics of the KLEE analysis, including the time spent on the solver. Finally we used `llvm-cov` from LLVM to compute the coverage and `calc-mccabe` from the Metrics tool to compute the cyclomatic complexity of LLVM bitcode.

The following list depicts the versions of the tools specified above:

- KLEE (klee, ktest-tool, klee-stats): 2.3-pre
- LLVM (clang, llvm-cov): 11.1.0
- WLLVM (wllvm, extract-bc): 1.3.1
- souper: commit 4a251bde91ccb20f5e2389c9962ff905156fc3e9
- STP solver: 2.3.3
- AFL++ (afl-clang-lto, afl-fuzz): 4.01a
- Tigress: 3.1
- Metrics (calc-mccabe): 11

5.1.2 Test Programs

The test programs chosen to be on our test set were varied in terms of functionality. The first two programs were taken of KLEE's examples, the second two's were taken from Darpa's Cyber Grand Challenge [80], and the last one was taken from the tigress examples. The KLEE examples and the tigress random function were programmed to find a specific input which allows the program to reach some winning state, and the CGC's programs were made with memory corruption vulnerabilities. Although KLEE ignores tests that lead to memory corruption, the programs were chosen because of their appropriate size for analysis. The following list summarizes this set:

- Maze (KLEE's examples)
- Regular Expressions (KLEE's examples)
- Barcoder (Darpa's Cyber Grand Challenge)
- Message Service (Darpa's Cyber Grand Challenge)
- Random Function (Tigress examples)

The Maze program [81], one of KLEE's examples, consists in guiding a starting position to the end of the maze by accepting the keys "w", "a", "s", and "d" to move through it. In this case, there is more than one solution, and the analysis frameworks may not find all solutions because it they stop when all paths have been covered, which can happen without having found all solutions. In our case it does not matter due to the fact we are trying to get the whole program covered, which has to have at least one solution, and not find all the solutions.

The Regular Expressions program [82] was also taken from KLEE's examples and its functionality is to test if a user can find a regular expression which matches a certain string. This specific example tests the regular expressions given by the user against the string hello and to reach the winning state the user must insert a regular expression that matches the string. The same situation of the Maze program happens on this one, where not all solutions may be found by the analysis frameworks.

The Barcoder program [83] was taken from Darpa's Cyber Grand Challenge and allows a user to generate or decode barcodes. When inputting a barcode, a user may input as text, bacoded ASCII representing the binary values or as a bitmap that digitally represents

the barcode. Once a barcode is cached it can be viewed as either an ASCII or bitmap representation. This program contains a format string and a stack-based buffer overflow vulnerabilities.

The Message Service program [84], also taken from Darpa's CGC, consists in a service with register and login features in which users can exchange messages between them. In addition to having the read and send messages features, they also have the list and delete messages functionalities. This program contains a heap-based buffer overflow and a Use After Free vulnerabilities.

The Random Function program [85] was taken from the tigress examples and its behaviour is for the analysis tools to find an input for which the program prints "You win!". This program does also suffer from the same problem of the Maze and Regular Expression programs where there exists more than one solution and not all are going to be found.

5.1.3 Experiments

5.1.3.1 Obfuscations

The first step in our experiments is the obfuscation phase. In this step the programs are obfuscated with the obfuscations chosen using an widely available obfuscator, called Tigress. It contains several obfuscations, also called transformations, in which it incorporates a considerable amount of options for each obfuscation. For our tests we chose the default options for each transformation.

The obfuscation chosen for our tests were the following:

- Virtualization
- Obfuscations
- Virtualization
- Control-Flow Flattening
- Opaque Predicates
- Literals Encoding
- Data Encoding
- Arithmetic Encoding

- Branch Encoding
- Anti Alias Analysis
- Anti Taint Analysis

By executing the scripts mentioned on Section 4.1 and following it's rules, we now have the original program obfuscated in the transformations mentioned in the previous list.

As usual, we are expecting that some obfuscations will yield worse results when analysing them, with virtualization being the most difficult to revert and encode arithmetic the easiest, at a first glance.

Note: The Jit, JitDynamic, EncodeExternal and SelfModify transformations were not used in our testing benchmarks due to KLEE not supporting them.

5.1.3.2 Optimizations and Compilation

The optimizations used were the 4 levels which compilers commonly offer (O0, O1, O2, O3), in our case being Clang, souper, and when analysing with KLEE, the KLEE optimize flag. Compilers such as Clang have usually 4 levels of optimization, the first being O0 which does not optimize the resulting code and generating the most debuggable binary, O1, an intermediate level between O0 and O2, O2 using most of the optimizations available, and O3, the same as O2 but enabling all optimizations, some which take longer to perform or may generate larger code (in an attempt to run faster).

The other optimizations that we are going to use are souper, which is a peephole optimizer for LLVM bitcode, and KLEE's optimize flag that optimizes the program with a subset of compiler optimizations before the analysis is run.

5.1.3.3 KLEE and AFL++ Analysis

The test set includes 5 programs and all of the programs are obfuscated with 9 transformations. Then they were optimized using the 4 levels of the compiler optimization and the KLEE --optimize flag. To add to this, the programs optimized were then optimized with souper, including KLEE's optimize flag. This means that for each analysis set, we are going to analyse the 5 programs where each one has 9 transformations, plus the original program, and each of those generated programs (10) has 9 optimizations in the case

of KLEE and 7 transformations in the case of AFL++, plus the generated programs without optimizations. In total, for each program, the number of analysis to run is 100 for KLEE and 80 for AFL++. Because each analysis set contains the 5 programs, you need to multiple the numbers by 5.

As we will see, these numbers are not exactly correct because not all obfuscations are able to be applied to all programs and sometimes the optimizations remove critical parts of the program, which makes the analysis of the program incorrect due to the fact the programs have different behaviours. This awaits an explanation because it's weird that optimizations are doing this, even the most less aggressive ones.

The results were obtained by running several benchmarks with different variables. One of those variables is the time, where the analysis were run with time limits of 10 and 30 minutes to investigate how much of the results were dependent on the time limit. Another variable is the limit of the input size to see if by giving a higher number for the analysis tools they behaved better or worse and in these tests we doubled the input size for each program. In the case of AFL++ the number of starting inputs are also examined to see how AFL++ behaves when it is given a higher number of inputs at the beginning of the analysis, where the default value is 10% of the inputs KLEE generated by analysing the original program and the other is 50% of those inputs.

For tool specific limits, KLEE had a memory limit of 6144 MB, and AFL++ was executed with 4 threads to mimic the number of CPU cores/threads of a low/medium power computer and to use all of the computing power of the virtual machine. The solver used in the tests was STP (Simple Theorem Prover) which is the one KLEE uses by default and the stack size limit was set to none before the analysis ran.

Although symbolic execution is more slow and does not scale very well with large programs, there is a advantage that it's got: the code KLEE analysis runs is on LLVM bitcode and not compiled code, which does have less information stripped from the code.

For the KLEE analysis, we noted that some analysis were terminating in a matter of seconds and some strange warnings were being thrown. After investigating we found that the STP solver had a very big stack size when solving some formulas, and the solution is to allow for an unlimited stack size before executing KLEE [76].

5.1.3.4 Metrics

The last step in our workflow is to extract the metrics of the analysis made and use that to generate plots and subsequently evaluate each analysis.

To analyse the behaviour of the analysis tools we extracted some metrics from their runs. The one we will focus the most is the coverage percentage, where the inputs the analysis generated are tested against the program without obfuscations and optimizations. For example: if the result is 75%, it means the inputs the analysis generated cover 75% of the original program. We established this measurement as the comparison variable to quantify the difference obfuscations and optimizations have on a program when analysed by symbolic execution and fuzzing tools. Then, to explain the different results and try to find a variable which dictates the coverage the analysis will yield, we extract and analyse more metrics in order to obtain an explanation of why that is happening. The first one is specific to KLEE and is about how much time of the analysis was spent waiting for the solver to reach a solution to the given theorem. Another is the cyclomatic complexity of the programs which measures the number of linearly-independent paths through a program, meaning that programs with a lower cyclomatic complexity are easier to understand.

We also ran several analysis in which we modified some variables too see if the analysis generated more tests and the difference that made in the final result. This is because different obfuscations and optimizations might reach the same paths and generate the same number of tests but need more time or more input to reach it. The variables were the maximum time the analysis take, the input size, better demonstrated on the Barcoder and Message Service programs which have a big input size, and the number of tests given to AFL++ as the initial inputs.

5.2 Results

In the following sections we discuss and investigate if the code optimizations reverted the obfuscations applied and which yield better results, and try to find a variable which explains the results obtained by trying to infer the coverage before the analysis is run. As we will see, the results are not what we initially expected using critical thinking, and we examine if by modifying the execution variables of the analysis tools the results are different, and if so, to what extent.

To begin with, some plots will not have all values due to the fact some of these programs were not behaving correctly. Some were crashing due to segmentation faults and others didn't worked as expected, e.g., some menus did not exist or the behaviour was not the same as in the original program, where the same input would generate different results from the original program. We will depict these issues later on this section.

5.2.1 Cyclomatic Complexity

First, let's look at the cyclomatic complexity of the different programs. Figure 5.1 presents the average cyclomatic complexity for all the programs. It is clear by the results that on average the cyclomatic complexity is lower the more optimization there is, specially starting from the O2 level, which is what we predicted. As expected, the virtualization obfuscation got the highest cyclomatic complexity of all programs on average. One curious result is that the virtualization obfuscation got lower cyclomatic complexity values on the O1 optimization level, with and without Souper, compared to the original program. Let's see the plots one by one to get a better picture of these values.

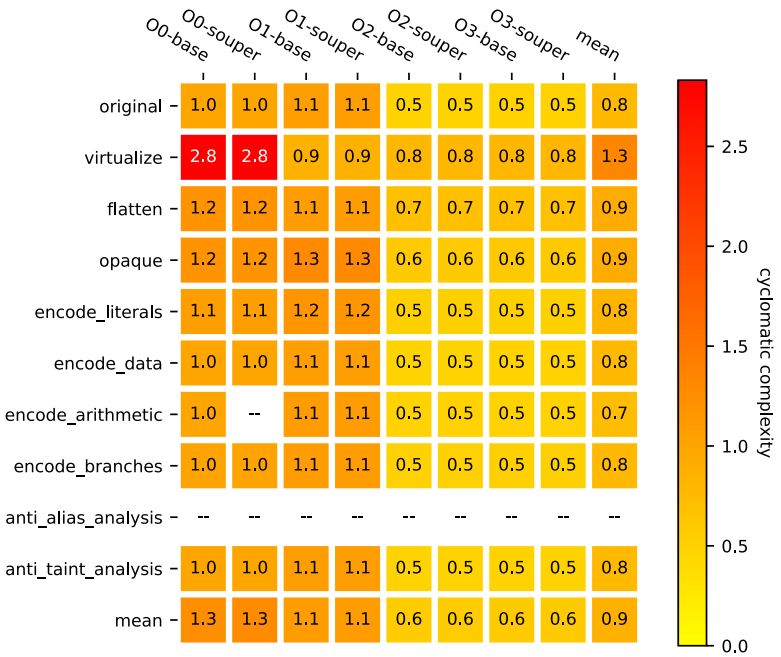


FIGURE 5.1: Average cyclomatic complexity for all programs

If we evaluate each program individually, we can see that the results are not the same for each one. In the case of the Maze program depicted on Figure 5.2, we can observe that

the virtualization transformation does not have a lower cyclomatic complexity than the programs without obfuscations, which is the opposite of the average results. By examining the Regular Expression's cyclomatic complexity we notice that the same behaviour is also true, e.i., the virtualization obfuscation does not have lower cyclomatic complexity compared to the programs without obfuscations.

Comparing the Maze and Regular Expression programs, we note that the first has it's cyclomatic complexity decreased the more code optimization is done, while on the last it starts to increase when the O1 and O2 levels are applied. Another difference between these two programs is that in the Maze program the literals encoding transformation increases the complexity by a factor of more than 2, and in the Regular Expression program this behaviour is seen on the opaque predicates transformation.

This can be explained by the Regular Expression program already having complex branches due to using recursion on the `matchhere` function and the opaque predicates transformation increases this complexity. For the Maze program related to the variable that hosts the maze being a literal string.

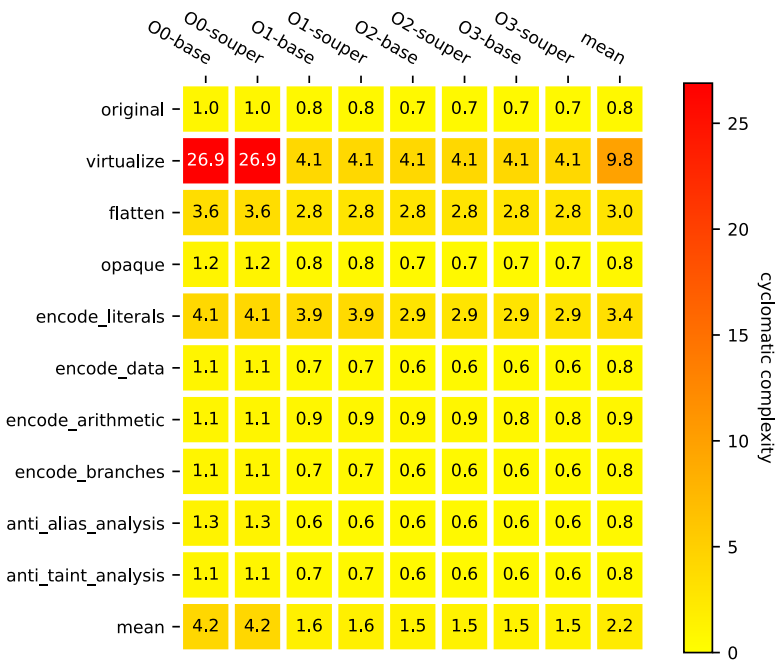


FIGURE 5.2: Cyclomatic complexity of the Maze program

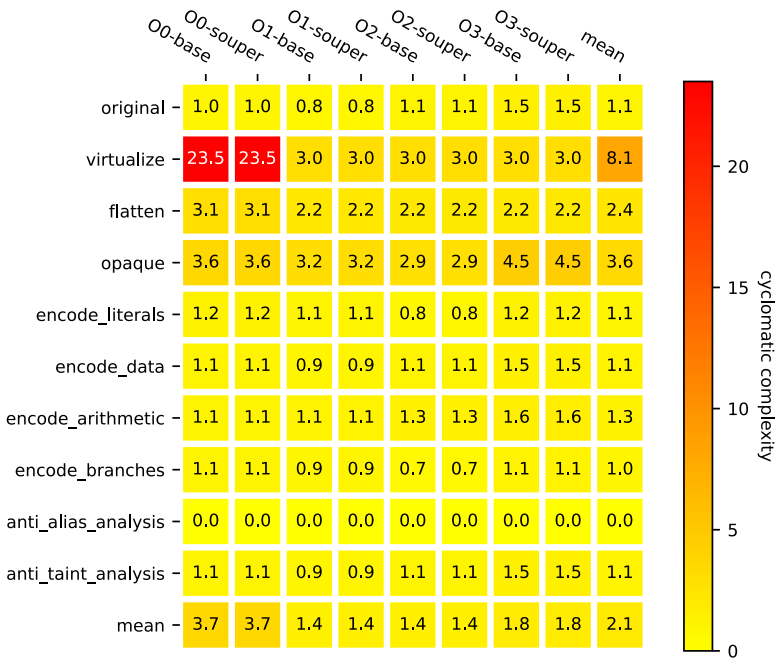


FIGURE 5.3: Cyclomatic complexity of the Regular Expression program

The Barcoder and Message Service programs have more features and are more complete programs overall, comparing to the others in the test set. The values for the Barcoder program on Figure 5.4 show that when the code optimization hits the O2 level, it's cyclomatic complexity is greatly reduced, just as in the Maze program (although starting in different optimization levels), while on the Message Service program depicted on Figure 5.5 the more optimized a program is, the greater is it's cyclomatic complexity. This is the same behaviour observed in the Maze and Regular Expression program, respectively. As expected, the obfuscation that had the highest complexity was the virtualization and the transformations that had the same complexity as the programs without obfuscations were the data encoding, arithmetic encoding, branch encoding, and anti taint analysis.

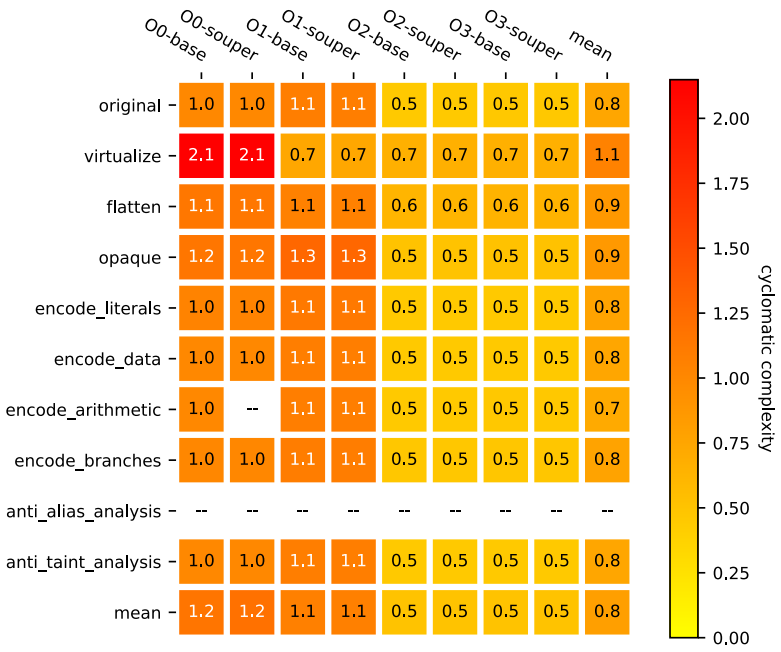


FIGURE 5.4: Cyclomatic complexity of the Barcoder program

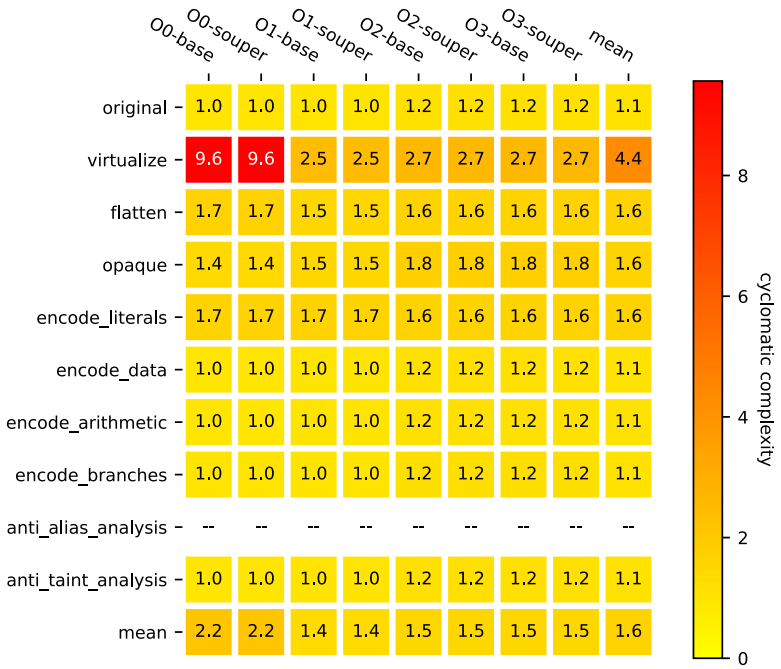


FIGURE 5.5: Cyclomatic complexity of the Message Service program

Looking at the cyclomatic complexity of the Random Function program on Figure 5.6, it is evident that the same behaviour of the Maze and Barcoder programs happen, where

the more optimized a program is, the lower is it's cyclomatic complexity. Aside from the virtualization obfuscation, the flatten transformation is the one that increased the complexity the most. This can be attributed to the fact that the Random Function program contains nested conditional instructions and they are inside while loops (which under the hood are also conditionals).

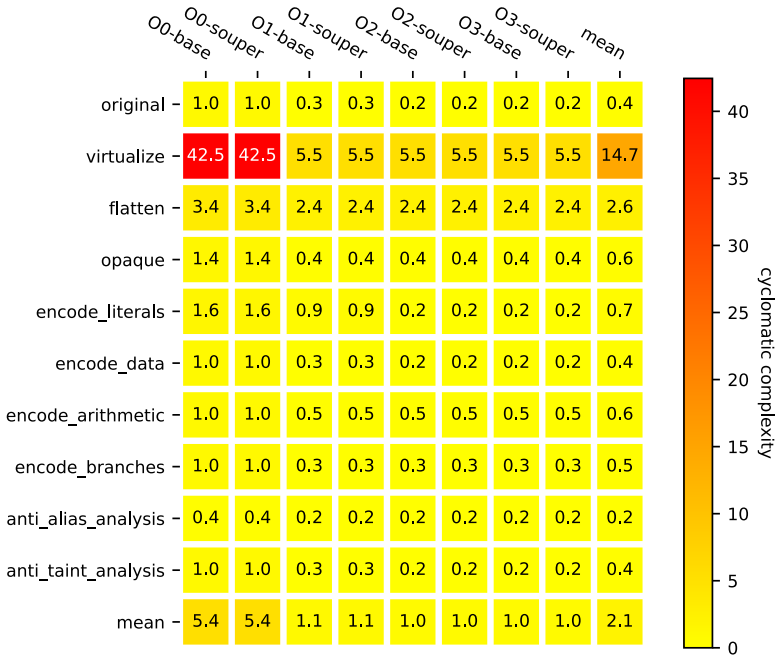


FIGURE 5.6: Cyclomatic complexity of the Random Function program

With these values in mind, we can begin to analyse the coverage results and see if there is any correlation between this variable and the coverage obtained.

5.2.2 KLEE

The first results we will analyse are the coverage KLEE got for all the programs on average when run with a timeout of 30 minutes. By examining the average values for all programs, on Figure 5.7, we note that on average the more a program is optimized, the less coverage it reaches. If we examine the values one by one, there are cases where the optimization helped KLEE to achieve a higher coverage result, specially when talking about the O0 optimization level combined with Souper and the O1 level. Another observation is that the virtualization transformation did not achieve a good coverage result, and as we will see later, this transformation is the one that got worst results in all the tests,

which is expected due to being a transformation that adds a big chunk of code. We can conclude that no affirmation can be done by this plot, merely by the fact that the values do not change so often, and sometimes the code optimizations hinders the analysis tools to achieve higher coverage while on others it helps them to attain better results, where this is dependent on the obfuscation and code optimization, not existing a common pattern.

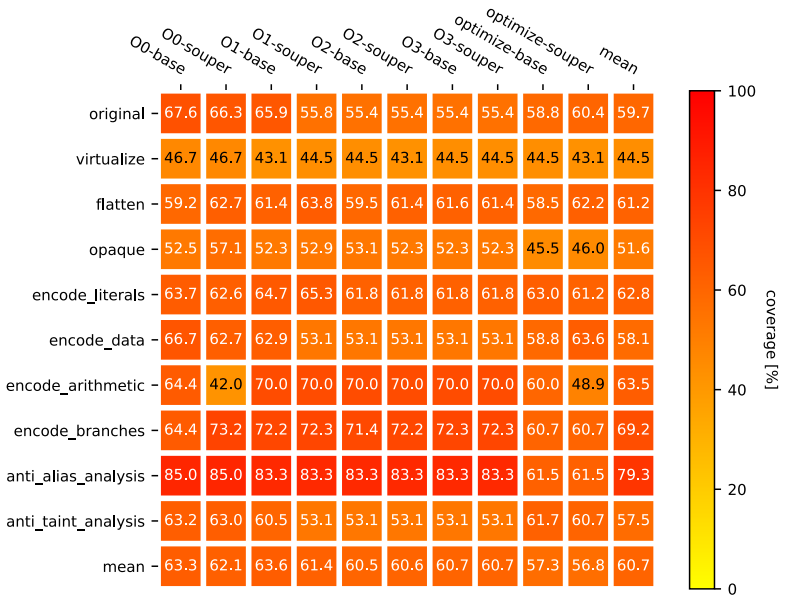


FIGURE 5.7: Average coverage of KLEE's analysis of all programs, with time limit of 30 minutes

The plots on Figures 5.8 and 5.9 depict the coverage KLEE got for the analysis on the programs Maze and Regular Expressions, respectively, when run with a timeout of 30 minutes. We can observe that in the case of the Maze program, the difference on coverage is not significant, except on the virtualization obfuscation. Another curious results is the flatten transformation, where in this case it achieved a higher coverage when compared to the programs without obfuscations. This can be attributed to the fact that most of the Maze program is contained inside a while loop, and the control flattening transformation substitutes the whole program for a switch case, where every basic block inside program is put on the same level. Although this might indicate that it should hinder the coverage obtained, it in fact helps KLEE to analyse the program due to not having the problem of path explosion that loops have. Another observation is that the programs obfuscated but without optimizations, excluding Souper, did achieve a higher coverage than the original program itself.

The results for the Regular Expressions program are similar to that of Maze, except the opaque predicates and anti alias analysis obfuscations achieved less coverage. Although some programs of these transformations, but not limited to, were removed because of erroneous behaviour, we can see that the ones that were not removed did not achieve a good coverage result. This can happen due to the fact that the behaviour of these programs is also wrong but was not noted when interacting with the program manually. And the same as before the virtualization transformation was the worst in terms of coverage.

The results of these two programs are not enough to conclude if the obfuscations were resilient against the optimizations.

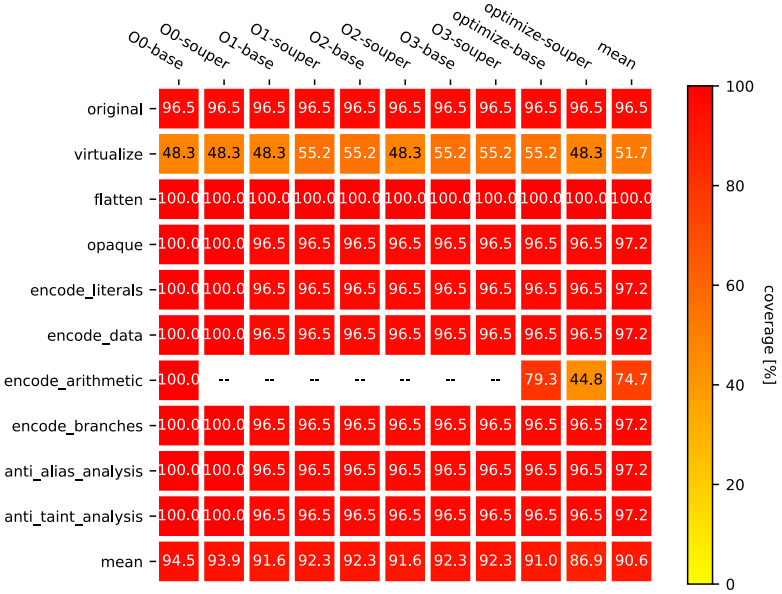


FIGURE 5.8: KLEE analysis coverage of the program Maze, with time limit of 30 minutes and input size of 40 bytes

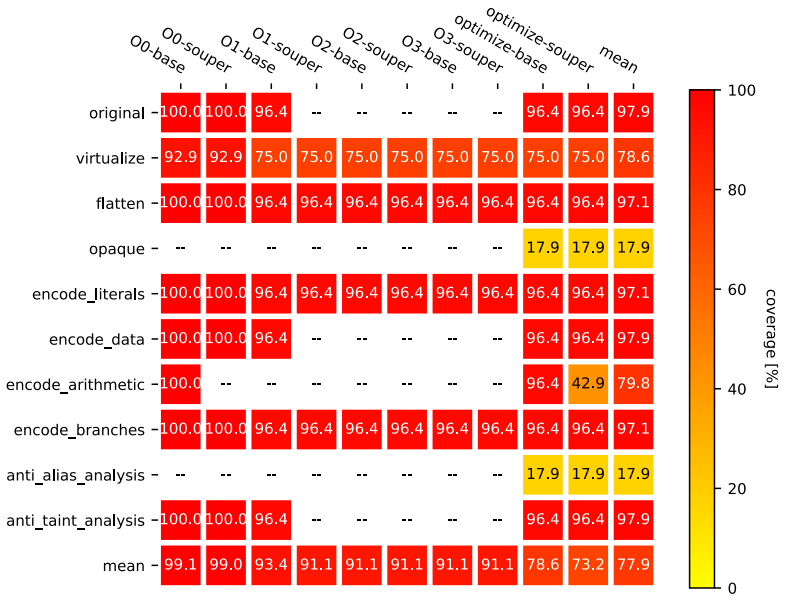


FIGURE 5.9: KLEE analysis coverage of the program Regular Expressions, with time limit of 30 minutes and input size of 8 bytes

Looking at the Barcoder coverage results on Figure 5.10 the obfuscation which yield the least coverage was virtualization, as expected, and the code optimization that had the least coverage was KLEE's optimize flag. This latter evidence is somewhat peculiar due to the fact KLEE's optimize flag only optimizes the program with selected optimizations, which should not hinder the analysis results, on the contrary.

The opposite happens on the Message Service program. By analysing Figure 5.11 we can conclude the code optimization that achieved a higher coverage was KLEE's optimize flag, with and without Souper, and in some combinations of obfuscations and code optimizations the coverage attained was higher than the programs without obfuscations. This can be easily spotted by looking at the colors in the heatmap or at the average values for all obfuscations.

As for the Barcoder program, the code optimizations did not improve the coverage obtained, and a straightforward observation is that the resilience of the obfuscations was high. But if we also look at the program without obfuscations, we can also see that the optimizations did not improve it's coverage, meaning that optimizations might be at fault here.

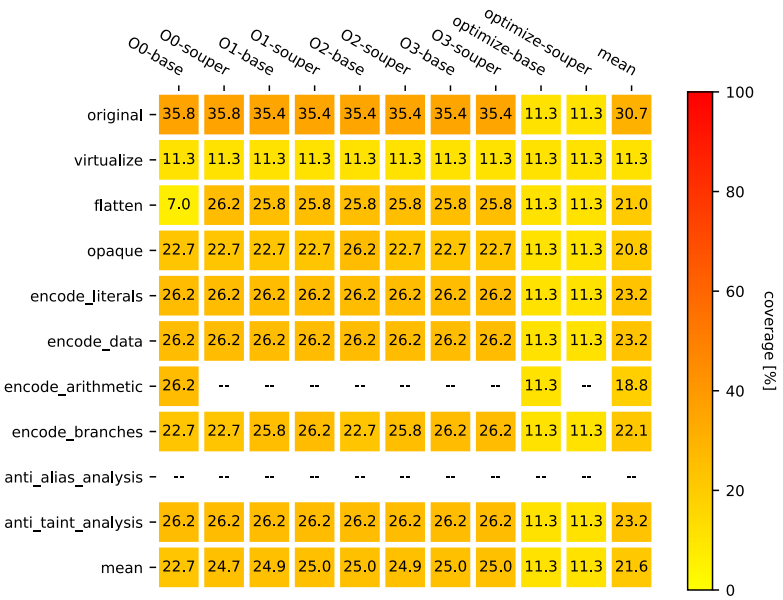


FIGURE 5.10: KLEE analysis coverage of the program Barcoder, with time limit of 30 minutes and input size of 5000 bytes

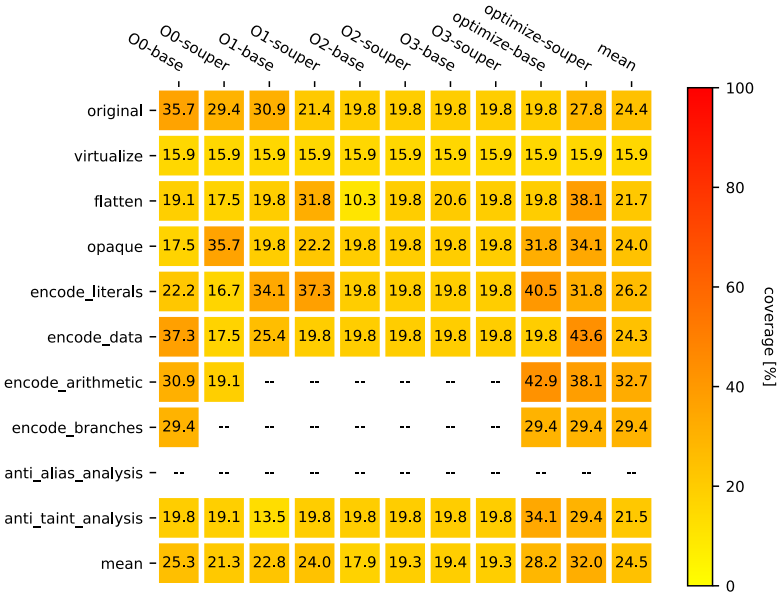


FIGURE 5.11: KLEE analysis coverage of the program Message Service, with time limit of 30 minutes and input size of 2500 bytes

For the last coverage results on KLEE's analysis, we will look at the Random Function program depicted on Figure 5.12, where the coverage is the same for almost all of the

programs, only having the virtualization and flattening obfuscations with an slightly decreased coverage.

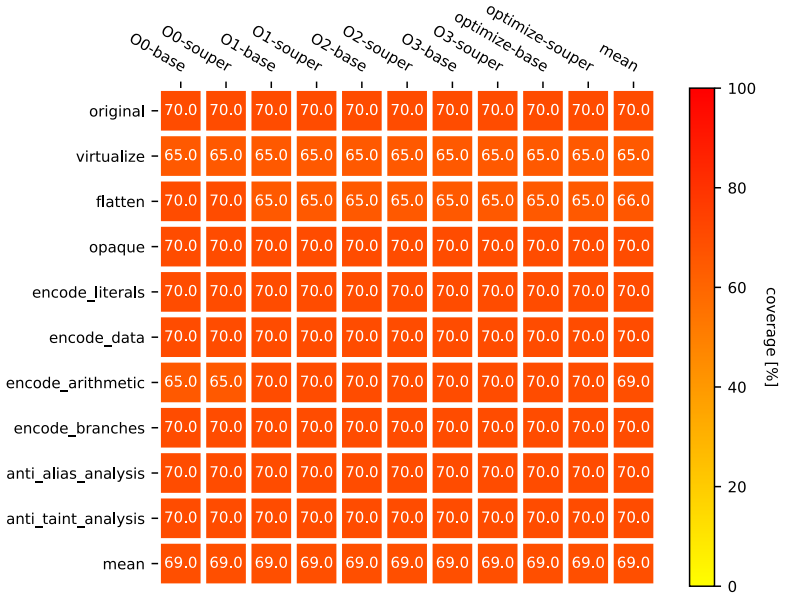


FIGURE 5.12: KLEE analysis coverage of the program Random Function, with time limit of 30 minutes and input size of 20 bytes

As the above discussion showed us, the obfuscations behave differently depending on the program. This is also true for the code optimizations and for the combinations of obfuscations and code optimizations.

5.2.2.1 Solver Time

One of the metrics we extracted from KLEE was the time the analysis spent on the solver. Although this metric alone is not enough to correlate it between the coverage obtained, we chose it because it can help explain some inconsistent results by the combination of obfuscations and code optimizations.

By examining the Figures 5.7 and 5.13, it is evident that the time spent on the solver and the coverage the analysis obtained are not correlated and do not help to explain the overall coverage results, albeit there is a pattern on the virtualization obfuscation.

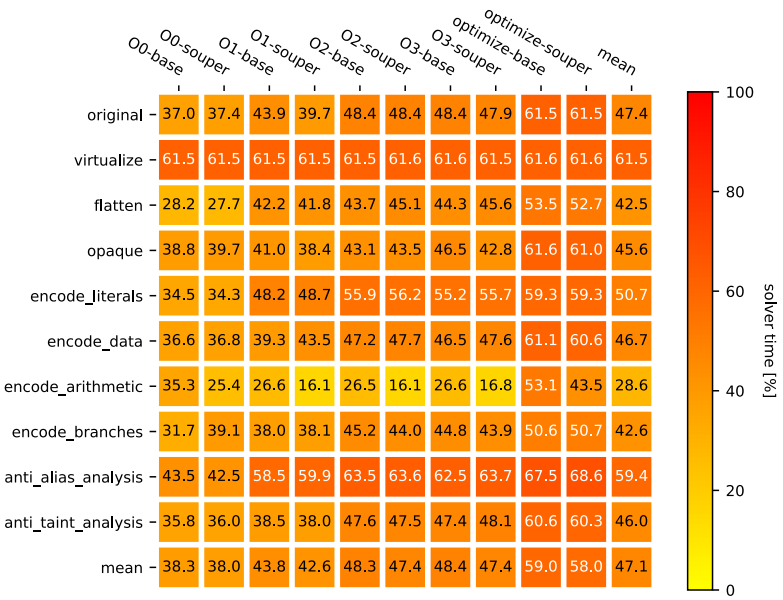


FIGURE 5.13: Average of KLEE's solver time for all analysis, with time limit of 30 minutes

Nevertheless, it is also necessary to look into each program individually. To confirm the conclusions of the last paragraph, we can examine the Figures 5.14 and 5.15 which depict the time spent on the solver for the Barcoder and Message Service programs, respectively. The evidence for the Barcoder program shows that only when the coverage is very low, the solver time is very high, albeit this pattern does not happen on all of the analysis that fit in this condition. If we look at the Message Service program and try to correlate it with the coverage, we can also conclude the time spent on the solver is not an indicative of how much coverage it obtained.

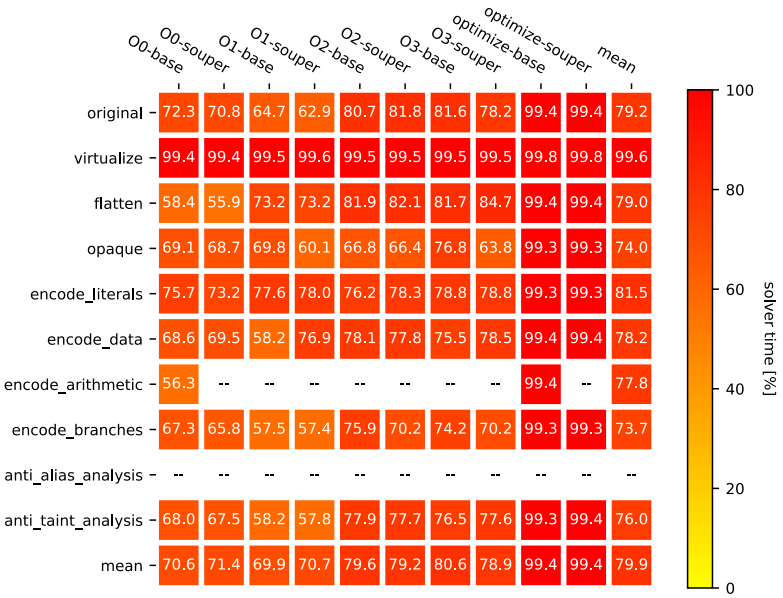


FIGURE 5.14: KLEE solver time for the Barcoder program, with time limit of 30 minutes and input size of 5000 bytes

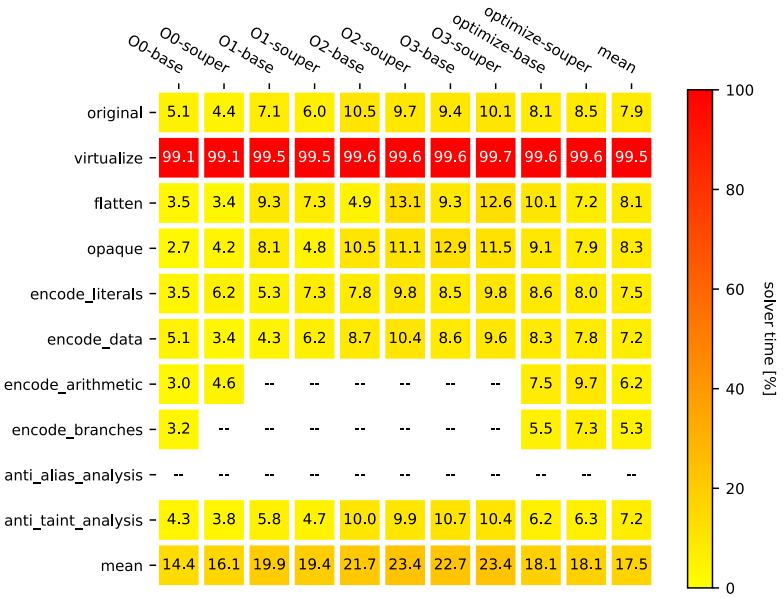


FIGURE 5.15: KLEE solver time for the Message Service program, with time limit of 30 minutes and input size of 2500 bytes

To answer the question of obfuscation resiliency, by examining Figure 5.13 we can confirm that in most of the obfuscations, the more a program was optimized the more the analysis spent on the solver. Although this affirmation is not true for the Message Service program,

which has a peak between the optimization levels O2 and O3 combined with Souper, the average values of all programs do confirm this conclusion.

5.2.2.2 Time Limit

Do the analysis behave differently when the maximum time for an analysis is higher? To try and answer this question we can examine Figure 5.16 that shows the coverage obtained on average for all programs but for a maximum running time of 10 minutes instead of 30. Comparing the Figures 5.16 and 5.7, we notice that just a slightly improvement was made when comparing the 10 and 30 minutes results, but the differences between the obfuscations were not changed. The Maze and Regular Expression programs did not have very different results when the time limit was 10 minutes, compared to the 30 minutes one, so their plots will not be presented.

The program that had more differences in this test was the Message Service program. If we look at Figure 5.17 and compare it with Figure 5.11, we can see that several programs without obfuscations did not achieve a higher coverage than programs with obfuscations, but when looking at the latter figure, in which the analysis ran for 30 minutes, we can see that some of these programs achieved higher coverage than their obfuscated counterparts.

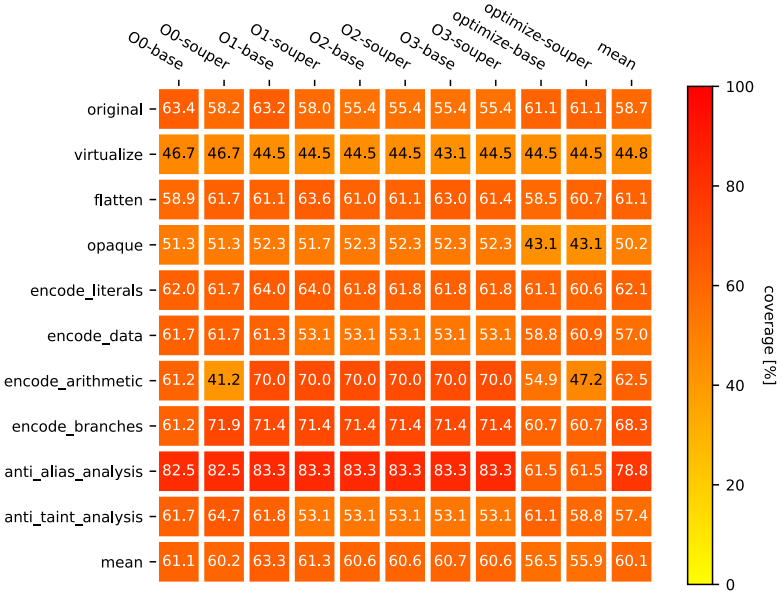


FIGURE 5.16: Average coverage of KLEE's analysis of all programs, with time limit of 10 minutes

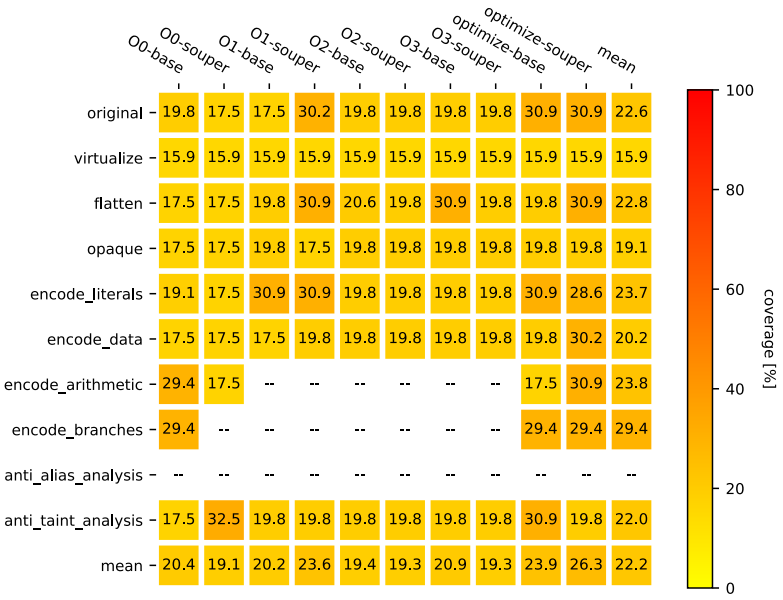


FIGURE 5.17: KLEE analysis coverage of the Message Service program, with time limit of 10 minutes and input size of 2500 bytes

The same happens on the Random Function program, depicted on Figure 5.18, where when run with a 10 minute limit, the programs without optimizations had a lower coverage results than their optimized counterparts. But if we look at the 30 minute limit plot, on Figure 5.12, we can see that the programs without optimizations achieved the same results as the optimized ones in most of the programs.

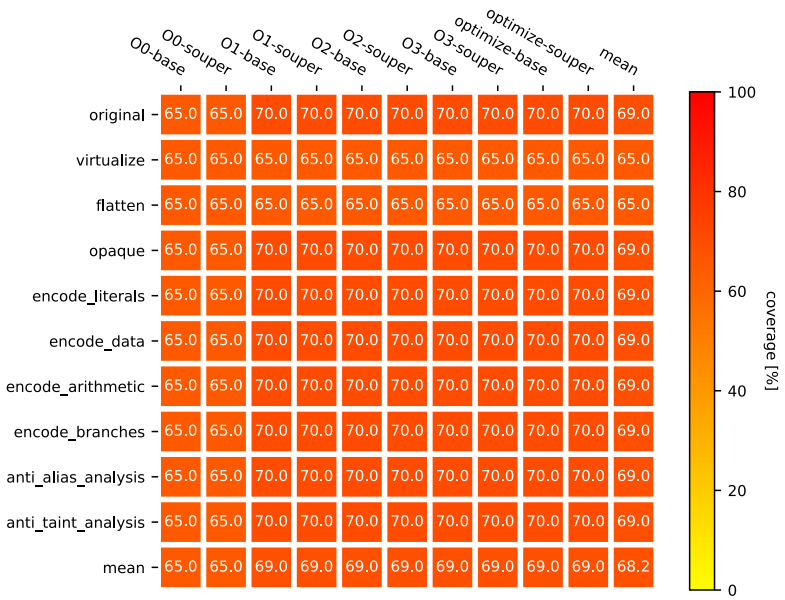


FIGURE 5.18: KLEE analysis coverage of the Random Function program, with time limit of 10 minutes and input size of 20 bytes

With the discussion above we can conclude the more time an analysis has, the more accurate it's results will be, not just in terms of coverage but in overall equilibrium, and that this variable is a very important one when it comes to having accurate results.

5.2.2.3 Input Size

In this section we will see the difference the input size has on the analysis. For this we ran the Barcoder and Message Service programs with a time limit of 10 minutes but with double the input size, from 5000 to 10000 bytes and from 2500 to 5000 bytes, respectively. The results are shown on Figures 5.20 and 5.21.

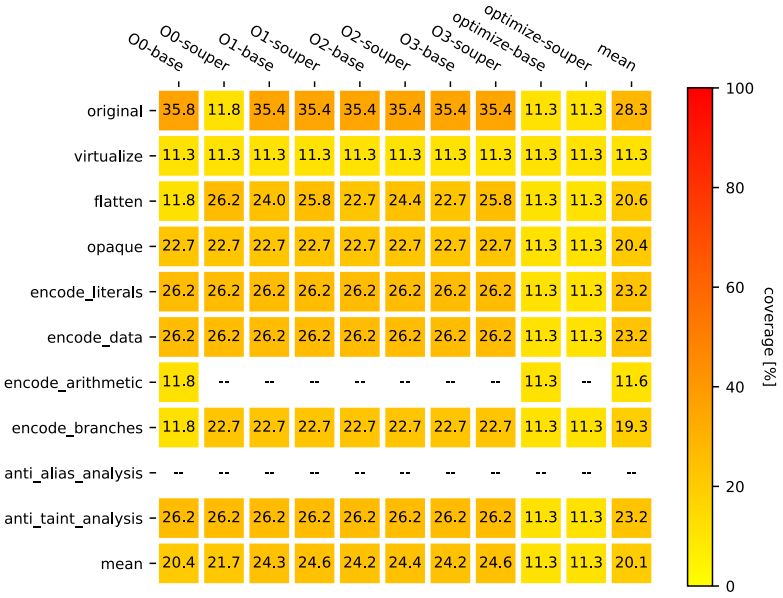


FIGURE 5.19: KLEE analysis coverage of the Barcoder program, with time limit of 10 minutes and input size of 5000 bytes

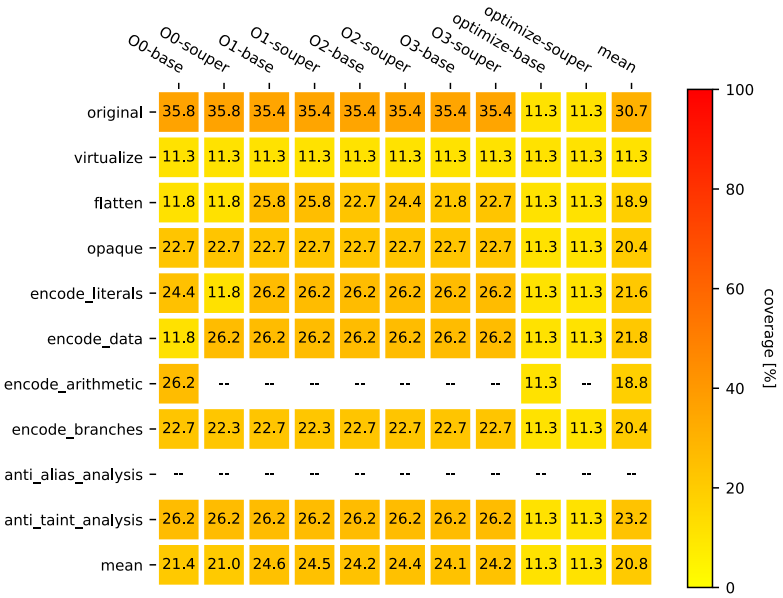


FIGURE 5.20: KLEE analysis coverage of the Barcoder program, with time limit of 10 minutes and input size of 10000 bytes

When comparing the Barcoder programs, Figures 5.19 and 5.20, we notice that there is a slight change to the results. Overall the coverage achieved is higher when the input is also higher, but when looking in detail we can observe that some obfuscations had improved coverage while others had their coverage decreased. The same is true when looking at

the code optimizations. The contrary is true on the Message Service program, depicted on Figure 5.21 and comparing it to Figure 5.17, where the bigger the input the lesser is the coverage achieved, although in a very limited value if we look at the average.

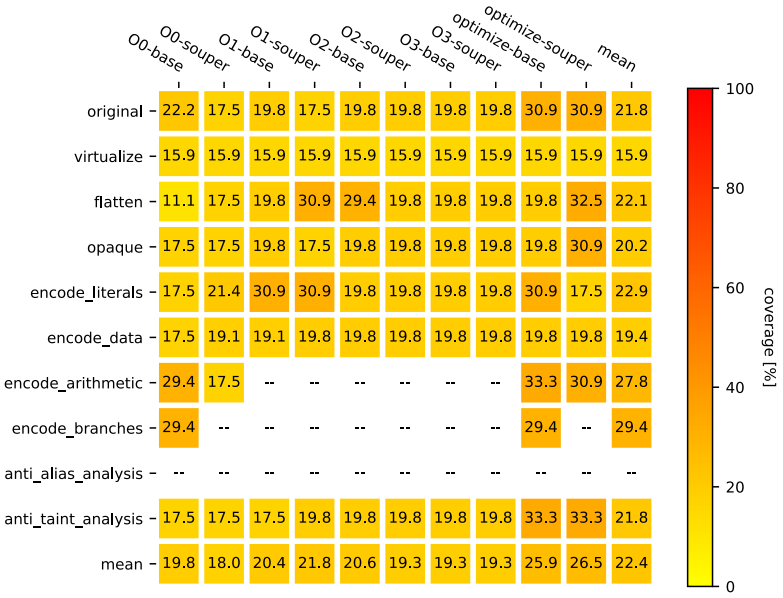


FIGURE 5.21: KLEE analysis coverage of the Message Service program, with time limit of 10 minutes and input size of 5000 bytes

This determines that the input size is not such a bigger factor on the outcome of the analysis as it is the maximum time an analysis can run, but on some cases if this value is too big it can hinder the analysis and yield worse coverage results.

5.2.3 AFL++

The second analysis tool used was the AFL++ fuzzer and the same tests for the KLEE symbolic execution framework were executed. On Figure 5.22, the average results obtained when running AFL++ with the programs in the test set and with a time limit of 30 minutes are shown. By looking at the plot, we can notice right away that these coverage results are much higher than in KLEE's analysis. One explanation to this is that because AFL++ is a fuzzer and does not have a solver to generate new inputs or explore new paths, it's speed in finding the available paths in a program is greater. Another curious remark is that the values are not much different from program to program except on some of them, whereas on the KLEE's analysis it was. Overall the same behaviour is detected as in the

KLEE analysis, where the more a program is optimized the less coverage it obtains, at least when looking at the average results.

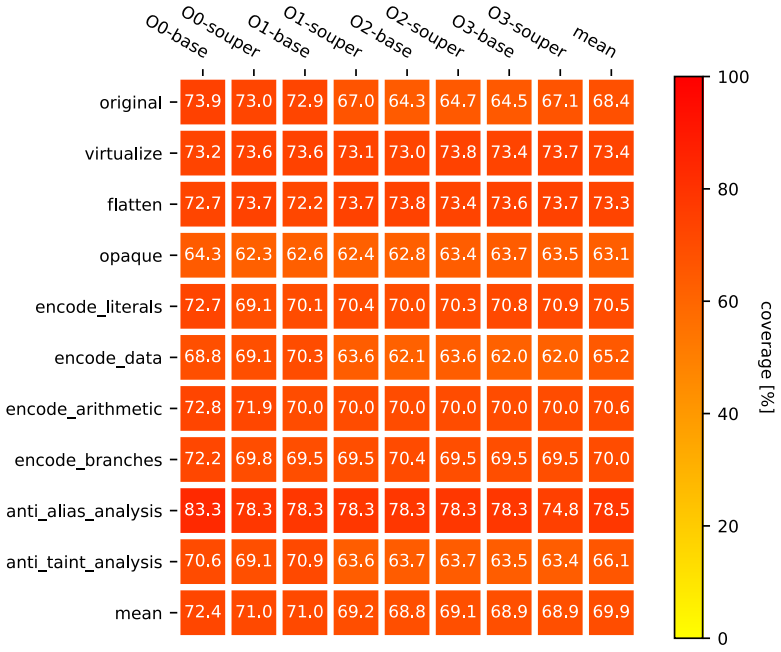


FIGURE 5.22: Average coverage of AFL++'s analysis of all programs, with time limit of 30 minutes

For the AFL++ analysis on the Maze and Regular expressions programs, depicted on Figures 5.23 and 5.24 respectively, we can see that little or no difference exists between the original program and the others.

The Maze and Regular Expressions programs are very small and uncovering most of the paths including in the obfuscated ones should not be hard. Due to this, in the Regular Expression program some obfuscations achieved higher coverage than the programs without obfuscations. This same situation happened on KLEE's analysis.

Looking at the results, one obfuscation that stood out is the virtualization, due to the fact that in KLEE's analysis this transformation yielded very small coverage percentage compared with the other obfuscations. This happens because fuzzers do not have the problems which symbolic execution have, mainly path explosions and complex constraints to be solved, which makes the analysis of some obfuscation having the same or better performance than the programs without obfuscation.

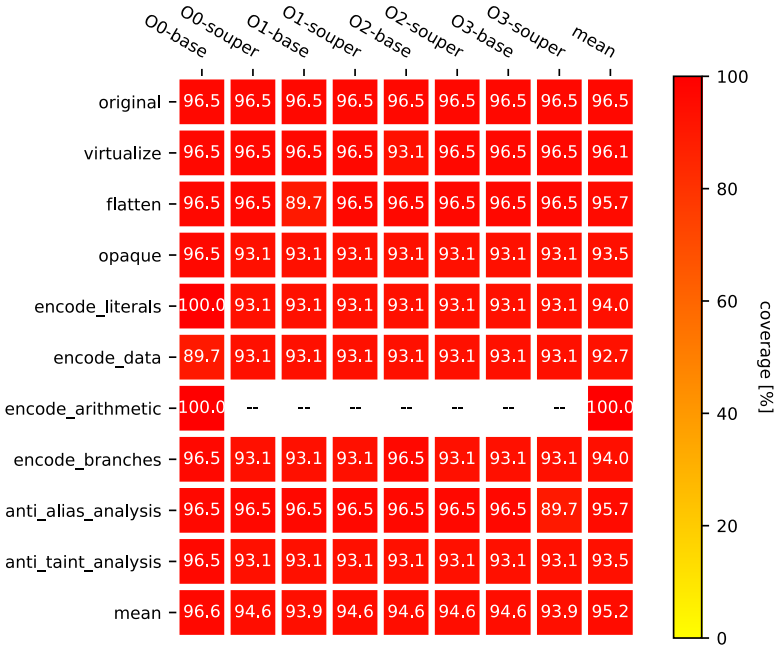


FIGURE 5.23: AFL++ analysis coverage of the Maze program, with time limit of 30 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 40 bytes

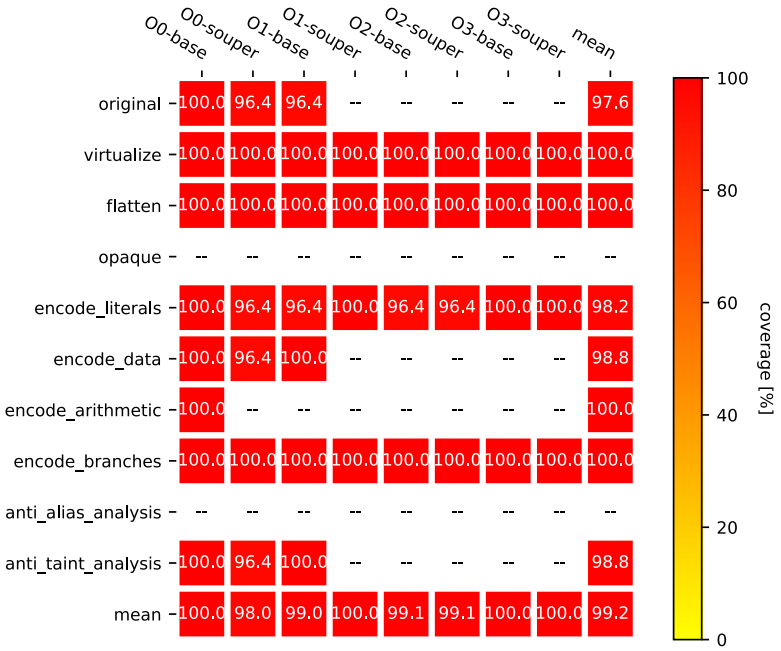


FIGURE 5.24: AFL++ analysis coverage of the Regular Expression program, with time limit of 30 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 8 bytes

Now, let's analyse the results for more complex programs, Barcoder and Message Service, depicted on Figures 5.25 and 5.26 respectively. The results for the Barcoder program are on pair with KLEE's analysis, but the Message Service program achieved a very high coverage for all programs. One question that arises is why the AFL++ analysis of the Message Service program attained a much higher coverage percentage compared to the AFL++ analysis of the Barcoder program, while on KLEE's analysis these values were on the same order of magnitude of the Barcoder program. Even if we compare the cyclomatic complexity of both programs there is no correlation between them.

One of the things we can also notice on these programs is that the virtualization obfuscation achieves the same or better coverage percentage than other obfuscations. Indeed, most of the obfuscations and code optimizations achieved almost the same coverage. Again, this is due to the fact fuzzers do not employ other complex software to be able to find more paths.

Another observation is that only the Barcoder program attained less coverage compared to the other programs. Even if we look into KLEE's analysis depicted on the previous subsection, the Barcoder program had almost the same coverage of the Message Service program on average, while in the AFL++ analysis this is not true.

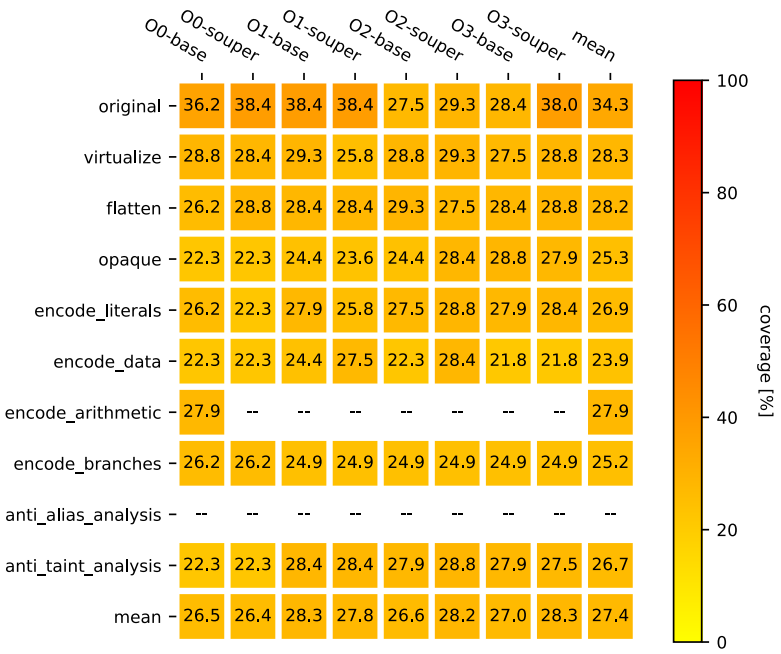


FIGURE 5.25: AFL++ analysis coverage of the Barcoder program, with time limit of 30 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 5000 bytes

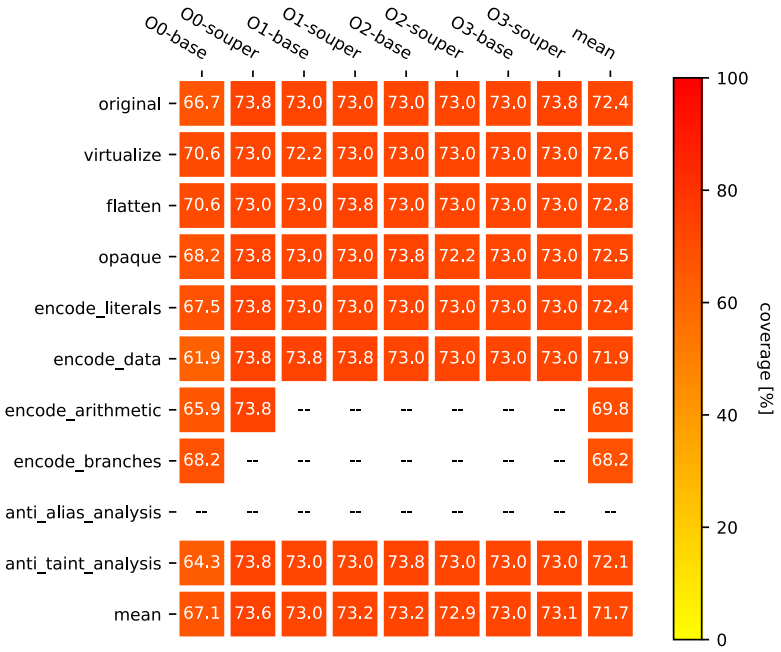


FIGURE 5.26: AFL++ analysis coverage of the Message Service program, with time limit of 30 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 2500 bytes

The results for the Random Functions program are shown in Figure 5.27 and all of the results are either 60% or 70%, and it can also be seen that the programs without optimizations yielded a higher coverage than those with optimizations. Also, the same happens as in the other programs, where the virtualization obfuscation was one of the obfuscations which had higher coverage results.

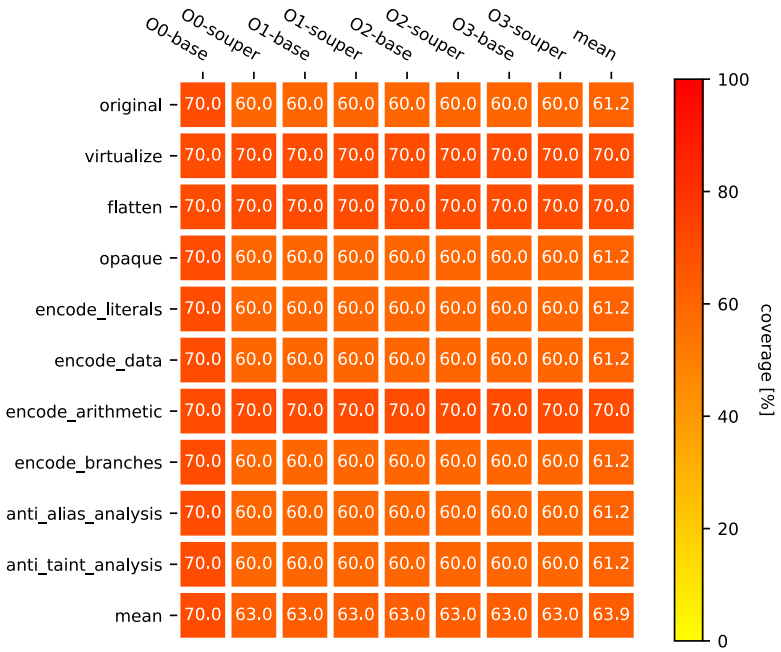


FIGURE 5.27: AFL++ analysis coverage of the Random Function program, with time limit of 30 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 20 bytes

5.2.3.1 Time Limit

This test consists in lowering the value of the maximum time the analysis can run and examining the results to see how much it hampers the coverage obtained. Just as in the KLEE analysis, we are trying to identify if and how much the time the analysis has to run has an impact on the coverage.

First, we can look at the average value for the coverage obtained when running the analysis with a time limit of 10 minutes. The results are present on Figure 5.28 and comparing them to the Figure 5.22 we can observe that the values are not so different, with an average coverage of 67% to 69.9%. This means that even the analysis taking the triple the time, it did not get much further than the 10 minutes limit test, albeit the results were more solid due to the fact that the programs without optimizations had increased coverage when ran

with a 30 minute limit. This is the same result that was noticed on KLEE's analysis and it shows that in order to obtain accurate results, the time limit is very important.

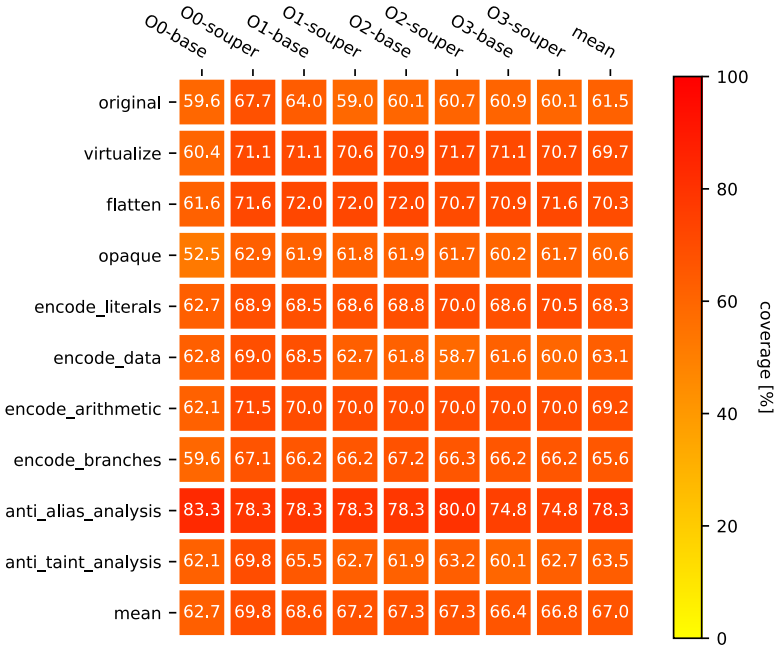


FIGURE 5.28: Average coverage for the AFL++ analysis of all programs, with time limit of 10 minutes and initial input of 10% of the KLEE analysis of the original program

When looking in detail on the analysis, the Maze and Regular Expressions program achieved very similar results with the limit of 10 or 30 minutes. If we look at the Regular Expressions program depicted on Figure 5.29 we can also observe that behaviour except on the virtualization obfuscation, where it increased from 65%, with a time limit of 10 minutes, to 70%, with a time limit of 30 minutes.

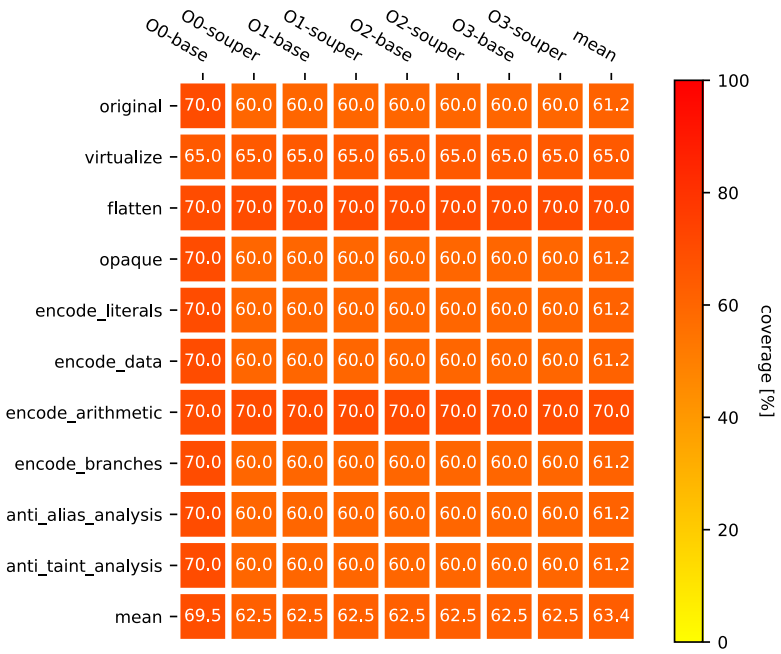


FIGURE 5.29: AFL++ analysis coverage of the Regular Expressions program, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 20 bytes

5.2.3.2 Input Size

In the next tests, we increased the size of the initial inputs while the percentage given to AFL++ remained the same. The Figure 5.30 depicts the results obtained on average. By examining it, we can conclude that the coverage obtained has been reduced dramatically, specially on the programs without optimizations and in the encode branches obfuscation. This means that for fuzzing tools, it is fundamental to define a good size of the initial inputs given. A curious result is that is it very evident the optimizations aided in the discovery of new inputs and successively the coverage attained, although the more optimized a program was it did not mean that it achieved better coverage. One of the explanations is that the programs without obfuscations contain some instructions that are slow to execute, hindering the results of the analysis.

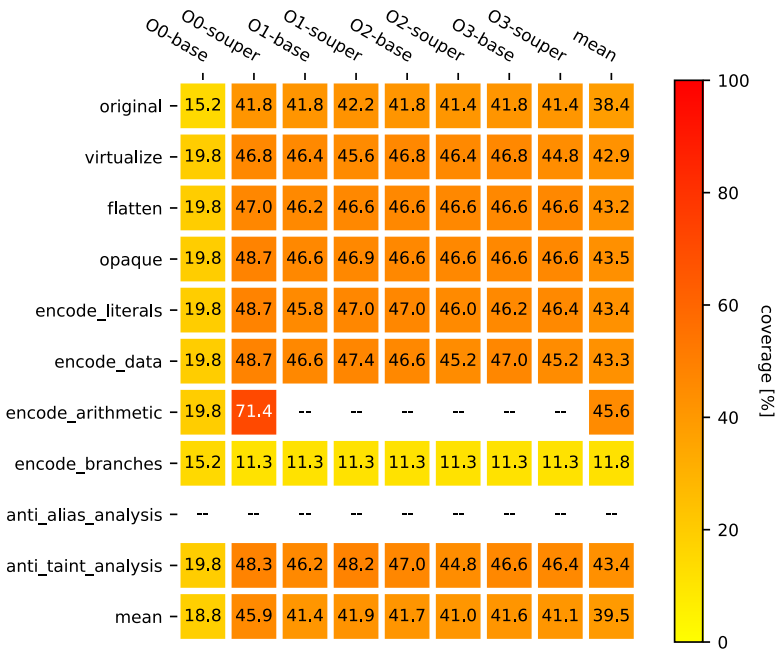


FIGURE 5.30: Average coverage for the AFL++ analysis of all programs, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, and input sizes double of the original

The programs without optimizations and the encode branches obfuscation had very questioning coverage results compared to the other programs, but our analysis into these values did not uncover abnormal behaviour on the programs nor in their analysis.

By looking at the analysis of the programs separately, we can see that this issue occurred on the Message Service program, as can be seen on Figure 5.31. Because this test was only performed on some of the programs, namely the Barcoder and Message Service programs, the average values are not the best approach to this type of analysis.

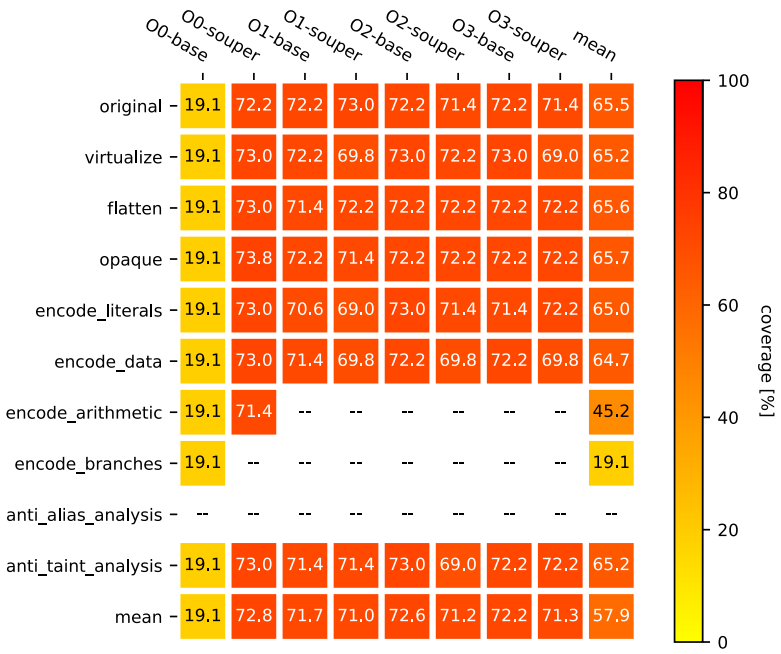


FIGURE 5.31: AFL++ analysis coverage of the Message Service program, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 5000 bytes

The plots on Figures 5.31 and 5.32 show the AFL++ analysis of the Message Service program with 5000 and 2500 bytes of input, respectively. From these plots we can see that by doubling the input size the coverage obtained was slightly higher on average, but no significant increase was obtained.

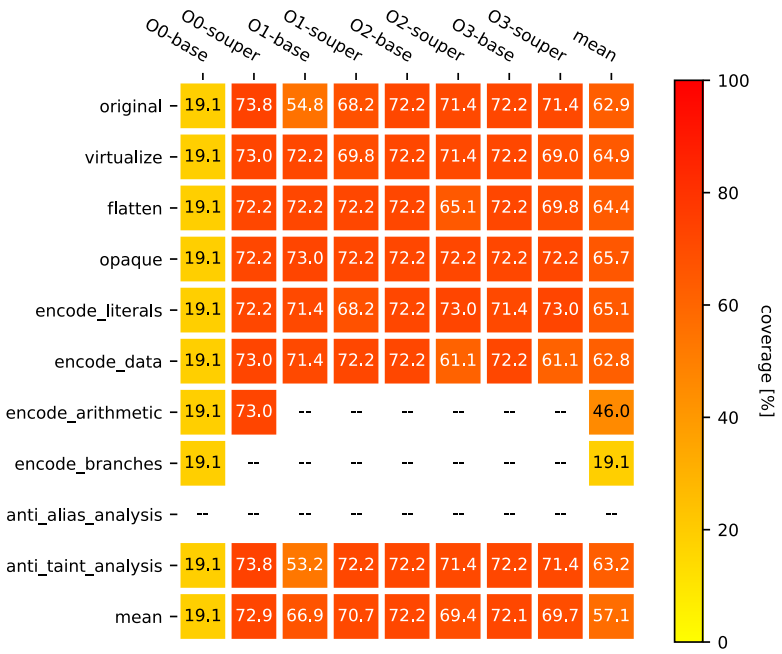


FIGURE 5.32: AFL++ analysis coverage of the Message Service program, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 2500 bytes

In the Barcoder program, the opposite happens, where by doubling the input, the coverage obtained was lower. This can be seen by comparing the Figures 5.33 and 5.34, where the former attained a lower coverage on average. This behaviour where different programs present different behaviours when analysed was already noticed on some of the analysis discussed earlier, and this one is no exception.

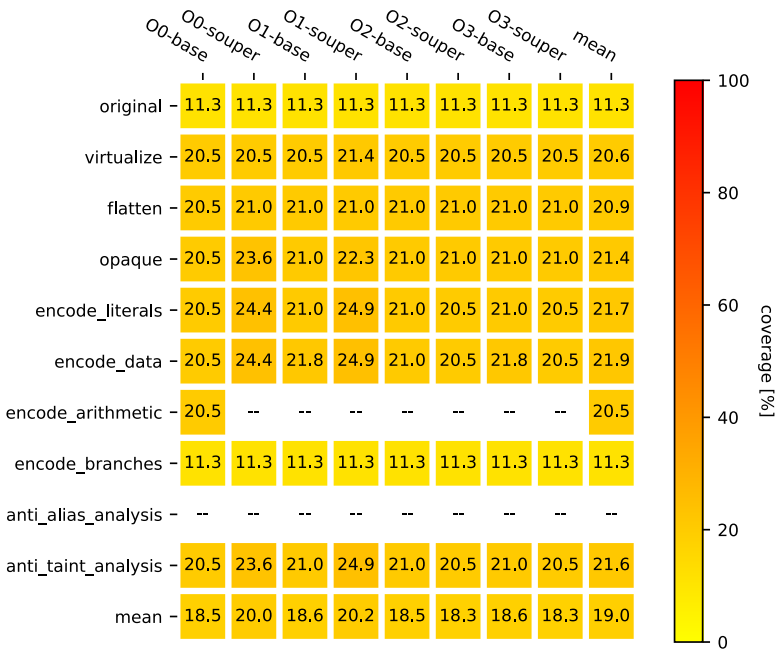


FIGURE 5.33: AFL++ analysis coverage of the Barcoder program, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 10000 bytes

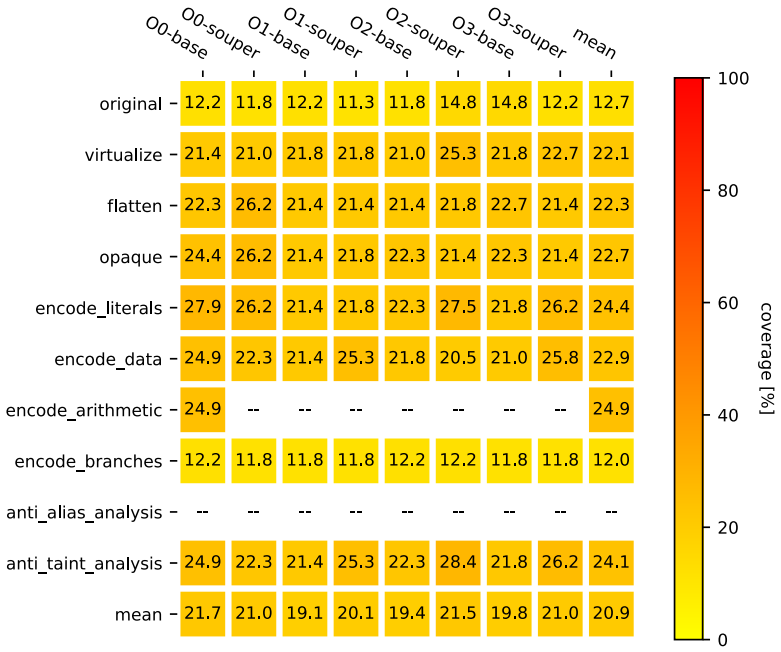


FIGURE 5.34: AFL++ analysis coverage of the Barcoder program, with time limit of 10 minutes, initial input of 10% of the KLEE analysis of the original program, with an input size of 5000 bytes

5.2.4 Final Discussion

An initial observation is that the plots that describe the average results are not the best in terms of accuracy due to the fact that small or big values in an analysis will undermine the average results. While examining all of the results obtained, we can state some key findings. One of them is the time an analysis runs is essential to attain accurate results. This was noted when looking at the tests with a time limit of 10 minutes, where the programs without optimizations yielded worse results than the others, which was not observed in the tests that contained a time limit of 30 minutes. Another key finding is that in more than half of the analysis, the more a program is optimized, the less coverage it obtained, even with the optimization that KLEE offers which should help the analysis framework to attain better results. We also noted that for the the AFL++ analysis, the input size is very important, where a bigger input size does not translate to a higher coverage, which means that a good input size is needed to achieve good coverage results when running AFL++. We also saw that the results could not be explained and correlated by the programs cyclomatic complexity in both of the analysis tools, neither by the time spent on the solver in the case of KLEE. The results obtained varied from program to program, obfuscation to obfuscation, and optimization to optimization, meaning that the results of the analysis is very dependent on the program being evaluated and a deeper study is needed as to why this happens.

Chapter 6

Conclusion

This research aimed to answer the question if the obfuscations were resilient to optimizations, and to what extent. To answer this question, it is evident that optimizations could not systematically revert the obfuscations applied, and on some cases they aggravated the analysis of the tools and depending on the program it had different results. This means that the results do not confirm the empirical sense that optimizations can somewhat revert obfuscations, which means the question cannot be easily answered and a deep dive into how the analysis tools work and the inner working of the analysed programs is mandatory to explain this behaviour, meaning that further research is needed. The research process contained in this thesis, mainly the methodology part, is a contribution on methodologies to test obfuscations and/or optimizations and if the former is resilient to the latter, including an empirical study using the proposed methodology. Our perspective is that of one who wants to analyse all possible execution paths in the program. We took this approach due to the fact that nowadays software is being more and more tested by automated analysis tools, e.g., during CI/CD, while also being highly used when reverse engineering a program, making this framework in line with this new approach. As discussed on the Chapter 5, the evidence suggests that depending on the program, the obfuscations and optimizations have different effects on the analysis of the tools, where in some cases the code optimization increased the coverage obtained, while on others it was the opposite. Also, no variable that we tested could describe these results. More than one test was made on each program by changing the execution variables of the analysis tools and it was found that the variable that has a higher weight in improving

the coverage is the maximum time an analysis runs. It was also found the input size is an important variable when analysing with AFl++, where a big input does not necessarily mean greater coverage, on the contrary.

Unfortunately, some of the tools we experimented when making this thesis were still very immature, being very often not maintained, not supporting the most recent versions of software and dependencies, and failing to run or throw errors that hinder all of the steps in this thesis. One example of this is the tools we used initially to lift compiled code back into LLVM IR to instrumentalize and analyse it with KLEE, but none of them provided accurate results or failed to run. This includes tools such as retdec [86], mcsema [87] and revng [88]. Another example is Tigress and Souper which failed to obfuscate or optimize some programs, respectively, by throwing unknown errors or executing indefinitely. Going into the optimzations tool set, we observed that some programs failed to retain their semantic and the same input gave different results depending on the program, meaning that when mixing obfuscations and optimizations the logic of the program was changed. Several time spent on the production of this thesis was trying to get all of the tools work without any errors and all of them supporting the same versions of the underlying software, may it be dependencies or OS's. Very often, the tools being tested were broken, or did not work reliably, meaning they failed when run on any significant real-world software. Another adversity was the interoperability of the tools tested, where the file formats each one accepted as input and output might differ, which also limited the set of tools we could choose in order to produce this thesis [89]. Although several tools were tested, the methodology proposed and respective implementation of this thesis was limited by these factors.

6.1 Future Work

Further research is needed to determine the causes/effects between obfuscations and optimizations on automated analysis tools. One point in what this thesis lacked was the testing of several obfuscations combined on the same program. Another approach to this testing would be to let the analysis reach 100% coverage, as done by Banuescu et al. [4], independent on the time it takes it to be fulfilled. For a detailed analysis, we would also need to examine individual optimisations triggered by a compiler at each level, and if possible, examine their effect individually.

Future work also includes a more comprehensive study on why different programs exhibit different behaviours when some analysis variables are changed, e.g., input size, time limit, etc., and applying the methodology on this thesis to a more extensive set of programs while additionally applying different obfuscation techniques for each transformation. One of the initial approaches of this thesis was to try and lift compiled code into an intermediate language, e.g., LLVM IR, so that we were not limited to having the source code of the program, but as experience showed these tools were not outputting the correct program, making the behaviour change or not even compile to machine code again.

Other metrics for the programs beyond the cyclomatic complexity can be extracted such as the program length, nesting complexity, data flow complexity, fan-in/out complexity, data structure complexity and object-oriented design metrics. This was done by Collberg et al. [3] and can be applied to this framework.

As stated on this chapter, some of the tools tested along the way, e.g., code lifters, Tigress and Souper, had errors when executed and some programs could not be generated. Also, some analysis could not be performed due to the generated program having different semantics, meaning that for the same input the generated program did not have the same output as the original program. This is due to the fact that we are using these tools in depth and in an unusual way. This also serves as future work, that is, to use these tools more extensively to catch all the errors and fix them.

Bibliography

[1] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening code obfuscation against automated attacks," *ArXiv*, vol. abs/2106.08913, 2021. [Cited on page 2.]

[2] H. Xu, Y. Zhou, J. Ming, and M. Lyu, "Layered obfuscation: a taxonomy of software obfuscation techniques for layered security," *Cybersecurity*, vol. 3, p. 9, 04 2020. [Cited on page 5.]

[3] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, 01 1997. [Cited on pages 5, 8, and 82.]

[4] S. Banescu, C. S. Collberg, and A. Pretschner, "Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning," in *USENIX Security Symposium*, 2017. [Cited on pages 5, 6, and 81.]

[5] S. Banescu, M. Ochoa, and A. Pretschner, "A for measuring software obfuscation resilience against automated attacks," 05 2015, pp. 45–51. [Cited on page 5.]

[6] K. Heffner and C. Collberg, "The obfuscation executive," in *Information Security*, K. Zhang and Y. Zheng, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 428–440. [Cited on page 6.]

[7] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 674–691. [Cited on page 6.]

[8] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual*

Conference on Computer Security Applications, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–200. [Online]. Available: https://doi.org/10.1145/2991079.2991114 [Cited on page 6.]

[9] P. Garba and M. Favaro, "Saturn – software deobfuscation framework based on llvm," 2019. [Cited on page 6.]

[10] J. Salwan, S. Bardin, and M.-L. Potet, *Symbolic Deobfuscation: From Virtualized Code Back to the Original*, 06 2018, pp. 372–392. [Cited on page 6.]

[11] S. B. Jonathan Salwan and M.-L. Potet, "Playing with binary analysis: Deobfuscation of vm based software protection," 2017, last accessed 12 November 2021. [Online]. Available: http://shell-storm.org/talks/SSTIC2017_Deobfuscation_of_VM_based_software_protection.pdf [Cited on page 7.]

[12] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software a semantics-based approach," 10 2011, pp. 275–284. [Cited on page 7.]

[13] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 643–659. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko [Cited on page 7.]

[14] B. Yadegari and S. K. Debray, "Symbolic execution of obfuscated code," *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015. [Cited on page 7.]

[15] B. Spasojević, "Using optimization algorithms for malware deobfuscation," Master's thesis, University of Zagreb - Faculty of Electrical Engineering and Computing, 2010. [Cited on page 7.]

[16] J. H. Suk, Y. B. Lee, and D. H. Lee, "Score: Source code optimization amp; reconstruction," *IEEE Access*, vol. 8, pp. 129 478–129 496, 2020. [Cited on page 7.]

[17] M. Liang, Z. Li, Q. Zeng, and Z. Fang, "Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization," in *Information and*

Communications Security, S. Qing, C. Mitchell, L. Chen, and D. Liu, Eds. Cham: Springer International Publishing, 2018, pp. 313–324. [Cited on page 8.]

[18] D. Canavese, L. Regano, C. Basile, and A. Viticchié, "Estimating software obfuscation potency with artificial neural networks," 09 2017, pp. 193–202. [Cited on page 8.]

[19] Christian Collberg, "The tigress c obfuscator," https://tigress.wtf/, 2021, last accessed 17 November 2021. [Cited on pages 9 and 29.]

[20] P. Godefroid, "Fuzzing: Hack, art, and science," *Commun. ACM*, vol. 63, no. 2, p. 70–76, jan 2020. [Online]. Available: https://doi.org/10.1145/3363824 [Cited on pages 15 and 16.]

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157. [Cited on pages 15 and 21.]

[22] Charlie Miller - Independent Security Evaluators, "Fuzzing with code coverage by example," https://fuzzinginfo.files.wordpress.com/2012/05/cmiller_toorcon2007.pdf, 2005, last accessed 17 November 2021. [Cited on page 15.]

[23] Michal Zalewski, "american fuzzy lop," https://lcamtuf.coredump.cx/afl/, 2021, last accessed 17 November 2021. [Cited on page 16.]

[24] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 818–825. [Cited on page 16.]

[25] B. Yadegari and S. Debray, "Bit-level taint analysis," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 255–264. [Cited on page 16.]

[26] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 725–741. [Cited on page 16.]

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016. [Cited on pages 16 and 21.]

[28] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Cited on page 17.]

[29] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05, 2005, p. 41. [Cited on page 17.]

[30] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657 [Cited on page 18.]

[31] T. Zhang, P. Wang, and X. Guo, "A survey of symbolic execution and its tool klee," *Procedia Computer Science*, vol. 166, pp. 330–334, 2020, proceedings of the 3rd International Conference on Mechatronics and Intelligent Robotics (ICMIR-2019). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705092030212X [Cited on pages 18 and 22.]

[32] Wikipedia contributors, "2016 cyber grand challenge — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=2016_Cyber_Grand_Challenge&oldid=1044678094, 2021, [Online; accessed 16-November-2021]. [Cited on page 18.]

[33] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft." *Queue*, vol. 10, no. 1, p. 20–27, jan 2012. [Online]. Available: https://doi.org/10.1145/2090147.2094081 [Cited on page 18.]

[34] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," 01 2010, pp. 317–331. [Cited on pages 20 and 21.]

[35] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, p. 82–90, feb 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408795 [Cited on page 20.]

[36] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R.

Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381. [Cited on page 20.]

[37] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 95–111. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041563 [Cited on page 20.]

[38] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 183–194. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831732 [Cited on page 20.]

[39] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254088 [Cited on page 20.]

[40] SMT Steering Committee, "Smt-comp 2021 — smt-comp," https://smt-comp.github.io/2021/benchmarks.html, 2021, [Online; accessed 16-November-2021]. [Cited on page 20.]

[41] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, 2007, p. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2007.41 [Cited on page 21.]

[42] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson, "The mayhem cyber reasoning system," *IEEE Security Privacy*, vol. 16, no. 2, pp. 52–60, 2018. [Cited on page 21.]

[43] Wikipedia contributors, "Satisfiability modulo theories — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Satisfiability_modulo_theories&oldid=1056299526, 2021, [Online; accessed 21-November-2021]. [Cited on page 21.]

[44] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," 01 2009. [Cited on page 21.]

[45] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, no. 3, p. 265–278, mar 2011. [Online]. Available: https://doi.org/10.1145/1961296.1950396 [Cited on page 21.]

[46] ——, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, feb 2012. [Online]. Available: https://doi.org/10.1145/2110356.2110358 [Cited on page 21.]

[47] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," *2012 IEEE Symposium on Security and Privacy*, pp. 380–394, 2012. [Cited on page 21.]

[48] CEA IT Security, "Miasm," https://miasm.re/, 2021, last accessed 17 November 2021. [Cited on page 21.]

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015. [Cited on page 21.]

[50] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, Rennes, France, Jun. 2015, pp. 31–54. [Cited on page 21.]

[51] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *USENIX Security Symposium*, 2020. [Cited on page 21.]

[52] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224. [Cited on pages 21 and 22.]

[53] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, dec 2008. [Online]. Available: https://doi.org/10.1145/1455518.1455522 [Cited on page 22.]

[54] T. C. Team, "clang - the clang c, c++, and objective-c compiler - clang 13 documentation," 2021, last accessed 15 November 2021. [Online]. Available: https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options [Cited on pages 22 and 30.]

[55] Incredibuild Software Ltd., "What is clang? - incredibuild," https://www.incredibuild.com/integrations/clang, 2021, last accessed 18 November 2021. [Cited on page 23.]

[56] The Clang Team, "Clang - features and goals," https://clang.llvm.org/features.html#libraryarch, 2021, last accessed 21 November 2021. [Cited on page 24.]

[57] ——, "Clang static analyzer," https://clang-analyzer.llvm.org/, 2021, last accessed 21 November 2021. [Cited on page 24.]

[58] Google, "google/sanitizers: Addresssanitizer, threadsanitizer, memorysanitizer," https://github.com/google/sanitizers, 2021, last accessed 21 November 2021. [Cited on page 24.]

[59] The KLEE Team, "Tutorial one - testing a small function," https://klee.github.io/tutorials/testing-function/#compiling-to-llvm-bitcode, 2021, last accessed 21 November 2021. [Cited on pages 24 and 38.]

[60] ——, "Options - overview of klee's main command-line options," https://klee.github.io/docs/options/, 2021, last accessed 21 November 2021. [Cited on pages 24 and 30.]

[61] Wikipedia contributors, "Peephole optimization — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Peephole_optimization&oldid=1052685619, 2021, [Online; accessed 18-November-2021]. [Cited on page 24.]

[62] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," 11 2017. [Cited on pages 25 and 30.]

[63] Vineet Nanda, "Mccabe's cyclomatic complexity: Calculate with flow graph (example)," https://www.tutorialspoint.com/mccabe-s-cyclomatic-complexity-calculate-with-flow-graph-example, 2021, last accessed 24 September 2022. [Cited on pages x, 26, and 27.]

[64] AFLplusplus, "Aflplusplus/readme.lto.md - afl-clang-lto - collision free instrumentation at link time," https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md, 2021, last accessed 29 November 2021. [Cited on page 30.]

[65] NumFOCUS, "pandas - python data analysis library," https://pandas.pydata.org/, 2022, last accessed 24 September 2022. [Cited on pages 31 and 41.]

[66] Tristan Ravitch, "travitch/whole-program-llvm: A wrapper script to build whole-program llvm bitcode files," https://github.com/travitch/whole-program-llvm, 2021, last accessed 29 November 2021. [Cited on page 33.]

[67] Free Software Foundation, Inc., "Coreutils - gnu core utilities," https://www.gnu.org/software/coreutils/, 2020, last accessed 29 September 2022. [Cited on page 33.]

[68] The KLEE Team, "Testing coreutils - tutorial on how to use klee to test gnu coreutils," https://klee.github.io/releases/docs/v1.3.0/tutorials/testing-coreutils/, 2016, last accessed 29 November 2021. [Cited on page 33.]

[69] Wikipedia, "Code coverage - wikipedia," https://en.wikipedia.org/wiki/Code_coverage, 2022, last accessed 24 September 2022. [Cited on page 33.]

[70] The KLEE Team, "Tools - overview of the auxiliary tools provided by klee," https://klee.github.io/docs/tools/, 2022, last accessed 28 September 2022. [Cited on page 34.]

[71] LLVM Project, "llvm-cov - emit coverage information - llvm 13 documentation," https://llvm.org/docs/CommandGuide/llvm-cov.html, 2021, last accessed 29 November 2021. [Cited on page 34.]

[72] P. Wägemann, T. Distler, P. Raffeck, and W. Schröder-Preikschat, "Towards code metrics for benchmarking timing analysis," in *Proceedings of the 37th Real-Time Systems Symposium Work-in-Progress Session (RTSS WiP '16)*, 2016, pp. 1–4. [Online]. Available: https://www4.cs.fau.de/Publications/2016/waegemann_16_rtss-wip.pdf [Cited on page 34.]

[73] HashiCorp, "Packer by hashicorp," https://www.packer.io/, 2021, last accessed 17 December 2021. [Cited on page 34.]

[74] ——, "Discover vagrant boxes - vagrant cloud," https://app.vagrantup.com/boxes/search, 2021, last accessed 17 December 2021. [Cited on page 34.]

[75] Oracle, "Oracle vm virtualbox," https://www.virtualbox.org/, 2022, last accessed 29 September 2022. [Cited on page 34.]

[76] Stefan Bucur, "[klee-dev] klee crash question," http://mailman.ic.ac.uk/pipermail/klee-dev/2012-December/000005.html, 2012, last accessed 24 September 2022. [Cited on pages 38 and 49.]

[77] The Matplotlib development team, "Matplotlib: Visualization with python," https://matplotlib.org/, 2022, last accessed 24 September 2022. [Cited on page 41.]

[78] NumPy, "Numpy," https://numpy.org/, 2022, last accessed 24 September 2022. [Cited on page 41.]

[79] OVH, "Ovhcloud vps - your virtual private server in the cloud — ovhcloud," https://www.ovhcloud.com/en/vps/, 2022, last accessed 17 August 2022. [Cited on page 44.]

[80] Trail of Bits, "trailofbits/cb-multios: Darpa challenges sets for linux, windows, and macos," https://github.com/trailofbits/cb-multios, 2021, last accessed 26 November 2021. [Cited on page 46.]

[81] feliam, "The symbolic maze!" https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/, 2010, last accessed 26 November 2021. [Cited on page 46.]

[82] The KLEE Team, "Tutorial two - testing a simple regular expression library," https://klee.github.io/tutorials/testing-regex/, 2021, last accessed 26 November 2021. [Cited on page 46.]

[83] B. Caswell, "Challenge information - kprca_00069," https://www.lungetech.com/cgc-corpus/challenges/KPRCA_00069/, 2017, last accessed 24 September 2022. [Cited on page 46.]

[84] ——, "Qualifier challenge - nrfin_00024," https://www.lungetech.com/cgc-corpus/challenges/NRFIN_00024/, 2017, last accessed 24 September 2022. [Cited on page 47.]

[85] Christian Collberg, "Random function," https://tigress.wtf/randomFuns.html, 2021, last accessed 26 November 2021. [Cited on page 47.]

[86] Avast Software, "avast/retdec: Retdec is a retargetable machine-code decompiler based on llvm." https://github.com/avast/retdec, 2022, last accessed 29 September 2022. [Cited on page 81.]

[87] Trail of Bits, "lifting-bits/mcsema: Framework for lifting x86, amd64, aarch64, sparc32, and sparc64 program binaries to llvm bitcode," https://github.com/lifting-bits/mcsema, 2022, last accessed 29 September 2022. [Cited on page 81.]

[88] rev.ng, "revng/revng: the core repository of the rev.ng project," https://github.com/revng/revng, 2022, last accessed 29 September 2022. [Cited on page 81.]

[89] Thomas Dullien, "Reverse engineering," https://docs.google.com/presentation/d/1ljVUiXVi2PfEdolGXr7Wpepj0x2RxaOo9rzMKWXebG4/edit, 2018, last accessed 29 September 2022. [Cited on page 81.]