Università
Ca'Foscari
Venezia

Master's Degree
in Computer Science

Final Thesis

Analysis and implementation of Software Similarity metrics

Supervisor
Prof. Paolo Falcarin
Graduand
Jonathan Gobbo
Matriculation number
870506
Academic Year
2022 / 2023

Analysis and implementation of Software Similarity metrics

Jonathan Gobbo

Abstract

Software obfuscation is the act of changing the inner workings of a program to make it more expensive in terms of time and effort to reverse engineer, while maintaining the same semantics. This practice has lately been employed for a variety of purposes, both legit and malicious, such as intellectual property protection and hiding malware internal details to avoid detection. Moreover, by using obfuscation it is possible to achieve Software diversification, a condition where different versions of the same program are generated and distributed, with the advantage of reducing code-reuse attacks.

The quality of these obfuscation and diversification techniques can be measured by using a wide variety of similarity metrics. In this work, several of these software similarity techniques, working on different abstraction levels, are implemented. Their performance is then evaluated by applying them to sample programs that were obfuscated with common obfuscation techniques.

Contents

Chapter 1

Introduction

Software Obfuscation and Diversification are two techniques that have long been used to protect software from man-at-the-end attacks. Obfuscation operates by changing the inner workings of a program to make it harder for a reverse engineer to understand how the program works. Diversification, on the other hand, focuses on delivering different versions of a software, so that an attacker, for example, would be unable to build an exploit that works on all the variants.

Being able to measure how similar two programs are is therefore very important in the context of these two techniques, and Software Similarity metrics achieve exactly that purpose. Using these metrics, it is possible to evaluate the effectiveness of an obfuscation configuration, while also allowing to search for a set of diversified variants of a software whose distance between each other is maximized.

This work explores the topic of Software Similarity by implementing a selection of metrics and testing them against an array of programs obfuscated using multiple obfuscation configurations.

In particular, chapter 2 provides a background on the knowledge of Software Obfuscation and Diversification. Regarding the first, a formal definition is given, and the most common obfuscating transformations are described. Then, Diversification is analyzed, providing information on the different ways it can be achieved.

Chapter 3 covers instead the problem of Software Similarity. After an introduction on the topic, an overview on how the metrics can be categorized is given. Then, the chapter describes in depth the set of metrics investigated in this work.

The following chapter, chapter 4, focuses on the project developed to test the metrics. Particularly, it introduces the structure of the project, it describes every tool used and how they work, explains how each metric is invoked and provides examples of the outputs. Additionally, it shows how the automated

testing is performed and how it is configured.

Finally, chapter 5 provides a description of how the experiments are set up, then presents the results obtained for each metric, while providing comments explaining the results. Then, a comparison of all the metric is shown, and the best performing metrics are identified.

Chapter 2

Background

2.1 Software Obfuscation

2.1.1 Definitions

As mentioned in the introduction, Software Obfuscation is the act of changing the implementation details of a program to make it more difficult to reverse-engineer. Obfuscation can be achieved by chaining together a list of obfuscating transformations.

Formally, an Obfuscating Transformation is a program transformation $P \rightarrow^{\tau} P'$, where P and P' are respectively the source and target programs, and the two have the same observable behaviour. This condition requires P and P' to have the same behaviour as seen by the user, while allowing side-effects like file creation and network activity. [4]

Applying a set of obfuscation transformations $\tau = \{\tau_1, ..., \tau_n\}$ on a program P has the objective of finding a new program P' such that its *quality* is maximized. The quality is an aggregation of the following measures [4] [5]:

- Obscurity (also known as Potency) measures how much more expensive it is for a reverse engineer to understand the obfuscated code with respect to the clean one. It is defined as $\tau_{pot}(P) = E(P')/E(P) - 1$, where E is a software complexity measure (e.g. the Halstead program length [9] or the McCabe cyclomatic complexity [16]);

- Resilience, similarly to obscurity, measures how expensive it is for an automatic tool such as a deobfuscator to undo the transformations;

- Stealth, which measures the statistical similarity of the code, or how *hidden* the obfuscation is;

- Cost measures the overhead in terms of execution time and space. It should be minimized.

An additional way of measuring the quality of obfuscation is Similarity. This method, which measures how similar two programs are, is the focus of this work and will be detailed in chapter 3.

2.1.2 Obfuscation Transformations

Here are introduced common obfuscation transformations as described in the article *A taxonomy of obfuscating transformations* by Collberg, Thomborson, and Low [4]. The following techniques are classified according to the target of the transformations: *layout*, *control* or *data*.

Layout Obfuscations

The Layout class provides trivial obfuscation transformations that achieve low obscurity but come with no performance impact. These are often useful when dealing with interpreted languages or in general with ones that do not compile to native code. These transformations include:

- Remove Comments: as the name suggests, it simply removes the comments from the code;

- Scramble Identifiers: removes or renames the identifiers. This can also be achieved on native binaries by *stripping* them of debug symbols.

Control Obfuscations

This class of obfuscations targets the control flow of a program. These can be further classified according to how they influence the flow: *aggregation*, *ordering* and *computation*.

A fundamental construct useful to implement several of the computation transformations is the Opaque Construct. These can be Opaque Variables and Opaque Predicates. Formally, V_p^q is a opaque variable if it has a property q at point p known at obfuscation time, while a predicate P is opaque at p if its outcome is known at obfuscation time. Notably, P_p^F and P_p^T respectively mean that the predicate always evaluates to false and true. Moreover, $P_p^?$ may evaluate to either true or false.

Aggregation These methods try to achieve confusion by meddling with the aggregation of the code the programmer originally intended:

- Inline method: replaces the call of a procedure with the code of the procedure itself;
- Outline statements: moves a sequence of statements into a new function;
- Method interleaving: merges multiple functions into a single one with an additional parameter that allows choosing the desired behaviour;
- Clone methods: clones a method so that it seems that different functions are being called;
- Loop unrolling: this technique is often used by the compiler to achieve loop parallelism and improve performance. It creates confusion by removing or reducing the number of iterations of a loop and replicating the code originally inside of it.

Ordering These transformations simply reorder statements, loops, expressions in the code to counter the principle of locality.

Computation Finally, computation transformations add and modify code, often using opaque constructs:

- Insert Dead code: this method uses opaque predicates to split basic blocks. For example, a P^T predicate can be used by moving the code inside the true branch, while the false branch will contain either nothing or some fake / buggy code. Alternatively, a $P^?$ predicate could contain semantically equivalent code on both branches;
- Extend Loop Conditions: adds an opaque predicate to the condition of a loop to make it harder to understand how many times it will cycle;
- Remove Library Calls: hides system library calls by replacing them with a version provided by the obfuscator;
- Add Redundant Operands: adds opaque variable to arithmetic expressions;
- Parallelize Code: creates useless processes or splits sequential code into multiple parallel processes.

Data Obfuscations

As the name implies, this class of transformations targets the data of a program. Just like control obfuscation, it is possible to further classify these methods into the following classes: *storage & encoding*, *aggregation* and *ordering*.

Storage & Encoding The following transformations target the storage structures and encoding of data:

- Change Encoding: the value of a variable is represented with a different encoding;
- Promote Variable: in Object Oriented languages, replaces the primitive type of a variable with its object counterpart (e.g. `int` to `Integer` in Java).
- Split Variables: splits a variable into multiple ones, with a function mapping their values;
- Static to Procedural Data: converts data into a function that produces that data.

Aggregation The purpose of the following transformations is to modify the data structures present in the program:

- Merge Scalar Variables: merges multiple variables into a single one that can fit all of them. Arithmetic operations are updated accordingly;
- Restructure Arrays: allows to split, merge, fold and flatten arrays;
- Modify Inheritance: splits classes and add useless ones.

Ordering Similarly to Control ordering obfuscations, these reorder methods, variables and parameters.

2.2 Software Diversification

Software Diversification refers to the act of modifying the structure of a program to allow the creation of different versions of the same program [11]. This has the advantage of making it harder for an attacker to develop an exploit that can work across all the diversified versions of the program, even though the vulnerabilities are not actually removed.

In "SoK: Automated software diversity" by Larsen et al., the authors answer to two important questions that define the implementation of diversification: what should be diversified and when should it be diversified [13].

Regarding the first question, the authors define the abstraction levels to which the diversifying transformations can be applied to:

1. Instruction level: these are transformations that involve small sequences of instructions, smaller than a basic block. For example, these may include replacing instructions with equivalent ones, reordering them, randomizing register usage and inserting `NOP` instructions;

2. Basic block level: as the name implies, these transformations operate on basic blocks. They include block reordering, insertion of opaque predicates and branching functions.

3. Loop level: these include modifying the condition of a loop, intersecting them and loop unrolling [11];

4. Function level: the transformations at function level include randomization of the stack layout, randomization of the function parameters, inlining, outlining, splitting and most importantly control flow flattening;

5. Program level: these are transformations targeting the program as a whole. They include reordering functions, randomizing the base address by means of Address Space Layout Randomization (ASLR), instruction set virtualization, randomization of library entry points. Moreover, a set of data randomization techniques are supported, such as constant blinding, randomization of the layout structure.

6. System level: these target the whole system running the program. It is possible, for example, to diversify the internal interfaces of the operating system.

It should be pointed out that many of these techniques correspond to obfuscating transformations. This shows that Obfuscation can be considered a way to achieve Diversification.

For the second question, the authors identify the following stages of the lifetime of a program as possible targets of diversification:

1. Implementation: diversification at this stage is characterized by the development of multiple versions of the same feature, with obvious cost increases;

2. Compilation and Linking: applying diversification at the compilation stage has the advantage of not requiring rewriting the source code. This can be implemented as a sequence of transformation passes applied on an intermediate representation (IR) of the program;
3. Installation: consists in diversification being applied during or just after the installation of a program. It needs a disassembler to be able to modify the software;
4. Loading: the diversification is applied when the program is loaded. A prime example is ASLR, which was mentioned earlier;
5. Execution: diversification is achieved at runtime for example by randomizing heap data position, or by dynamically rewriting the binary;
6. Updating: this consists in delivering a different diversified version of the program at each update, with the advantage of making it more difficult for an attacker to transfer their knowledge obtained through reverse engineering to the new version.

2.3 Related Work

Plenty of research has been done on the topics of Obfuscation and Diversification. For the issue of Software Obfuscation, in "Stochastic optimization of program obfuscation" researchers have devised a Markov Chain Monte Carlo based method to find the optimal sequence of obfuscating transformations [14].

In "Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors", the authors propose an obfuscation scheme that operates both on hardware and software level, with the objective of protecting embedded systems from disclosure attacks [8].

Regarding Software Diversification, in "Internal interface diversification as a method against malware" the authors investigate the usage of diversification on the internal interfaces of the operating system, namely the system calls, library functions and the command line interpreter, for the purpose of defending from malware [23]. Similarly, in "Code Diversification Mechanisms for Internet of Things (Revised Version 2)" the authors discuss the feasibility of applying diversification in Internet of Things (IoT) environments, that are characterized by limited resources.

In "Constraint-based software diversification for efficient mitigation of code-reuse attacks", the trade-off between diversification and code quality was studied. The authors proposed a constraint-based method able to achieve diversification while respecting a set of quality requirements [29].

Additionally, the authors of "Composite software diversification" studied the effect of applying different diversification techniques to the same program and developed a procedure to find the best performing composition [30].

Chapter 3

Software Similarity

3.1 Definitions

Measuring Software Similarity has become a necessity for a wide range of disciplines. For example, one could use these kind of metrics to detect cloned programs and find instances of plagiarism [17], or it could be used to find software defects [18]. Moreover, similarity is also useful in the context of software obfuscation and diversification: by measuring how similar two versions of a program are, it is possible to optimize the obfuscation configuration such that the distance between the original program and its obfuscated version is maximized. In the context of diversity, similarity can be used to search for a set of diversified versions of a program so that for each one its distance from all the others is maximized [1].

An attacker could also benefit from similarity metrics, for example to easily identify changes in the code of a program after an update.

According to the article "A comparison of code similarity analysers" by Ragkhitwetsagul, Krinke, and Clark, the Software Similarity metrics can be classified in the following categories: metrics-based, text-based, token-based, tree-based and graph-based.

Metric-based approaches are based on software metrics such as Halstead [9] and McCabe [16], but are regarded to be worse than the other types [22]. Text-based techniques work by comparing strings of code, while token-based approaches evolves the previous by transforming the code into a sequence of tokens that represent an abstraction level over the code.

Tree-based and Graph-based algorithms, as the name implies, use tree and graph structures generated from the program to compute the similarity. Such structures include Abstract Syntax Trees (AST), Program Dependency Graphs (PDGs) and Control Flow Graphs (CFGs).

3.2 Metrics

3.2.1 Normalized Compression Distance

The Normalized Compression Distance is a compression based distance algorithm. The idea behind this metric is using a compression algorithm to compress the concatenation of two sequences of bytes b_1 and b_2 and comparing its size with the singularly compressed b_1 and b_2. Ideally, if the two byte sequences contain common information, the compressed concatenation will be smaller than the sum of b_1 and b_2 compressed separately [7].

Formally, the NCD similarity metric is defined as:

$$S(b_1, b_2) = 1 - NCD(b_1, b_2) = 1 - \frac{C(b_1 b_2) - min(C(b_1), C(b_2))}{max(C(b_1), C(b_2))}$$

where C is a compression algorithm. The choice of the compression algorithm can greatly influence the result, since the size of the history buffer varies. For example, according to the Linux manpage of `rzip`, the buffer for `gzip`, `bzip2` and `rzip` is $32kB$, $900kB$ and $900MB$ respectively [25]. Additionally, in "Search based clustering for protecting software with diversified updates" by Ceccato et al. the authors show that the idem-potency property (i.e. $NCD(b, b) = 0$) for `rzip` does not hold with files larger than $448MB$, meaning that the metric would become unreliable with larger data [1].

In chapter 5 the three aforementioned compression algorithms will be tested against the same programs to understand how they compare against each other.

3.2.2 Longest Common Subsequence

The Longest Common Subsequence algorithm, finds, given two sequences $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_m)$, the longest subsequence present in both sequences. Intuitively, a subsequence of A is a sequence of elements that belong to A and preserve the same order in which they appear in A. [6]

For example, given $A = (1, 3, 4, 6, 8, 9)$ and $B = (0, 2, 3, 6, 7, 9)$, the sequence $S_1 = (1, 3, 9)$ is a subsequence of A but not B, $S_2 = (3, 6)$ is a common subsequence but not the longest, while $S_3 = (3, 6, 9)$ is the longest common subsequence between A and B.

When dealing with the problem of finding the longest common subsequence between two code functions, an element in a sequence represents an indivisible line of code.

The length of a LCS can be used to build a similarity metric [20]. Simply, the similarity between two functions, represented as sequences of textual lines of code l_1 and l_2, is given by double the length of the longest common substring

between l_1 and l_2, divided by the sum of the length of the two sequences (as in the number of lines of code). Formally, the score is given by [26]:

$$S(l_1, l_2) = \frac{|LCS(l_1, l_2)| * 2.0}{|l_1| + |l_2|}$$

3.2.3 Opcode Frequency

The following similarity metric has been proposed in the paper "Metamorphic virus detection in Portable Executables using opcodes statistical feature" by Rad and Masrom with the purpose of detecting metamorphic viruses [21]. According to the authors, this technique should resist obfuscation techniques such as garbage code insertion, register usage exchange, instruction replacement and permutation.

The idea behind this metric to compute the similarity between two programs is the opcode frequency histogram. This kind of histogram counts occurrences of each opcode (without operands) in a function. An example is shown in figure 3.1.

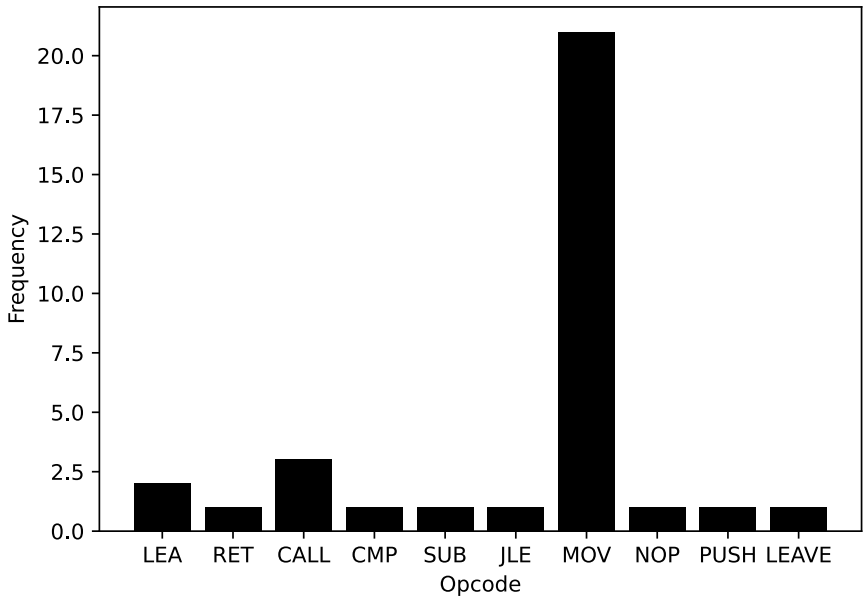

Figure 3.1: Opcode Histogram - Quicksort function

To compute the similarity between programs P_1 and P_2, these histograms are generated for every function in the two programs, normalized, and then pair-

wise compared to find, for each function in the first program, the best matching function in the second. The histogram distance is computed with the Euclidian form distance:

$$d(x, y) = \sum_{i=1}^{n} (x_i - y_i)^2$$

where x and y are respectively histograms in P_1 and P_2, while x_i and y_i are the frequencies in x and y of the same opcode. The similarity is computed as the average of the distances of the best matches.

Finally, since P_1 and P_2 might differ in the number of functions and the algorithm searches the best match in P_2 for each function in P_1, it is possible that the symmetry property does not hold. To improve the result, the final similarity is computed as:

$$S\{P_1, P_2\} = \frac{S(P_1, P_2) + S(P_2, P_1)}{2}$$

meaning that the algorithm is invoked twice by swapping the two programs, and then the final score is computed as the average of the two results.

3.2.4 ROP Gadget Survival

Return-Oriented Programming is an exploitation technique in which an attacker can redirect control flow without having to inject code. Such an attack is constructed by combining in a chain a list of so called *gadgets*. A gadget is a short sequence of instructions found in the program that ends in a return statement. By exploiting the code already present in the program, this technique defeats the $W \oplus X$ protection, which forbids execution of code in memory regions not marked as executable [24].

In "ROP gadget prevalence and survival under compiler-based binary diversification schemes" by Coffman et al., the authors investigate the survival of gadgets across diversified programs, which can be used by an attacker to build a common exploit for all the variants [2]. The percentage of surviving gadgets could be interpreted as a measure of similarity.

To compute the gadget survival percentage, the authors propose two metrics:

- Survivor: this metric, originally introduced in "Profile-guided automated software diversity" by Homescu et al., considers a gadget as a surviving one if it has the same sequence of bytes and its location, as an offset from the base address, is the same [10];
- Bag of Gadgets: this metric drops the requirement of having the location in memory to match and considers a gadget as a survivor if it simply has the same sequence of bytes, regardless of its location in the program.

While this is not sufficient information to build a common gadget chain, the authors state that this would be applicable in case an attacker is able to discover gadgets at runtime (JIT-ROP).

3.2.5 Code Abstraction

The following method, described in "Detecting source code similarity using code abstraction" by Park et al., is not a similarity metric but rather a technique to abstract the source code of a program to improve the results of the actual similarity metrics, while also improving execution times thanks to the reduced size of the resulting code [19].

The idea is to delete lines of code that are more easily targeted by obfuscating transformations, while keeping or simplifying the parts that are more difficult to obfuscate. This is achieved by following a set of abstraction rules:

1. Comments: single and multiple line comments are deleted;
2. Variable Declaration: all variable declarations are removed, since they can easily be renamed or moved around;
3. Strings: any kind of string is replaced with an empty string (i.e. `""`);
4. Expressions: expression statements are deleted, except when located inside a function call;
5. Return: all return statements are removed;
6. Function Calls: deletes local function calls, which are easier to obfuscate. External and system calls are instead kept, since those are harder to obfuscate;
7. Conditional structures: `if`, `else` and `switch` statements are simplified by removing their boolean conditions, since those are easy to obfuscate. Only the structure is kept;
8. Loop structures: similarly, `for`, `while` and `do-while` statements are simplified by removing their boolean conditions, while their structure is maintained.

Listings 3.1 and 3.2 show how the abstraction algorithm perform on the main function of a quicksort implementation: the comment and the variable assignments are deleted with the exception of the array, since it is assigned with a call to an external function. The strings are replaced with empty counterparts, the condition in the `for` loop is erased and the local function calls to `display` and

quicksort are removed. The calls to printf, scanf, malloc, getchar and free are kept since their implementation is not in the program. Finally, the return statement is removed.

The resulting code in listing 3.2 appears much simpler. If this process is repeated on the obfuscated version of the same function, the comparison of the two abstracted versions through a similarity metric will presumably result in a higher score. This is investigated in chapter 5.

Listing 3.1: Original

```
// quicksort main
int main()
{
  int n;
  printf("Size:");
  scanf("%d", &n);

  printf("Elements:");
  int i;
  int *arr =
    malloc(sizeof(int) * n);
  for (i = 0; i < n; i++)
  {
    scanf("%d", &arr[i]);
  }

  printf("Original: ");
  display(arr, n);

  quickSort(arr, 0, n - 1);

  printf("Sorted: ");
  display(arr, n);
  getchar();
  free(arr);
  return 0;
}
```

Listing 3.2: Abstracted

```
int main()
{

  printf("");
  scanf("", &n);

  printf("");

  int *arr =
    malloc(sizeof(int) * n);
  for (;;)
  {
    scanf("", &arr[i]);
  }

  printf("");




  printf("");

  getchar();
  free(arr);

}
```

3.3 Other Metrics

Here are mentioned some additional metrics that have been proposed in the literature but are not object of experimentation in this work.

In "Common program similarity metric method for anti-obfuscation", the authors define a similarity metric based on the Reductive Instruction Dependence Graph (RIDG). This graph structure represents the transitive reduction of a Instruction Dependency Graph, and according to the authors it should be resistant to code obfuscations [32].

The paper "A similarity metric method of obfuscated malware using function-call graph" describes a technique based on the comparison of Function Call Graphs. Additionally, their algorithm tries to match functions according to other informations, such as the usage of external functions and the similarity of the opcode used in the functions [31].

In "A deep learning approach to program similarity", the researchers compute the similarity between binaries by representing them as images and using a Deep Learning image classification model [15].

Chapter 4

Project

The purpose of this chapter is to present the project developed to test the Software Similarity metrics described in the previous chapter. The software has been published on GitHub[1].

4.1 Introduction

The objective of this project is to implement a set of similarity metrics and test them against a range of obfuscated binaries, to understand which metric counters the effect of the obfuscating transformations better. The project is structured as follows:

1. `src`: This folder contains the set of scripts that compute the similarity metrics, the code that wraps Ghidra's capabilities and other helper libraries. Its content will be described in section 4.2;

2. `test`: This folder contains everything necessary to automate the generation of each obfuscated sample and to automatically test them against every similarity metrics. It will be described in section 4.3;

3. `tools`: Finally, this folder simply contains the tools that are used by the scripts in `test`, namely Ghidra, Tigress and Obfuscator-LLVM.

[1] `https://github.com/GobboJ/GhidraSimilarity`

4.2 Similarity scripts

4.2.1 Tools

Ghidra

The similarity metrics, with the exception of one, are implemented as Ghidra scripts. Ghidra[2] is a reverse-engineering tool built by the National Security Agency of the United States of America and publicly released in 2019 on GitHub. This software offers the capability to disassemble and decompile a program, which are fundamental to compute the metrics.

While Ghidra's core is written in C++, the frontend and scripting system are built in Java. Ghidra provides an extensive Java API to process binaries and interact with their content. Moreover, this API is also accessible through Python thanks to Jython, an implementation of Python in Java that runs on the JVM and allows to access classes written in Java.

This project's scripts are written in Python and are invoked by using the headless Ghidra analyzer interface as follows:

```
./analyzeHeadless <project_folder> <project>
    -scriptpath <script_folder>
    -postscript <script> [<script_parameters>]*
```

where:

- `<project_folder>`: The folder containing the Ghidra project;
- `<project>`: The name of the project to be processed;
- `<script_folder>`: The folder containing the similarity metrics scripts;
- `<script>`: The Python similarity script to be executed;
- `[<script_parameters>]*`: a list of parameters to be passed to the invoked script. Each script has its own different set of parameters.

A Ghidra project can be either created using the graphical front-end or by invoking the same headless analyzer:

```
./analyzeHeadless <project_folder> <project> -import <samples>
```

where, as before, `<project_folder>` and `<project>` indicate respectively the path and name of the project to be created, while `<samples>` is the path of the executables to be imported and analyzed.

[2] https://github.com/NationalSecurityAgency/ghidra

ROPgadget

ROPgadget[3] is another useful tool for this project, necessary to compute the last similarity metric. ROPgadget's purpose is to find gadgets in a binary to build a Return Oriented Programming (ROP) attack. It is built in Python and internally uses the Capstone[4] disassembly engine. ROPgadget can be invoked like this:

```
ROPgadget --binary <file>
```

with `<file>` being the binary to be analyzed. Many parameters are supported, such as `--all` to allow duplicate gadgets, `--ropchain` to try and build a ROP chain with the detected gadgets and `--depth <nbytes>` to set the maximum depth search size. An example of output is reported in listing 4.1.

4.2.2 Scripts

Normalized Compression Distance Similarity

The `compression_similarity.py` script implements the Normalized Compression Distance similarity algorithm as described in chapter 3. It can be invoked by the Ghidra headless analyzer as follows:

```
./analyzeHeadless <project_folder> <project>
    -scriptpath <script_folder>
    -postscript compression_similarity.py <algorithm>
        <abstraction_level> <reference_program> <csv>
```

where `<project_folder>`, `<project>` and `<script_folder>` indicate the folder of the project, the name of the project and the folder containing the script. More importantly, the script has the following parameters:

- `<algorithm>`: specifies which compression algorithm should be used;
- `<abstraction_level>`: specifies which abstraction level of the programs should be used;
- `<reference_program>`: indicates which of the programs in the project is to be considered as reference. This implies the reference being compared against all other samples;
- `<csv>`: indicates the path of the output csv file containing the results of the comparisons.

[3] https://github.com/JonathanSalwan/ROPgadget
[4] https://www.capstone-engine.org/

Listing 4.1: Output of ROPgadget

```
jonathan@ryzen ~> ROPgadget --binary ls
Gadgets information
============================================================
0x0000000000402177 : adc al, 0 ; add byte ptr [rax], al ; jmp
   0x402020
0x0000000000413274 : adc al, 0 ; add byte ptr [rax], al ; jmp
   0x413551
0x000000000040feac : adc al, 0x24 ; mov rsi, qword ptr [r13 +
   0x10] ; mov rdi, r14 ; call qword ptr [r13 + 0x30]
0x0000000000409bdd : adc al, 0x34 ; add rbp, rsi ; jmp 0
   x409c86
0x00000000004164ba : adc al, 0x45 ; xor ecx, ecx ; jmp 0
   x416651
0x00000000004147b8 : adc al, 0x8e ; mov ebx, r12d ; je 0
   x4147dc ; jmp 0x414880
...
0x00000000004164bb : xor r9d, r9d ; jmp 0x416651
0x0000000000412966 : xor r9d, r9d ; mov r8d, dword ptr [rsp +
   0x28] ; jmp 0x412a3b
0x000000000040f768 : xorps xmm0, xmm0 ; cvtsi2ss xmm0, r12 ;
   jmp 0x40f78b
0x000000000041035d : xorps xmm0, xmm0 ; cvtsi2ss xmm0, rcx ;
   jmp 0x41037f
0x00000000004103ff : xorps xmm2, xmm2 ; jmp 0x410409

Unique gadgets found: 6802
```

The supported compression algorithms are three: `gzip`, `bz2` and `rzip`. The first two are provided by the Python 2.7 standard library[5], while the third is implemented as a wrapper over the Linux system package `rzip`.

The abstraction level influences what the algorithm will compress:

- `bytes`: the algorithm compresses the entire binary files;
- `opcode_bytes`: the algorithm compresses only the concatenation of bytes of the functions detected by Ghidra;
- `opcode_text`: the algorithm compresses the concatenation of the textual mnemonic representation of the disassembled function detected by Ghidra;
- `decompiled`: the algorithm compresses the concatenation of the decompiled listing of every function;
- `simplified`: like the previous, but the decompiled code is simplified according to the code abstraction algorithm described in chapter 3.

Example The listing 4.2 contains the output of an invocation of the algorithm, where the `rzip` algorithm is used paired with the `bytes` abstraction level. In the example it is comparing the clean quicksort sample (`00_quicksort`) with a series of obfuscated variants.

Longest Common Subsequence Similarity

The `lcs_similarity.py` script provides the Longest Common Subsequence based similarity metric, as seen in chapter 3. Similarly to the previous algorithm, it can be invoked as:

```
./analyzeHeadless <project_folder> <project>
    -scriptpath <script_folder>
    -postscript lcs_similarity.py <abstraction_level>
        <reference_program> <csv>
```

with the meaning of the parameters being the same as the previous one.

Being this a text-based metric, it offers different abstraction levels:

- `opcode`: compares the textual mnemonic representation of the disassembled functions;
- `opcode_no_param`: like the previous, but it only considers the opcode without the operands;

[5] https://docs.python.org/2.7/library/archiving.html

Listing 4.2: NCD Similarity - Output Example

```
############################################################
NCD similarity metric
Arguments: Namespace(abstraction_level=u'bytes', algorithm=u'
   rzip', csv=u'out.csv', reference_program=u'00_quicksort')
############################################################
[!] Similarity(00_quicksort, 00_quicksort) = 0.995352651722
[!] Similarity(00_quicksort, 01_quicksort) = 0.729608381142
[!] Similarity(00_quicksort, 02_quicksort) = 0.52510422331
[!] Similarity(00_quicksort, 03_quicksort) = 0.751250329034
[!] Similarity(00_quicksort, 04_quicksort) = 0.455323193916
[!] Similarity(00_quicksort, 10_quicksort) = 0.569436850738
[!] Similarity(00_quicksort, 11_quicksort) = 0.533656297807
[!] Similarity(00_quicksort, 12_quicksort) = 0.576377523186
[!] Similarity(00_quicksort, 13_quicksort) = 0.491253294992
[!] Similarity(00_quicksort, 14_quicksort) = 0.411068000817
```

- `decompile`: compares the decompiled functions;

- `simplified`: like the previous, but the function are simplified according to the abstraction algorithm as seen on chapter 3.

The implementation of the Longest Common Subsequence algorithm is provided by Ghidra with the `ReducingListBasedLcs`[6] class. This is an optimized version of LCS, whose complexity, according to the documentation, is $O(n^2)$.

Example The listing 4.3 is an example of invocation of the LCS similarity metric, where the abstraction level is set to `decompile` and the reference program to be compared against all others is `00_quicksort`. Unlike the previous metric, this one compares each pair of functions and for each one in the reference program it prints the best matching function in the second program. The similarity score is the average of the similarity of the matches. In this example, every function was matched to the correct counterpart.

[6] https://ghidra.re/ghidra_docs/api/generic/algorithms/ReducingListBasedLcs.html

Listing 4.3: LCS Similarity - Output Example

```
############################################################
LCS similarity metric
Arguments: Namespace(abstraction_level=u'decompile', csv=u'out
   .csv', reference_program=u'00_quicksort')
############################################################
...
============================================================
Comparing 00_quicksort, 12_quicksort
============================================================
[!] Match (register_tm_clones, register_tm_clones) = 1.0
[!] Match (_init, _init) = 1.0
[!] Match (_start, _start) = 1.0
[!] Match (swap, swap) = 1.0
[!] Match (_dl_relocate_static_pie, _dl_relocate_static_pie) =
    1.0
[!] Match (main, main) = 0.285714285714
[!] Match (quickSort, quickSort) = 1.0
[!] Match (deregister_tm_clones, deregister_tm_clones) = 1.0
[!] Match (.annobin_abi_note.c, .annobin_abi_note.c) = 1.0
[!] Match (partition, partition) = 0.4375
[!] Match (printArray, printArray) = 0.555555555556
[!] Match (__do_global_dtors_aux, __do_global_dtors_aux) = 1.0
[!] Match (_fini, _fini) = 1.0
[!] Match (FUN_00401020, FUN_00401020) = 1.0
[!] Similarity(00_quicksort, 12_quicksort) = 0.877054988662
```

Opcode Frequency Histogram Similarity

The Opcode Frequency Histogram similarity metric is implemented in the script named `opcode_frequency_similarity.py` and it is invoked as follow:

```
./analyzeHeadless <project_folder> <project>
    -scriptpath <script_folder>
    -postscript opcode_frequency_similarity.py
        <reference_program> <csv>
```

This script does not provide any customization, therefore it only requires to be given a reference program and the csv output file path.

Example The listing 4.4 contains an example of invocation of the opcode similarity metric. Like the previous one, it prints the best matching functions and the final average similarity score.

ROP Gadget Survival

Finally, the `rop_similarity.py` script provides the ROP Gadget Survival metric. Contrary to the previous scripts, this one does not use Ghidra. Instead, it uses ROPgadget and it is invoked as a standard python script:

```
python3 rop_similarity.py <folder> <metric>
    <reference_program> <csv>
```

where:

- `<folder>`: The path of the folder containing the samples to be processed;
- `<metric>`: The chosen metric. Can be either `bag_of_gadgets` or `survivor`, which were explained in chapter 3;
- `<reference_program>`: The program to compare against all the others;
- `<csv>`: The path of the output csv file containing the similarity scores.

Example The example in listing 4.5 illustrates the output of the invocation of the ROP similarity script with the `bag_of_gadgets` metric. Contrary to the previous example, this one does not matches function but just returns the overall similarity between the binaries.

Listing 4.4: Opcode Frequency Similarity - Output Example

```
################################################################
Opcode frequency similarity metric
Arguments: Namespace(csv=u'out.csv', reference_program=u'00
   _quicksort')
################################################################
...
================================================================
Comparing 00_quicksort, 01_quicksort
================================================================
[!] Match (register_tm_clones, register_tm_clones) = 1.0
[!] Match (_init, _init) = 1.0
[!] Match (_start, _start) = 1.0
[!] Match (swap, swap) = 0.823438343375
[!] Match (_dl_relocate_static_pie, _dl_relocate_static_pie) =
    1.0
[!] Match (main, main) = 0.900000784927
[!] Match (quickSort, main) = 0.848005978466
[!] Match (deregister_tm_clones, deregister_tm_clones) = 1.0
[!] Match (.annobin_abi_note.c, .annobin_abi_note.c) = 1.0
[!] Match (partition, partition) = 0.757896647296
[!] Match (printArray, printArray) = 0.903711743504
[!] Match (__do_global_dtors_aux, __do_global_dtors_aux) = 1.0
[!] Match (_fini, _fini) = 1.0
[!] Match (FUN_00401020, FUN_00401020) = 1.0
...
[!] Similarity(00_quicksort, 01_quicksort) = 0.93912330845
```

Listing 4.5: ROP Similarity - Output Example

```
############################################################
ROP similarity metric
Arguments: Namespace(folder='samples/quicksort',
    metric='bag_of_gadgets', reference_sample='00_quicksort',
       csv='a.csv')
############################################################
[!] Loading gadgets from: 00_quicksort
[+] Binary loaded
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
[!] Similarity(00_quicksort, 00_quicksort) = 1.0
----------------------
[!] Loading gadgets from: 01_quicksort
[+] Binary loaded
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
[!] Similarity(00_quicksort, 01_quicksort) =
   0.6195652173913043
----------------------
[!] Loading gadgets from: 14_quicksort
[+] Binary loaded
[+] Loading gadgets, please wait...
[+] Gadgets loaded !
[!] Similarity(00_quicksort, 14_quicksort) =
   0.32608695652173914
```

Helper Libraries

Ghidra wrapper The `ghidra_wrapper.py` file contains a series of methods that wrap Ghidra capabilities to provide a simpler interface to be used by the similarity scripts. In particular, the following methods are provided:

1. `get_program_list(state)`: returns the list of every program in the project;
2. `find_reference_program(program_list, reference_program_name)`: returns a program given its name and a list of all the programs;
3. `get_program_bytes(program)`: given a program it returns its bytes;
4. `get_external_function_names(program)`: returns a list of names of external functions, meaning functions that do not have an implementation inside the program, such as syscalls and library functions;
5. `get_functions(program)`: returns the list of all the functions in a program that are not external;
6. `get_functions_bytes(program)`: returns the bytes of the concatenation of all non-external functions in the program;
7. `get_opcode_text(program)`: returns the bytes of the textual representation of the concatenation of every non-external function in the program;
8. `get_opcode_listings(program, remove_parameters=False)`: returns a list of all the disassembled functions in the program. If `remove_parameters` is `True`, the opcode operands are removed;
9. `get_histograms(program)`: returns a list of opcode frequency histogram of the functions in the program.

Additionally, `ghidra_wrapper.py` contains a class named `DecompilerWrapper` that provides the `get_decompiled_code(self, program, simplify=False)` method. This method returns a list of decompiled functions, and optionally applies the code abstraction algorithm described in chapter 3 if the parameter `simplify` is set to `True`. This class also takes care of caching the decompiled functions to file. This is useful because decompilation is an expensive operation in terms of time, and subsequent usage of the decompiled code by the similarity script benefits greatly from caching in terms of time.

Function	Target	Substitution
Multiline comment	`"\/\*[\w\s]*\*\/"`	`""`
Single line comment	`"\/\/[^\n]*$"`	`""`
Variable declaration	`"[\w ]+ [*\w]+(?:,[ *\w]*)*;"`	`""`
Array declaration	`"[\w ]+ [\w]+ ?\[\w*\];"`	`""`
Strings	`"\".*\""`	`"\"\""`
Return	`"return.*;"`	`""`
If	`"if ?\(.*\)"`	`"if ()"`
Switch	`"switch ?\(.* ?\)"`	`"switch ()"`
For	`"for ?\(.*;.*;.*\)"`	`"for (;;)"`
While	`"while ?\(.* ?\) ?{"`	`"while ()"`
Variable assignment	`".+ = (.*);"`	`keep_syscall`
Local function calls	`"([\w]+)\s*\(.*\);"`	`remove_local_calls`

Table 4.1: List of code abstraction rules

Simplify The `simplify.py` script provides an implementation of the code abstraction algorithm described in chapter 3. Its `simplify_code` function takes a decompiled function as a parameter and simplifies it according to the set of regular expressions shown in table 4.1.

Note that the last two rules, `keep_syscall` and `remove_local_calls` are functions that change behaviour according to the capturing group content. Specifically, the first checks if the variable assignment contains a system call. If it does, the assignment is kept, else it is removed. The second function instead checks whether a function call is external (either a system call or a library) or local. Local function calls are removed while the others are kept.

It should be pointed out that regular expression is not the optimal solution to this problem. Considering how much coding style can vary, finding a pattern that works as expected every time is hard. Fortunately, Ghidra's decompiler is consistent with the code style, allowing these regular expressions to achieve their purpose, even though they might not work correctly for arbitrarily styled code.

Other Scripts The last two python scripts, `printing.py` and `csv_writer.py`, simply provide a common way to print information about the similarity metrics and to write the results to a csv file.

4.3 Sample Generation

4.3.1 Tools

To generate the obfuscated samples, two obfuscators were used: Tigress[7] and Obfuscator-LLVM[8].

Tigress

Tigress is a C obfuscator built in OCaml developed by Collberg [3], that supports a set of obfuscating transformations that can be combined into an obfuscation script. Tigress is source-to-source, meaning that given a source file it will produce another obfuscated source file. An important limitation of Tigress is that it can only be applied to single file programs. While it is possible to merge source files into a single one, this is infeasible for larger projects, especially for those with complex build systems.

Example An example of a Tigress script is shown in listing 4.6. The script is composed as a sequence of transformations that are applied in the specified order. For each transformation it specifies to which functions they should be applied to, and additionally other transformation specific parameters. This particular script applies Opaque Predicates, Branch Functions, and Encoded Arithmetic transformations to the `test.c` program, targeting in particular the `fac` and `fib` functions.

Obfuscator-LLVM

Obfuscator-LLVM, developed by Junod et al., is an obfuscator implemented on top of LLVM, it applies its transformations on an intermediate representation of the program during compilation, and as such does not produce an obfuscated source code file [12]. However, contrary to Tigress, it is possible to apply it to programs with more than one source file and complex build systems.

To apply Obfuscator-LLVM's obfuscations, it is possible to invoke it as:

```
./<path-to-ollvm>/clang <source> -o <out> [<transformations>]
```

with `<path-to-ollvm>` being the path were Obfuscator-LLVM is installed, `<source>` and `<out>` respectively being the input source file and the output binary file, and `[<transformations>]` being a list of transformations to be applied.

[7] https://tigress.wtf/
[8] https://github.com/obfuscator-llvm/obfuscator

Listing 4.6: Tigress - Script Example

```
tigress --Seed=42 --Statistics=0 --Verbosity=0 \
    --Environment=x86_64:Linux:Gcc:4.6  \
 --Transform=InitEntropy \
    --Functions=init_tigress \
    --InitEntropyKinds=vars \
 --Transform=InitOpaque \
    --Functions=init_tigress \
    --InitOpaqueStructs=list,array,env  \
 --Transform=InitBranchFuns \
    --InitBranchFunsCount=1 \
 --Transform=AddOpaque \
    --Functions=fac,fib \
    --AddOpaqueStructs=list \
    --AddOpaqueKinds=true \
 --Transform=AntiBranchAnalysis \
   --Functions=fac,fib \
   --AntiBranchAnalysisKinds=branchFuns \
   --AntiBranchAnalysisObfuscateBranchFunCall=false \
   --AntiBranchAnalysisBranchFunFlatten=true \
 --Transform=EncodeArithmetic \
    --Functions=fac,fib  \
test.c --out=output1.c
```

Specifically, Obfuscator-LLVM supports the following transformations:

- `-mllvm -fla`: Control Flow Flattening, which flattens the control flow of the program;
- `-mllvm -sub`: Instruction Substitution, that replaces additions, subtraction, boolean operators with more complex computations;
- `-mllvm -bcf`: Bogus Control Flow, which encapsulates basic blocks in opaque predicates.

These also support a few parameters, mostly to set the number of iterations for which the transformation should be applied, or to set the probability of it to be applied.

In this project, Obfuscator-LLVM is used to obfuscate some real-world samples, like the GNU Coreutils[9]. The build system of Coreutils is Makefile, so to compile it, it is necessary to set the environment variables

```
export CC="<path-to-ollvm>/clang"
export CFLAGS+="-mllvm -fla -mllvm -sub -mllvm -bcf"
```

before issueing the `./configure` and `make` commands. This will ensure that `make` will compile the project using Obfuscator-LLVM.

4.3.2 Configuration files

The generation of the obfuscated samples and the automatic testing with every similarity metric is described by three `json` files:
`programs.json`, `obfuscations.json` and `measurements.json`.

Programs the `program.json` file describes which programs are to be obfuscated with Tigress and Obfuscator-LLVM.

For each program to be processed by Tigress, it specifies the name to be used as a base to name every generated sample, the source file to be obfuscated and the list of functions that should be targeted by the transformations. For the programs to be obfuscated by Obfuscator-LLVM (only coreutils), it instead specifies the name, folder and the list of programs to be targeted in the Coreutils suite.

The entire content of the file is shown in listing 4.7.

[9] https://www.gnu.org/software/coreutils/

Listing 4.7: programs.json

```
{
  "tigress-programs": [
    {
      "name": "dijkstra",
      "filename": "dijkstra.c",
      "functions": "enqueue,dequeue,queue_has_something,
         dijkstra"
    },
    {
      "name": "quicksort",
      "filename": "quicksort.c",
      "functions": "swap,partition,quickSort,display"
    },
    {
      "name": "hash_blake2b",
      "filename": "hash_blake2b.c",
      "functions": "test,assert_bytes,blake2b,BLAKE2B,F,G"
    }
  ],

  "obfuscator-llvm-programs": [
    {
      "name": "coreutils",
      "folder": "coreutils",
      "executables": [
        "base64", "chown", "dd", "sha256sum"
      ]
    }
  ]
}
```

Obfuscations the `obfuscations.json` file contains every single obfuscation configuration to be applied to the programs.

The example in listing 4.8 has been shortened for brevity. For every obfuscation configuration are specified the name to be appended to the program name, and the parameters that should be specified when invoking Tigress or Obfuscator-LLVM. The parameters in curly brackets are templates to be populated by the scripts described later.

Measurements the last configuration file is `measurements.json`.

The example shown in listing 4.9 is shortened for brevity. For every similarity metric script, this `json` file lists its name, the parameters to pass to the Ghidra headless analyzer or to the python interpreter, and a list of configurations, if the script supports them.

4.3.3 Scripts

Finally, here are described the scripts that generate and test the samples according to the `json` configuration files. There are four scripts:

1. `generate_samples.py`: applies every obfuscation configuration listed in `obfuscations.json` to each program in `programs.json` by invoking Tigress and Obfuscator-LLVM. The resulting executables are moved to the `samples` folder, while the obfuscated source files generated by Tigress are moved into the folder `sources/obfuscated`;

2. `generate_ghidra_projects.py`: for each program in `programs.json`, grabs the corresponding executables from `samples` and generates a Ghidra project inside the `projects` folder;

3. `run_tests.py`: for each program in `programs.json`, invokes every measurement configuration specified in `measurements.json`, and outputs the csv files in the `output` folder;

4. `clean_all.py`: simply removes any file generated by the previous scripts.

Listing 4.8: obfuscations.json

```
{
  "tigress-obfuscations": [
    {
      "name": "01_flatten",
      "params": "--Environment=x86_64:Linux:Gcc:4.6
        --Transform=Flatten --FlattenDispatch=switch
        --Functions={functions} --out={output_file}
        {input_file}"
    },
    {
      "name": "02_split",
      "params": "--Environment=x86_64:Linux:Gcc:4.6
        --Transform=Split --SplitCount=10
        --Functions={functions} --out={output_file}
        {input_file}"
    },
    {
      "name": "03_merge",
      "params": "--Environment=x86_64:Linux:Gcc:4.6
        --Transform=Merge --Functions={functions}
        --out={output_file} {input_file}"
    }, ...
  ],
  "obfuscator-llvm": [
    {
      "name": "01_control_flow_flattening",
      "params": "-mllvm -fla"
    },
    {
      "name": "02_instruction_substitution",
      "params": "-mllvm -sub"
    }, ...
  ]
}
```

Listing 4.9: measurements.json

```
{
  "ghidra_scripts": [
    {
      "name": "ncd",
      "call": "{folder} {project} -scriptpath {script_folder}
         -postscript compression_similarity.py {parameters}
        {reference} {csv}",
      "configs": [
        {
          "title": "gzip_bytes",
          "params": "gzip bytes"
        },
        {
          "title": "gzip_opcode_bytes",
          "params": "gzip opcode_bytes"
        }, ...
      ]
    }, ...
  ],
  "python_scripts": [
    {
      "name": "rop",
      "call": "rop_similarity.py {folder} {parameters} {
         reference} {csv}",
      "configs": [
        {
          "title": "bag_of_gadgets",
          "params": "bag_of_gadgets"
        }, ...
      ]
    }
  ]
}
```

Chapter 5

Experimental Results

5.1 Setup

To study the capabilities of the similarity metrics, seven programs were tested. These programs are listed in table 5.1, and for each one of them is specified the size of the unobfuscated binary and the number of functions contained according to the Ghidra analyzer. Note that the number of functions may include external or compiler generated functions, therefore this value does not represent the actual number of functions contained in the source code, but it still can provide an estimation of the size of the program.

The first three programs are `dijkstra.c`, `hash_blake2b.c` and `quicksort.c` [27], whose code is reported in appendix A, and are obfuscated with the Tigress obfuscator. Table 5.2 lists, for each one of them, the functions target of the obfuscations.

The other four programs, `base64`, `chown`, `dd`, `sha256sum` are very common command line tools part of the GNU Coreutils project. These are more complex programs, considerably larger and with many more functions, meant to represent a *real-world* scenario, that will be obfuscated with Obfuscator-LLVM.

Table 5.3 lists every obfuscation configuration applied to the programs. In particular, T_i are the Tigress transformations, while O_i represent the Obfuscator-LLVM ones. Moreover, T_0 and O_0 mean that no transformation is applied and the unobfuscated program is compared with itself. Finally, T_8 and O_4 are configurations that combine multiple transformations.

All the programs listed in table 5.1 are obfuscated with every obfuscation configuration detailed in table 5.3, and are then measured using every possible configuration of parameters of the similarity metrics described in chapter 4.

Program	# Functions	Size
Dijkstra	33	25KiB
Hash_blake2b	39	30KiB
Quicksort	41	25KiB
Base64	252	150KiB
Chown	362	245KiB
DD	367	453KiB
Sha256sum	272	184KiB

Table 5.1: List of the tested programs

Program	Target Functions
Dijkstra	enqueue,dequeue,queue_has_something,dijkstra
Hash_blake2b	swap,partition,quickSort,display
Quicksort	test,assert_bytes,blake2b,BLAKE2B,F,G

Table 5.2: Target functions for Tigress obfuscated programs

	Obfuscation
T_0	None
T_1	Flattening
T_2	Split
T_3	Merge
T_4	Opaque Predicate
T_5	Encode Literals
T_6	Encode Arithmetics
T_7	Randomize Args
T_8	Recipe: Opaque Predicates, Branch Functions, and Encoded Arithmetic
O_0	None
O_1	Flattening
O_2	Instruction Substitution
O_3	Bogus Control Flow
O_4	Mixed: Flattening, Instruction Substitution and Bogus Control Flow

Table 5.3: Obfuscation configurations

5.2 Results

5.2.1 Normalized Compression Distance Similarity

Best Compression Algorithm

The Normalized Compression Distance similarity metric depends on the choice of two parameters: the abstraction level and the compression algorithm. Tables 5.4, 5.5 and 5.6 highlight how the compression algorithms compare for each abstraction level. The results for the identical comparisons (T_0 and O_0) confirm what was described in chapter 3: the larger size of the history buffer allows `rzip` to almost consistently achieve the better result out of the three, usually scoring or nearing 100% similarity. The `gzip` algorithm seems to generally score better results than `bz2` for the smaller samples, while it fails when dealing with the larger ones. For all the other transformations, the scores of the three compression algorithm tend to be closer, with `rzip` generally scoring marginally better.

Given its higher reliability, all the subsequent tables featuring NCD will assume to be using the `rzip` algorithm.

Results

Tables 5.7 and 5.8 show how the `rzip`-based NCD similarity metric performs on different abstraction levels of the programs. It can be observed that the `bytes` abstraction layer consistently achieves a better result against `opcode_bytes` across all tests, both for Tigress and Obfuscator-LLVM samples. This can be explained by the fact that the second only considers bytes in detected functions, while the first involves every byte of the program, including data. Since data is not involved in the transformations, it will resist the transformations, improving the results when using the `bytes` abstraction layer. The `opcode_text` representation marginally improves the result of `opcode_bytes`, which could be explained by the fact that the textual representation of the disassembled code is bigger in size and allows groups of letters to be compressed. Once again, the `decompiled` abstraction layer improves the results of `opcode_text` across all tests. When compared against `bytes`, it manages to do slightly better for the Obfuscator-LLVM samples, while it falls behind when considering the Tigress obfuscated programs. Finally, the `simplified` representation achieves its purpose by improving the results of the `decompiled` abstraction level. In particular, for the Tigress samples it consistently achieves better results than `decompiled`, while also performing better than `bytes` in some cases. When dealing with the Obfuscator-LLVM test cases, `simplified` performs better than any other abstraction level across every single test.

		Dijkstra			Hash-blake2b			Quicksort		
		bz2	gzip	rzip	bz2	gzip	rzip	bz2	gzip	rzip
Bytes	T_0	0.84	0.93	1.0	0.85	0.95	1.0	0.85	0.93	1.0
	T_1	0.59	0.6	0.68	0.32	0.12	0.36	0.64	0.65	0.73
	T_2	0.5	0.51	0.58	0.17	0.04	0.2	0.5	0.5	0.57
	T_3	0.63	0.65	0.73	0.31	0.1	0.35	0.66	0.67	0.75
	T_4	0.48	0.45	0.53	0.32	0.09	0.34	0.51	0.48	0.56
	T_5	0.5	0.45	0.53	0.34	0.07	0.37	0.52	0.48	0.55
	T_6	0.57	0.58	0.65	0.31	0.08	0.34	0.64	0.66	0.73
	T_7	0.62	0.63	0.71	0.36	0.11	0.39	0.66	0.68	0.75
	T_8	0.4	0.38	0.45	0.21	0.04	0.24	0.48	0.49	0.55
Op. Bytes	T_0	0.83	0.98	0.96	0.87	0.97	0.99	0.87	0.97	0.97
	T_1	0.41	0.43	0.45	0.15	0.11	0.17	0.38	0.4	0.47
	T_2	0.34	0.32	0.37	0.11	0.07	0.12	0.31	0.27	0.34
	T_3	0.42	0.45	0.46	0.11	0.06	0.13	0.36	0.38	0.43
	T_4	0.28	0.25	0.33	0.19	0.16	0.2	0.27	0.25	0.3
	T_5	0.33	0.29	0.36	0.22	0.2	0.25	0.35	0.36	0.37
	T_6	0.29	0.3	0.34	0.14	0.11	0.15	0.37	0.38	0.4
	T_7	0.44	0.46	0.52	0.18	0.14	0.2	0.41	0.44	0.51
	T_8	0.28	0.31	0.32	0.15	0.09	0.16	0.25	0.28	0.32
Op. Text	T_0	0.84	0.94	1.0	0.76	0.92	1.0	0.83	0.93	0.99
	T_1	0.53	0.53	0.52	0.18	0.15	0.19	0.52	0.51	0.54
	T_2	0.43	0.39	0.41	0.13	0.07	0.11	0.42	0.36	0.39
	T_3	0.53	0.53	0.56	0.13	0.1	0.14	0.5	0.48	0.5
	T_4	0.4	0.36	0.39	0.19	0.07	0.23	0.39	0.35	0.4
	T_5	0.42	0.37	0.38	0.22	0.05	0.27	0.49	0.43	0.51
	T_6	0.44	0.39	0.43	0.15	0.06	0.16	0.47	0.46	0.49
	T_7	0.54	0.54	0.56	0.2	0.1	0.22	0.57	0.52	0.56
	T_8	0.42	0.39	0.43	0.18	0.12	0.18	0.44	0.41	0.48

Table 5.4: Normalized Compression Distance
Compression algorithm comparison - Tigress samples

		Dijkstra			Hash-blake2b			Quicksort		
		bz2	gzip	rzip	bz2	gzip	rzip	bz2	gzip	rzip
Decompiled	T_0	0.82	0.96	0.98	0.79	0.95	0.99	0.84	0.96	0.98
	T_1	0.57	0.6	0.6	0.17	0.11	0.17	0.53	0.57	0.6
	T_2	0.49	0.46	0.47	0.14	0.07	0.15	0.44	0.38	0.39
	T_3	0.58	0.61	0.63	0.16	0.09	0.15	0.57	0.58	0.61
	T_4	0.49	0.55	0.54	0.2	0.15	0.22	0.44	0.47	0.45
	T_5	0.48	0.51	0.52	0.2	0.17	0.23	0.46	0.47	0.5
	T_6	0.52	0.57	0.56	0.18	0.12	0.18	0.54	0.6	0.6
	T_7	0.68	0.77	0.75	0.23	0.14	0.23	0.65	0.71	0.67
	T_8	0.56	0.6	0.61	0.18	0.18	0.22	0.55	0.59	0.59
Simplified	T_0	0.84	0.96	0.98	0.84	0.96	0.97	0.83	0.96	0.97
	T_1	0.65	0.7	0.66	0.56	0.53	0.51	0.63	0.71	0.65
	T_2	0.55	0.57	0.54	0.3	0.22	0.24	0.52	0.51	0.47
	T_3	0.68	0.73	0.71	0.61	0.58	0.61	0.65	0.72	0.71
	T_4	0.57	0.65	0.62	0.6	0.61	0.59	0.56	0.63	0.58
	T_5	0.59	0.65	0.62	0.54	0.58	0.59	0.59	0.65	0.62
	T_6	0.59	0.67	0.65	0.49	0.54	0.53	0.67	0.78	0.73
	T_7	0.72	0.81	0.76	0.69	0.68	0.65	0.73	0.81	0.73
	T_8	0.62	0.68	0.68	0.55	0.54	0.53	0.63	0.71	0.69

Table 5.5: Normalized Compression Distance
Compression algorithm comparison - Tigress samples (cont.)

		base64			chown			dd			sha256sum		
		bz2	gzip	rzip	bz2	gzip	rzip	bz2	gzip	rzip	bz2	gzip	rzip
Bytes	O_0	0.79	0.01	1.0	0.78	0.0	1.0	0.77	0.0	1.0	0.78	0.01	1.0
	O_1	0.02	0.0	0.04	0.01	0.0	0.02	0.01	0.0	0.02	0.03	0.0	0.04
	O_2	0.05	0.0	0.07	0.04	0.0	0.05	0.02	0.0	0.03	0.03	0.0	0.05
	O_3	0.05	0.0	0.06	0.02	0.0	0.04	0.01	0.0	0.02	0.04	0.01	0.06
	O_4	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.01
Op. Bytes	O_0	0.82	0.97	1.0	0.8	0.04	1.0	0.76	0.0	1.0	0.8	0.98	1.0
	O_1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	O_2	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.0
	O_3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	O_4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Op. Text	O_0	0.71	0.05	1.0	0.7	0.02	1.0	0.03	0.01	1.0	0.72	0.02	1.0
	O_1	0.0	0.0	0.01	0.0	0.0	0.0	0.0	0.0	0.01	0.0	0.0	0.0
	O_2	0.0	0.01	0.03	0.0	0.01	0.02	0.0	0.0	0.0	0.0	0.01	0.0
	O_3	0.0	0.0	0.02	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.02
	O_4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.0	0.0	0.0
Decomp.	O_0	0.77	0.04	1.0	0.75	0.02	1.0	0.8	0.01	1.0	0.77	0.02	1.0
	O_1	0.04	0.01	0.05	0.03	0.0	0.04	0.0	0.0	0.05	0.06	0.01	0.07
	O_2	0.2	0.02	0.2	0.14	0.01	0.15	0.11	0.0	0.11	0.07	0.01	0.08
	O_3	0.14	0.01	0.13	0.08	0.01	0.09	0.09	0.0	0.09	0.13	0.01	0.12
	O_4	0.02	0.0	0.03	0.0	0.0	0.02	0.02	0.0	0.02	0.02	0.0	0.02
Simpl.	O_0	0.76	0.92	1.0	0.75	0.1	1.0	0.77	0.02	0.99	0.77	0.92	1.0
	O_1	0.16	0.01	0.14	0.15	0.01	0.12	0.28	0.0	0.17	0.18	0.03	0.16
	O_2	0.29	0.22	0.28	0.23	0.07	0.22	0.19	0.01	0.18	0.29	0.22	0.27
	O_3	0.24	0.08	0.22	0.17	0.04	0.17	0.16	0.0	0.14	0.22	0.07	0.21
	O_4	0.12	0.0	0.08	0.08	0.0	0.06	0.21	0.01	0.08	0.09	0.01	0.09

Table 5.6: Normalized Compression Distance
Compression algorithm comparison - Obfuscator-LLVM samples

It should be pointed out that, except for the identical comparisons, the Normalized Compression Distance algorithm performed poorly on the Obfuscator-LLVM samples, with the best result being a similarity of 28% on the comparison between the clean sample of `base64` and its obfuscated variant O_2, using the `simplified` abstraction level.

5.2.2 Longest Common Subsequence Similarity

The results obtained for the Longest Common Subsequence similarity metric are shown in tables 5.9, 5.10, respectively for the samples generated with Tigress and Obfuscator-LLVM.

For all tests, the identical comparison achieved 100% similarity. It should be noted that initially this wasn't the case for the Obfuscator-LLVM samples, which, given their larger size, caused Ghidra's `ReducingListBasedLcs` implementation of LCS to reach the size limit for some comparisons, therefore returning empty subsequences. This was fixed by invoking `ReducingListBasedLcs`'s `setSizeLimit` method and setting the maximum size as the largest integer (i.e. $2^{31} - 1$). This change also required to increase Ghidra's maximum memory to $4GB$ in the `analyzeHeadless` file to avoid crashes.

When comparing the first two abstraction levels, `opcode` and `opcode_no_param`, the results show that the latter always performs better than the first. This can be easily explainable by the fact that removing the operands allows the same opcode to match even if they originally had different operands.

Regarding the `decompiled` representation, it performs better than `opcode` across all tests, while when compared against `opcode_no_param` it scores similarly for the Tigress-obfuscated samples, but falls behind when considering the samples obfuscated with Obfuscator-LLVM.

Finally, the `simplified` abstraction level again achieves its purpose of improving the score of the `decompiled` versions, scoring similarly to `opcode_no_param` for the Tigress samples, and surpassing it for the Obfuscator-LLVM ones.

Therefore, the best resulting abstraction levels for the Longest Common Subsequence algorithms are `simplified` and `opcode_no_param`.

5.2.3 Opcode Frequency Histograms

The results for the Opcode Frequency algorithm are listed in tables 5.11 and 5.12. This algorithm does not have any parameter to be set, therefore the tables simply contain the results achieved for each tested program with every defined obfuscation. Both Tigress-obfuscated and OLLVM-obfuscated samples achieved

		Bytes	Opcode Bytes	Opcode Text	Decompiled	Simplified
Dijkstra	T_0	1.0	0.96	1.0	0.98	0.98
	T_1	0.68	0.45	0.52	0.6	0.66
	T_2	0.58	0.37	0.41	0.47	0.54
	T_3	0.73	0.46	0.56	0.63	0.71
	T_4	0.53	0.33	0.39	0.54	0.62
	T_5	0.53	0.36	0.38	0.52	0.62
	T_6	0.65	0.34	0.43	0.56	0.65
	T_7	0.71	0.52	0.56	0.75	0.76
	T_8	0.45	0.32	0.43	0.61	0.68
Hash_blake2b	T_0	1.0	0.99	1.0	0.99	0.97
	T_1	0.36	0.17	0.19	0.17	0.51
	T_2	0.2	0.12	0.11	0.15	0.24
	T_3	0.35	0.13	0.14	0.15	0.61
	T_4	0.34	0.2	0.23	0.22	0.59
	T_5	0.37	0.25	0.27	0.23	0.59
	T_6	0.34	0.15	0.16	0.18	0.53
	T_7	0.39	0.2	0.22	0.23	0.65
	T_8	0.24	0.16	0.18	0.22	0.53
Quicksort	T_0	1.0	0.97	0.99	0.98	0.97
	T_1	0.73	0.47	0.54	0.6	0.65
	T_2	0.57	0.34	0.39	0.39	0.47
	T_3	0.75	0.43	0.5	0.61	0.71
	T_4	0.56	0.3	0.4	0.45	0.58
	T_5	0.55	0.37	0.51	0.5	0.62
	T_6	0.73	0.4	0.49	0.6	0.73
	T_7	0.75	0.51	0.56	0.67	0.73
	T_8	0.55	0.32	0.48	0.59	0.69

Table 5.7: Normalized Compression Distance - rzip
Tigress samples

		Bytes	Opcode Bytes	Opcode Text	Decompiled	Simplified
Base64	O_0	1.0	1.0	1.0	1.0	1.0
	O_1	0.04	0.0	0.01	0.05	0.14
	O_2	0.07	0.01	0.03	0.2	0.28
	O_3	0.06	0.0	0.02	0.13	0.22
	O_4	0.01	0.0	0.0	0.03	0.08
Chown	O_0	1.0	1.0	1.0	1.0	1.0
	O_1	0.02	0.0	0.0	0.04	0.12
	O_2	0.05	0.0	0.02	0.15	0.22
	O_3	0.04	0.0	0.0	0.09	0.17
	O_4	0.0	0.0	0.0	0.02	0.06
DD	O_0	1.0	1.0	1.0	1.0	0.99
	O_1	0.02	0.0	0.01	0.05	0.17
	O_2	0.03	0.0	0.0	0.11	0.18
	O_3	0.02	0.0	0.0	0.09	0.14
	O_4	0.01	0.0	0.02	0.02	0.08
Sha256sum	O_0	1.0	1.0	1.0	1.0	1.0
	O_1	0.04	0.0	0.0	0.07	0.16
	O_2	0.05	0.0	0.0	0.08	0.27
	O_3	0.06	0.0	0.02	0.12	0.21
	O_4	0.01	0.0	0.0	0.02	0.09

Table 5.8: Normalized Compression Distance - rzip
Obfuscator-LLVM samples

		Opcode	Op. no param	Decomp	Simplified
Dijkstra	T_0	1.0	1.0	1.0	1.0
	T_1	0.85	0.93	0.88	0.91
	T_2	0.87	0.96	0.91	0.97
	T_3	0.85	0.92	0.85	0.89
	T_4	0.74	0.88	0.82	0.87
	T_5	0.79	0.97	0.97	0.97
	T_6	0.81	0.88	0.88	0.93
	T_7	0.88	0.96	0.98	1.0
	T_8	0.76	0.93	0.83	0.87
Hash_blake2b	T_0	1.0	1.0	1.0	1.0
	T_1	0.65	0.89	0.72	0.81
	T_2	0.65	0.9	0.75	0.87
	T_3	0.59	0.84	0.7	0.79
	T_4	0.64	0.9	0.76	0.84
	T_5	0.75	0.95	0.86	0.93
	T_6	0.63	0.84	0.78	0.9
	T_7	0.7	0.96	0.8	0.92
	T_8	0.58	0.86	0.7	0.78
Quicksort	T_0	1.0	1.0	1.0	1.0
	T_1	0.83	0.92	0.81	0.89
	T_2	0.8	0.94	0.82	0.91
	T_3	0.76	0.85	0.81	0.87
	T_4	0.74	0.89	0.79	0.85
	T_5	0.83	0.98	0.93	0.94
	T_6	0.86	0.91	0.86	0.93
	T_7	0.83	0.98	0.85	0.93
	T_8	0.73	0.92	0.79	0.86

Table 5.9: Longest Common Subsequence
Tigress samples

		Opcode	Op. no Param	Decomp	Simplified
Base64	O_0	1.0	1.0	1.0	1.0
	O_1	0.2	0.62	0.4	0.65
	O_2	0.21	0.63	0.45	0.7
	O_3	0.2	0.62	0.42	0.67
	O_4	0.21	0.61	0.37	0.66
Chown	O_0	1.0	1.0	1.0	1.0
	O_1	0.19	0.59	0.35	0.62
	O_2	0.2	0.6	0.39	0.68
	O_3	0.2	0.59	0.37	0.66
	O_4	0.2	0.58	0.33	0.62
DD	O_0	1.0	1.0	1.0	1.0
	O_1	0.18	0.61	0.37	0.62
	O_2	0.19	0.61	0.41	0.67
	O_3	0.19	0.61	0.38	0.64
	O_4	0.2	0.61	0.34	0.62
Sha256sum	O_0	1.0	1.0	1.0	1.0
	O_1	0.19	0.62	0.39	0.66
	O_2	0.2	0.62	0.43	0.7
	O_3	0.2	0.62	0.4	0.68
	O_4	0.19	0.61	0.35	0.65

Table 5.10: Longest Common Subsequence
Obfuscator-LLVM samples

high similarity scores across all transformation configurations. These results will be compared against other metrics in section 5.3.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Dijkstra	1.0	0.95	0.95	0.97	0.92	0.97	0.89	0.97	0.95
Hash_blake2b	1.0	0.94	0.93	0.89	0.92	0.96	0.87	0.96	0.89
Quicksort	1.0	0.96	0.98	0.97	0.93	0.98	0.89	1.0	0.97

Table 5.11: Opcode Frequency Histograms
Tigress samples

	O_0	O_1	O_2	O_3	O_4
Base64	1.0	0.71	0.74	0.7	0.75
Chown	1.0	0.75	0.79	0.74	0.77
DD	1.0	0.76	0.81	0.78	0.83
Sha256sum	1.0	0.72	0.74	0.72	0.74

Table 5.12: Opcode Frequency Histograms
Obfuscator-LLVM samples

5.2.4 ROP Gadget Survival

Tables 5.13 and 5.14 show how the ROP Gadget Survival algorithm performed against Tigress and Obfuscator-LLVM samples. The tables highlight the difference between the results obtained the `bag_of_gadgets` and `survivor` metrics.

Expectedly, the `bag_of_gadgets` achieves a better score compared to `survivor` across all tests. This is explained by the fact that the `survivor` metric can at most obtain a result equivalent to `bag_of_gadgets`, since it only adds the additional requirement for the gadgets to be placed in the same address.

Notably, the results for the Obfuscator-LLVM samples show scores, except the identity comparisons, consistently under 10%, suggesting this metric being unreliable.

	Dijkstra		Hash_blake2b		Quicksort	
	bag	surv.	bag	surv.	bag	surv.
T_0	1.0	1.0	1.0	1.0	1.0	1.0
T_1	0.63	0.43	0.34	0.17	0.69	0.54
T_2	0.65	0.47	0.34	0.17	0.74	0.54
T_3	0.61	0.47	0.32	0.17	0.68	0.59
T_4	0.44	0.22	0.28	0.1	0.61	0.32
T_5	0.47	0.19	0.36	0.17	0.62	0.28
T_6	0.56	0.43	0.31	0.17	0.77	0.54
T_7	0.64	0.43	0.36	0.17	0.77	0.54
T_8	0.51	0.22	0.3	0.17	0.62	0.47

Table 5.13: ROP Gadget Survival
Tigress samples

	Base64		Chown		DD		Sha256sum	
	bag	surv.	bag	surv.	bag	surv.	bag	surv.
O_0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
O_1	0.08	0.05	0.05	0.04	0.05	0.04	0.07	0.05
O_2	0.1	0.08	0.05	0.04	0.01	0.01	0.08	0.05
O_3	0.08	0.05	0.06	0.04	0.02	0.01	0.08	0.05
O_4	0.08	0.05	0.06	0.04	0.1	0.08	0.09	0.05

Table 5.14: ROP Gadget Survival
Obfuscator-LLVM samples

5.3 Metrics Comparison

Finally, tables 5.15 and 5.16 contain the comparison between the results obtained with each metric, respectively for the Tigress and Obfuscator-LLVM obfuscated programs.

A few assumptions are made: when Normalized Compression Distance is mentioned, it refers to the `rzip` results. The ROP Gadget Survival results are reported as using the `bytes` abstraction level, since ROPgadget analyzes the whole binary, and specifically refer to the `bag_of_gadgets` results. The Opcode Frequency metric is categorized under the `opcode_no_param` abstraction level.

As was mentioned before, the only metric failing to achieve 100% similarity for the identity comparison is the Normalized Compression Distance algorithm. This is only true for the smaller Tigress-obfuscated programs, while it does achieve 100% for the larger Obfuscator-LLVM ones. In particular, NCD fails to achieve the maximum result when an abstraction level that reduces the amount of data compared is used. This is especially noticeable for the `opcode_bytes` level, which provides the smallest amount of data. For these reasons, the most reasonable explanation is that for smaller data, the metadata introduced by the compression algorithm reduces the effectiveness of the algorithm.

As for the actual transformations, the results show that the best performing metric across all tests is the Opcode Frequency similarity metric, closely followed by the Longest Common Substring operating on both the same `opcode_no_param` abstraction level, and the `simplified` one. These three achieve about 90% or higher similarity in every Tigress-obfuscated test, while the score for the O-LLVM ones is usually 60% or higher.

To conclude, the code abstraction algorithm described in chapter 3 was shown to improve the results of the `decompiled` abstraction level across all compatible similarity metrics.

		Bytes		Op. By.	Op. Text		Op. no Par.		Decomp.		Simplif.	
		ncd	rop	ncd	ncd	lcs	lcs	freq	ncd	lcs	ncd	lcs
Dijkstra	T_0	1.0	1.0	0.96	1.0	1.0	1.0	1.0	0.98	1.0	0.98	1.0
	T_1	0.68	0.63	0.45	0.52	0.85	0.93	0.95	0.6	0.88	0.66	0.91
	T_2	0.58	0.65	0.37	0.41	0.87	0.96	0.95	0.47	0.91	0.54	0.97
	T_3	0.73	0.61	0.46	0.56	0.85	0.92	0.97	0.63	0.85	0.71	0.89
	T_4	0.53	0.44	0.33	0.39	0.74	0.88	0.92	0.54	0.82	0.62	0.87
	T_5	0.53	0.47	0.36	0.38	0.79	0.97	0.97	0.52	0.97	0.62	0.97
	T_6	0.65	0.56	0.34	0.43	0.81	0.88	0.89	0.56	0.88	0.65	0.93
	T_7	0.71	0.64	0.52	0.56	0.88	0.96	0.97	0.75	0.98	0.76	1.0
	T_8	0.45	0.51	0.32	0.43	0.76	0.93	0.95	0.61	0.83	0.68	0.87
Hash_blake2b	T_0	1.0	1.0	0.99	1.0	1.0	1.0	1.0	0.99	1.0	0.97	1.0
	T_1	0.36	0.34	0.17	0.19	0.65	0.89	0.94	0.17	0.72	0.51	0.81
	T_2	0.2	0.34	0.12	0.11	0.65	0.9	0.93	0.15	0.75	0.24	0.87
	T_3	0.35	0.32	0.13	0.14	0.59	0.84	0.89	0.15	0.7	0.61	0.79
	T_4	0.34	0.28	0.2	0.23	0.64	0.9	0.92	0.22	0.76	0.59	0.84
	T_5	0.37	0.36	0.25	0.27	0.75	0.95	0.96	0.23	0.86	0.59	0.93
	T_6	0.34	0.31	0.15	0.16	0.63	0.84	0.87	0.18	0.78	0.53	0.9
	T_7	0.39	0.36	0.2	0.22	0.7	0.96	0.96	0.23	0.8	0.65	0.92
	T_8	0.24	0.3	0.16	0.18	0.58	0.86	0.89	0.22	0.7	0.53	0.78
Quicksort	T_0	1.0	1.0	0.97	0.99	1.0	1.0	1.0	0.98	1.0	0.97	1.0
	T_1	0.73	0.69	0.47	0.54	0.83	0.92	0.96	0.6	0.81	0.65	0.89
	T_2	0.57	0.74	0.34	0.39	0.8	0.94	0.98	0.39	0.82	0.47	0.91
	T_3	0.75	0.68	0.43	0.5	0.76	0.85	0.97	0.61	0.81	0.71	0.87
	T_4	0.56	0.61	0.3	0.4	0.74	0.89	0.93	0.45	0.79	0.58	0.85
	T_5	0.55	0.62	0.37	0.51	0.83	0.98	0.98	0.5	0.93	0.62	0.94
	T_6	0.73	0.77	0.4	0.49	0.86	0.91	0.89	0.6	0.86	0.73	0.93
	T_7	0.75	0.77	0.51	0.56	0.83	0.98	1.0	0.67	0.85	0.73	0.93
	T_8	0.55	0.62	0.32	0.48	0.73	0.92	0.97	0.59	0.79	0.69	0.86

Table 5.15: Metrics comparison
Tigress samples

		Bytes		Op. By.	Op. Text		Op. no Par.		Decomp.		Simplif.	
		ncd	rop	ncd	ncd	lcs	lcs	freq	ncd	lcs	ncd	lcs
Base64	O_0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	O_1	0.04	0.08	0.0	0.01	0.2	0.62	0.71	0.05	0.4	0.14	0.65
	O_2	0.07	0.1	0.01	0.03	0.21	0.63	0.74	0.2	0.45	0.28	0.7
	O_3	0.06	0.08	0.0	0.02	0.2	0.62	0.7	0.13	0.42	0.22	0.67
	O_4	0.01	0.08	0.0	0.0	0.21	0.61	0.75	0.03	0.37	0.08	0.66
Chown	O_0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	O_1	0.02	0.05	0.0	0.0	0.19	0.59	0.75	0.04	0.35	0.12	0.62
	O_2	0.05	0.05	0.0	0.02	0.2	0.6	0.79	0.15	0.39	0.22	0.68
	O_3	0.04	0.06	0.0	0.0	0.2	0.59	0.74	0.09	0.37	0.17	0.66
	O_4	0.0	0.06	0.0	0.0	0.2	0.58	0.77	0.02	0.33	0.06	0.62
DD	O_0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	O_1	0.02	0.04	0.0	0.01	0.18	0.61	0.76	0.05	0.37	0.17	0.62
	O_2	0.03	0.01	0.0	0.0	0.19	0.61	0.81	0.11	0.41	0.18	0.67
	O_3	0.02	0.01	0.0	0.0	0.19	0.59	0.78	0.09	0.38	0.14	0.64
	O_4	0.01	0.08	0.0	0.02	0.2	0.59	0.83	0.02	0.34	0.08	0.62
Sha256sum	O_0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	O_1	0.04	0.07	0.0	0.0	0.19	0.62	0.72	0.07	0.39	0.16	0.66
	O_2	0.05	0.08	0.0	0.0	0.2	0.62	0.74	0.08	0.43	0.27	0.7
	O_3	0.06	0.08	0.0	0.02	0.2	0.62	0.72	0.12	0.4	0.21	0.68
	O_4	0.01	0.09	0.0	0.0	0.19	0.61	0.74	0.02	0.35	0.09	0.65

Table 5.16: Metrics comparison
Obfuscator-LLVM samples

Chapter 6

Conclusions

In this work, the problem of Software Similarity was examined. Several metrics were described and implemented in the project as a series of Ghidra scripts. Each of these metrics were then tested against a set of programs with varying size, obfuscated with two known obfuscators, Tigress and Obfuscator-LLVM.

The results show that, overall, the best performing similarity method is the Opcode Frequency Histogram metric, followed by Longest Common Subsequence algorithm when either the opcode representation without parameters or the simplified decompilation is used.

On the contrary, the data demonstrates that Normalized Compression Distance is an unreliable metric, especially for the larger, OLLVM-obfuscated samples. Additionally, the ROP Gadget Survival metric was also shown to perform poorly with the larger programs.

This is a large field of study, and the opportunities to improve this work are plenty. For example, more metrics could be implemented and more obfuscation configurations tested. Another improvement could be testing the similarity of the programs against known different programs, for which the similarity should be minimized.

Appendix A

Sources

Listing A.1: dijkstra.c [27]

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 20
#define INF 999

int mat[MAX][MAX];
int V;

int dist[MAX];

int q[MAX];
int qp = 0;

void enqueue(int v) { q[qp++] = v; }

int cf(void *a, void *b)
{
    int *x = (int *)a;
    int *y = (int *)b;
    return *y - *x;
}

int dequeue()
{
    qsort(q, qp, sizeof(int), cf);
    return q[--qp];
}
```

```
29
30 int queue_has_something() { return (qp > 0); }
31
32 int visited[MAX];
33 int vp = 0;
34
35 void dijkstra(int s)
36 {
37     dist[s] = 0;
38     int i;
39     for (i = 0; i < V; ++i)
40     {
41         if (i != s)
42         {
43             dist[i] = INF;
44         }
45         enqueue(i);
46     }
47     while (queue_has_something())
48     {
49         int u = dequeue();
50         visited[vp++] = u;
51         for (i = 0; i < V; ++i)
52         {
53             if (mat[u][i])
54             {
55                 if (dist[i] > dist[u] + mat[u][i])
56                 {
57                     dist[i] = dist[u] + mat[u][i];
58                 }
59             }
60         }
61     }
62 }
63
64 int main(int argc, char const *argv[])
65 {
66     printf("Enter the number of vertices: ");
67     scanf(" %d", &V);
68     printf("Enter the adj matrix: ");
69     int i, j;
70     for (i = 0; i < V; ++i)
71     {
```

```
72          for (j = 0; j < V; ++j)
73          {
74              scanf(" %d", &mat[i][j]);
75          }
76      }
77
78      dijkstra(0);
79
80      printf("\nNode\tDist\n");
81      for (i = 0; i < V; ++i)
82      {
83          printf("%d\t%d\n", i, dist[i]);
84      }
85
86      return 0;
87  }
```

Listing A.2: hash_blake2b.c [27]

```
#include <assert.h>    /// for asserts
#include <inttypes.h>  /// for fixed-width integer types e.g. uint64_t and uint8_t
#include <stdio.h>     /// for IO
#include <stdlib.h>    /// for malloc, calloc, and free. As well as size_t

#ifdef __GNUC__
#pragma GCC diagnostic ignored "-Wshift-count-overflow"
#elif _MSC_VER
#pragma warning(disable : 4293)
#endif

#define bb 128
#define KK_MAX 64
#define NN_MAX 64
#define CEIL(a, b) (((a) / (b)) + ((a) % (b) != 0))
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define ROTR64(n, offset) (((n) >> (offset)) ^ ((n) << (64 - (offset))))
#define U128_ZERO \
    {             \
        0, 0      \
    }

typedef uint64_t u128[2];
typedef uint64_t block_t[bb / sizeof(uint64_t)];

static const uint8_t R1 = 32;  ///< Rotation constant 1 for mixing function G
static const uint8_t R2 = 24;  ///< Rotation constant 2 for mixing function G
static const uint8_t R3 = 16;  ///< Rotation constant 3 for mixing function G
static const uint8_t R4 = 63;  ///< Rotation constant 4 for mixing function G

static const uint64_t blake2b_iv[8] = {
    0x6A09E667F3BCC908, 0xBB67AE8584CAA73B, 0x3C6EF372FE94F82B,
    0xA54FF53A5F1D36F1, 0x510E527FADE682D1, 0
```

```
        x9B05688C2B3E6C1F,
    0x1F83D9ABFB41BD6B, 0x5BE0CD19137E2179};

static const uint8_t blake2b_sigma[12][16] = {
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15},
    {14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5, 3},
    {11, 8, 12, 0, 5, 2, 15, 13, 10, 14, 3, 6, 7, 1, 9, 4},
    {7, 9, 3, 1, 13, 12, 11, 14, 2, 6, 5, 10, 4, 0, 15, 8},
    {9, 0, 5, 7, 2, 4, 10, 15, 14, 1, 11, 12, 6, 8, 3, 13},
    {2, 12, 6, 10, 0, 11, 8, 3, 4, 13, 7, 5, 15, 14, 1, 9},
    {12, 5, 1, 15, 14, 13, 4, 10, 0, 7, 6, 3, 9, 2, 8, 11},
    {13, 11, 7, 14, 12, 1, 3, 9, 5, 0, 15, 4, 8, 6, 2, 10},
    {6, 15, 14, 9, 11, 3, 0, 8, 12, 2, 13, 7, 1, 4, 10, 5},
    {10, 2, 8, 4, 7, 6, 1, 5, 15, 11, 9, 14, 3, 12, 13, 0},
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15},
    {14, 10, 4, 8, 9, 15, 13, 6, 1, 12, 0, 2, 11, 7, 5,
     3}};  ///< word schedule permutations for each round of
         the algorithm

static inline void u128_fill(u128 dest, size_t n)
{
    dest[0] = n & UINT64_MAX;

    if (sizeof(n) > 8)
    {
        dest[1] = n >> 64;
    }
    else
    {
        dest[1] = 0;
    }
}

static inline void u128_increment(u128 dest, uint64_t n)
{
    /* Check for overflow */
    if (UINT64_MAX - dest[0] <= n)
    {
        dest[1]++;
    }

    dest[0] += n;
}
```

```
76
77 static void G(block_t v, uint8_t a, uint8_t b, uint8_t c,
       uint8_t d, uint64_t x,
78                 uint64_t y)
79 {
80     v[a] += v[b] + x;
81     v[d] = ROTR64(v[d] ^ v[a], R1);
82     v[c] += v[d];
83     v[b] = ROTR64(v[b] ^ v[c], R2);
84     v[a] += v[b] + y;
85     v[d] = ROTR64(v[d] ^ v[a], R3);
86     v[c] += v[d];
87     v[b] = ROTR64(v[b] ^ v[c], R4);
88 }
89
90 static void F(uint64_t h[8], block_t m, u128 t, int f)
91 {
92     int i;
93     block_t v;
94
95     /* v[0..7] := h[0..7] */
96     for (i = 0; i < 8; i++)
97     {
98         v[i] = h[i];
99     }
100    /* v[8..15] := IV[0..7] */
101    for (; i < 16; i++)
102    {
103        v[i] = blake2b_iv[i - 8];
104    }
105
106    v[12] ^= t[0]; /* v[12] ^ (t mod 2**w) */
107    v[13] ^= t[1]; /* v[13] ^ (t >> w) */
108
109    if (f)
110    {
111        v[14] = ~v[14];
112    }
113
114    for (i = 0; i < 12; i++)
115    {
116        const uint8_t *s = blake2b_sigma[i];
117
```

```
118         G(v, 0, 4, 8, 12, m[s[0]], m[s[1]]);
119         G(v, 1, 5, 9, 13, m[s[2]], m[s[3]]);
120         G(v, 2, 6, 10, 14, m[s[4]], m[s[5]]);
121         G(v, 3, 7, 11, 15, m[s[6]], m[s[7]]);
122
123         G(v, 0, 5, 10, 15, m[s[8]], m[s[9]]);
124         G(v, 1, 6, 11, 12, m[s[10]], m[s[11]]);
125         G(v, 2, 7, 8, 13, m[s[12]], m[s[13]]);
126         G(v, 3, 4, 9, 14, m[s[14]], m[s[15]]);
127     }
128
129     for (i = 0; i < 8; i++)
130     {
131         h[i] ^= v[i] ^ v[i + 8];
132     }
133 }
134
135 static int BLAKE2B(uint8_t *dest, block_t *d, size_t dd,
        u128 ll, uint8_t kk,
136                    uint8_t nn)
137 {
138     uint8_t bytes[8];
139     uint64_t i, j;
140     uint64_t h[8];
141     u128 t = U128_ZERO;
142
143     /* h[0..7] = IV[0..7] */
144     for (i = 0; i < 8; i++)
145     {
146         h[i] = blake2b_iv[i];
147     }
148
149     h[0] ^= 0x01010000 ^ (kk << 8) ^ nn;
150
151     if (dd > 1)
152     {
153         for (i = 0; i < dd - 1; i++)
154         {
155             u128_increment(t, bb);
156             F(h, d[i], t, 0);
157         }
158     }
159
```

```
160     if (kk != 0)
161     {
162         u128_increment(ll, bb);
163     }
164     F(h, d[dd - 1], ll, 1);
165 
166     /* copy bytes from h to destination buffer */
167     for (i = 0; i < nn; i++)
168     {
169         if (i % sizeof(uint64_t) == 0)
170         {
171             /* copy values from uint64 to 8 u8's */
172             for (j = 0; j < sizeof(uint64_t); j++)
173             {
174                 uint16_t offset = 8 * j;
175                 uint64_t mask = 0xFF;
176                 mask <<= offset;
177 
178                 bytes[j] = (h[i / 8] & (mask)) >> offset;
179             }
180         }
181 
182         dest[i] = bytes[i % 8];
183     }
184 
185     return 0;
186 }
187 
188 uint8_t *blake2b(const uint8_t *message, size_t len, const
        uint8_t *key,
189                  uint8_t kk, uint8_t nn)
190 {
191     uint8_t *dest = NULL;
192     uint64_t long_hold;
193     size_t dd, has_key, i;
194     size_t block_index, word_in_block;
195     u128 ll;
196     block_t *blocks;
197 
198     if (message == NULL)
199     {
200         len = 0;
201     }
```

```
202     if (key == NULL)
203     {
204         kk = 0;
205     }
206 
207     kk = MIN(kk, KK_MAX);
208     nn = MIN(nn, NN_MAX);
209 
210     dd = MAX(CEIL(kk, bb) + CEIL(len, bb), 1);
211 
212     blocks = calloc(dd, sizeof(block_t));
213     if (blocks == NULL)
214     {
215         return NULL;
216     }
217 
218     dest = malloc(nn * sizeof(uint8_t));
219     if (dest == NULL)
220     {
221         free(blocks);
222         return NULL;
223     }
224 
225     /* If there is a secret key it occupies the first block
           */
226     for (i = 0; i < kk; i++)
227     {
228         long_hold = key[i];
229         long_hold <<= 8 * (i % 8);
230 
231         word_in_block = (i % bb) / 8;
232         /* block_index will always be 0 because kk <= 64 and
               bb = 128*/
233         blocks[0][word_in_block] |= long_hold;
234     }
235 
236     has_key = kk > 0 ? 1 : 0;
237 
238     for (i = 0; i < len; i++)
239     {
240         /* long_hold exists because the bit-shifting will
               overflow if we don't
241          * store the value */
```

```
242         long_hold = message[i];
243         long_hold <<= 8 * (i % 8);
244 
245         block_index = has_key + (i / bb);
246         word_in_block = (i % bb) / 8;
247 
248         blocks[block_index][word_in_block] |= long_hold;
249     }
250 
251     u128_fill(ll, len);
252 
253     BLAKE2B(dest, blocks, dd, ll, kk, nn);
254 
255     free(blocks);
256 
257     return dest;
258 }
259 
260 static void assert_bytes(const uint8_t *expected, const
        uint8_t *actual,
261                          uint8_t len)
262 {
263     uint8_t i;
264 
265     assert(expected != NULL);
266     assert(actual != NULL);
267     assert(len > 0);
268 
269     for (i = 0; i < len; i++)
270     {
271         assert(expected[i] == actual[i]);
272     }
273 }
274 
275 static void test()
276 {
277     uint8_t *digest = NULL;
278 
279     /* "abc" example straight out of RFC-7693 */
280     uint8_t abc[3] = {'a', 'b', 'c'};
281     uint8_t abc_answer[64] = {
282         0xBA, 0x80, 0xA5, 0x3F, 0x98, 0x1C, 0x4D, 0x0D, 0x6A
                , 0x27, 0x97,
```

```
283         0xB6, 0x9F, 0x12, 0xF6, 0xE9, 0x4C, 0x21, 0x2F, 0x14
                , 0x68, 0x5A,
284         0xC4, 0xB7, 0x4B, 0x12, 0xBB, 0x6F, 0xDB, 0xFF, 0xA2
                , 0xD1, 0x7D,
285         0x87, 0xC5, 0x39, 0x2A, 0xAB, 0x79, 0x2D, 0xC2, 0x52
                , 0xD5, 0xDE,
286         0x45, 0x33, 0xCC, 0x95, 0x18, 0xD3, 0x8A, 0xA8, 0xDB
                , 0xF1, 0x92,
287         0x5A, 0xB9, 0x23, 0x86, 0xED, 0xD4, 0x00, 0x99, 0x23
                };
288
289     digest = blake2b(abc, 3, NULL, 0, 64);
290     assert_bytes(abc_answer, digest, 64);
291
292     free(digest);
293
294     uint8_t key[64] = {
295         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
                , 0x09, 0x0a,
296         0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13
                , 0x14, 0x15,
297         0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e
                , 0x1f, 0x20,
298         0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29
                , 0x2a, 0x2b,
299         0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34
                , 0x35, 0x36,
300         0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f
                };
301     uint8_t key_answer[64] = {
302         0x10, 0xeb, 0xb6, 0x77, 0x00, 0xb1, 0x86, 0x8e, 0xfb
                , 0x44, 0x17,
303         0x98, 0x7a, 0xcf, 0x46, 0x90, 0xae, 0x9d, 0x97, 0x2f
                , 0xb7, 0xa5,
304         0x90, 0xc2, 0xf0, 0x28, 0x71, 0x79, 0x9a, 0xaa, 0x47
                , 0x86, 0xb5,
305         0xe9, 0x96, 0xe8, 0xf0, 0xf4, 0xeb, 0x98, 0x1f, 0xc2
                , 0x14, 0xb0,
306         0x05, 0xf4, 0x2d, 0x2f, 0xf4, 0x23, 0x34, 0x99, 0x39
                , 0x16, 0x53,
307         0xdf, 0x7a, 0xef, 0xcb, 0xc1, 0x3f, 0xc5, 0x15, 0x68
                };
308
```

```
309     digest = blake2b(NULL, 0, key, 64, 64);
310     assert_bytes(key_answer, digest, 64);
311
312     free(digest);
313
314     uint8_t zero[1] = {0};
315     uint8_t zero_key[64] = {
316         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
                , 0x09, 0x0a,
317         0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13
                , 0x14, 0x15,
318         0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e
                , 0x1f, 0x20,
319         0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29
                , 0x2a, 0x2b,
320         0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34
                , 0x35, 0x36,
321         0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f
                };
322     uint8_t zero_answer[64] = {
323         0x96, 0x1f, 0x6d, 0xd1, 0xe4, 0xdd, 0x30, 0xf6, 0x39
                , 0x01, 0x69,
324         0x0c, 0x51, 0x2e, 0x78, 0xe4, 0xb4, 0x5e, 0x47, 0x42
                , 0xed, 0x19,
325         0x7c, 0x3c, 0x5e, 0x45, 0xc5, 0x49, 0xfd, 0x25, 0xf2
                , 0xe4, 0x18,
326         0x7b, 0x0b, 0xc9, 0xfe, 0x30, 0x49, 0x2b, 0x16, 0xb0
                , 0xd0, 0xbc,
327         0x4e, 0xf9, 0xb0, 0xf3, 0x4c, 0x70, 0x03, 0xfa, 0xc0
                , 0x9a, 0x5e,
328         0xf1, 0x53, 0x2e, 0x69, 0x43, 0x02, 0x34, 0xce, 0xbd
                };
329
330     digest = blake2b(zero, 1, zero_key, 64, 64);
331     assert_bytes(zero_answer, digest, 64);
332
333     free(digest);
334
335     uint8_t filled[64] = {
336         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
                , 0x09, 0x0a,
337         0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13
                , 0x14, 0x15,
```

```
338         0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e
                , 0x1f, 0x20,
339         0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29
                , 0x2a, 0x2b,
340         0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34
                , 0x35, 0x36,
341         0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f
               };
342     uint8_t filled_key[64] = {
343         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
                , 0x09, 0x0a,
344         0x0b, 0x0c, 0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13
                , 0x14, 0x15,
345         0x16, 0x17, 0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e
                , 0x1f, 0x20,
346         0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x29
                , 0x2a, 0x2b,
347         0x2c, 0x2d, 0x2e, 0x2f, 0x30, 0x31, 0x32, 0x33, 0x34
                , 0x35, 0x36,
348         0x37, 0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f
               };
349     uint8_t filled_answer[64] = {
350         0x65, 0x67, 0x6d, 0x80, 0x06, 0x17, 0x97, 0x2f, 0xbd
                , 0x87, 0xe4,
351         0xb9, 0x51, 0x4e, 0x1c, 0x67, 0x40, 0x2b, 0x7a, 0x33
                , 0x10, 0x96,
352         0xd3, 0xbf, 0xac, 0x22, 0xf1, 0xab, 0xb9, 0x53, 0x74
                , 0xab, 0xc9,
353         0x42, 0xf1, 0x6e, 0x9a, 0xb0, 0xea, 0xd3, 0x3b, 0x87
                , 0xc9, 0x19,
354         0x68, 0xa6, 0xe5, 0x09, 0xe1, 0x19, 0xff, 0x07, 0x78
                , 0x7b, 0x3e,
355         0xf4, 0x83, 0xe1, 0xdc, 0xdc, 0xcf, 0x6e, 0x30, 0x22
               };
356
357     digest = blake2b(filled, 64, filled_key, 64, 64);
358     assert_bytes(filled_answer, digest, 64);
359
360     free(digest);
361
362     printf("All tests have successfully passed!\n");
363 }
364
```

```
365 int main()
366 {
367     test();
368     return 0;
369 }
```

Listing A.3: quicksort.c [27]

```
#include <stdio.h>
#include <stdlib.h>

/*Displays the array, passed to this method*/
void display(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }

    printf("\n");
}

/*Swap function to swap two values*/
void swap(int *first, int *second)
{
    int temp = *first;
    *first = *second;
    *second = temp;
}

/*Partition method which selects a pivot
  and places each element which is less than the pivot value
      to its left
  and the elements greater than the pivot value to its right
  arr[] --- array to be partitioned
  lower --- lower index
  upper --- upper index
*/
int partition(int arr[], int lower, int upper)
{
    int i = (lower - 1);

    int pivot = arr[upper];  // Selects last element as the
       pivot value

    int j;
    for (j = lower; j < upper; j++)
    {
        if (arr[j] <= pivot)
```

```
41          {  // if current element is smaller than the pivot
42
43              i++;  // increment the index of smaller element
44              swap(&arr[i], &arr[j]);
45          }
46      }
47
48      swap(&arr[i + 1], &arr[upper]);  // places the last
           element i.e, the pivot
49                                       // to its correct
                                            position
50
51      return (i + 1);
52  }
53
54  /*This is where the sorting of the array takes place
55      arr[] --- Array to be sorted
56      lower --- Starting index
57      upper --- Ending index
58  */
59  void quickSort(int arr[], int lower, int upper)
60  {
61      if (upper > lower)
62      {
63          // partitioning index is returned by the partition
               method , partition
64          // element is at its correct poition
65
66          int partitionIndex = partition(arr, lower, upper);
67
68          // Sorting elements before and after the partition
               index
69          quickSort(arr, lower, partitionIndex - 1);
70          quickSort(arr, partitionIndex + 1, upper);
71      }
72  }
73
74  int main()
75  {
76      int n;
77      printf("Enter size of array:\n");
78      scanf("%d", &n);  // E.g. 8
79
```

```
80      printf("Enter the elements of the array\n");
81      int i;
82      int *arr = (int *)malloc(sizeof(int) * n);
83      for (i = 0; i < n; i++)
84      {
85          scanf("%d", &arr[i]);
86      }
87
88      printf("Original array: ");
89      display(arr, n);  // Original array : 10 11 9 8 4 7 3 8
90
91      quickSort(arr, 0, n - 1);
92
93      printf("Sorted array: ");
94      display(arr, n);  // Sorted array : 3 4 7 8 8 9 10 11
95      getchar();
96      free(arr);
97      return 0;
98  }
```

Bibliography

[1] Mariano Ceccato et al. "Search based clustering for protecting software with diversified updates". In: *Search Based Software Engineering: 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings 8*. Springer. 2016, pp. 159–175.

[2] Joel Coffman et al. "ROP gadget prevalence and survival under compiler-based binary diversification schemes". In: *Proceedings of the 2016 ACM Workshop on Software PROtection*. 2016, pp. 15–26.

[3] Christian Collberg. "Tigress: A Source-to-Source-ish Obfuscation Tool". In: *Proc. 8th Workshop on Software Security, Protection, and Reverse Engineering*. 2018.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. Department of Computer Science, The University of Auckland, New Zealand, 1997.

[5] Christian S. Collberg and Clark Thomborson. "Watermarking, tamper-proofing, and obfuscation-tools for software protection". In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 735–746.

[6] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[7] Manuel Freire, Manuel Cebrian, and Emilio del Rosal. "Uncovering plagiarism networks". In: *arXiv preprint cs/0703136* (2007).

[8] Marc Fyrbiak et al. "Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors". In: *IEEE Transactions on Computers* 67.3 (2017), pp. 307–321.

[9] Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[10] Andrei Homescu et al. "Profile-guided automated software diversity". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2013, pp. 1–11.

[11] Shohreh Hosseinzadeh et al. "Diversification and obfuscation techniques for software security: A systematic literature review". In: *Information and Software Technology* 104 (2018), pp. 72–93.

[12] Pascal Junod et al. "Obfuscator-LLVM – Software Protection for the Masses". In: *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. Ed. by Brecht Wyseur. IEEE, 2015, pp. 3–9. DOI: 10.1109/SPRO.2015.10.

[13] Per Larsen et al. "SoK: Automated software diversity". In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 276–291.

[14] Han Liu et al. "Stochastic optimization of program obfuscation". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 221–231.

[15] Niccolò Marastoni, Roberto Giacobazzi, and Mila Dalla Preda. "A deep learning approach to program similarity". In: *Proceedings of the 1st international workshop on machine learning and software engineering in symbiosis*. 2018, pp. 26–35.

[16] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.

[17] Matija Novak, Mike Joy, and Dragutin Kermek. "Source-code similarity detection and detection tools used in academia: a systematic review". In: *ACM Transactions on Computing Education (TOCE)* 19.3 (2019), pp. 1–37.

[18] Ahmet Okutan. "Use of source code similarity metrics in software defect prediction". In: *arXiv preprint arXiv:1808.10033* (2018).

[19] Seongsoo Park et al. "Detecting source code similarity using code abstraction". In: *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*. 2013, pp. 1–9.

[20] Mike Paterson and Vlado Dančìk. "Longest common subsequences". In: *Mathematical Foundations of Computer Science 1994: 19th International Symposium, MFCS'94 Košice, Slovakia, August 22–26, 1994 Proceedings 19*. Springer. 1994, pp. 127–142.

[21] Babak Bashari Rad and Maslin Masrom. "Metamorphic virus detection in Portable Executables using opcodes statistical feature". In: *arXiv preprint arXiv:1104.3229* (2011).

[22] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. "A comparison of code similarity analysers". In: *Empirical Software Engineering* 23 (2018), pp. 2464–2519.

[23] Sampsa Rauti et al. “Internal interface diversification as a method against malware”. In: *Journal of Cyber Security Technology* 5.1 (2021), pp. 15–40.

[24] Ryan Roemer et al. “Return-oriented programming: Systems, languages, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), pp. 1–34.

[25] *rzip(1): large-file compression program - Linux man page*. URL: `https://linux.die.net/man/1/rzip`.

[26] Lucas Pereira da Silva and Patrìcia Vilain. “LCCSS: A similarity metric for identifying similar test code”. In: *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*. 2020, pp. 91–100.

[27] TheAlgorithms. *TheAlgorithms/C: Collection of various algorithms in mathematics, Machine Learning, computer science, physics, etc implemented in C for educational purposes.* URL: `https://github.com/TheAlgorithms/C`.

[28] Shukun Tokas, Olaf Owe, and Christian Johansen. “Code Diversification Mechanisms for Internet of Things (Revised Version 2)”. In: *Research report http://urn. nb. no/URN: NBN: no-35645* (2020).

[29] Rodothea Myrsini Tsoupidi, Roberto Castañeda Lozano, and Benoit Baudry. “Constraint-based software diversification for efficient mitigation of code-reuse attacks”. In: *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7–11, 2020, Proceedings 26*. Springer. 2020, pp. 791–808.

[30] Shuai Wang, Pei Wang, and Dinghao Wu. “Composite software diversification”. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2017, pp. 284–294.

[31] Ming Xu et al. “A similarity metric method of obfuscated malware using function-call graph”. In: *Journal of Computer Virology and Hacking Techniques* 9 (2013), pp. 35–47.

[32] Xiaochuan Zhang, Jianmin Pang, and Xiaonan Liu. “Common program similarity metric method for anti-obfuscation”. In: *IEEE Access* 6 (2018), pp. 47557–47565.