Information Security

Obfuscation: Automatic Benchmark Compilation Dynamic Evaluation Framework

Diploma Thesis

For attainment of the academic degree of

Diplom-Ingenieur/in

submitted by

Florian LIENHART

is191840

in the

University Course Information Security at St. Pölten University of Applied Sciences

The interior of this work has been composed in LaTeX.

Supervision

Advisor: Dipl.-Ing. Patrick Kochberger, BSc

Assistance: -

St. Pölten, August 9, 2021

(Signature author) (Signature advisor)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Ein wichtiger Teil der Softwareentwicklung ist die Verschleierung von Software. Sei es, um das Verhalten von Schadsoftware zu verschleiern oder um das geistige Eigentum zu schützen und AnalystInnen das Reverse Engineering der Programmlogik zu erschweren.
Aus diesem Grund hat sich in den letzten Jahren ein Katz-und-Maus-Spiel zwischen AnalystInnen und Obfuskationsmethoden, zum Schutz von Software, entwickelt. So gibt es eine Vielzahl unterschiedlicher Obfuskations-Techniken mit unterschiedlichen Zielen, was sie zu schützen versuchen, wie zum Beispiel schutz vor automatischer Analyse von Software. In dieser Arbeit wird untersucht, wie praktikabel diese Obfuskationsmethoden sind und wie sie die daraus resultierende ausführbaren Dateien beeinflussen.
Das Programm Obfuscation abcdef wurde im Zuge dieser Arbeit entwickelt, um eine Möglichkeit zu etablieren, Obfuskations-Techniken für deren bestimmten Verwendungszweck auf einfache Weise zu prüfen, ob eine Obfuskations-Technik geeignet ist und die endgültige ausführbare Datei praktikabel ist und sich wie beabsichtigt verhält. Das Framework ist in der Lage, die Laufzeit, die CPU-Nutzung, die Nutzung des virtuellen Speichers, die Entropie von Dateien, den Mime-Typ und die Dateigröße von ausführbaren Dateien zu messen. Außerdem können verschiedene Compiler ausgewählt werden um die Leistung verschiedener Binärdateien in Bezug auf Verschleierung, Optimierung Compilerunabhängig zu vergleichen.

Abstract

A big part of software engineering is software obfuscation. Whether to disguise the behaviour of malware or protect the intellectual protperty and complicate the analysis process in order to comprehend the logic of the program.

For this reason, a game of cat and mouse has developed over the years between analysts and obfuscation methods to protect software. Thus there exist a lot of different obfuscation methods with different objectives what they try to protect, for example automatic analysis of software. In this thesis the practicability of these obfuscation methods is investigated and their influence on the resulting executbale files.

The Obfuscation abcdef was developed alongside this thesis to provide a way to evaluate obfusaction techniques for their specific use-cases and to test the resulting executable on practicability and if it behaves as intended. It is capable of measuring the runtime, Central processing unit (CPU) usage, virtual memory usage, entropy of files, mime type and file size of executable files. Additionally different compilers can be selected to perform compiler-independent comparison of different binaries in terms of obfuscation and optimization.

Contents

1 Introduction . . . 1

1.1 Thesis Outline . . . 2

1.2 Research Question . . . 2

2 Prerequisits . . . 3

2.1 Static analysis . . . 3

2.1.1 Manual static analysis . . . 3

2.1.2 Automated static analysis . . . 3

2.2 Dynamic analysis . . . 4

2.2.1 Instrumentation based dynamic analysis . . . 4

2.2.2 VM profiling based dynamic analysis . . . 4

2.2.3 Aspect oriented programming . . . 4

2.3 Obfuscation . . . 4

2.3.1 Reverse Engineering . . . 5

2.3.2 Classification of obfuscation techniques . . . 5

2.3.3 Data Obfuscation . . . 5

2.3.4 Static code rewriting . . . 6

2.3.5 Dynamic code rewriting . . . 8

2.3.6 Defeating obfuscation . . . 10

2.4 Malware . . . 11

2.4.1 Polymorphism . . . 11

2.4.2 Metamorphism . . . 12

2.5 Digital rigths management (DRM) . . . 12

2.6 Compiler . . . 12

2.6.1 Compilation Steps . . . 13

2.6.2 Compiler Optimization . . . 13
2.6.3 Compiler Testing . . . 13
2.7 Benchmark . . . 15
2.7.1 Creating Benchmarks . . . 15
2.7.2 Benchmark Suites . . . 15
2.7.3 Obfuscation benchmarking . . . 17
2.8 Docker . . . 17
2.8.1 Container compared to other types of virtualisation . . . 17

3 Related Work . . . 19
3.1 Symbolic Execution . . . 19
3.2 Tools . . . 20
3.3 Tigress . . . 20
3.4 Compiler . . . 20
3.5 Benchmarks . . . 21
3.6 Test programs . . . 21
3.7 Sandbox . . . 22
3.8 Malware Repositories . . . 22

4 Framework . . . 23
4.1 Overview . . . 23
4.1.1 Visual representation of the framework . . . 25
4.2 Dependency Check . . . 27
4.3 Structure of the folders . . . 27
4.4 Style check of the scripts . . . 28
4.5 Compile . . . 28
4.5.1 Compile Scripts . . . 28
4.6 Test cases . . . 29
4.7 Compare return values . . . 30
4.8 Measure Performance of the Programs . . . 31
4.8.1 Module measure . . . 32

5 Benchmark . . . 33

5.1 Sample set 33
5.2 General benchmark setup 34
5.3 Correct runs of test programs 34
5.4 Measurements of entropy, file size and runtime 35
5.4.1 Runtime of the programs 36
5.4.2 File size of the programs 38
5.4.3 Entropy of the programs 40

6 Conclusion 41
6.1 Future Work 42

List of Figures 43

List of Tables 44

Glossary 47

Bibliography 51

1 Introduction

According to the Kaspersky security bulletin 2020-2021 [1] 70% of the computer connected to the internet in the EU experienced at least one malware attack and over 56000 unique users in the EU were attacked by ransomware. The number of malware derivates is growing steadily, one of the reasons is the redesign regarding flexible working places. Another reason is, that obfuscation is often used in malware development to hide the malware from Antivirus or Endpoint Detection and Response (EDR) solutions. Due to obfuscation it is possible to modify malware that it is semantically the same but always appears different by execution, which raises the effort to detect the fraudulent behaviour of the malware.

Obfuscation is also used to protect the intellectual property of companies like software solutions and computer games. According to DataProts Piracy Statistics from 2021 [2] almost 20% of computer users confessed to having pirated software at least once. Between 2015 and 2017 the software industry lost over $46 billion to piracy. Thus there is a constant battle between the software industry and cyber criminals. With constant increasing protection mechanisms and a fast paced industry the performance of the software should not suffer from its protection, especially if some of these protection mechanisms are combined, it can be relevant for the performance and the semantic of the software. Thus it is important that software manufacturer are able to compare different techniques and choose the best suitable and performant obfuscation technique to protect their software.

Most of the obfuscation research is done with small dummy programs where the traceability/reproducibility is easy and researcher can understand every detail oft the program. For research purpose it is often only necessary to proof if and how obfuscation techniques work. In such a research case it would unnecessarily bloat the work if bigger software is used. However obfuscation techniques are effecting the performance and some of them also the semantics of the test software which is often not considered. Thus this thesis was created to provide a framework which uses different compiler and obfuscation techniques to aide with the analysis of bigger software and show the difference between software of various sizes. The framework

named obfuscation abcdef is also capable of measuring the performance statically and dynamically to help researchers compare compiler and obfuscation methods to show if this techniques are practicable for their use cases. The goal was to provide a central framework which consists of various existing obfuscation tools and compiler to provide a uniform workflow in which those tools get executed automatically to make the output comparable.

1.1 Thesis Outline

This document is organized in several parts and it gives an overview of obfuscation techniques and a detailed explanation of the framework developed alongside this paper. It is possible to perform static as well as dynamic analyses with the framework. Furthermore this work demonstrates how different obfuscation techniques and compiler options affect the behaviour of test programs dependent on their file size and the number of arguments they process. This thesis focueses on the benchmarks like memory usage, runtime, filesize and entropy of these compiled binaries

In chapter 1 the current problem with obfuscation and malware is introduced. The chapter 2 describes the basics to understand the terms and techniques to use the framework and how the obfuscation techniques work. It also describes terms used to measure the programs which originate from the framework. Related work and in what way this thesis differs from them is described in chapter 3. chapter 4 describes the obfuscation abcdef and details about its inner working. This chapter is mandatory for these users who want to modify or contribute to the framework.
In chapter 5 the details about the measurement and the performance difference between big and small test programs is discussed and chapter 6 concludes the topic with the outcome of the benchmark measurements and future work.

1.2 Research Question

The main focus of this paper is to show how different obfuscation techniques and optimization level effect the performance of programs by letting known algorithms work with different size of input data and to offer a framework to the community, which is able to evaluate their techniques and algorithms in use. Detailed attention was paid to the runtime, file size and entropy of the compiled test programs.

2 Prerequisits

In this section the technical background, details and concepts of obfuscation are explained. To fully understand the framework, developed alongside of this thesis, a basic understanding of benchmarking is required and will get elaborated in this chapter as well. The fundamentals of compilers are also necessary for this thesis and get explained in this chapter. This chapter will end with an overview of Docker [3].

2.1 Static analysis

By static code analysis the software is not executed, thus the name static analysis. This analysis method is the opposite of dynamic analysis explained in section 2.2. As it is not safe to run arbitrary software, static analysis is often the preferred way to start analysis of malware, because the impact of the malware samples is unknown and so it is safer to start with static analysis, where the programs do not get executed. Static analysis can be performed either manually or automatically. [4]

2.1.1 Manual static analysis

Manual static analysis is very time consuming, because a human has to analyse the software without running it. Not only the software code can be reviewed, it is also possible to extract data from the executable file or check hashes if someone else already analysed the same software [5].

2.1.2 Automated static analysis

Automated static analysis is most of the time the preferred way due to its speed. Tools are also capable of evaluating programs much more frequently [4]. It is not possible to analyse software only with automatic static analysis methods because this tools rely on pattern matching and can not determine the exact logic behind. Tools can also produce false negatives or false positives which would falsify the analysis results.

2.2 Dynamic analysis

Dynamic analysis is a practicable technique to analyse software. It analyses the program while it gets executed. The advantage of dynamical analysis is the software is inspected while execution, thus the software reveals more information like run-time, polymorphism and the correlations between threads and processes. There are different dynamic analysis techniques which will get discussed in the next chapters. [6]

2.2.1 Instrumentation based dynamic analysis

This technique inserts code at any stage of the compilation process. This process is called instrumentation and is used to simplify the process of analysing the behaviour at runtime. This code can be inserted to monitor the performance or conduct other analysis methods [6].

2.2.2 VM profiling based dynamic analysis

Via debugging and profiling inside the Virtual Machine (VM) the dynamic analysis is performed. The profiler gives detailed information about the inner operation of a program. The heavy lifting is carried out by the profiler which only needs an interface on the VM. Technique benchmarks of the heap, the memory and other run-time specific operations can get examined with this analysis method [6].

2.2.3 Aspect oriented programming

With Aspect oriented programming (AOP) the code for analysis must not get injected like by instrumentation based dynamic analysis, furthermore the instrumentation part is a built-in feature of the programming language itself. Programming languages like C++ and Java have their own AOP extensions. Thus it is easier for developer to inject their profiles into existing applications. This technique is not relevant for this thesis and got inserted for the sake of completeness [6].

2.3 Obfuscation

Some software needs protection against intellectual property theft, or to hide the functionality of itself. Thus several obfuscation techniques got developed. The idea behind obfuscation is to make the recovery of the internal software logic difficult. Most of the time this is achieved by introducing redundancies into the original program. The semantics of the software must be the same as in the original program. The software is cloaked by using opaque constants, which are add blocks of junk code that will never get executed. Another

method is to confuse the static analyzer by using one-way functions. Regardless which obfuscation approach is applied, all of them have different vulnerabilities. Opaque predicates can be defeated by *symbolic execution*. Every obfuscation technique can be circumvented with enough time, determination, effort and skill. [7]

2.3.1 Reverse Engineering

Most of the times software is distributed in binary form, which is hard to read and understand for humans. But with the right tools and techniques it is possible to analyze the code. This process is called reverse engineering, it aims to recover higher-level representations (e.g. assembly code or pseudo code) of the software to analyze its nature [8].

2.3.2 Classification of obfuscation techniques

On the one hand obfuscation transformation is classified according to the data or information it protects inside the software. The simpler ones target the lexical structure of the software like formatting the source-code or change the names of variables used. On the other hand obfuscation techniques are categorised on the basis of the operation they perform on the data or information. These kind of techniques manipulate the aggregation of control or the data in use. Most of the time the logic, which got introduced by the programmer, gets broken up or new constructs get added by inserting bogus data. The positional order of data also gets scrambled to cloak the relation between two or more objects located next to each other as there could be useful information for a reverse engineer and for the semantics of a program it is not mandatory in which order these objects get declared. [9]

2.3.3 Data Obfuscation

This category of obfuscation techniques modify how data is stored in software to hide it and make analysis more expensive. Since the data is in modified form on the hard disk, the software must convert the data to its original state at runtime. Some of these techniques are described by Collberg et al. [10].

Change data sequence

To make it harder for analysts to identify variables they can get split into several pieces. Two functions are involved in this process. The first one obfuscates the data, whereas the second one obtains the original value

at runtime. Boolean variables can be split into multiple Boolean variables. This technique can also be used for obfuscating integer and string variables. Multiple arrays can either be merged into one array, folding arrays by increasing the number of dimensions or decreasing the number of dimensions. One array can get split into multiple subarrays to hide its coherence.
Another fundamental obfuscation technique is to reorder components of data. Most of the time are logically related data structures physically close inside the binary. This relation is broken up to make the data structures harder to understand for analysts. [10]

Encoding

Strings and other data often provide useful information for analysts which can speed up the reverse engineering process. This process also uses two functions, one to encode static data and the other one to dynamically decode the data. Zhou et al. [11] introduced a variant of data encoding with mixed Boolean-Arithmetic.

Conversion of static data to procedures

In this obfuscation method static data gets replaced with a function. This function transforms the data back to its original form at runtime. This method can be used with static strings to not leak information about the binary or its purpose for static analysis.

2.3.4 Static code rewriting

By static rewriting the program gets modified during compilation. This chapter uses the same approach as subsection 2.3.3. The only difference between static code rewriting and data obfuscation is the target of the obfuscation, which are binary code obfuscation and data obfuscation respectively.

Instruction replacement

Many instructions can be replaced by one or more semantically equivalent instructions. It preserves the semantic of the program. For example, instead of adding 3 to the value in *rax* register (*add rax, 3*) it is possible to increment the *rax* register three times (*inc rax; inc rax; inc rax*). It is possible to write shellcode [12] where all instructions are in the range of American Standard Code for Information Interchange (ASCII) characters and thus can form grammatically correct English sentences [13].

Opaque predicates

Opaque predicates is a Boolean-valued function where the value is known to the obfuscator at obfuscation time. On the other hand it is hard to determine the outcome for the deobfuscator. Additional code is added if this obfuscation technique is used. Thus they have to be cheap and resistant against deobfuscation attacks. If opaque predicates are created it is very hard for deobfuscation tools to crack. This technique is hard to solve for static analysis tools but if the program ins analysed in run-time it is possible to circumvent the obfuscation method. [14] Opaque predicates have a static behavior and can be defeated by running the program. To prevent this weakness Palsberg et al. [15] introduced *dynamic opaque predicates* where opaque predicates evaluate to different results between different runs of the software.

Inserting dead code

Code can be rewritten by inserting dead or irrelevant code, code which gets never executed is inserted in the binary in the obfuscation step. This pieces of code cannot get reached and will never get executed. This can make analysis of a program more time consuming because there is more code to analyze. Especially in combination with section 2.3.4 the insertion of dead or irrelevant code makes the analysis of a binary hard. [9] Irrelevant code insertion [16] is a concept which places instructions like *xchg rax, rax* inside the binary. These instruction have no effect on the semantic of the program but can make analysis of the software more complex. The difference between dead and irrelevant code is, that the irrelevant code can actually be reached and executed.

Loop transformation

Loop transformation improves the performance and space use of software [17]. Whereas some of them increase the complexity of loops and therefore are well suited for obfuscation. Some of the purpose of loop transformation is to use the cache more efficient or split a loop into more loops to increase the speed of the software. The same techniques can be applied to introduce more complexity to the software and make analysis more elaborate.

Function splitting

On one hand the control flow can be split into two or more paths which look different from an analyst's perspective but semantically these paths are equivalent. Whereas on the other hand it is possible to merge more functions into one, this function contains the parameters of both functions. Some instructions reoccur

more often than others. If functions are similar it is possible to use the equivalent instructions in both functions. This concept was developed by Jacob et al. [18].

Control flow obfuscation

As the name intends this method obfuscates the control flow graph. In this technique a dispatcher is used which determines which instructions gets executed next. This is determined on the basis of opaque variables [19]. Control flow graph flattening can be implemented in several ways. It can modify the *CALL* instructions [20] or by using traps and signal handler which redirects the control flow to where it is intended to be [21].
Other static obfuscation techniques are:

1. Reordering
2. Aliasing
3. Name scrambling
4. Parallelize Code
5. Remove library calls
6. Breaking relations

[22]

2.3.5 Dynamic code rewriting

Dynamic algorithms transform the program at runtime instead of obfuscation time like the static methods described in subsection 2.3.4. It is possible to turn the static obfuscation methods into dynamic methods by including the obfuscation in the software itself [23]. In a nutshell examples of this category of obfuscation techniques execute different code than the statically visible code which is unveiled by static analysis.

Packing and Encryption

Packers are programs which take an executable file as input and perform compression on it to obfuscate the content of the executable file. This stub, which is a section of code, is stored in a new executable file with a second stub, which performs the decompression on the previously obfuscated executable inside the new executable. Thus the decompressed file is only available in memory and never gets stored on the hard disk and therefore an analysis of the initial executable is not possible with static analysis techniques. Cryptors are similar to packers, whereas packers compress the input executable, cryptors obfuscate the input file by encrypting it. The Cryptor runs a decryption routine at runtime to deobfuscate the encrypted content of the

executable and afterwards jumps to the decrypted stub and executes it [24]. There are several open source as well as commercial packers available. These tools are used by malware authors to achieve polymorphism. Polymorphism is a technique to create randomly created copies of files, which are in fact a mutation and thus harder to detect by automatic analysis tools [25].

Dynamic code modification

By dynamic code modification template functions are placed in the executable file. At runtime these template function get modified to contain the original function which gets executed in memory. Therefore static analysis tools are not capable of analysing this obfuscation technique [9]. A variation of this technique has been introduced by Kanzaki et al. [26] where code containing errors gets patched at runtime to run correctly.

Environmental requirements

Statically generated keys for encryption have to get delivered with the binary to decrypt its content. This is the weakness of the obfuscation technique discussed in section 2.3.5. With the key it is possible for an analyst to decrypt the data or code inside the executable file. If the key is otherwise created dynamically via environmental dependencies where the key gets generated from several classes of environmental data the key is not delivered within the binary. Only if all environmental criteria are met the information can be decrypted, otherwise no information is revealed by the executable file [27]. As introduced by Sharif et al. [28] it is possible to to trigger a different behaviour of malware if certain conditions are triggered. This *conditional code obfuscation* must fulfill a specific branch condition which gets triggered by an one-way hash function. If this condition is met a specific code gets executed which encrypts the code with the previously derived key from the satisfied condition.

Virtualisation based obfuscation

Virtualisation based obfuscation uses a custom VM interpreter. The target program's functionality gets converted into byte code of the VM [29] [30]. In this setup the target software gets executed in an controlled environment inside the VM and the instructions are only known to the VM, which makes analysis of this obfuscation technique expensive. It is possible to implement polymorphism using virtual obfuscation, by using a different VM for each part of the program [31].

Anti analysis

Anti analysis obfuscation techniques try to prevent analysis by disrupting debugger and disassembler. Debugger can be detected by using timing analysis, detecting breakpoints from debugger and their impact on the system or using Operating System (OS) Application Programming Interface (API). By using impossible disassembly or jump targets with the same destination it is possible to confuse a disassembler. The result is that the disassembler generated code is distorted. It is also possible to insert constant conditions to jump to a location inside an assembly instruction because the branch which evaluates to false will most of the time be processed first by a disassembler. If the condition always evaluates to true the disassembler interprets the code in a false way [5].

2.3.6 Defeating obfuscation

As obfuscation techniques evolved the opposing side also discovered new techniques to defeat obfuscation techniques. One technique has its roots in fuzzing and is called symbolic execution. It can be used for reverse engineering and circumvent obfuscation techniques.

Symbolic Execution

Symbolic execution is used in software testing to detect as many paths in the program's control-flow-graph as possible. For each path discovered an individual set of input gets generated to reach the part of the code and check if the software is prone to errors or uncaught exception. The generation of test cases is one major strength of symbolic execution because it provides a specific test case which triggers the bug.
Symbolic execution uses *symbolic values* instead of specific data values as input. The computed output by a program is expressed as a function or the symbolic input value. Executable paths in the control flow are a sequence of *true* and *false* branches. Program paths can then get visualised in a tree representation and the goal of symbolic execution is to traverse all possible knots of the tree and explore all leaves of the tree [32].

Deobfuscation with Dynamic symbolic execution (DSE)

Dynamic symbolic execution (DSE) is when symbolic execution cannot handle some concepts of the programming language like self-modification, and some program paths need to get discovered by a dynamic approach and get added to the control-flow graph dynamically. This technique is a good candidate to defeat simple obfuscation techniques, like packing or self-modification.
Symbolic deobfuscation is based on DSE to defeat packed software or reconstruct the control flow graph of

software to make deobfuscation easier. Through the symbolic values it is possible to simplify the control-flow-graph of obfuscated software.
Dynamic symbolic execution also has its weakness. It is possible to use techniques in order to make the analysis of DSE slow or cause it to fail. According to [33] anti DSE techniques are categorised into three categories:
Complex constraints: Path constrains can be designed so they are impossible to solve for symbolic execution. With non-linear operations it is possible to prevent DSE from discovering all paths.
Path divergence: It is possible to defeat DSE obfuscation if the symbolic execution engine does not compute all path constraints correctly. This can lead that paths are missed or wrong paths are taken.
Path explosion: Every path constraint has to be calculated and stored in memory in order to explore them. If the number of possible paths is too large it is not possible to explore them in a realistic amount of time. [34]

Deobfuscation with LLVM

With LLVM it is possible to lift binaries into the *LLVM-IR* which is the internal language of LLVM. If a software is in the *LLVM-IR* the instructions get converted to a SMT formula to use a SMT solver following a analysis of the program with symbolic execution with the program KLEE [35]. [36]

2.4 Malware

Malware is a combination of the word malicious and software which is an accurate description it is basically malicious unwanted software. This kind of software causes damage or harms systems intentionally, it is furthermore capable of replicating, propagating or executing of itself. After infecting computer systems malware endangers the confidentiality and integrity of these systems. To avoid detection malware often uses polymorphic or metamorphic techniques [37].

2.4.1 Polymorphism

A polymorphic malware is able to mutate itself when it infects new host systems which makes the identification for detection systems difficult. These virus change their decryption or decompressing routines, creating many instances of itself in different form.
Polymorphic engines are a software which modifies other software by creating a different version of the initial program by changing the code. The semantic behaviour of the initial program is equal to the newly created one. This behaviour is often used to avoid detection from antivirus software and is explained in

section 2.3.5 [38].

2.4.2 Metamorphism

Metamorphic code is capable of rewriting its own code with each infection similar to polymorphism but metamorphic code does not modify the packing or encryption routine it changes its own code and reprograms itself. This is a technique to hide the software from detection software and make analysis of the program harder. Metamorphic obfuscation techniques are discussed in subsection 2.3.5 [38].

2.5 Digital rigths management (DRM)

Digital rigths management (DRM) enables a secure exchange method of digital products, like copyright-protected video, text, music over electronic media or the internet. This mechanism is for the content owners to distribute the material securely and authorize users. [39]
DRM is a heterogeneous terminology, which most of the time means protection mechanisms like copy-protection measures, access control, copyright protection etc. Unfortunately almost every DRM implementation was circumvented. Thus the entertainment industry forced a law behind DRM to prohibit the removal of these protections [40]. Obfuscation techniques such as encryption and packing and others are the foundation of DRM these techniques were explained in detail in section 2.3.

2.6 Compiler

Every Computer system runs software which is written in programming languages. Programming languages describe operations a system should perform but between executing a software, the program needs to get translated into a form that a computer is able to understand. The software that performs that translation is called *compiler*. For most programming languages one or more compiler exist which can translate source-code, written in the specified programming language, to an executable file can get executed by a computer. There is another way to execute source-code on a computer. An *interpreter* is a language processor which does not produce a program as output it rather interprets the source-code directly and executes the instructions [41].

2.6.1 Compilation Steps

A compiler is actually a set of programs which have all different tasks to create an executable program.
Preprocessor: If a preprocessor exists, it is responsible for gathering all source files when the source-code is split into multiple source files. It is also possible to define macros in source files, these macros get expanded by the preprocessor.
Compile: This step is called compile step, it produces *assembly* code which is easier to debug and easier to manage. Sometimes the name can cause confusion because it is the same as the whole concept of translating source-code into a program.
Assembler: An assembler produces position independent code as output.
Linker: The linker joins the object files and libraries together because large programs are often compiled in pieces. Furthermore the linker resolves external memory addresses, which reside in other files.
Loader: In the end the loader loads the files into memory and resolves the entry point and passes the execution to the program so it can get executed by the OS [41] [42].

2.6.2 Compiler Optimization

Compiler optimization is a process which modifies the output of a compiler (executable file) so that some of the attributes are minimized or the efficiency and speed is maximized. Some of them could be time minimization, memory minimization or work as power efficient as possible. The latter one is especially relevant for portable computers like mobile phones.
Optimizing compiler output aims to produce the best suited machine code from source-code files, where best suited depends on the aforementioned requirements and on the application itself. Furthermore some of the optimization techniques are mutually exclusive, it is not often possible to make the code faster, more memory efficient and minimize the power consumption.
Algorithms for optimization differ in their complexity as well. A optimization method can be removing a simple loop constant or more complex one such as removing a complete subroutine. Changes due to optimization modify the logic the programmer initially wrote into a more efficient variation which can change the semantic of the program. [43]

2.6.3 Compiler Testing

It is commonly hard to evaluate the effectiveness of compiler because it has many dependencies like hardware, the OS, which optimization goal the program has etc. Even if these categories are the same it is hard

to determine on which measurement the quality should get measured. Benchmarks, Code Characteristics, compiler effectiveness, instruction-level parallelism only to name a few [44]. One challenge in compiler testing is the construction of test programs. Like in every other software test, test cases are a central point of compiler testing. There are three main challenges for constructing test programs. *Validity of the test programs:* Due to language constructs it is not trivial to generate test programs. If invalid test programs get generated the process is not useful for compiler testing. As described in subsection 2.6.1 a compilation consists of multiple steps and if the process is aborted in the initial steps the compiler is not tested completely. *Diversity of the test programs:* To get a high code coverage of the underlying compiler the test programs have to be diverse. As syntactical diverse programs trigger different paths in the compiler code. If more paths are covered more bugs can get revealed. By using various language constructs in the test programs the greater the chance to generate an invalid test program gets. *Specific requirements imposed by a testing method:* If for instance two compiler should get compared by inputting the same test program, the program should not contain undefined behaviour. Another test case is if compiler crashes are inspected, it is possible to have undfined behaviour in it. [45] [17]

What you see is not what you execute (WYSINWYX)

As described in section 2.6 programs are nowadays most of the time written in high-level programming languages and get translated to machine code. If the analysis of code is performed on source-code level it is possible that the resulting executable after compilation behaves different due to compiler optimization. For example if a password string is saved in memory via a pointer and the location where the pointer refers to gets overwritten with *NULL* Bytes, followed by a *free* of the pointer the compiler might optimize the code and omit the string modification, leaving the password in memory. This phenomenon is called What you see is not what you execute (WYSINWYX) and can lead to bugs in:

1. memory-layout details
2. register usage
3. execution order
4. optimization performance
5. artifacts of compiler bugs

Many security exploits rely on this behaviour and can be crucial in aspect of security of the systems where this programs are running. [46]

2.7 Benchmark

Software benchmarking is to compare performance metrics like cost, cycle time, productivity or quality to another software. Benchmarking provides a snapshot at a given moment and shows how the performance of a program is compared to the other test candidates. [47]
Software benchmarks make software comparable to other. This is mandatory due to competition on the market and some environments are time critical, thus the software which is used in those environments is time critical as well.

2.7.1 Creating Benchmarks

For performance testing usually benchmarks get created. Which means a workload that tests the system under real conditions. During this test, it is determined how the system will behave in practice.
During benchmarking one problem arises. How can be determined if the benchmark data is actual representative? Via monitoring it can be determined how the system was used earlier and most likely will be used in the future.
Additionally it must be considered if the benchmark should test a normal workload or a stress workload to push the system to its limits. The duration of the test is also a critical factor. Such test time frames can get from several seconds to more than 20 hours. Thus the benchmark is highly dependent on the environment and the system it examines. Hence there is not one way to do benchmarking. [48]

2.7.2 Benchmark Suites

The goal of benchmark suites is to provide a wide variety of different computation patterns with state of the art algorithms for different situations. The environment like multicore CPUs, power limits and several accelerators like Graphics processing unit (GPU), Field-programmable gate array (FPGA) and STI Cells has to be taken into account in benchmark suits. [49]

S2CBench

With 13 programs the S2CBench collection has the three main objectives to enable the comparison of High Level Synthesis (HLS) tools, test language support features, synthesis optimization techniques and performance and finally help researchers analyze and compare their own techniques.
Every program of the 13 programs in the S2CBench suite is designed to test a specific feature. All programs are categorized according to one of the following application domains:

1. Automotive and Industrial
2. Quick sort design sorts (qsort)
3. Sobel filter
4. Security
5. Advanced Encryption Standard (AES) cypher
6. kasumi
7. md5
8. Snow
9. Telecommunications
10. Adaptive differential pulse-code modulation
11. Fast Fourier Transform
12. Consumer
13. Fir
14. Decimation
15. Interpolation
16. Inverse discrete cosine transformation
17. Disparity

The design of the benchmark ranges from small single process designs like quick sort of FIR filter to larger multiprocessor designs like kasumi and disparity. [50]

Rodinia

The Rodinia benchmark suite mostly targets GPU and CPU. For selection of the benchmarks the Berkeley Dwarfs [51] are used. The Berkeley Dwarfs are defined at a high level of abstraction to allow reasoning about the program behaviour.
Features of the Rodinia suite are, the four included applications and five kernels. They are running parallelized for multicore CPU systems. Furthermore optimization techniques take advantage of the on-chip resources if the applications are run. The workloads consist of parallelism, data access patterns and data sharing characteristics. As in the C2Bench suite the Rodinia suite also covers many representative application domains for applications. Another feature is that the applications within one dwarf show different features. [49]

2.7.3 Obfuscation benchmarking

Benchmarking compares performance metrics of one program, like described in section 2.7, to another. Through obfuscation these metrics change depending on the obfuscation technique in use. Banescu et. al [52] examined how these obfuscation techniques can be defeated by symbolic execution. The goal of this research was to compare symbolic execution with programs and the obfuscated counterparts and which free obfuscation technique can stand attacks through symbolic execution.

2.8 Docker

Docker [3] is an open source platform. It is capable of running applications and aids in developing and distribution of software. Container are the instances in which the application built in docker with all its dependencies are packaged. Docker can interfere with other third-party instruments which makes it easy to deploy and mange docker containers. Applications get virtualised within container environments, this is done by adding an extra layer of deployment. Hence it is easy to test the code and deploy it to the production environment with the aide of the light weighed design of docker. [53]

2.8.1 Container compared to other types of virtualisation

The difference between containers and hypervisor-based virtualization is that that the former performs virtualization on operating system level, whereas the latter solution virtualizes at hardware level. The outcome is the same but there exist some major differences.
On the one hand hypervisors come in two different flavours Type 1 and Type 2, where the first runs directly bare metal on hardware and the second introduces an addition layer within a guest OS. Representatives of the Type 1 group are open-source Xen and VMware ESX. Type 2 hypervisor are Oracle Virtual Box and Vmware Server.
Container on the other hand protect parts of the underlying OS and make them available to the container. If two container are run on the same host system it is not possible for them to recognize the other container with which they are sharing resources, because of the isolated processes, network layer etc. [54]

3 Related Work

Obfuscation modifies the program internal data and code, with the goal to hide its logic or protect the internal data of the program. One key element in modifying the program is to keep the unmodified and the obfuscated program semantically equivalent, which means the result of the program stays the same [55] [56].
Most of the research in obfuscation is done with small test programs or dummy programs to showcase the purpose of the technique. This small programs are often used because it is easier to understand the details of the program which is in fact not always possible in larger programs. Especially the practicability of the obfuscation techniques with larger programs is often not taken into account which is also an important factor for the evaluation of obfuscation techniques. In conjunction with this thesis a framework was created to test the ability of obfuscation techniques and compilers in creating programs.

Software Obfuscation is widely used in Malware and is still almost unexplored compared to the state-of-the-art code analysis. There is also a big difference between obfuscation techniques regarding the effectiveness of de-obfuscation techniques and code analysis tools. Therefore different defence strategies are more or less effective against various obfuscation techniques [22].

3.1 Symbolic Execution

Symbolic Execution is widely known in software testing [32]. This technique explores several program paths in a predefined time frame. It generates some input values for the explored paths and checks if any errors are present. Current analysis methods like DSE are able to defeat software obfuscation techniques. Dynamic analysis and the inference ability of static analysis provide a robust foundation for the success of symbolic deobfuscation. It is unclear how strong the protection mechanism are against DSE-based attacks [55]. One key goal of the framework provided in this thesis is to support the inspection process of different obfuscation methods with different input programs, and moreover to compare the performance of various obfuscation methods.

3.2 Tools

Klee [57] is capable of generating tests which achieve a high coverage on complex programs. It is a popular source-level DSE tool. Other binary-level DSE tools are *ANGR* [58] which provides a basis for all kinds of analysis types with static and dynamic obfuscation techniques. *Binsec* [59] is another tool which uses DSE to explore paths of programs and feed them in an automatic solver. The solution is used as a new test input to expand the amount of paths. Another tool is *fuzzball* [60] or *S2E* [61]. The latter is a platform which also uses symbolic execution and is used for prototyping custom analyses.

3.3 Tigress

Tigress [62] is a freely available state-of-the-art obfuscation tool capable of performing many standard obfuscation techniques on source-code. It allows a precise control over which obfuscation technique is used. Additionally it is possible to mix more than one obfuscation method [63]. This tool is included in the framework of this thesis and was used in the initial phase to perform the feasibility tests.

3.4 Compiler

Compilers are a crucial part of each system because every software running on a computer has been processed by a compiler or a compiler-like tool. Thus the importance of the correctness of compilers is a influential aspect in Information Technology (IT). Compiler testing [44] and performance is also a big part of the obfuscation abcdef which uses different compilers and obfuscation techniques to inspect the performance and feasibility of test programs. To perform a variety of analysis, like comparison of performance, it was mandatory to include different compilers [64]. It is possible to select different compiler or obfuscation techniques for programs.
A phenomenon called WYSINWYX [46] occurs when source-code programs get translated to machine-code programs and thereby a mismatch is created between what the programmer meant and what gets executed by the processor. This behaviour also infects the obfuscation abcdef, because through the various obfuscation techniques used it is possible that the semantic of the programs gets distorted. One approach to test compiler is the differential testing where at least two compiler get designed and implemented with the

same specification. The results of these compilers are compared to identify compiler bugs [45]. Another approach for testing compiler is *Metamorphic testing* [65]. It constructs *metamorphic relations* which effect the input of the compiler would change the output.

3.5 Benchmarks

A tremendous part of software engineering nowadays is Software Performance Engineering (SPE), which is about devolution software systems that meet certain performance requirements [66]. Benchmarking is an operation whose aim is the comparison of performance metrics which include cost, cycle time, productivity or quality to another [47]. Synthesizable SystemC Benchmark (S2cbench) [50] suite consists of 12 + 1 programs. Each program is designed to test a particular feature belonging to a category according to its application domain. The main objectives of the S2cbench suite are to enable the comparison of commercial HLS tools, test tools features classified as language support, tool performance and synthesis optimization techniques. The last objective is to help researchers compare and analyze their own techniques which is related to the obfuscation abcdef developed in this paper. Rodinia [49] is another benchmark suite. It focuses on programs running on accelerators such as GPUs or FPGAs. The application in this framework have been implemented for for GPUs and multicore CPUs using *CUDA* and *OpenMP*. The main focus of this thesis was to determine the correctness and performance of test programs. Additionally it was analyzed how different obfuscation techniques effect the performance of the test programs and if different size of test data processed by this programs effect the performance.

3.6 Test programs

Practical code obfuscation with benchmarks and how virtualisation based obfuscation is defeated by symbolic execution based analysis and deobfuscation methods is shown in the work from Banescu et. al. [67]. Along with this work a set of small programs was provided [68]. Benchmarking is a critical process in design due to the high requirements for performance. These benchmarks target different areas of computation like *integer* performance and *floating point* performance [69]. Benchmark examples were distributed in various categories [70]. Some of these formats are the *FSM format* which uses *KISS2* format or multi-level formats like *BLIF*. The obfuscation abcdef uses sort and hash algorithms which come out-of-the-box with the framework.

3.7 Sandbox

In state-of-the-art malware analysis, sandboxes [71] are indispensable. Sandbox software virtualises an environment to securely run programs in an controlled environment. Virtual machines, containerized environments and emulated environments are often referred to as sandboxes due to their similarities. A Sandbox is capable of analyzing the target program without the aid of human interference [72]. The framework created alongside this thesis is similar to sandbox software. However, the framework described in this thesis does not virtualise an environment, it is similar to a sandbox due to the fact, that both software solutions are able to process a high amount of test programs in a short time without much configuration changes in-between. Additionally the framework measures the test programs in terms of performance with dynamically and statically analysis.

3.8 Malware Repositories

Like the drebin dataset [73] the framework in this thesis contains several basic programs. Unlike the drebin dataset, wich contains over 5500 samples from 179 different malware families, the programs coming with the obfuscation abcdef consist of sort and hash programs. The samples inside the drebin dataset where gathered from 2010 to 2012 via a MobileSanbox project and comprise of Android malware. As previously discussed the framework is similar to malware sandboxes like VirusTotal [74] where about 70 different malware engines perform analysis on an uploaded file or cuckoo [75] which runs the test program in a sandbox environment and returns a report afterwards.

4 Framework

The goal of this thesis is to show, if programs can be obfuscated. Furthermore, if obfuscation methods influence the practicability of the software. For performance measurements and checks, if the software runs correctly after being obfuscated, a framework was created. The framework is explained detailed in the following chapters.

4.1 Overview

Generally the framework provides scripts to compile software with different compiler, like clang, gcc and tinycc and obfuscation tools. It creates a directory structure and allows to generate test cases for the programs as input. The scripts in the framework can get executed as standalone programs or combined with other scripts. The following steps are possible:

1. Check the dependencies of the framework section 4.2.
2. Check the structure which the framework needs to work section 4.3.
3. Check the style of the shell and *python3* [76] scripts section 4.4.
4. Compile the source files section 4.5.
5. Generate test cases for the programs section 4.6.
6. Compare the return values of the programs and their output section 4.7.
7. Measure the performance of the programs section 4.8.
8. Compare the results of the previous step subsection 4.8.1.

The framework consists of several shell scripts, which are responsible for starting and controlling the execution of the *python3* scripts. The shell scripts are parsing the names from the files and pass them over to the *python3* scripts as program arguments.

The actual work of checking the return values, the output, which is printed to *stout* and the performance measurements is done by *python3* scripts. For starting the framework or a modular use of it a *makefile* [77] is used. The Configuration for the framework is in the *config* directory. Every program gets compiled by searching its source code in the *src* directory. For naming conventions the framework parses the names

of the compiled programs from the source files. The absolute path in which the framework was installed has to be added in the *config/config.sh* file, this file is included in every shell script and contains important variables, like the install directory and the compiler version to use, and other functions. For each compilation a new directory is created in the *out* folder, this folder has the naming convention *run_timestamp*. The source codes of the programs, which get compiled and measured, are located in the *src* directory. For each program the source file has to be in a directory within the aforementioned *src* directory. The *compile* folder contains the scripts, which initiate the compiler [64] or tools, like tigress. The name of these scripts will be used to create the directories in the *out/run_timestamps* folder, where the compiled programs are located after compilation. Additionally the optimization level and other compiler flags [78] can be configured in the *compile* scripts. In the *compare* folder the python3 programs reside for the comparison of the return values and the output, the program to measure the performance and record the statistics of the program file itself and a python3 script, which compares the collected data.

4.1.1 Visual representation of the framework

For visual representation and a better understanding of how the framework, works the graphics Figure 4.1, Figure 4.2 and Figure 4.3 were created. On the left side in the graphics the inputs for the processes or script are shown. The processes, which are conducted by scripts, are represented in squares in the middle of the image. If the scripts have outputs, they are shown in diamond forms on the right side of the graphics.
Basically the framework is divided into 3 main steps, checking the content, prepare analyses and last perform the analyses.

In the first part the dependencies like *gcc*, *tigress* and *python3* are checked, if they are installed on the current system. The obfuscation abcdef also needs a predefined folder structure, where scripts are executed and outputs are copied to, the second step is performed in the *structure check* phase. When basic scripts of the framework are rewritten, the *style check* is responsible for checking the source-code for coding conventions in order to eliminate errors.

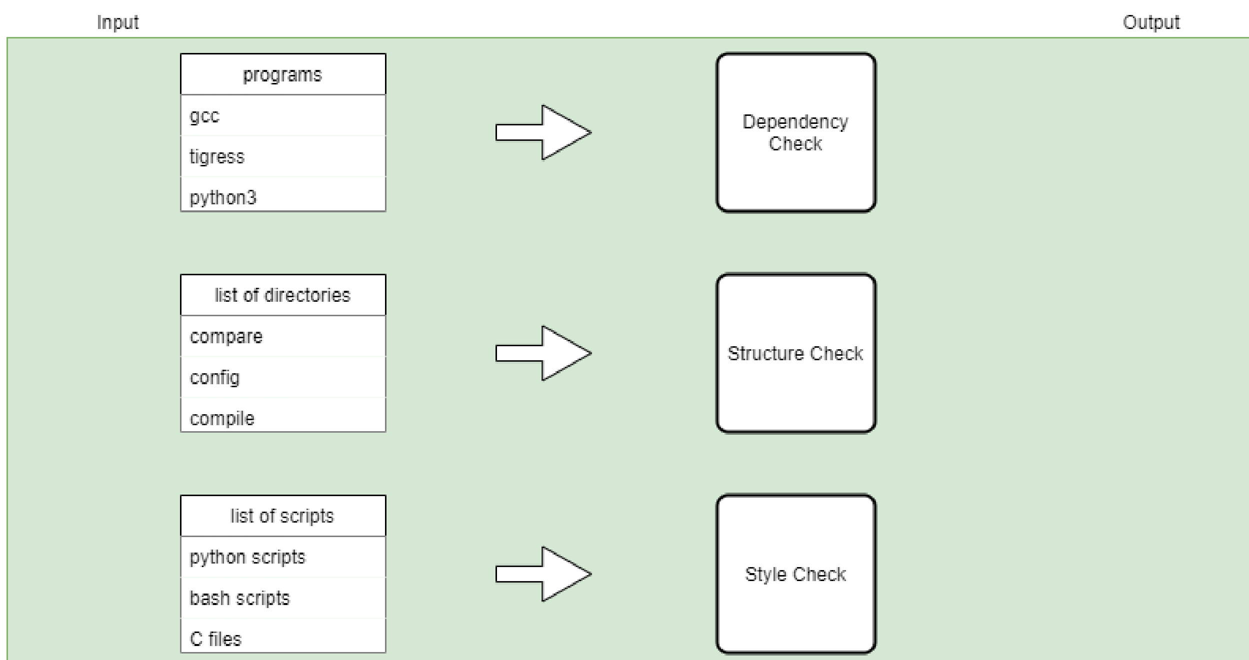

Figure 4.1: Part 1 Content checking: The general contents and scripts are checked in this part. On the left side of the image the inputs for the processes, which is represented in the forms in the middle, are shown.

The second part of the framework is responsible for the creation of binaries with the intended compiler and

tools. First the C programs get selected by the compile scripts. These contain instructions, how the C files get compiled and moreover the obfuscation technique that should be used to obfuscate the executable file, is defined in these files. After this step the compiled binaries, which are ready to get executed, are stored on the disk in the run folder. Every time the *compile source* step is executed a new run directory gets created, where all following steps store their output in. The currently used test programs require arguments to run, these are either provided via file input or command line arguments. These arguments get created in the *generate testcases* step and get passed to the program either via command line arguments or file input. The last step in the second phase is to run the programs a few times and check, if they provide the same return value and output. The results of this step are saved to disk in a Comma-Separated Values File (CSV-File) file in the *ret_compare* folder, which exists in every run directory.

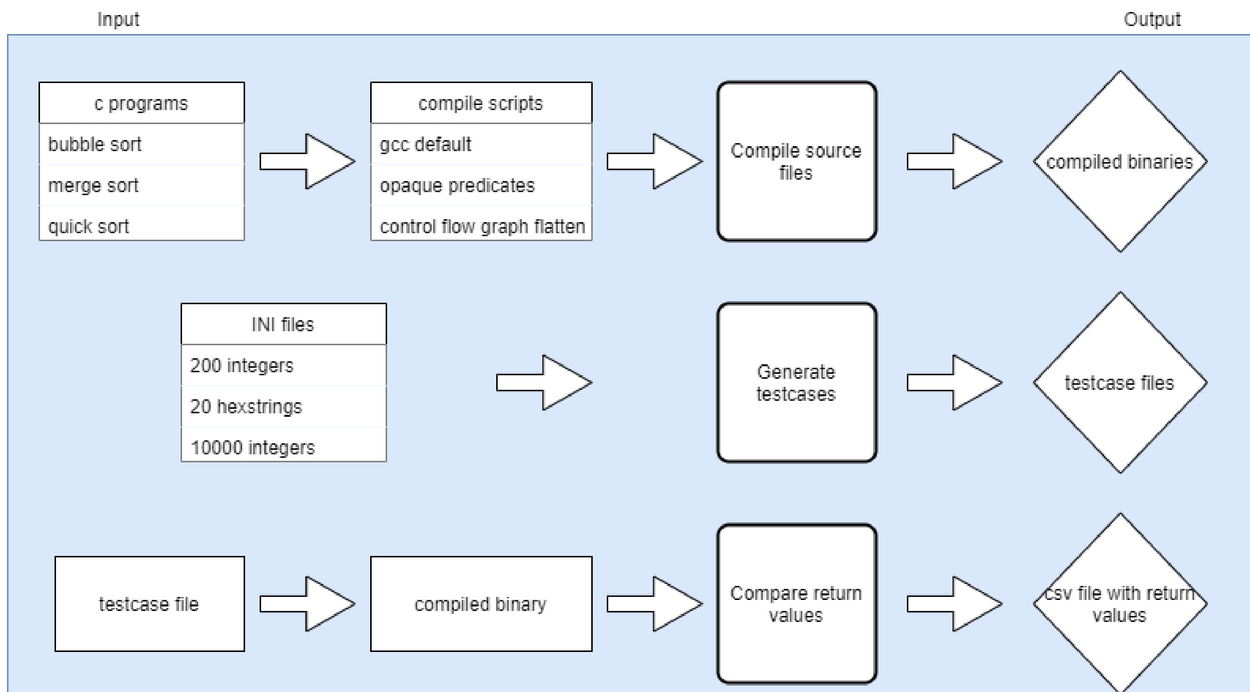

Figure 4.2: Second part of the framework: prepare for analyses. In this part the files get compiled and the inputs for the test programs are generated. Basic checks are performed, if the programs execute and exit gracefully. On the left side the inputs in form of rectangles are shown. The processes are square forms in the middle. The diamonds on the right hand side represent the output of the processes.

In the last phase, which is enclosed in the red square, the actual performance measurements are recorded. The previously compiled binaries are executed with the arguments from the test case files. After several

executions of one program the results are written to disk in a CSV-File or JavaScript Object Notation (JSON) file. The last step is to open those CSV-File files and compare the results and modify the output to only display the suitable results.

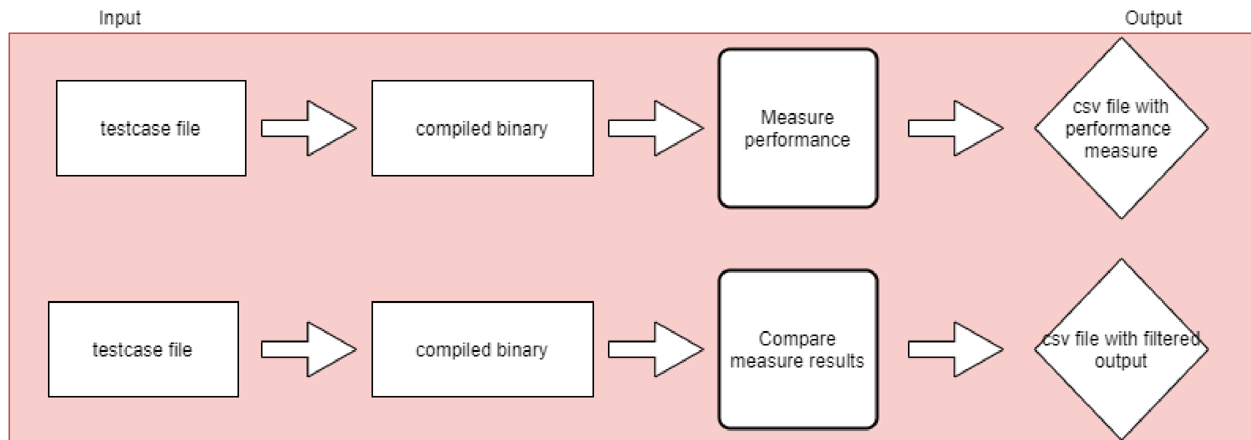

Figure 4.3: Part 3 of the framework: Performance measuring and analyses. On the left the input to the binaries and the binaries themselves are shown as rectangles. In the middle of the graphic the processes and on the right side the output in form of diamonds are represented.

4.2 Dependency Check

This part of the framework is responsible for checking, if the requirements are met to compile all programs and afterwards measure the results. In the current version the scripts checks, if *GNU Compiler Collection (GCC)* [79], *tigress*, *python3* and the following *python3* modules are installed:

1. Python magic
2. Pandas
3. Psutil
4. Capstone
5. Elftools

4.3 Structure of the folders

This script's purpose is to verify, if every directory exists in the root directory of the framework. These directories are the aforementioned *compare, compile, config, out and src* folders. If these checks succeeded the scripts verifies if the folders contain the scripts and source code files. In the *compile* folder the scripts

have to be executable, this gets also examined by the script.

4.4 Style check of the scripts

To avoid errors this script checks, if all programs and scripts in the framework are written properly. This step helps to prevent errors, which result from sloppy programming. It checks the python3 scripts, the bash scripts and the C source-code files for errors. This script does not have to be executed every time the framework is used, as all other steps, it should rather help researchers, who are modifying parts of the framework to avoid errors in the code and thus reduce the time troubleshooting.

4.5 Compile

After all checks exited without errors and the files are in their specific directory, the script, accountable for compiling the source files, gets executed. This script generates the main output directory, where the compiled programs will be located. Basically the directory name consists of two parts, the first part always starts with "run" run and the second part is a timestamp consisting of year, month, day, hour, minute and second respectively. As all programs need input either from a file or command line arguments a folder for these files is created in the run directory. For every compile script in the *compile* directory the name of this script is taken and truncated to only contain the obfuscation methods. With this string a new directory is created inside the run directory. Inside this folder all programs compiled with the same compiler and tool will be located. That is how the programs are grouped logically on base of their obfuscation and compile method. Every compile script gets executed for each program in the directories in the *src* folder. Additionally the compile time gets measured and the result gets written to a log file in the run directory, where the final program after compilation resides.

4.5.1 Compile Scripts

A central part of compile scripts is the *config.sh*, which resides in the *config* directory in the root directory of the framework. Inside of this script the terminal colors and warning color map can be configured. Moreover, the configuration for the compiler and tigress software is centralized in the *config.sh* script. This script determines the selection of compiler, its version and the obfuscation method, which should be used. For tigress is it mandatory to have the function names of the test programs. This information is crucial to select the parts of the code which should get obfuscated and inside the *config.sh* script a function is defined to

parse the functions out of the source code files from the test programs.

Which obfuscation methods are used is defined by the files in the *compile* directory. These files are symbolic links to centralised scripts in the same directory, which contain the configuration for the specific compiler or tool to use. The centralised scripts parse the options and check the exported variables from the *configs.sh* for the options, which should get used to compile the test programs. The symbolic link files refer to these centralised scripts. The naming convention of the symbolic links determine the options according to the name. Afterwards two examples are given:

- compile-gcc-oslatest-O0.sh
- compile-tigress-3_1-virtualize_gcc_oslatest_O3.sh

The first part of the name is the compile or obfuscation tool which should be used and the version of the software. If the test program gets obfuscated, the obfuscation method comes afterwards, as can be seen in the second entry in subsection 4.5.1. The last mandatory part is the optimization level of the compiler in use.

4.6 Test cases

Test cases can be created on their own without executing the whole workflow. If no command line arguments are provided, the script *make_testcases.sh* finds the name of the latest created directory with compiled programs in it and creates test cases for this run. In the second step the script checks, if a test case directory already exists. Otherwise it creates that folder. With the previously parsed directory name as argument the *python3* script *testcases.py* gets executed. This script generates the test cases for every program. An obligatory prerequisite for this script is the structure of the test cases. This information is parsed from the Initialization/Configuration File (INI-File). In the following listing a example entry can be seen:

```
[selectionsort_file]
testcase=fileinput
size=100
type=int
args=<in>
outfilename=selectionsort_file.outfile
```

The line *test case* defines the method, if the entry is *fileinput* the program needs a file name of a file as argument. Otherwise the test case entry defines the type of the input. In the current version the value of the entry can be of either *int*, *string* or *hexval*. The *size* argument is also mandatory because it defines the length

of the test case file. When the source program needs a file as input, the input type of the input is defined in the *type* variable. The *args* variable is used to define the ordering of the arguments. If the source program needs the input file and the output file as arguments, the ordering can be defined in that variable. When an output filename is provided, the name of this file has to be given in the *outfilename* variable.

The *testcase.py* script performs basic checks, if the INI-File is in the correct format, as described above. Furthermore, the two mandatory variables *testcase* and *size* must be present in the INI-File.
After the basic checks the script verifies, which test case types are present in the INI-File. Currently the following test case formats are supported: *int*, *string*, *hexval*, which is a string containing Hexadecimal values. These test case variations are supported to be either passed as command line arguments or as file input to the test programs. After the test case type is parsed, the according function in the modules_testclass.py is invoked to create the according files. If one or more test programs expect the same test case file format, for example a string with 100 Bytes, only one file is created. All test programs, which await the same test case, use the same file to be consistent during the test runs, where the performance and the correctness is revisited. All implemented test case classes work similarly. The length of the string or hexadecimal value is taken from the INI-File, then the random ASCII-string or the Hexadecimal string with the previously retrieved length is created and written to a file in the run directory in the *testcases* folder. For the *int* test case, the length is the amount of integer numbers written to the file, not the length of the file. There is no difference between the test case files, which are passed as arguments to the test programs and the test cases, which are passed as file input. The test case files get stored with the following name pattern:

```
"test class"_length_.args
```

4.7 Compare return values

To determine if programs are able to run and if they are able to run correctly and exit gracefully, the output and the return values of a program gets verified. The central script responsible to start this section of the framework is *start_compare_return.sh*. After the various compiled programs with different obfuscation methods get run and their return values, standard output and file output get written to a CSV-File [80].

In the first steps the script *start_compare_return.sh* checks, if an argument was provided to the script. If so it sets the current compiled directory to the value of the first argument. Otherwise the last directory, which got compiled, is taken for the comparison of the return values. After the initial step the obfuscation methods

used and the obfuscated programs get parsed from the run directory. The program names get passed to the python3 script *compare_return.py* as arguments. The python3 script executes the programs and saves the returned values inside a CSV-File. This is done for every executable in the run directory. Additionally only the programs with the same source file and different obfuscation or compiler are compared.

For all scripts in the *compare* directory it is mandatory to parse the paths to the test program names, compile directory, argument files, INI-File, name of the output CSV-Files and the absolute path to them. This is done with a helper class, which provides the methods to parse these information. One key difference of the test programs is how the test cases are provided, either per file input or as command line arguments. For that reason the script has to parse this information from the INI-File. If the test program awaits the test case as file input, the name of the test case is provided, otherwise the file containing the right test case is opened and the content is passed to the test program via command line arguments. This is done for each obfuscation/compile method of each program. The programs with the same source file are run by the script and the output and exit code get filled in a pandas data frame, which gets written to disk in form of a CSV-File. The CSV-File gets written to a folder named *compare_ret* in the run directory. The output of the same test program with other obfuscation/compilation techniques gets saved to the same CSV-File. These files have the following naming convention:

```
"test␣program␣name"_return_val.csv
```

4.8 Measure Performance of the Programs

For the performance measurements the script *start_measure.sh* is responsible. In the first step it parses the directory for the last directory where the test programs reside in, if this directory is not provided via command line arguments. After the correct folder has been provided or found by the script, it iterates over all directories within the run directory and executes the *python3* script *measure.py* with the programs to run as command line arguments. The script *measure.py* awaits a second command line argument beside the path to the executable, which is the *-JSON* flag. It decides, if the output is a CSV-File or in JSON format. The *python3* script *measure.py* uses the helper class, which is responsible for parsing folder names, program names, argument files, INI-File and output file names. The second class is in the *module_metric.py* file and does the heavy lifting by providing the logic to perform the static and dynamic measures on the test programs.

4.8.1 Module measure

The last script in the framework opens the previously saved CSV-File files and saves them into pandas data frames. In the actual version of the framework this script displays the runtime, virtual memory usage, data, return value, file size on disk and the entropy of the binary file. Additionally the average, median, maximum and minimum values are printed to stdout. The format of the script is adapted to display the same programs (merge sort, quick sort, etc) with all compile techniques next to each other. With the formated output the difference in performance is more clear.
With the data saved in the pandas data frames the script draws plots for the file size, runtime and entropy of the test programs. For the entropy and the file size bar plots get generated to compare the attributes to the same program compiled with other obfuscation techniques and optimization levels. In the current setting the framework runs all programs six times and all six runtime elements are taken to draw a box plot graph. These graphs get saved to a newly created *results* folder inside the current run directory, where for every program a folder exists for separation.

5 Benchmark

In this section, the benchamrk results are presented and analyzed.

5.1 Sample set

For benchmarking with the obfuscation abcdef, a sample set was taken from the github repository [67]. The samples are limit to the hash and sort software. These samples were modified to accept arguments provided via command line or input file. The output and return values of the program were modified, in order to provide a uniform comparable output of the programs with which the obfuscation abcdef can deal with. Input files were integrated to the framework, because command line arguments are limited in size and it was the purpose to investigate how these programs perform in obfuscated form with large input. All programs are build on a modular basis because *tigress* needs the name of the function that are obfuscated during the compile step. This sample set was specifically crafted for benchmarking obfuscation techniques and the output is deterministic.

The following hash algorithms are included in the test sample set:

- bkdrhash
- bphash
- dekhash
- djbhash
- elfhash
- fnvhash
- jshash
- pjwhash
- rshash
- sdbmhash

For the category sort algorithms, the subsequent five are included in the benchmark tests.

- bubblesort

- insertionsort
- mergesort
- quicksort
- selectionsort

5.2 General benchmark setup

Obfuscation abcdef in the current version examines the latest GCC version and three obfuscation methods, which are generated by *tigress* and *tinycc*. For the GCC compiled programs the optimization level zero to three are investigated. The same optimization level are taken into account by the programs, which get compiled with different obfuscation techniques. For benchmarking the following obfuscation techniques are investigated in more detail: control-flow graph flattening, opaque predicates and virtualization. Test programs compiled with *tinycc* are compiled with the version 0.9.27 and the latest available version which is installed dynamically during installation of the framework.
The following properties of the test programmes were observed in detail during the benchmark: Virtual memory, the runtime, return value, file size and the entropy of the executable file. All test programs are run six times in the current setting and the performance of each execution is ascertained. The number of executions per program is six to prevent outliers. Additionally the average, median, minimum and maximum are calculated for each program attribute. The end results are provided via pandas data frames and exported to a CSV-File, where all benchmarks can be compared.

5.3 Correct runs of test programs

The test programs got compiled with the compiler/optimization techniques listed in this section. There are 30 programs in total, which were run twice during the tests. One time with an argument size of 100 and the second time with an argument size of 10,000. All test programs exited gracefully and the output was exactly the same, thus it is assumed, that the programs run correctly and the optimization and obfuscation did not affect the semantics of the programs.

- compile technique
- gcc 9.3.0 optimization level 0
- gcc 9.3.0 optimization level 1
- gcc 9.3.0 optimization level 2
- gcc 9.3.0 optimization level 3

- tigress 3.1 control-flow flattening with gcc optimization level 0
- tigress 3.1 control-flow flattening with gcc optimization level 1
- tigress 3.1 control-flow flattening with gcc optimization level 2
- tigress 3.1 control-flow flattening with gcc optimization level 3
- tigress 3.1 opaque predicates with gcc optimization level 0
- tigress 3.1 opaque predicates with gcc optimization level 1
- tigress 3.1 opaque predicates with gcc optimization level 2
- tigress 3.1 opaque predicates with gcc optimization level 3
- tigress 3.1 virtualization with gcc optimization level 0
- tigress 3.1 virtualization with gcc optimization level 1
- tigress 3.1 virtualization with gcc optimization level 2
- tigress 3.1 virtualization with gcc optimization level 3
- tinycc 0.9.27 default
- tinycc oslatest default

5.4 Measurements of entropy, file size and runtime

The programs compiled using virtualisation are the biggest, based on the file size. It is also not possible to find differences between programs using inputs from files and those which use command line based on the criteria above.

5.4.1 Runtime of the programs

For the visualization of the runtime the programs generate box plots, moreover it was conducted the longer the runtime of the program is the less the measurements get falsified from operation performed by the OS. The hash programs with input of 10,000 string length, the hash programs with input size of 100 and the sort programs with the input size of 100 are too fast to distinguish, which optimization and obfuscation technique is slower than the others, as can be seen in Figure 5.1 with the jshash as example.

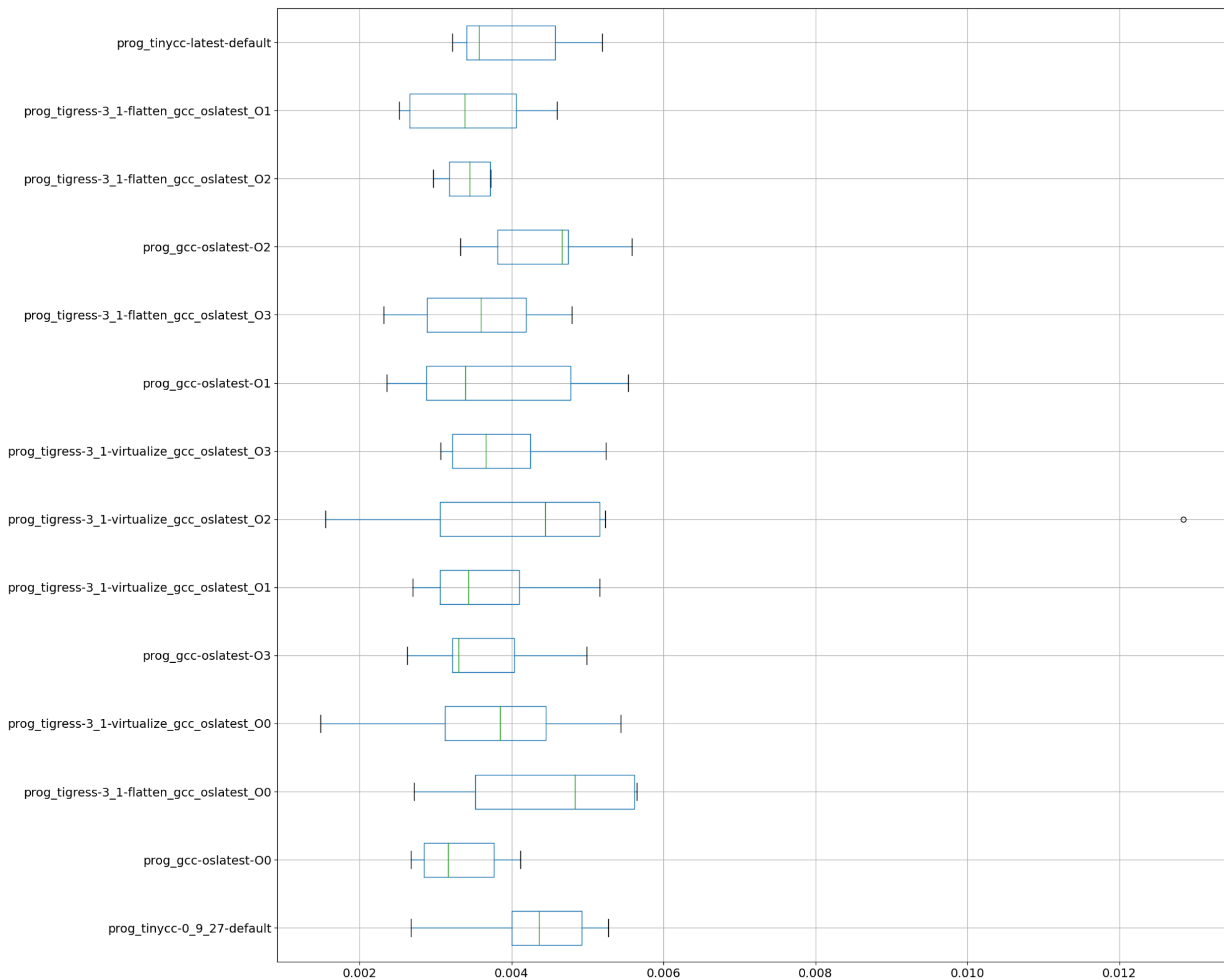

Figure 5.1: Runtime of jshash in seconds with string length of 100.

The sort programs with input of 10,000 integers are slower than the other programs and thus provide more informative results as can be seen in Figure 5.2. From this figure it can be deduced, that the virtualisation obfuscation method with no optimization is tremendously slower than all other programs. Generally the virtualisation obfuscation technique slows down the runtime for all programs.

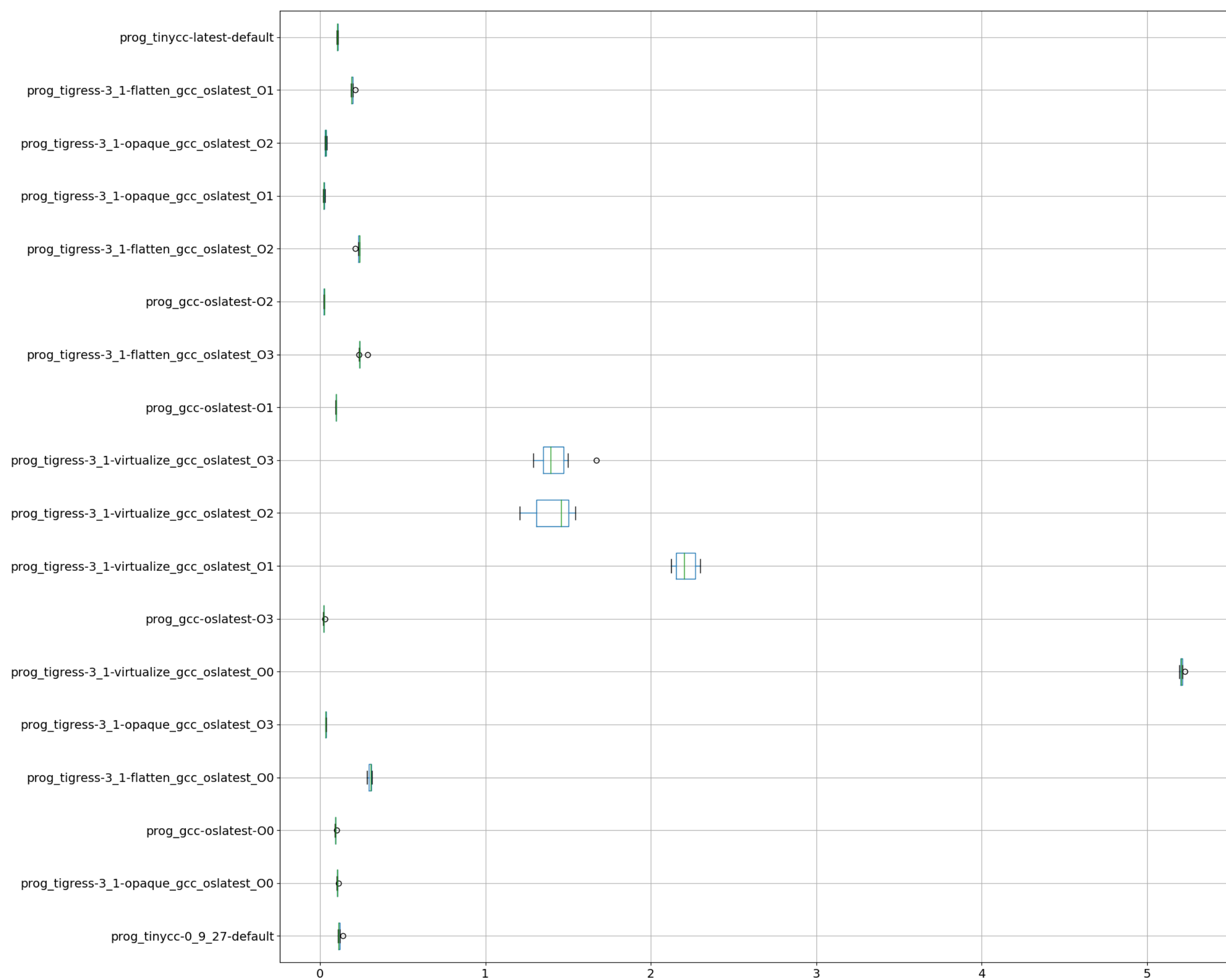

Figure 5.2: Runtime in seconds of selectionsort program with 10,000 integers as input.

5.4.2 File size of the programs

By comparing the file size of the test programs, it is evident, that some obfuscation techniques generate smaller binaries than others. This behaviour is consistently evident in both hash programs and sort programs. Different sort programs compiled with the same obfuscation technique and optimisation level are relatively equal in size to the same program compiled and optimized with other techniques. Also the file sizes are constant over different test programs. For example smdbhash and elfhash, both compiled with tinycc 0.9.27, are almost the same size. As can be seen in Figure 5.3 the virtualization obfuscation technique is always the biggest in terms of file size, whereas the programs compiled with tinycc are much smaller. In general the tinycc programs are the smallest, followed by the gcc and control-flow graph flattening programs. The second largest are the test programs compiled with opaque predicates and the largest programs are the ones containing the virtualisation obfuscation technique. The figure Figure 5.3 is used as an example, the results have a similar distribution for every test program.

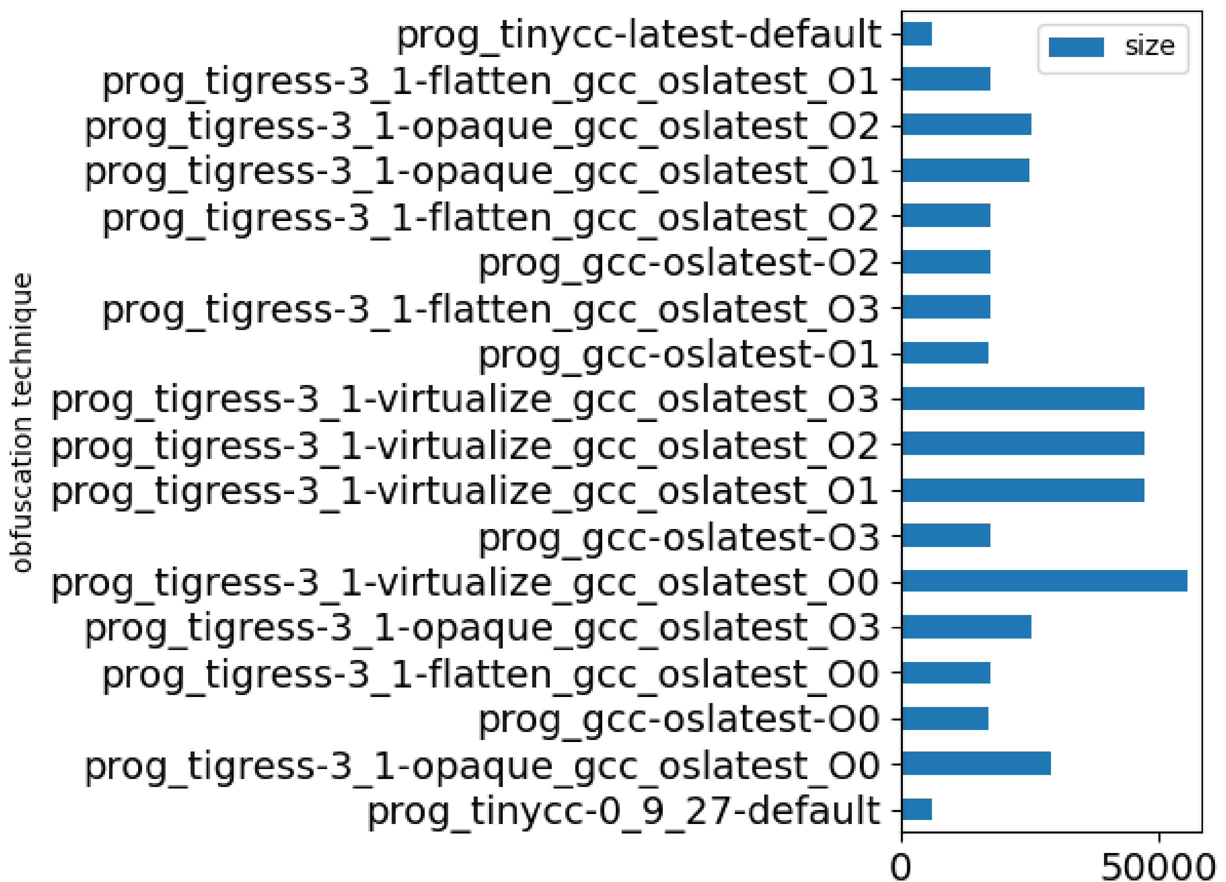

Figure 5.3: File size in Bytes of mergesort with file input.

The hash programs are relatively the same on source code level, thus they do not differ regarding their file size. In terms of optimisation level, the compiler flag does sometimes not influence the file size e.g. for programs compiled with gcc and the control-flow graph obfuscation technique. If the programs are compiled with virtualization, the one without optimization is by far the largest as can be seen in Figure 5.4

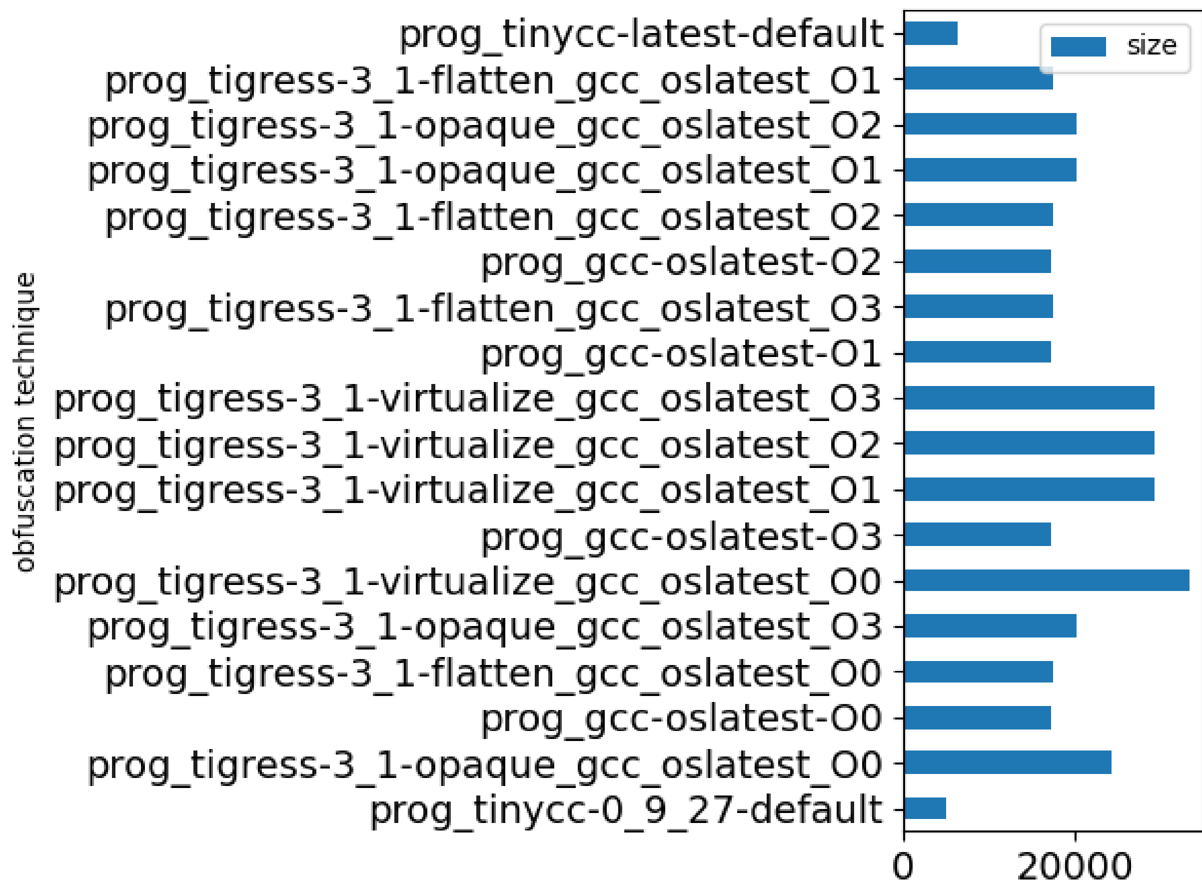

Figure 5.4: djbhash file size in bytes.

5.4.3 Entropy of the programs

The behaviour of the test programs regarding the entropy are very similar to the ones regarding the file size described in subsection 5.4.2. The hash programs are almost the same, therefore the entropy is constant between different hash test programs.

The entropy of the programs is consistent across different test programs, as their characteristics are almost the same. Furthermore the relation between the different compile and optimization techniques stay the same. For example programs with opaque predicates show the largest entropy among the obfuscation techniques used.

Another aspect is, that the programs compiled with tinycc have the highest entropy. The reason for this behavior is the small file size, which is related to the high entropy in this case. In figure Figure 5.5 those properties can be seen by the example of the pjwhash program.

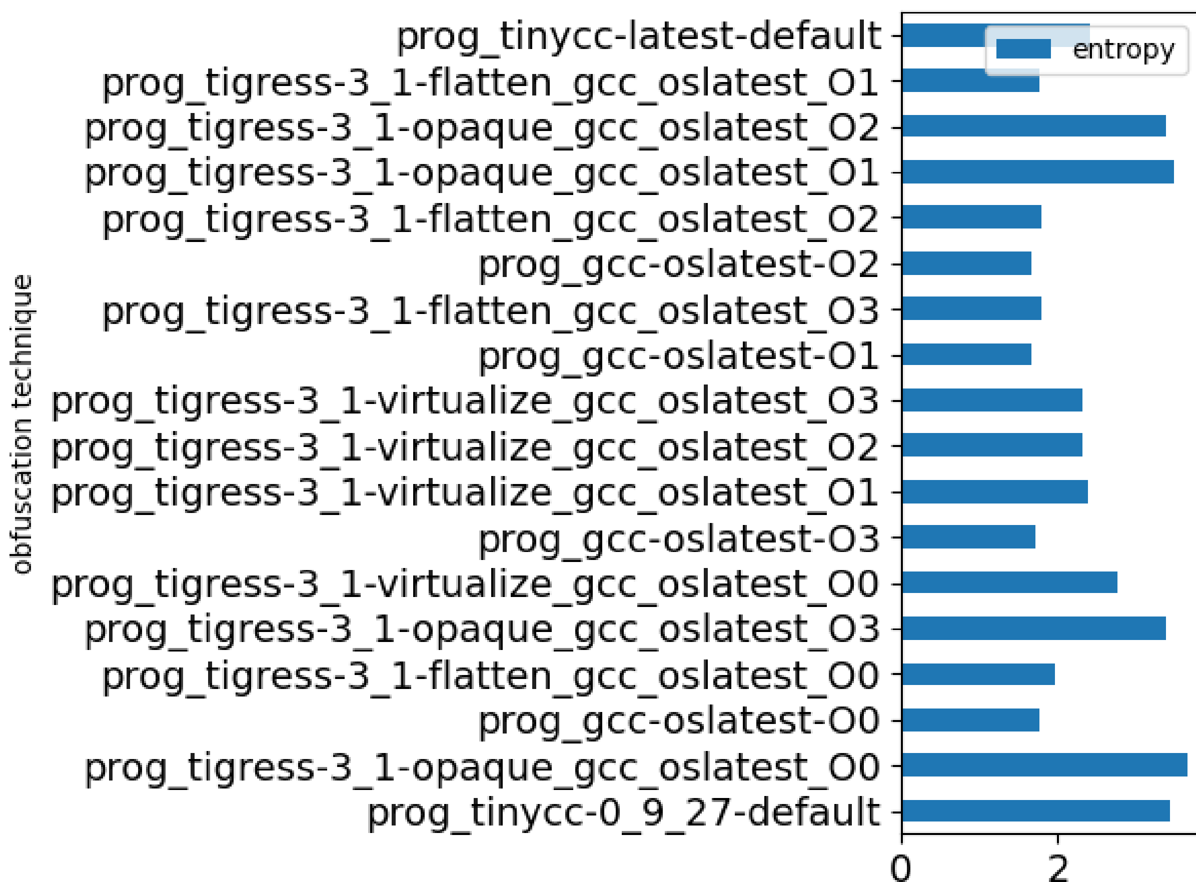

Figure 5.5: File entropy of pjwhash

6 Conclusion

In this thesis an overview of obfuscation techniques was given. For the evaluation of different obfuscation techniques a framework, named obfuscation abcdef was developed to provide a fast approach to compile and test obfuscation techniques on programs and measure their output. The framework is capable of measuring the following properties of an executable file:

- virtual memory usage
- runtime
- CPU usage
- File size
- File entropy
- Mime type

The file size, file entropy and runtime were analyzed in detail. It can be concluded that the source code of programs does not directly affect the entropy of the final binary, only the compiler or obfuscator influences the entropy. This behaviour is equally if the file size is analyzed.
If the test programs are big enough, the optimization flags of the compiler can limit the size and runtime of the final executable. This was evident by analyzing the sort algorithm programs with 10,000 arguments, where the virtualization obfuscation method without optimization was slower by the factor 100, than the same program compiled with GCC using optimization. If the programs are too small, a difference cannot be noted regarding the file size, entropy and runtime also the optimization does not affect small executables.
The *tinycc* compiler creates very small binaries, which have a much higher entropy, than the other executables. The two attributes entropy and file size are coherent, because a small file will result in an high entropy. If optimization is used, there is no impact noted by the programs compiled with GCC and with the obfuscation method control-flow graph flattening.

6.1 Future Work

The obfuscation abcdef is a simple way to evaluate executables and their performance. The current version is limited to specific programs, because all programs receive the same input file, as it was configured in the INI-File, thus if it gets modified the other programs are working with the modified input file. Furthermore, the current test cases are only written for integer values, strings and hexadecimal strings. If steganographic tools are chosen as test programs the test cases for pictures must get added.

In this thesis, the obfuscation abcdef was currently only tested with small sized programs, however as shown small executable files are hard to analyze, because their runtime is very small and the obfuscation techniques and optimization levels do not have a high impact so the practicability with huge programs was not considered. Moreover, the obfuscation techniques were limited to control-flow graph flattening, opaque predicates and virtualization, more exhaustive tests with more obfuscation techniques and combinations should be considered and the tool in use was tigress 3.1, although the framework can be extended in order to use different tools.
Further research could show if the entropy, file size and compiler in use can provide information about the implemented obfuscation technique.

List of Figures

4.1 Part 1 Content checking: The general contents and scripts are checked in this part. On the left side of the image the inputs for the processes, which is represented in the forms in the middle, are shown. 25

4.2 Second part of the framework: prepare for analyses. In this part the files get compiled and the inputs for the test programs are generated. Basic checks are performed, if the programs execute and exit gracefully. On the left side the inputs in form of rectangles are shown. The processes are square forms in the middle. The diamonds on the right hand side represent the output of the processes. 26

4.3 Part 3 of the framework: Performance measuring and analyses. On the left the input to the binaries and the binaries themselves are shown as rectangles. In the middle of the graphic the processes and on the right side the output in form of diamonds are represented. 27

5.1 Runtime of jshash in seconds with string length of 100. 36

5.2 Runtime in seconds of selectionsort program with 10,000 integers as input. 37

5.3 File size in Bytes of mergesort with file input. 38

5.4 djbhash file size in bytes. 39

5.5 File entropy of pjwhash 40

List of Tables

Listings

user_data/Z_listings/testcase_ini_example.txt 29

Glossary

AES	Advanced Encryption Standard
AOP	Aspect oriented programming
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CFG	Control-flow graph
CPU	Central processing unit
CSV-File	Comma-Separated Values File
DES	Data Encryption Standard
DRM	Digital rigths management
DSE	dynamic symbolic execution
EDR	Endpoint Detection and Response
FPGA	Field-programmable gate array
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPU	Graphics processing unit
HLS	High Level Synthesis

INI-File	Initialization/Configuration File
IT	Information Technology
JSON	JavaScript Object Notation
LLVM	LLVM is a toolkit which convert instruction into a form that is understood and can be executed by computers. With LLVM it is possible to create new computer languages and improve them. It is possible to lift binaries into LLVM Intermediate Representation Language to modify them or use them otherwise.
Malware	Malicious computer software
MATE	man-at-the-end
obfuscation abcdef	obfuscation automatic benchmark compilation dynamic evaluation Framework, is the name of the framework developed alongside this thesis. It is capable of measuring the performance and the possibility of obfuscation techniques with test programs.
OS	Operating System
S2cbench	Synthesizable SystemC Benchmark
shellcode	Shellcode is a small piece of opcodes which get injected into an exploited program and afterwards gets executed. It is written in assembly language and gets translated to hexadecimal opcodes

SPE	Software Performance Engineering
VM	Virtual Machine
WYSINWYX	What you see is not what you execute

Bibliography

[1] *Kaspersky Security Bulletin 2020-2021. EU statistics*. [Online]. Available: https://securelist.com/kaspersky-security-bulletin-2020-2021-eu-statistics/102335/ (visited on 07/11/2021).

[2] *Piracy Is Back: Piracy Statistics for 2021*, en-US, Mar. 2021. [Online]. Available: https://dataprot.net/statistics/piracy-statistics/ (visited on 07/11/2021).

[3] *Empowering App Development for Developers | Docker*, en. [Online]. Available: https://www.docker.com/ (visited on 07/11/2021).

[4] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

[5] Michael Sikorski and Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st. USA: No Starch Press, 2012, ISBN: 1593272901.

[6] Anjana Gosain and Ganga Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, Eds., Cham: Springer International Publishing, 2015, pp. 113–122, ISBN: 978-3-319-11933-5.

[7] Hui Xu, Yangfan Zhou, and Michael Lyu, "N-version obfuscation," in *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security*, ser. CPSS '16, Xi'an, China: Association for Computing Machinery, 2016, pp. 22–33, ISBN: 9781450342889. DOI: 10.1145/2899015.2899026. [Online]. Available: https://doi.org/10.1145/2899015.2899026.

[8] Sebastian Schrittwieser and Stefan Katzenbeisser, *Code Obfuscation against Static and Dynamic Reverse Engineering*.

[9] Christian Collberg, Clark Thomborson, and Douglas Low, "A taxonomy of obfuscating transformations," *http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial*, Jan. 1997.

[10] C. Collberg, C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," in *Proceedings of the 1998 International Conference on Computer Languages (Cat. No.98CB36225)*, 1998, pp. 28–38. DOI: 10.1109/ICCL.1998.674154.

[11] Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *Information Security Applications*, Sehun Kim, Moti Yung, and Hyung-Woo Lee, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75, ISBN: 978-3-540-77535-5.

[12] Chris Anley, Jack Koziol, Felix Linder, and Gerardo Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. USA: John Wiley &; Sons, Inc., 2007, ISBN: 047008023X.

[13] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus, "English shellcode," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, Chicago, Illinois, USA: Association for Computing Machinery, 2009, pp. 524–533, ISBN: 9781605588940. DOI: 10.1145/1653662.1653725. [Online]. Available: https://doi.org/10.1145/1653662.1653725.

[14] Christian Collberg, Clark Thomborson, and Douglas Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, San Diego, California, USA: Association for Computing Machinery, 1998, pp. 184–196, ISBN: 0897919793. DOI: 10.1145/268946.268962. [Online]. Available: https://doi.org/10.1145/268946.268962.

[15] J. Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang, "Experience with software watermarking," in *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000, pp. 308–316. DOI: 10.1109/ACSAC.2000.898885.

[16] Frederick B. Cohen, "Operating system protection through program evolution," *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, Oct. 1993, ISSN: 0167-4048. DOI: 10.1016/0167-4048(93)90054-9. [Online]. Available: https://doi.org/10.1016/0167-4048(93)90054-9.

[17] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994, ISSN: 0360-0300. DOI: 10.1145/197405.197406. [Online]. Available: https://doi.org/10.1145/197405.197406.

[18] Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan, “Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings,” in *Proceedings of the 9th Workshop on Multimedia & Security*, ser. MM&Sec ’07, Dallas, Texas, USA: Association for Computing Machinery, 2007, pp. 129–140, ISBN: 9781595938572. DOI: 10.1145/1288869.1288887. [Online]. Available: https://doi.org/10.1145/1288869.1288887.

[19] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson, “Software tamper resistance: Obstructing static analysis of programs,” USA, Tech. Rep., 2000.

[20] Cullen Linn and Saumya Debray, “Obfuscation of executable code to improve resistance to static disassembly,” in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS ’03, Washington D.C., USA: Association for Computing Machinery, 2003, pp. 290–299, ISBN: 1581137389. DOI: 10.1145/948109.948149. [Online]. Available: https://doi.org/10.1145/948109.948149.

[21] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews, “Binary obfuscation using signals,” in *16th USENIX Security Symposium (USENIX Security 07)*, Boston, MA: USENIX Association, Aug. 2007. [Online]. Available: https://www.usenix.org/conference/16th-usenix-security-symposium/binary-obfuscation-using-signals.

[22] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl, “Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?” en, *ACM Computing Surveys*, vol. 49, no. 1, pp. 1–37, 2016, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2886012. [Online]. Available: https://dl.acm.org/doi/10.1145/2886012 (visited on 07/04/2021).

[23] Christian Collberg, *Surreptitious software : obfuscation, watermarking, and tamperproofing for program protection*, eng, 1. print.., ser. Software security series. 2009, ISBN: 9780321549259.

[24] Monnappa K A, *Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware*, eng. Birmingham: Packt Publishing, Limited, 2018, ISBN: 1788392507.

[25] ——, *Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware*, eng. Birmingham: Packt Publishing, Limited, 2018, ISBN: 1788392507.

[26] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, “Exploiting self-modification mechanism for program protection,” in *in Proc. 27th IEEE Computer Software and Applications Conference*, 2003, pp. 170–179.

[27] James Riordan and Bruce Schneier, "Environmental Key Generation Towards Clueless Agents," in *Mobile Agents and Security*, Giovanni Vigna, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 15–24, ISBN: 978-3-540-68671-2. DOI: 10.1007/3-540-68671-1_2. [Online]. Available: https://doi.org/10.1007/3-540-68671-1_2.

[28] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," en, p. 13,

[29] S.T. King and P.M. Chen, "Subvirt: Implementing malware with virtual machines," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, 2006, 14 pp.-327. DOI: 10.1109/SP.2006.38.

[30] Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson, "A Secure and Robust Approach to Software Tamper Resistance," in *Information Hiding*, Rainer Boehme, Philip W. L. Fong, and Reihaneh Safavi-Naini, Eds., vol. 6387, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 33–47, ISBN: 978-3-642-16434-7 978-3-642-16435-4. [Online]. Available: http://link.springer.com/10.1007/978-3-642-16435-4_3 (visited on 07/16/2021).

[31] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan, "Proteus: Virtualization for diversified tamper-resistance," in *Proceedings of the ACM Workshop on Digital Rights Management*, ser. DRM '06, Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 47–58, ISBN: 159593555X. DOI: 10.1145/1179509.1179521. [Online]. Available: https://doi.org/10.1145/1179509.1179521.

[32] Cristian Cadar and Koushik Sen, "Symbolic execution for software testing: Three decades later," en, *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2408776.2408795. [Online]. Available: https://dl.acm.org/doi/10.1145/2408776.2408795 (visited on 07/04/2021).

[33] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[34] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion, "Obfuscation: Where are we in anti-dse protections? (a first attempt)," in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, ser. SSPREW9 '19, San Juan, Puerto Rico, USA:

Association for Computing Machinery, 2019, ISBN: 9781450377461. DOI: 10.1145/3371307.3371309. [Online]. Available: https://doi.org/10.1145/3371307.3371309.

[35] Cristian Cadar, Daniel Dunbar, Dawson R Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *OSDI*, vol. 8, 2008, pp. 209–224.

[36] Peter Garba and Matteo Favaro, "Saturn - software deobfuscation framework based on llvm," *ArXiv*, vol. abs/1909.01752, 2019. DOI: 10.1145/3338503.3357721.

[37] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub, "A survey on malware and malware detection systems," *International Journal of Computer Applications*, vol. 67, no. 16, 2013.

[38] Xufang Li, Peter K.K. Loh, and Freddy Tan, "Mechanisms of polymorphic and metamorphic viruses," in *2011 European Intelligence and Security Informatics Conference*, 2011, pp. 149–154. DOI: 10.1109/EISIC.2011.77.

[39] Jiaming He and Hongbin Zhang, "Digital right management model based on cryptography and digital watermarking," in *2008 International Conference on Computer Science and Software Engineering*, vol. 3, 2008, pp. 656–660. DOI: 10.1109/CSSE.2008.1000.

[40] Matěj Myška *et al.*, "The true story of drm," *Masaryk University Journal of Law and Technology*, vol. 3, no. 2, pp. 267–278, 2009.

[41] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[42] Peter Van der Linden, *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

[43] Shengtong Zhong, Yang Shen, and Fei Hao, "Tuning compiler optimization options via simulated annealing," in *2009 Second International Conference on Future Information Technology and Management Engineering*, 2009, pp. 305–308. DOI: 10.1109/FITME.2009.81.

[44] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3363562. [Online]. Available: https://doi.org/10.1145/3363562.

[45] William M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[46] G Balakrishnan, T Reps, D Melski, and T Teitelbaum, "WYSINWYX: What You See Is Not What You eXecute," en, p. 11,

[47] Hong-Li Wu, Yong Zhong, and Yong Chen, "A software reliability prediction model based on benchmark measurement," in *2010 2nd IEEE International Conference on Information Management and Engineering*, 2010, pp. 131–134. DOI: 10.1109/ICIME.2010.5478245.

[48] Filippos I. Vokolos and Elaine J. Weyuker, "Performance testing of software systems," in *Proceedings of the 1st International Workshop on Software and Performance*, ser. WOSP '98, Santa Fe, New Mexico, USA: Association for Computing Machinery, 1998, pp. 80–87, ISBN: 1581130600. DOI: 10.1145/287318.287337. [Online]. Available: https://doi.org/10.1145/287318.287337.

[49] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[50] Benjamin Carrion Schafer and Anushree Mahapatra, "S2cbench: Synthesizable systemc benchmark suite for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 53–56, 2014. DOI: 10.1109/LES.2014.2320556.

[51] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[52] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 189–200.

[53] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[54] Dirk Merkel *et al.*, "Docker: Lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[55] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion, "Obfuscation: Where are we in anti-DSE protections? (a first attempt)," en, in *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering - SSPREW9 '19*, San Juan, Puerto Rico: ACM Press, 2019, pp. 1–8, ISBN: 978-1-4503-7746-1. DOI: 10.1145/3371307.3371309. [Online].

Available: http://dl.acm.org/citation.cfm?doid=3371307.3371309 (visited on 07/04/2021).

[56] Boaz Barak, "Hopes, fears, and software obfuscation," en, *Communications of the ACM*, vol. 59, no. 3, pp. 88–96, Feb. 2016, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2757276. [Online]. Available: https://dl.acm.org/doi/10.1145/2757276 (visited on 07/04/2021).

[57] Cristian Cadar, Daniel Dunbar, and Dawson 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, San Diego, California: USENIX Association, 2008, pp. 209–224.

[58] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni 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. DOI: 10.1109/SP.2016.17.

[59] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion, "Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43.

[60] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song, "Statically-directed dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 12–22, ISBN: 9781450305624. DOI: 10.1145/2001420.2001423. [Online]. Available: https://doi.org/10.1145/2001420.2001423.

[61] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012, ISSN: 0734-2071. DOI: 10.1145/2110356.2110358. [Online]. Available: https://doi.org/10.1145/2110356.2110358.

[62] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, Orlando, Florida, USA: Association for Computing Machinery, 2012, pp. 319–328, ISBN: 9781450313124. DOI: 10.1145/2420950.2420997. [Online]. Available: https://doi.org/10.1145/2420950.2420997.

[63] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion, "How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections)," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19, San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 177–189, ISBN: 9781450376280. DOI: 10.1145/3359789.3359812. [Online]. Available: https://doi.org/10.1145/3359789.3359812.

[64] Dick Grune, Ed., *Modern compiler design*, eng, 2. ed. New York: Springer, 2012, ISBN: 978-1-4614-4698-9.

[65] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu, "Metamorphic testing: A new approach for generating next test cases," *CoRR*, vol. abs/2002.12543, 2020. arXiv: 2002.12543. [Online]. Available: https://arxiv.org/abs/2002.12543.

[66] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019, ISSN: 0360-0300. DOI: 10.1145/3365001. [Online]. Available: https://doi.org/10.1145/3365001.

[67] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16, Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 189–200, ISBN: 9781450347716. DOI: 10.1145/2991079.2991114. [Online]. Available: https://doi.org/10.1145/2991079.2991114.

[68] *Tum-i4/obfuscation-benchmarks*, original-date: 2015-09-15T20:43:05Z, Jul. 2021. [Online]. Available: https://github.com/tum-i4/obfuscation-benchmarks (visited on 07/10/2021).

[69] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[70] Saeyang Yang, *Logic synthesis and optimization benchmarks user guide version 3.0*, 1991.

[71] Zhen Li, Jun-Feng Tian, and Feng-Xian Wang, "Sandbox system based on role and virtualization," in *2009 International Symposium on Information Engineering and Electronic Commerce*, 2009, pp. 342–346. DOI: 10.1109/IEEC.2009.77.

[72] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste, "Malware dynamic analysis evasion techniques: A survey," *ACM Comput. Surv.*, vol. 52, no. 6, Nov. 2019, ISSN: 0360-0300. DOI: 10.1145/3365001. [Online]. Available: https://doi.org/10.1145/3365001.

[73] (). The Drebin Dataset, [Online]. Available: https://www.sec.tu-bs.de/~danarp/drebin/ (visited on 07/08/2021).

[74] (). VirusTotal, [Online]. Available: https://www.virustotal.com/gui/ (visited on 07/08/2021).

[75] (). Cuckoo Sandbox - Automated Malware Analysis, [Online]. Available: https://cuckoosandbox.org/ (visited on 07/08/2021).

[76] (). Welcome to Python.org. en, [Online]. Available: https://www.python.org/ (visited on 06/28/2021).

[77] (). Make - GNU Project - Free Software Foundation, [Online]. Available: https://www.gnu.org/software/make/ (visited on 06/20/2021).

[78] (). Option Summary (Using the GNU Compiler Collection (GCC)), [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html (visited on 06/20/2021).

[79] (). GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF), [Online]. Available: https://gcc.gnu.org/ (visited on 06/28/2021).

[80] (). Rfc4180, [Online]. Available: https://datatracker.ietf.org/doc/html/rfc4180 (visited on 07/01/2021).