Politecnico di Torino

Master's Degree in Computer Engineering

Master's Degree Thesis

Overhead Prediction in Obfuscated Programs

Supervisors
Prof. Cataldo Basile
Prof. Antonio Lioy
Dott. Daniele Canavese

Candidate
Ilio Di Pietro
matricola 266393

Academic year 2021-2022

Contents

Chapter 1

Introduction

In today's connected world, security is a constant concern. Any compromise to integrity, authentication, confidentiality, and availability makes software insecure. Software systems can be attacked to steal information, monitor content, introduce vulnerabilities, and damage its behavior.
The challenging aspect of this field is that there is no dividing line between what is considered safe and what is not, so the idea of software security involves a proactive approach in its resolution: "Security is a process, not a product" [1].

Although there are innumerable types of attacks, two main categories can be distinguished: *internal* and *external*.
Most *external attacks* happen to steal confidential information through the use of malware such as worms, Trojan horses, and phishing, but also to make a service unusable for a certain period of time, such as DoS or DDoS.

On the other hand, *internal attacks* involve software users themselves. The adversary is no longer a third party between two trusted parties, but rather one of them with physical, local, or remote access to the target software. The goals of a MATE (Man-At-The-End) attack include violating the confidentiality of algorithms or other data inside of a software program and/or the integrity of the software behavior as intended by the developer. Practically, any device under the control of an end user that runs proprietary software is exposed to MATE attacks[1]. Software developers are also afraid of the prospect that a competitor can extract proprietary algorithms and data structures from their applications to incorporate them into their own programs [2]. In this scenario, there are two general ways to protect intellectual property: legally or technically.

Legally means getting copyrights or signing legal contracts against creating duplicates, but in many cases it does not represent the best choice, especially for small companies or even worse for single developers, due to the cost and maintenance of such a solution.

[1] Bruce Schneier, Crypto-Gram, 2005

Technically means the owners of the software will make reverse engineering of their code impossible or at least not feasible in terms of time and resources. One of the most promising techniques to limit the effect of a reverse engineering attack is *code obfuscation*.

Obfuscation consists of code transformations that make a program more difficult to understand by changing its structure while preserving the original functionalities [3]. This means that we pass the code we want to protect through a *obfuscator*, which is a program that performs semantic-preserving code manipulation to transform the original code into a much more complex version.

It must be said that code obfuscation cannot always completely protect an application from malicious attacks. Moreover, it is not mandatory for an attacker to retrieve the original source code of the application, as it is sufficient for the reverse-engineered code to be at least comprehensible to the malicious user. In fact, given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application [2].
This last sentence would seem to nullify everything previously said about protection against MATE attacks: What could be the point of applying obfuscation to code if, with the right resources, it is possible to overcome that obstacle and trace back, if not to the original code, at least to an equally comprehensible version of it?
Time is the answer: since it is not possible to protect code for an indefinite period of time, obfuscation is aimed at those specific applications that need immediate and maximum protection as soon as they are released and for a short period, after which it would no longer be worth attacking. For example, a newly released film needs protection only during the first few weeks, as most of the revenue is concentrated in that time period. The same applies to a game or in-game content available for a limited period, such as skins.

The downside of this technique is definitely the drop in performance of obfuscated programs: building a more complex version of the application means that the obfuscator has to add more and more complex computations, many of which introduce loops or data structures with the sole purpose of confusing the attacker: the execution time required for an obfuscated program will always be greater than its original version.

The steps generally followed when applying obfuscation to the code consist of applying as much protection as possible and running the code to see if performance is acceptable. If it is too slow, go back to remove some transformations and try running it again. This is done until a fair compromise between protection and performance is found.
This process can be repeated an indefinite number of times, resulting in thousands of code executions and taking hours and hours to reach a solution.
This results in the release of the application to the public with a not inconsiderable delay, causing a considerable loss of profit.

In this scenario, being able to understand a priori whether these performance losses may be negligible, or whether the impact of one or more transformations is too costly in terms of time and resources, could be a game changer:developers would be able to figure out which techniques to apply without running the code, saving a great deal of time and

being able to release their applications sooner.

Therefore, the objective of this thesis is to build a machine learning model capable of predicting the overhead caused by the application of various obfuscation techniques before they are applied to the code under analysis .

The necessary data on the execution of vanilla and obfuscated programs, needed to construct a complete dataset, were collected by another student during the development of his thesis [4]. In his work, each selected program was executed with a set of different inputs to collect a wide variety of execution traces. Subsequently, each application was obfuscated using Tigress[2], a tool developed by the University of Arizona. In the end, information on time measurements was taken by running these obfuscated applications, as previously done for their vanilla counterparts. This whole process led to the creation of a large data collection.

Although having a huge quantity of execution traces, the challenging part was creating a complete dataset, making them suitable for a machine learning model: this was the starting point of this thesis work. The first step was to clean the traces by removing unnecessary information and to retain only the relevant data. Since the obfuscation tool used work on individual functions and since a single trace lists all the instructions regarding the overall execution of one application with some specific inputs, the next step was to decompose the trace, reconstructing the call tree and identifying all the functions that are executed within it.
Parallelly, to correlate each function with its obfuscated counterpart, information concerning the overhead caused by the application of certain previously chosen obfuscation techniques was computed by correlating the time measurement regarding vanilla applications with their obfuscated counterpart. Subsequently, this information was added to the traces.
In this way, each trace will represent a single function, with the sequence of the instruction executed and the overhead values at the end.

The choice of the network was obviously influenced by the type of data.
Due to the sequential nature of the data, in which each instruction is linked to the next by a well-defined temporal relation, the choice of a traditional neural network would not be correct, as a conventional architecture cannot handle temporal sequences: this scenario requires a different solution.
in particular, a type of architecture capable of exhibiting temporal dynamic behavior will be required, in which the various inputs will depend not only on the current data under analysis, but also on the previous history.

The proposed solution involves the use of a particular architecture, called *Long-Short Term Memory* (LSTM).
To deal with the vast but limited memory of LSTM, various steps were necessary to reduce

[2] https://tigress.wtf/index.html

the length of the traces, including simplifying and compressing the instructions.

Subsequent selection of only data with consistent and coherent values led to the rejection of a part of the dataset that would otherwise have contributed to an increased probability of network errors.

Finally, after having defined an LSTM network, the constructed dataset was used to train and evaluate the model.

1.1 Thesis organization

The remainder of the thesis is structured as follows:

- Chapters 2 and 3 describe the background theory needed to understand the topics touched on in the thesis;
- Chapter 4 describes the state-of-the-art and existing work related to the main topics of this thesis;
- Chapter 5 describes the possible fields of application of obfuscation and the associated problems in its practical use, formalizing the motivations that led to the development of the topic proposed in this work;
- Chapter 6 discusses the problem of overhead prediction, outlining its design principles;
- Chapter 7 analyzes in detail every aspect of the solution presented in this thesis. The implementation solutions and the problems addressed were explained in depth;
- Chapter 8 explains the results of the work and proposes some ideas for future improvements.

Chapter 2

Background on data collection

The purpose of this chapter is to introduce the reader to the main topics related the data collection phase, as well as all topics related to the construction of the dataset.

An important part of this work, related to data collection, involves the execution of various programs, as well as the application of obfuscation and run-time tracking.
The development of this solution was the result of a complex work carried out by Stefano Alberto in his thesis [4] .
Since the data produced by him will be the basis on which the entire work of this thesis will rest, understanding how they were generated and their structure is crucial. To fully understand the various steps that led to the collection of information concerning the execution and fuzzing of programs, it is necessary to introduce a few technical topics, such as obfuscation and fuzzing, which are shown below.

2.1 Obfuscation

Any software, regardless of how secure and well designed it is, will be distributed in the form of executable code. People who have legal access to this code will have full control over the system and have the possibility to inspect it, running all types of tool such as disassemblers, simulators, decompilers, etc.

The reasons for reverse engineering a program may be multiple: for example, an attacker might be interested in extracting secret information that should not be revealed, such as cryptographic keys or algorithms that are considered a trade secret. Another reason for reverse engineering may be the alteration of the code to modify its behavior, making hidden functionalities of the program accessible, or unlocking functionalities that were originally blocked for certain types of devices.

Despite technological progress, protecting against MATE attacks can be very challenging for many reasons. Firstly, the attacker is human, so he uses motivation and creativity to reverse engineer the code. Furthermore, having white-box and limitless access to software can create opportunities for theft of intellectual property through software piracy, as

well as security breaches by allowing attackers to discover vulnerabilities in an application. Finally, all protections focus on delaying the attacker until a certain period of time and not making the code unhackable, which is still extremely complex to do today. Subsequently, no piece of software is likely to survive unscathed for a long period of time.

In this scenario, however, an attractive attempt to protect is represented by software obfuscation. *Software Obfuscation* refers to a large number of semantic-preserving transformation techniques aimed at changing the form of the code in such a way as to prevent the understanding of its algorithms and data structures or to prevent the extraction of some valuable information from it. It makes a program more difficult to understand and reverse engineer, without affecting the original software behaviour.
The level of security from reverse engineering that an obfuscator adds to an application depends on [2]:

- the sophistication of the transformation used by the obfuscator;
- the power of the available deobfuscation algorithms;
- the amount of resources available by the deobfuscator.

Collberg et al.[2] gives a more formal definition of an obfuscating transformation:

Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source code P into a target program P'.
$P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order of $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation, the following conditions must hold:

1. If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
2. Otherwise, P' must terminate and produce the same output as P.

Observable behavior means what the user expects from the code; hence, the obfuscated code may have side effects that are not in common with the original program, such as creating new files or sending messages. As P' turns out to be a more complex version of P, the obfuscated program will always be slower and / or more resource-consuming than the *vanilla* one. This leads to performance degradation that is often not negligible.

Obfuscation methods are classified according to the information they target [2].

- *Layout obfuscation*: this transformation targets the appearance of the code. It may manipulate indentation, variable names, add or delete comments used in the code. The removed information or the changes in formatting cannot be recovered, so the original code cannot be reconstructed from the obfuscated version. Generally, it introduces minimal confusion to the attacker, as there is very little semantics introduced by the formatting of the code;
- *Data obfuscation*: this transformation targets the data structure of a program either by replacing the name of a variable with a complex expression or manipulating the form in which the data are stored. Example are:

  - *storage obfuscation*, in which unnatural storage methods are chosen for the data;
  - *encoding obfuscation*, in which the encodings of the variables are changed, like replacing a variable with an expression;
  - *aggregation obfuscation*, in which it is altered how data are grouped together, like splitting an array into many sub-arrays.

  The effectiveness of all these techniques varies depending on the number of variables changed and the complexity of the changes made;

- *Control obfuscation*: Its purpose is to modify the control-flow[1] of a program with transformations that keep the basic logic of the application unchanged. These transformations may affect *aggregation*, *ordering*, or *computation* of the control-flow:

  - *Control aggregation transformations* break up computations that logically belong together or merge computations that do not;
  - *Control ordering transformations* randomize the order in which computations are performed;
  - *Control computation transformations* insert new code or make some algorithmic change to the source of the application [2].

- *Preventive obfuscation*: different types of transformation in which the purpose is to make known automatic deobfuscation techniques more difficult or to exploit known weaknesses in deobfuscators.

The chosen obfuscation transformations are:

- *code flattening*
- *opaque predicate*

Code flattening

Code Flattening transformations are special types of transformation that obscure the control-flow structures of the source code of the application by hiding its real structure.
In particular, it targets branches as they are constructs that can be easily detected by the attacker since they have a similar structure in most programming languages. Once identified, the entire control flow of the program can be easily reconstructed.
The first step in flattening a function consists in splitting the code in single basic blocks. Then, all the basic blocks are put next to each other by destroying the original nested structure, using, for example, a switch case statement, and then insert this structure into a loop.
In this way, it is possible to make it difficult to identify the targets of the various jumps and, consequently, reconstruct the control flow graph of the application. Figure 2.1 shows a visual representation of code flattening, while Figure 2.2 shows a simple example.

[1] the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated

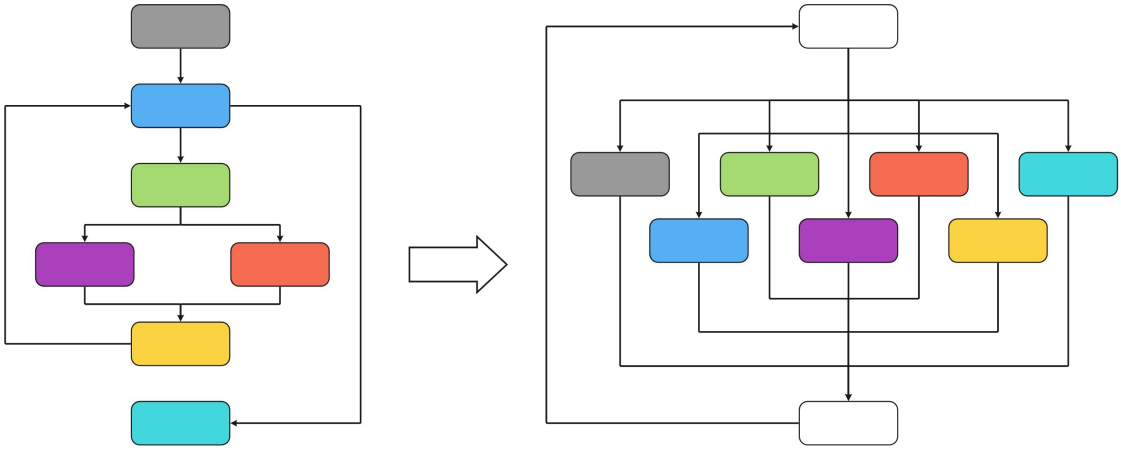


Figure 2.1: Visual representation of flatten transformation

```
i = 1
s = 0

while i <= 100:
   s += i
   i += 1
```

```
swVar = 1

while swVar != 0:
   switch swVar:
      case 1:
         i = 1
         s = 0
         swVar = 1
         break
      case 2:
         if i <= 100:
            swVar = 3
         else:
            swVar = 0
         break
      case 3:
         s += i
         i += 1
         swVar = 2
         break
```

Figure 2.2: Example of flatten transformation

Opaque predicate

The obfuscation transformation involving the addition of opaque predicates, as the previous one, belongs to all those transformations that aim to obfuscate the control-flow of the source code. In this case, opaque predicates, opaque variables, and other misleading constructs

are introduced to confuse the decompilers. A variable V is opaque if it has some property q that is known a priori to the obfuscator, but that is difficult for the deobfuscator to deduce. Similarly, a predicate P is opaque if a deobfuscator can deduce its outcome with great difficulty, while this outcome is well known by the obfuscator [2].
The strength of the technique and the increase in the execution time of the application added by these transformations depend on the complexity of the opaque predicates and their positioning in the code. Figure 2.3 shows a visual representation of the addition of an opaque predicate, while Figure 2.4 shows a simple example.

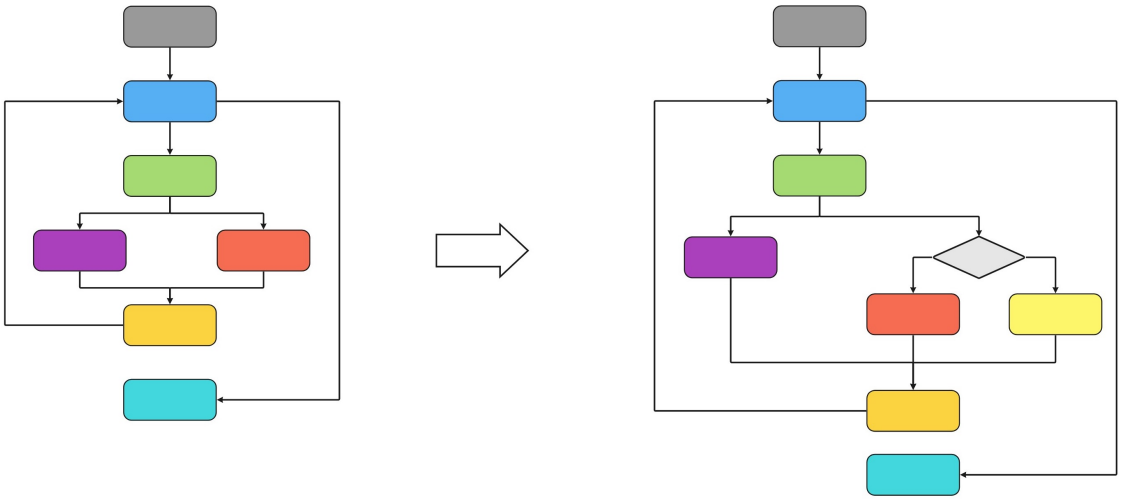


Figure 2.3: Visual representation of opaque transformation

```
int a, b

if (7*a*a - 1 != b*b):
   #always true
   originalCode()
else:
   deadCode()
```

Figure 2.4: Example of opaque predicate

Final considerations on code obfuscation

In conclusion, obfuscation makes the program more difficult for an attacker to understand, as its real code and its real logic is drowned in large amounts of dead and bogus code.

However, it should be noted that there is no guarantee that the obfuscated code will be completely immune to reverse engineering. In fact, given enough time and resources, an attacker will always be able to retrieve useful information from the obfuscated code.

Moreover, although research in this field is progressing, measuring the quality and strength of obfuscation transformations remains an open challenge as it is not yet clear how to evaluate or how good these techniques are, and, furthermore, many of the existing measures of complexity are very vague or based partly on human cognitive abilities, such as *potency*, a measure of how a human reader will be confused in reading a obfuscated code.
On the other hand, code analysis tools are also becoming increasingly sophisticated, implementing techniques that even make it possible to partially reconstruct code from potentially obfuscated binaries.

However, while theoretical results indicate that provably secure obfuscation in general is impossible, its widespread application in malware and commercial software shows that it is nevertheless popular in practice [5].

2.2 Test cases generation (fuzzing)

A program may consist of numerous paths that are only taken under certain circumstances and in the presence of certain data. It may contain loops that are repeated more or less times, depending on the conditions previously fulfilled. It may even contain dead paths that can cause unexpected interruption of the application.
In general, it can be said that the behavior of a program is highly dependent on the input data on which it will run.

For this reason, to obtain a good description of the behavior of a program, firstly, it is necessary to find a set of input data that allows for the most complete execution possible, looking for all possible combinations so that as many paths as possible can be explored. For this purpose, a *fuzzer* was used.

Fuzzing is an automated software testing technique that is generally used to find coding errors or unexpected behavior. It consists of randomly feeding invalid and unexpected inputs and data into a computer program to determine its failure if it happens. This makes it possible to verify the integrity of a program and to better ensure the absence of any kind of vulnerability.
Once the program has been fed with these data, it is monitored, and if an exception or a crash occurs, then the input is saved. This so-called *interesting case* can then be analyzed to find which part of the code is responsible for the crash. Figure 2.5 shows the main steps of a standard fuzzer.

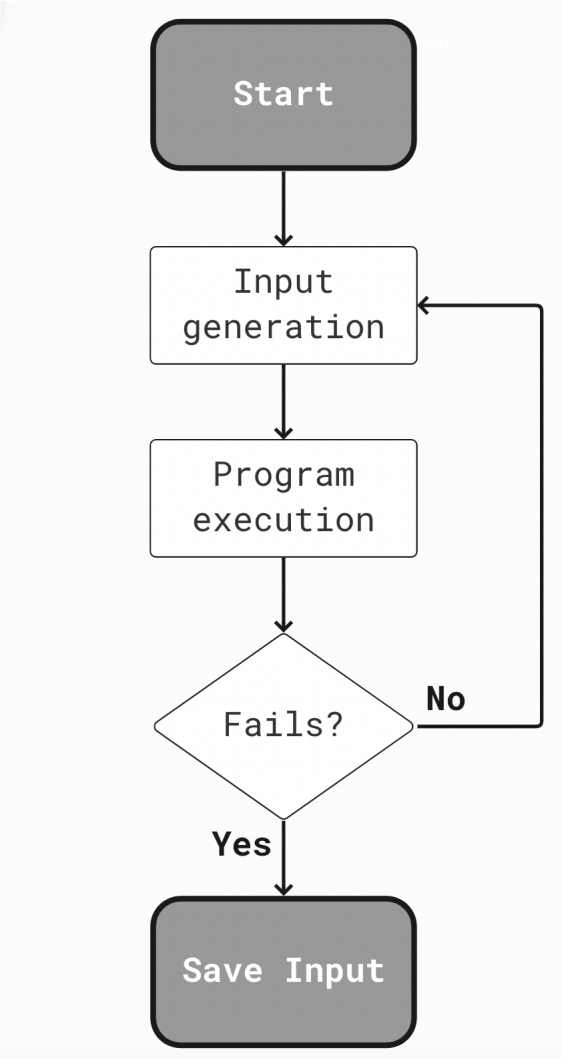


Figure 2.5: Fuzzing process

The principal advantage of using such technique is that a program can be tested without human intervention: a fuzzer can be left running for a long period of time, during which only those cases where errors have been found in the code will be notified.
The main categories into which fuzzing can be divided are [6]:

- *Blackbox Fuzzing*: the fuzzer tries to generate possible inputs without knowing the structure of the program. It basically feeds the program with random data and tries to use its output to try to understand what is happening inside, thus creating a smarter and more efficient input. This implementation is fast but relatively shallow: lack of knowledge about the application and random mutations of input data make it very unlikely that certain code paths will be reached in the tested code;
- *Whitebox Fuzzing*: the fuzzer tries to determine which are the best inputs to generate by analyzing the internal structure of the code. The aim is to maximize code coverage,

i.e. how many paths are reached. Whenever a conditional instruction such as if or while is executed, the code splits into a branch where the instruction is true and another where it is false. The purpose is that if there is an error hidden in some branch, the fuzzer must be able to reach it. To perform this kind of analysis, the program must be instrumented during compilation, i.e. calling a special function every time a branch is run, which logs it as having been run. This solution is certainly more effective than the first, but high overhead is introduced due to program analysis;

- *Graybox Fuzzing*: This approach tries to find a compromise between the respective pros and cons of the approaches mentioned above. It uses lightweight code instrumentation instead of full analysis to calculate code coverage and track the program's state. The input space is explored through mutation. Starting with seed inputs, the fuzzer mutates them using a predefined set of generic mutation operators. The outputs are then examined to determine whether they are sufficient *interesting* and, in these cases, the inputs that generated them will be further mutated to explore more different solutions.

The proposed solution uses *AFL*[2], a *feedback-based fuzzer*. It is a Graybox fuzzer that uses information on code coverage to generate new inputs. In this way, it can cover more paths in the software than a normal fuzzer.

However, since the original objective of this work was to collect data on program execution, the fuzzer was not used in a standard way but rather to collect a set of inputs for an application and build a test suite.

2.3 Data collection

This section aims to go into more detail regarding the collection of data concerning the execution of various programs and their performance when obfuscated. As explained previously, these results were achieved by Stefano Alberto [4]. The detailed workflow of the dataset creation can be seen in Figure 2.6

[2] https://github.com/google/AFL

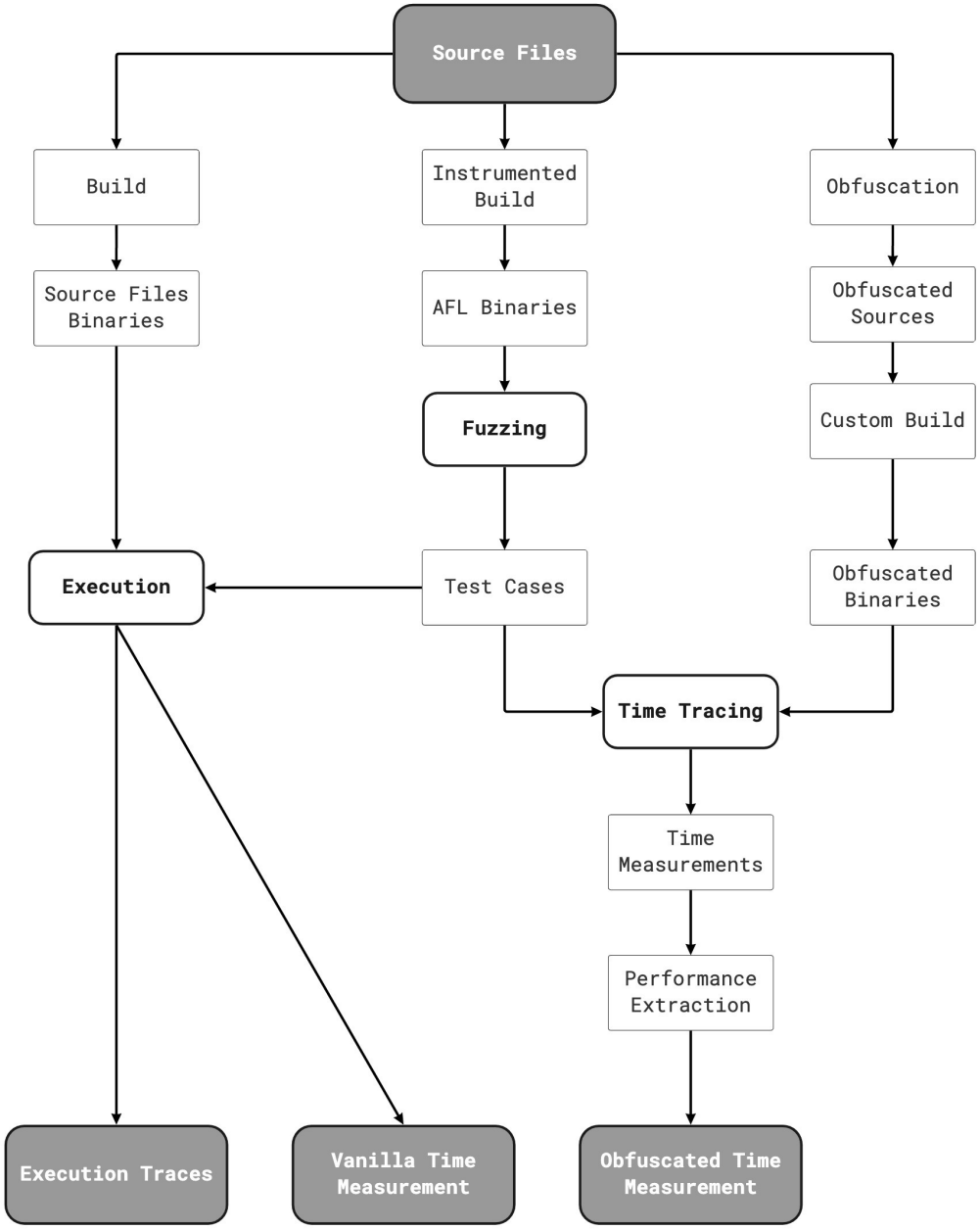


Figure 2.6: Data collection

2.3.1 Application selection

The first step was obviously the choice of the application to analyze.
They should be as heterogeneous as possible, so that very different execution traces can be obtained. This will help to collect as much information as possible and thus better generalize the dataset.
In addition, they will be subject to certain constraints: in fact, a program will only be

suitable if it meets the following requirements:

- *open source*, to apply code obfuscation without breaking the license terms of the application;
- *written in C language*: the obfuscation tool chosen, Tigress, is language-dependent because it needs to perform transformations on the application's source code. Specifically it allows the application of various obfuscation techniques on C code, even if it can show problems when used with code that has advanced C language features. In particular, therefore, Tigress works with programs written with the C99 version of the C language;
- *compatible with Tigress*, otherwise it cannot be obfuscated;
- *deterministic and repeatable execution*: since each execution must be traced for different versions of the same application, it is indispensable that the task execution depends only on the given input and not on external factors that could change between different runs.

To obtain realistic data about real world scenarios, only real software is used.

The chosen application where reported in Table 2.1.

Program Name	*Description*	*Lines Of Code*
aha	Converts ANSI colors to HTM	1075
ascii	Interactive ASCII name and synonym chart	445
colorize	Colorizes text on terminal with ANSI escape sequences	1490
d48	Disassembler for 8048/8041 code	5669
d52	Disassembler for 8052 code	7015
dz80	Disassembler for Z80/8080/8085 code	6517
id3ren	id3 tagger and renamer	2038
prips	Prints the IP addresses in a given range	524

Table 2.1: Selected programs

2.3.2 Application build

Once the set of applications to be analyzed has been chosen, the next mandatory step is to build the source files for these applications. This phase involves the use of a compiler, usually GCC[3], which, in summary, translates the source code of an application, i.e., the written code, into object code, which represents a sequence of statements in the machine

[3] https://gcc.gnu.org/

language.
This process consists of 4 steps:

- *preprocessing*: at this stage, all comments are removed from the source code, all header files (.h) are included, and the macros are replaced with their values. The output is a ".i" file;
- *compiling*: the compiler takes the ".i" file and generates a ".s" file, which contains assembly code;
- *assembling*: the assembler transforms the assembly code into binary code, producing an object file ".o";
- *linking*: the linker links all object files from all source codes of the initial application and links all function calls with their definitions. In the end, the executable file is generated.

However, it is possible to tune the behavior of a compiler through a set of options that affect the workflow of building phases, for example, executing only some of the steps listed before and the single phase, i.e., changing some aspects of preprocessing or compiling phases.
The relevant option to take into account for this work will be the optimization option and the instrumentation option, which refer to the compilation phase and will allow custom implementations to be generated.
Since the building of an application can require some specific options depending on application requirements, in general, what is done is providing a *makefile*, a specific file that describes the steps needed to build the entire application.
Each application is built multiple times to generate different binaries that will be used in different subsequent steps. A custom compilation is needed for the fuzzing phase, to add the instrumentation needed by the fuzzer to trace the application, and, accordingly, to choose how to mutate the generated input. The build commands also need to be modified to address a problem regarding the application of the obfuscation.
Since generally, when complex applications are built, several files .c are generated, but Tigress needs only one file .c to perform a single obfuscation transformation, some options had to be modified via a makefile to merge the various files .c into a single one.
Then, each application is compiled without instrumentation to generate the binaries used in the profiling step.
The output is a set of compiled binaries.

2.3.3 Fuzzing

As explained in Section 2.2, a program may consist of numerous paths that are only taken under certain circumstances and in the presence of certain data.
Choosing input data randomly, without considering the structure and semantics of the code, could lead to the execution of only certain paths or worse, to the raising of some exception or the premature termination of the code. All this would lead to the extraction of traces with poor information inside, which would then be useless later.
The chosen strategy to deal with this problem involves the use of a *feedback-based fuzzer* to

generate a set of test cases for each of the selected applications, and, in particular, *AFL*[4], which is one of the most popular.
Since AFL requires a custom build to understand the structure of the application and the execution path of every run, the first step involves the use of a special AFL utility to build the C source code with the required instrumentation. In particular, the instrumented binaries are obtained by simply using the command "afl-gcc" instead of "gcc". It is now possible to run the fuzzer through all selected applications.

The general AFL algorithm can be summarized as follows:

1. Load user-supplied initial test cases into the queue;
2. Take next input file from the queue;
3. Attempt to trim the test case to the smallest size that does not alter the measured behavior of the program;
4. Repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies;
5. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue;
6. Go to 2.

Therefore, with the initial test cases provided by the user, the fuzzing process can start. The instrumented application is run multiple times with new mutated inputs. Every time an input is considered interesting by the fuzzer (a crash or a timeout happens), it is saved in the queue to be mutated when generating other inputs.
However, the main purpose of a fuzzer is to report to the user unexpected behaviors of the application under analysis; therefore, since the aim of this phase is instead to build a good set of test cases to cover as many execution paths as possible, a different approach needs to be used. This new approach consists of changing the operating logic of the fuzzer so that whenever a new path is found in the code, the input that generated it is saved and stored in a dedicated folder.
Additionally, to cleanly terminate the execution of the fuzzer, a threshold of saved test cases was inserted beyond which execution must stop. Without this addition, the fuzzer would have continued to run until a certain number of crashes were reached, which was not intended in this case. The whole logic of the fuzzing process is shown in Figure 2.7.

2.3.4 Obfuscation

This section aims to explain the various steps needed to practically obfuscate the source code of the selected applications using the Tigress obfuscator.

[4] https://github.com/google/AFL

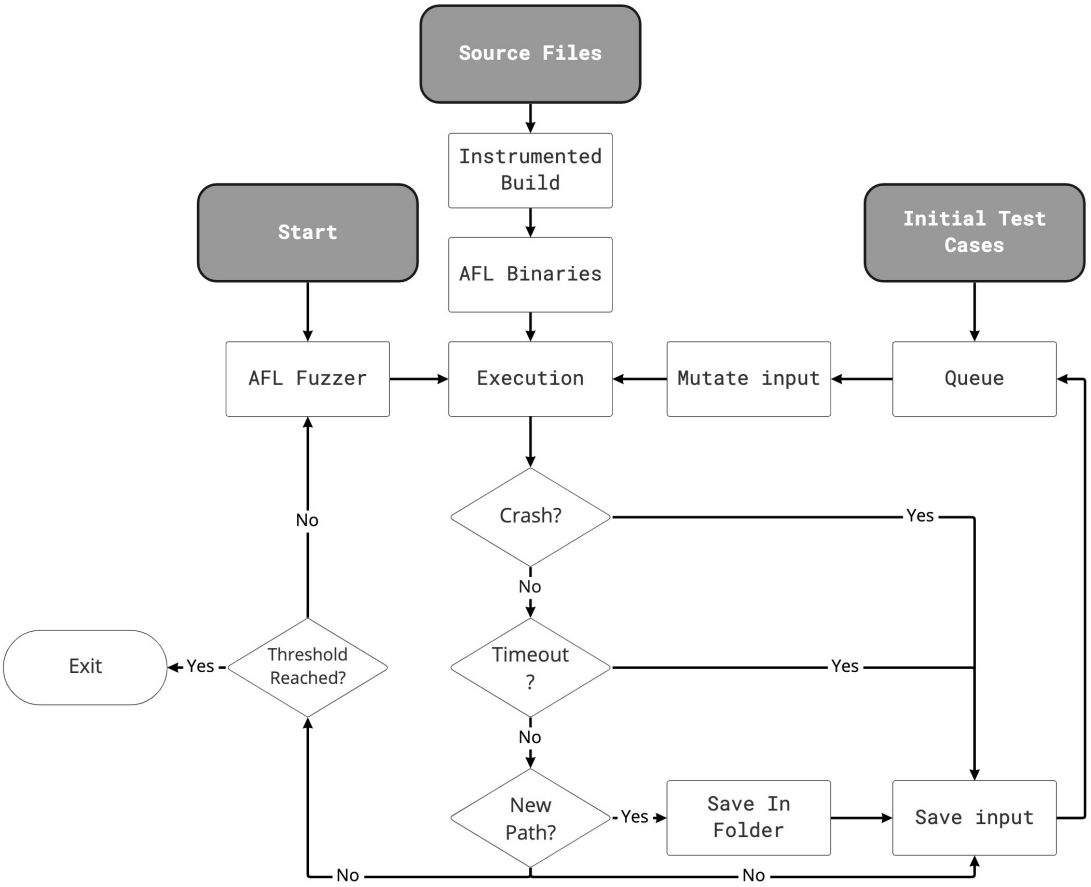


Figure 2.7: AFL loop

One of the peculiarities of Tigress is that it can obfuscate a single function at a time. Since for the purpose of this work this feature represents a limitation, to address this problem, each obfuscation technique chosen is applied to all the functions defined in the source code. In this way, it is possible to collect information about each function of a program in a single run.
Another constraint imposed by the obfuscator is, as explained in Section 2.3.2, that it can be used only on a single C file. The solution, as discussed, consists of merging all .c files into a single one.
Finally, to properly use Tigress, the header file *tigress.h* must be included in the source code, and the function *init_tigress* must first be invoked in the code. With this last step, it is possible to run the obfuscator without problems.

2.3.5 Execution tracing

Once all the applications were built and a set of input cases was generated for each one, the next step is to run all the applications to extract useful information on their execution

and performance.
A trace will contain a complete representation of an execution of a specific application with a specific set of inputs among those generated by the fuzzer. This was done by implementing a custom solution in which all needed information was extracted from the execution. Instruction after instruction, the information about the executed function, the offset from the beginning of the function, and the actual executed instruction (in the form of an opcode) are saved in the output file.
This procedure was repeated for each test case of a program and for all programs, choosing to ignore functions called up by standard libraries to keep track only of functions written specifically for the program under analysis.
It should be noted that these extractions were only carried out for unprotected files, i.e., those where no obfuscation transformation was applied. This is for the simple reason that the network that will be built later must be able to predict the overhead added by the application of obfuscation before applying it. Therefore, the starting point will be the vanilla version of the program.

2.3.6 Performance extraction

So far, each generated trace represents the execution of a program with a specific set of inputs. This means that for a single program, there will be as many traces as the possible inputs previously found by the fuzzer. As they are, these traces are still far from being used in a machine learning model, but can already be seen, in some ways, as the set of inputs to be fed into our network. Therefore, if, on the one hand, there are the input data, what is missing are the values that the network will have to predict and that will in some way link the execution of the vanilla programs with their obfuscated counterpart.

To have a measure of performance, the first step is to measure the time in terms of clock cycles required for each execution we are interested in.
With custom instrumentation, it is possible for the compiler to call a user-defined function just after each function entry and just before each function exit.
This user-defined function allows to get the clock cycles and to save them in two separate vectors: one when entering and one when exiting the function. In this way, it is possible to measure the execution time of a function in terms of the number of clock cycles used without interrupting the execution of the program and without adding a considerable overhead that would have distorted the obtained values.

As mentioned above, interest in measuring performance is limited to the function that belongs to the application under analysis, since all code executed outside it will not be considered by the obfuscator when applying the transformations.
In fact, to measure the drop in performance between a function and its obfuscated version, a ratio between the two measured timings will be considered. Thus, by also taking the execution time of the external code into account in the computation of the first measurement, the two measures would not take into consideration the execution of the same code, and therefore their comparison will be meaningless.
Consequently, there is no reason to keep track of the execution time of the various calls to external library functions.
The solution involves the use of the same user-defined function used before to trace the

execution time of calls only to external library functions. In this way, the output generated when the instrumented application is running contains all the data about each time the application entered and exited from each function, also the library ones.

To obtain the time spent by every single function inside of a program without considering all the time spent in all the other functions called during its execution, the needed step is to transform the series of timings in entering and exiting the functions into an ordered call tree. In this way, it is possible to understand if a function was called inside another. Reconstructing the call tree of the entire program makes it possible to obtain all individual execution times for each function.

One critical aspect to take into account is that in general the execution time of an application depends not only on the internal data and the application code, but also on external factors, such as the operating system and its scheduler, the status of the various levels of cache[5] and main memory, or various interrupts[6] that can greatly delay the time spent within a program. To reduce as much as possible the effect of these factors, possible countermeasures concern the reduction of processes running in parallel and reducing all possible background processes, avoiding, in particular, as much as possible any type of I/O interrupt.
Furthermore, the applications were executed multiple times (100 in particular) to reduce the random fluctuation in the timing measurements.
All this information will be stored in a special file that will contain, in particular, the list of functions called with the number of clock cycles spent in execution, for both vanilla and obfuscated versions.

2.4 Data compression

Data compression is the process of encoding, rearranging, or modifying data in which the primary objective is to minimize the amount of data to be transmitted. An example of data compression involves transforming a string of characters from some representation (such as ASCII) into a new string (e.g., of bits) that contains the same information but whose length is as small as possible [7]. When dealing with a large amount of data, its storage and transmission is likely to increase at an enormous rate. To overcome these problems, data compression transforms the original data into a compact form by recognizing and using existing patterns into it. The reduction of file size allows to store more information in the same storage space and to reduce the transmission time.

In general, a compression technique can affect the quality of the data depending on the chosen criteria and the type of requirement imposed when the compressed data are reconstructed [8]. In this case, it is common to divide these techniques into 2 groups:

[5] A cache is a smaller, faster memory, located closer to a processor core, which stores copies of the data from frequently used main memory locations.

[6] an interrupt is a request for the processor to interrupt currently executing code, so that the event can be processed in a timely manner.

- *lossless compression*: removes bits by locating and removing statistical redundancies. No information is lost, so the reconstructed data is identical to the original. It is used in applications where loss of information is undesirable, such as text or medical imaging. Lossless compression will often have a smaller compression ratio, with the benefit of not losing any data in the file;

- *lossy compression*: file size is reduced by permanently eliminating certain information. Specifically, it eliminates redundant or unnecessary information and reduces the complexity of existing information. Lossy compression can achieve much higher compression ratios, at the cost of possible degradation of file quality.

2.4.1 Run-length encoding

Run-Length Encoding (RLE) [9], is a common lossless data compression technique. It compresses data by reducing repetitive and consecutive information called runs.
The key idea of this algorithm is to scan the data to be compressed and for each item record the *run-length*, that is, the number of times it occurs, followed by the item itself. For example:

- the string *AAAAAAFDDCCCCCCCAEEEEEEEEEEEEEEEEE* will be encoded as *6A1F2D7C1A17E*;

- the string *AAAAHHHEEM, HAHA.* will be encoded as *4A3H2E1M1,1 1H1A1H1A1.*

From these examples, it is possible to notice that RLE compression is only efficient with data that contain many repetitions: Run-Length Encoding is especially useful for data that contain many runs. On the other hand, with files that do not have many runs, it could greatly increase the file size.
Therefore, the compression ratio of the algorithm depends on the data.

This algorithm can be summarized in pseudocode as follows:

```
1  i ← 0
2  j ← 0
3  result ← ' '
4  while i < length(string) do
5  |   cnt ← 1
6  |   j ← i + 1
7  |   while string[i] == string[j] and j < length(string) do
8  |   |   cnt ← cnt + 1
9  |   |   j ← j + 1
10 |   end
11 |   result ← 'cnt string[i]'
12 end
13 return result
```

2.5 Code disassembly

Disassembly is the process of recovering a symbolic representation of the instructions of the machine code of a program from its binary representation [10]. The goal of these techniques is to produce a higher-level representation, usually in assembly code, of a program that allows a human reader to understand the structure of the program.
The assembly code file can be used in reverse engineering processes to establish the logical flows of the computer program or its vulnerabilities in the real-world running environment [11].
Decompilation and reverse engineering is often prohibited by software license agreements, as the source code behind the software that is released to the public is something the programmer has created in a private way. On the other hand, decompilers can also be of great benefit to programmers, since they allow, for example, reconstructing lost source code from a binary executable.
The task of recovering these instructions is often complicated by the presence of non-executable data, such as jump tables, alignment bytes, etc.,in the instruction stream, which, however, are needed to identify the various instructions. The presence of variable-length instructions, commonly found in CISC architectures such as the widely used Intel x86, results in an additional degree of complexity and renders simple heuristics for extracting instruction sequences ineffective [12].

Disassembly techniques can be categorized into two main classes:

- *static techniques* analyze the binary structure statically, parsing the instruction opcodes as they are found in the binary image;
- *dynamic techniques* monitor execution traces of an application to identify the executed instructions and recover a (partial) disassembled version of the binary.

Both static and dynamic approaches have advantages and disadvantages. Static analysis takes into account the complete program, while dynamic analysis can only operate on the instructions that were executed in a particular set of runs. Therefore, it is impossible to guarantee that the entire executable is covered when dynamic analysis is used.

2.5.1 Capstone

Capstone[7] is an open source disassembly framework based on the LLVM[8] compiler infrastructure, designed to provide a simple, lightweight API that handles most popular architectures, including x86/ x86-x64, ARMS, MIPS, and others. Although the use of this tool is very simple, it provides details on the disassembled instruction, including instruction opcodes, mnemonics, as well as some semantics of the disassembled instruction, such as a list of implicit registers read and written.

To use Capstone, the first step involves choosing the hardware architecture and the hardware mode, which in the case of x86 architecture are the possible access modes of the

[7] https://www.capstone-engine.org/

[8] https://www.llvm.org/

registers, i.e. 16, 32, or 64-bit.
Basically, Capstone takes a memory buffer containing a block of code bytes as input and outputs the correspondent disassembled instruction.

Chapter 3

Background on deep learning

The purpose of this chapter is to introduce the reader to the main topics related to artificial intelligence, as well as the solutions chosen during the preprocessing phase to improve network performance.

3.1 Introduction to deep learning

Over the recent years, Deep Learning [13] has had a tremendous impact in various fields of science. Neural networks can creatively generate texts, music pieces, and even paintings in the style of the old masters. These achievements are based on many years of research on neural networks and Machine Learning.

To better understand what a neural network is and how it works, it is necessary to make a brief digression and define some terminology, starting with the concept of artificial intelligence.

- *Artificial Intelligence* (AI) is a science field that focuses on the development of algorithms capable of processing information to make future predictions. Therefore, it can be any technique that enables computers to mimic human behavior;

- *Machine Learning* (ML) is a subset of AI that focuses on teaching an algorithm to learn from data without explicitly programming how to process input information.

- *Deep Learning* (DL) takes the idea of ML even further. It is a subset of ML that focuses on using Neural Networks to automatically extract useful patterns from raw data and then using them to learn to perform specific tasks.

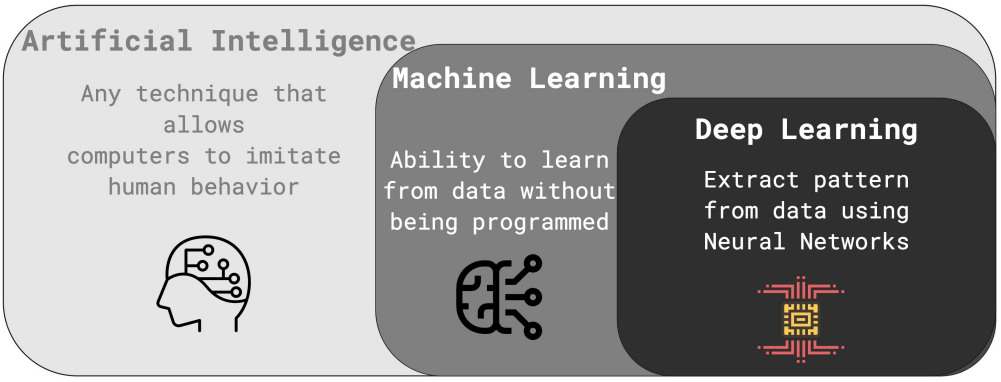


Figure 3.1: General subdivision of AI, ML and DL

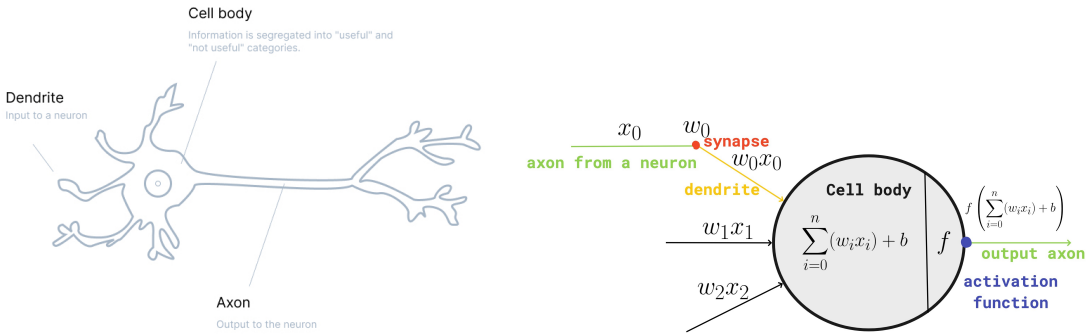
Although the fundamental building blocks of deep learning and their algorithms have existed for decades, they have only been rediscovered in the last few years. This is basically for three reasons:

- These algorithms are hungry for data, and today we are living in the world of big data, where we have more of it than ever;
- Machine Learning models require high computational capacity and can benefit from modern advances in GPU architectures;
- progress in the development of powerful and efficient libraries made it possible to develop these algorithms.

The remainder of this section aims to introduce the reader to the basic concept of Deep Learning, which will help to understand the observations described later in this thesis. In particular, we will start with the concept of neuron and Perceptron and then focus on the general architecture of a neural network.

3.1.1 Perceptron

Perceptron [14] can be seen as the basic building block of deep learning: it is basically a binary classifier that consists of a single neuron. The analogy with a biological neuron is quite simple, as Figure 3.2 shows. A biological neuron is made up of multiple dendrites, a nucleus, and an axon. When a stimulus is sent to the brain, it is received through the synapse located at the extremity of the dendrite. It is transported to the nucleus when processed together with other signals coming from other receptors. After all these signals have been processed, the nucleus will emit an output signal through its single axon.



(a) Biological neuron (b) Artificial neuron

Figure 3.2: Comparison between neurons

On the other hand, forward propagation of information through a single artificial neuron starts with the definition of a set of *inputs* $[x_0, ..., x_n]$. Each of these inputs is multiplied by their corresponding *weights* $[w_0, ..., w_n]$ and then added together. The single number that comes out of this operation is added to a *bias* term b and the output is passed through a non-linear *activation function* f and the result is the *prediction* $\hat{y}$. Mathematically speaking:

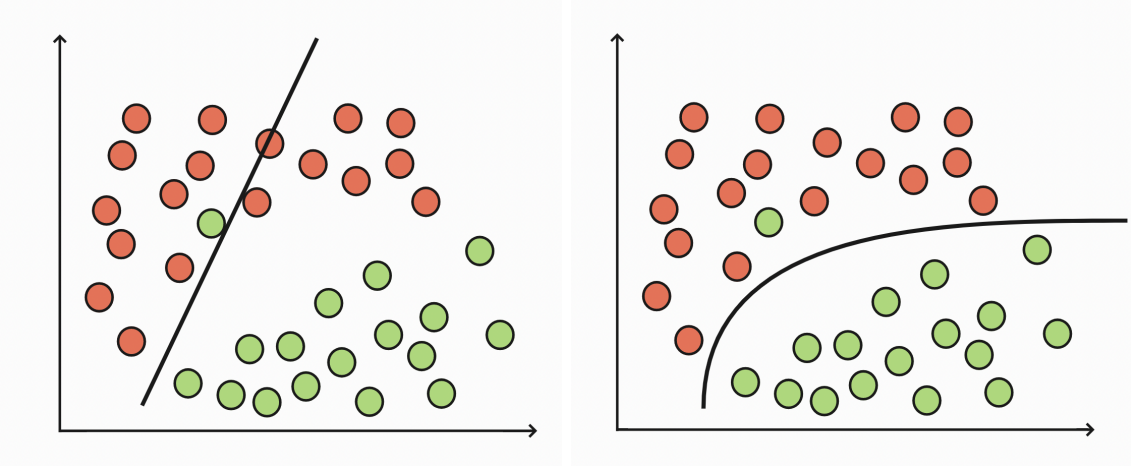
$$\hat{y} = f\left(\sum_{i=0}^{n}(w_i x_i) + b\right) \tag{3.1}$$

We can rewrite this equation using linear algebra:

$$\hat{y} = f(b + \mathbf{X}^T\mathbf{W}) \tag{3.2}$$

where $\mathbf{X} = \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix}$ and $\mathbf{W} = \begin{bmatrix} w_0 \\ \vdots \\ w_n \end{bmatrix}$.

The importance of the activation function is to introduce non-linearity into our system [15]. In real life, almost all data are non-linear and, without an activation function, a model would not be able to deal with these problems. Imagining to separate the red point from the green ones, having access only to a linear function (Figure 7.16a), there will be no possibility of producing good results, no matter how deep or complex the network is: the linear activation function will always produce a linear result. The addition of non-linearity allows arbitrarily complex functions to be approximated (Figure 7.16b).



(a) linear activation function (b) non-linear activation function

Figure 3.3: Comparison between decision boundaries

Particular importance is given to the *ReLU* activation function [15], which is probably the most widely used. It is defined as follows:

$$ReLU = \begin{cases} 0 & \text{for} \quad x \leq 0 \\ x & \text{for} \quad x > 0 \end{cases} \tag{3.3}$$

As can be seen, the output of this function will be the input itself if it is positive, 0 otherwise. There are some advantages as a consequence. First, it is easy to compute its derivative (since it is the slope, so it will be 0 for negative values and 1 for positive ones). Additionally, another benefit of the ReLU activation function is that it is capable of output a real zero value: the negative inputs that output 0 allow activation of only some neurons. This speeds up the learning phase.

3.1.2 Neural networks

Having described how a single neuron is made and how it works, it is now possible to build *Neural Networks*.
Starting from a single neuron, a multi-output Neural Network can be defined simply by adding more Perceptron to the initial configuration [16], as Figure 3.4 shows.
A Neural Network is made up of 3 components (layers):

- *input layer*: it takes raw input from the dataset. No computation is performed at this layer. The nodes here simply pass the information to the hidden layer;

- *hidden layer*: as the name suggests, the nodes in this layer are not exposed, so we cannot directly see the output of these nodes. The hidden layer performs all kinds of computation on the features entered through the input layer and transfers the result to the output layer;

- *output layer*: final layer of the network that brings the information learned through the hidden layer and makes the final prediction.

In particular, each of the circles in the hidden and output layers represents a single neuron, with its own set of weights and bias, as already seen. The output goes into the activation function and becomes the input of the next layer. A layer in which every input is connected to every output is called *Dense* or *Fully Connected*.

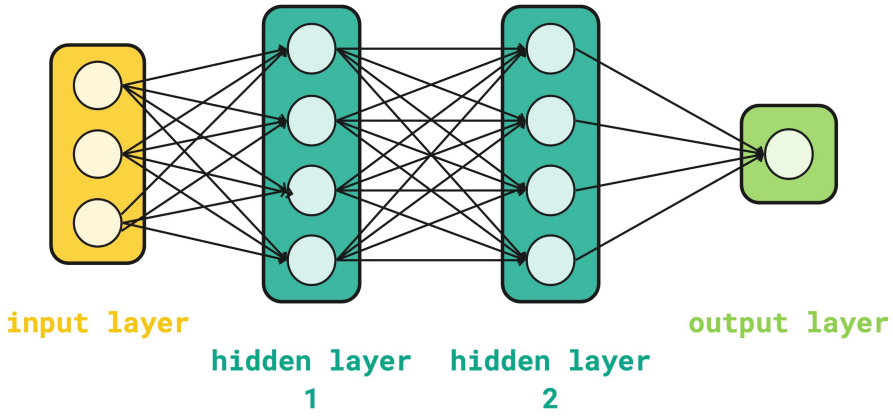


Figure 3.4: 3-layer Neural Network

Loss function

At this point, imagining to make a prediction, it could be possible to feed the network with an input vector and see what the output will be. At the moment, the model is not trained, so the prediction will almost certainly be wrong: the network has no idea what our input data mean because it has never seen anything before. The network has to learn to handle the input data and obtain reasonable results. To do this, it has to be defined what it means to get a wrong prediction, and this is basically the purpose of *loss function* [17].
The loss function specifies how good our neural network is for a certain task. The intuitive way to compute it is to take a sample of data, pass it through the network to get the prediction, and compare it with the actual number we want to get.

$$\mathcal{L}(f(x;\mathbf{W}),y) \tag{3.4}$$

where $f(x;\mathbf{W})$ is the prediction and y is the true value.
Thus, the closer the prediction is to the actual value, the lower the loss will be, and, of course, the farther away they are, the more error there will be.
Assuming to have more than one sample, the aim is to understand how the model performs on average, so *empirical loss* will be computed, which is just the mean of all individual loss functions from our dataset:

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^{(i)};\mathbf{W}),y^{(i)}) \tag{3.5}$$

Where $\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, ..., \mathbf{W}^{(n)}\}$ is the group of all $\mathbf{W}$ in every layer.
When training the network, the aim is to minimize this function.

There are many loss functions, but, in particular, when dealing with continuous real numerical values, *Mean Squared Error loss* (MSE) is the common choice [17]. It measures the squared difference between the ground truth and our predictions, averaged over the entire dataset. The mathematical definition is the following:

$$\mathbf{J}(\mathbf{W}) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - f(x^{(i)}; \mathbf{W}))^2 \tag{3.6}$$

Training neural networks

Having described the architecture of the neural network and defined a measure to quantify its errors, the next step is to put all these elements together to understand what it means to train a network. The aim of the training phase is to find a set of weights $\mathbf{W}^*$ that will give the minimum loss function on average throughout the entire dataset:

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)}) \tag{3.7}$$

$$\mathbf{W}^* = \underset{\mathbf{W}}{\operatorname{argmin}}\, \mathbf{J}(\mathbf{W})$$

To solve this minimization problem, the derivative with respect to $\mathbf{W}$ will be computed, which tells the direction to take to maximize the loss and therefore takes a step in the opposite direction, which is the right direction to minimize this function. This process will be repeated taking into account the point just computed and making a further step towards the opposite direction of the derivative. This will continue until convergence to a local minimum is achieved. This algorithm, called *Gradient Descend*, can be summarized in pseudocode as follows:

```
1 Randomly initialize W
2 while not convergence do
3 |   compute ∂J(W)/∂W
4 |   W ← W − η ∂J(W)/∂W /* Update weights */
5 end
6 return W
```

Gradient computation is performed for all weights and is called *backpropagation technique*: starting from the output, the update is carried out level by level until it returns to the input level. The first step is to quantify the error made by the network to tell when it is making a mistake. To do that, the prediction and the true value will be compared. If there is a significant discrepancy between them, the loss will reach high values and it will be necessary to move closer to the true answer.
An extremely important parameter is η, *learning rate*. It determines the amplitude of the step that must be taken in the direction of the gradient. It influences both learning time

and convergence. If it is too small, then the model may get stuck in a local minimum, because the steps towards the gradient will be too small. However, if it is too large, the model may diverge. In practice, one common way to choose a good learning rate is to try a number of different ones and see which work best. Alternatively, it can be used to what is called *adaptive learning rate* in which the learning rate is not fixed but can change taking into account various aspects, such as how large the gradient is or how fast the learning phase is happening.

Adam optimizer

Adam [18] is a method for efficient stochastic optimization that can be considered state-of-the-art among gradient optimizers. Adam uses an adaptive learning rate based not only on the moving average of the first moment (m) but also on that of the second moment (v). Moments are described as an update term that depends on iterations and the gradient of the loss function. This optimizer uses two parameters (β_1 and β_2) to control the decay rate of moments. It also introduces a smoothing factor to improve the numerical stability of the algorithm.

The algorithm updates the exponential moving averages of the gradient (m_t) and the squared gradient (v_t) where the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these moving averages. The moving averages themselves are estimates of the 1st moment (the mean) and the 2nd raw moment (the uncentered variance) of the gradient. These moving averages are initialized as vectors of 0's, leading to moment estimates that are biased toward zero, and bias-corrected estimates $\hat{m}_b$ and $\hat{v}_b$ are computed for the very purpose of minimizing the effect of the initialization bias.

Having $f(\theta)$ a noisy objective function: a stochastic scalar function that is differentiable with respect to parameters θ, the purpose of the algorithm is to minimize the expected value of this function, $\mathbb{E}[f(\theta)]$ with respect to parameters θ.
The overall algorithm requires:

- α: Stepsize
- $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for moment estimates
- $f(\theta)$: a stochastic scalar function that is differentiable with respect to parameters θ
- θ_0: Initial parameter vector

And its pseudocode is as follows:

$m_0 \leftarrow 0$ /* Initialize 1st moment vector */
$v_0 \leftarrow 0$ /* Initialize 2nd moment vector */
$t \leftarrow 0$ /* Initialize timestep */
while θ_t *not converged* **do**
  $t \leftarrow t + 1$
  $g_t \leftarrow \nabla\theta f_t(\theta_{t-1})$ /* Get gradients w.r.t. stochastic objective at timestep t */
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ /* Update biased first moment estimate */
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ /* Update biased second raw moment estimate */
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ /* Compute bias-corrected first moment estimate */
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ /* Compute bias-corrected second raw moment estimate */
  $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ /* Update parameters */
end
return θ_t

Where $f_1(\theta), ..., , f_T(\theta)$ denote the realisations of the stochastic function at subsequent timesteps $1, ..., T$.

Overfitting and underfitting

The main purpose of any machine learning algorithm, as far as a supervised learning problem is concerned, is to be able to correctly classify a new observation that has never been seen before and is not part of the data used to train the network. This concept is called *generalization*.

Usually, when training a model, we feed it with samples from the test set and compute error measurement and statistics to obtain the training error and then try to reduce it as much as possible.
However, what is crucial when testing the network with new observations is reducing the generalization error as much as possible. Typically, the generalization error of a machine learning model is estimated by measuring its performance on the test set. This set includes samples separated from the train set, and thus has never been analyzed by the model. In summary, the two factors to understand how well a machine learning model performs are:

- small training error;
- small gap between training and test error.

In this scenario, two of the main problems related to machine learning are overfitting and underfitting [19].
Overfitting occurs when a model or machine learning algorithm captures the noise in the data. This means that the neural network at a certain time during the training phase does

not improve its ability to solve the problem anymore but just starts to learn some random regularity contained in the set of training patterns. Overfitting is often the result of an over-complicated model.

Instead, *underfitting* occurs when the model is incapable of capturing the variability and overall trend of the data. It is often the result of an over-simplified model.

Figure 3.5 shows the typical relationship between capacity (i.e. its ability to fit a wide variety of functions, so it can also be seen as the complexity of the mode) and training and test error.

When capacity is low, both training and test errors are high. This is what the underfitting looks like. As capacity increases, the training error decreases, and, initially, the test error also decreases. But then it starts to increase as a result of overfitting. Therefore, the optimal model capacity is the one in which the test error is minimum [20].

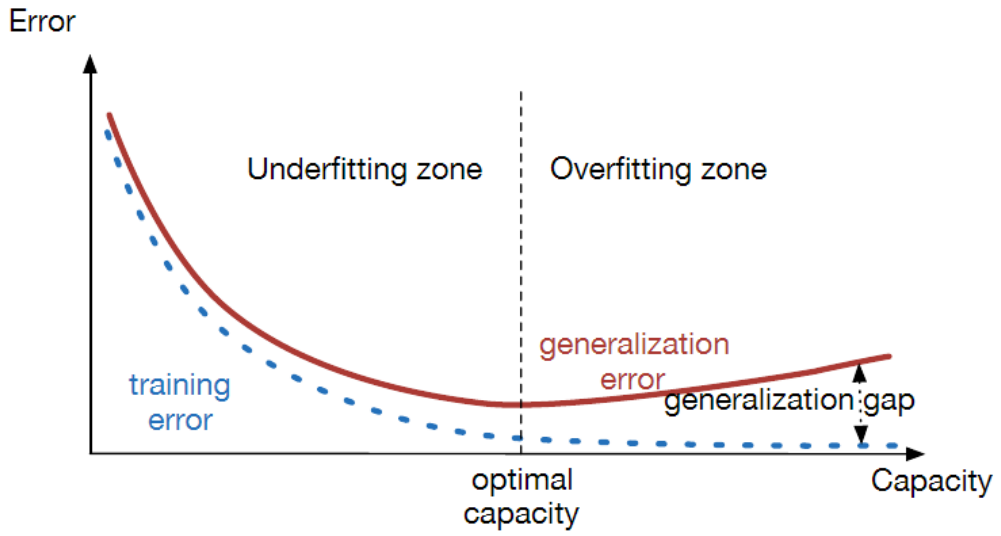


Figure 3.5: Underfitting vs. overfitting

3.1.3 Recurrent neural networks

As seen so far, Artificial Neural Networks are computational models inspired by the brain. They model very complex functions with many parameters that can associate an input to a desired output. They are properly trained to find the right values for these parameters to mathematically transform each input into its right output.

Neural Networks work mainly with vector data types or images. However, not all data can be effectively presented in this way. For instance, text or time series are better modeled as time sequences. Some possible applications of sequences of data can be as follows:

- understanding a question and replying with more or less complex phrases;

- looking at an image and giving it a description or a label;
- forecasting the future trajectory of a ball giving its past position and its current one.

etc.

Standard neural networks go from input to output in one direction. Therefore, they cannot maintain information about previous events when a sequence of inputs is given to it. A newer network architecture is required that is capable of understanding the dependencies between the individual elements of the sequence and the previous ones.

The first possible solution to this problem is given by *Recurrent Neural Networks* (RNNs) [21].

Figure 3.6 shows a simplified version of a feed-forward Neural Network composed only of a collapsed layer in the green box, an input vector of length m and an output vector of length n.

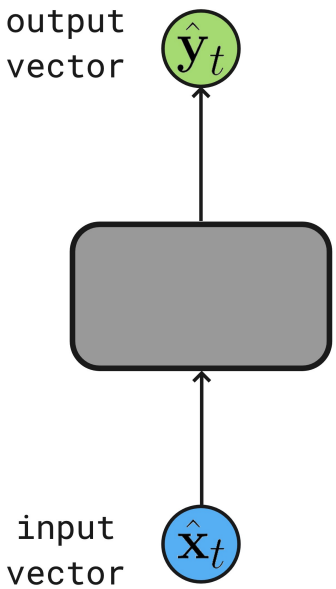


Figure 3.6: Feed-forward Neural Network

It could be possible to feed a sequence to this model simply by concatenating as many of these blocks as the time steps in the sequence, as shown in Figure 3.7 on the left. However, in its current form, such an architecture has no concept of memory or notion of a relationship between time steps; therefore, the output vector at a particular time step will be a function of only the input at that time step and will not take into account past information.

What is needed is a way to relate the network computations at a particular time step to its prior history. It is done by adding a so-called *internal state* $\mathbf{h}_t$, a special value that is maintained over time and can be passed forward over time. This parameter can capture important information about a specific time step and add this information to the next input in the chain. The result is that the output at time t now depends on both the input $\mathbf{x}_t$ and the past memory $\mathbf{h}_{t-1}$ (see Figure 3.7 on the right).

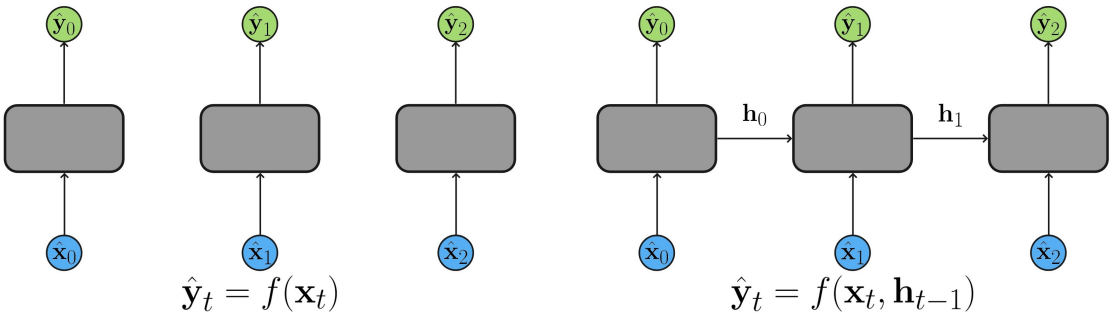


Figure 3.7: Stateless vs stateful architecture

The same architecture shown in Figure 3.7 can be visualized in a more compact way, when the relationship between different time steps is represented by a loop (Figure 3.8).

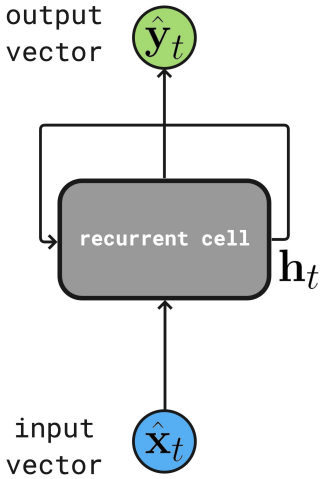


Figure 3.8: RNN cell

RNN can maintain the internal state h_t and apply at each time stamp a function f_W parameterised by a set of values $\mathbf{W}$ used to update this internal state. The key concept is that the new value of h_t is based on the internal state of the previous time step, as well as the input of the current time step. More formally:

$$\mathbf{h}_t = f_W(\mathbf{x}_t, \mathbf{h}_{t-1})$$

This set of weights $\mathbf{W}$ will be learned through the network during the course of training. $\mathbf{W}$ will be the same for all time steps considered in the sequence. The update of the hidden state and the output of the network are described by the following formula:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}^T \mathbf{h}_{t-1} + \mathbf{W}_{xh}^T \mathbf{x}_t) \tag{3.8}$$

Where $\mathbf{W}_{hh}$ represents the matrix of weights related to the hidden state and $\mathbf{W}_{hx}$ the one related to the input. Note that these matrices are independent because we have two different inputs to the state update equation. The output for a given time step is defined as follows:

$$\hat{\mathbf{y}}_t = \mathbf{W}_{hy}^T \mathbf{h}_t \tag{3.9}$$

Where $\mathbf{W}_{hy}$ is the weight matrix related to the output.

The next step is to understand how to measure the goodness of the model. As seen in Section 3.1.2, a loss function is needed. The unrolled version of the RNN may be more helpful. It is possible to compute a loss for each of these individual time steps based on what the output at that time is. Finally, all of these losses can be summed to generate an overall loss.

It is possible to apply this idea of sequence modeling to many tasks:

- *Many to one*: taking a sequential input and mapping it into a single output;
- *One to many*: taking a single input and generating a sequence of outputs;
- *Many to many*: taking a sequential input and generating another sequential output;

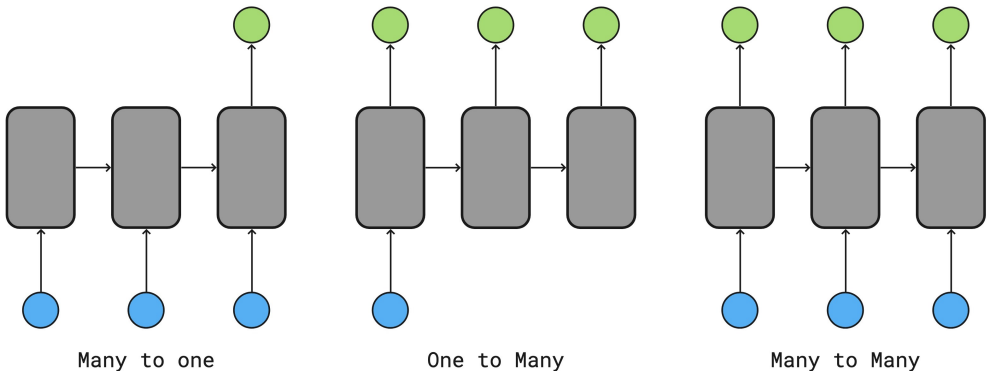


Figure 3.9: different tasks using RNN architecture

Backpropagation through time

As explained above (see Section 3.1.2), the main aspects of training a neural network can be summarized as follows:

1. a forward pass through the network is done, going from input to output, computing the gradients;
2. the gradient is propagated back downward through the network, taking the derivative of the loss with respect to the weights learned by the model;
3. the parameters are updated to minimize the loss.

As regards recurrent neural networks, the forward pass consists of going forward across time, computing the individual loss for each individual time step, and then summing them together. The error will also be backpropagated individually across time steps and then across all time steps to the point where we are currently in the sequence, to the beginning [22].

Gradient issues

During the backward pass, the gradient computation involves numerous matrix multiplications that involve the weight matrices seen previously. With too many weights or gradient values that are $>> 1$, during training, the gradient can explode.
It is also possible to have the opposite problem, with values $<< 1$.
The latter problem is called *vanishing gradient* and can be extremely problematic. Multiplying many small numbers together will bias the network to focus on short-term dependencies and ignore long sequences that exist in the dataset. There exist some different ways to mitigate this problem; one of them is to change the network architecture using more complex recurrent units that can handle longer dependencies [22].

3.1.4 Long short term memory

Long Short-Term Memory (LSTMs) [23] are a special type of neural network designed to solve the problem of vanishing gradient and to retain information of longer sequences. LSTM, like basic RNN, also has a chain-like structure, but its internal repeating structure is more complex, as can be seen in Figure 3.10.

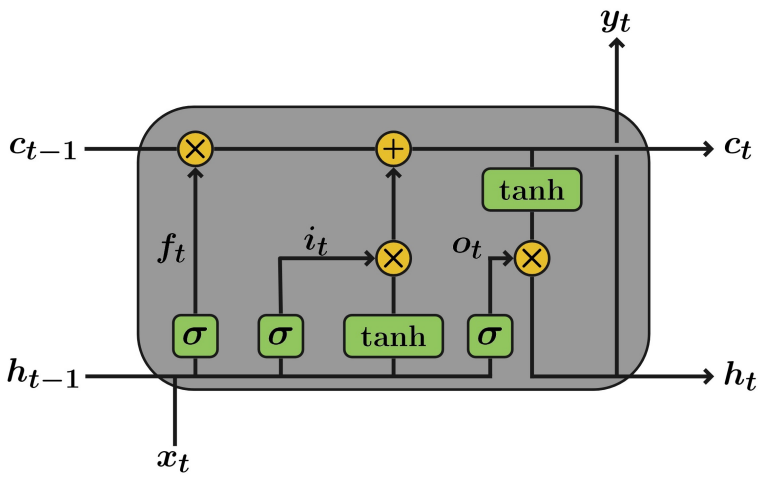


Figure 3.10: LSTM cell

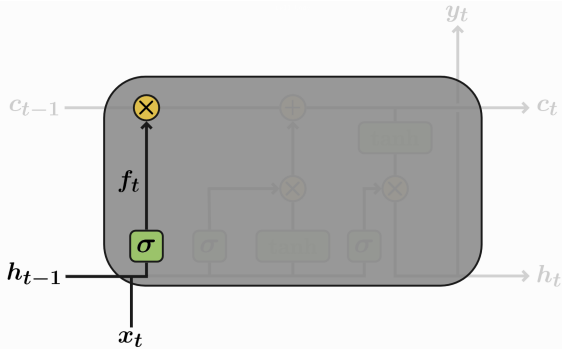
The key idea behind its structure is the introduction of a *gate*, a structure that has the purpose of selectively adding or removing information from the state. This is done using standard operations, such as the application of activation functions and matrix multiplications. Ultimately, gates can control what information passes through the recurrent cell. For example, a gate composed of a sigmoid activation function will force anything that passes through it between 0 and 1, so in a certain way it can decide how much information coming from the input can pass.
LSTMs use this type of operation to process information by:

1. forget the irrelevant information;
2. storing the most relevant new information;

3. updating the internal cell state;

4. generating an output.

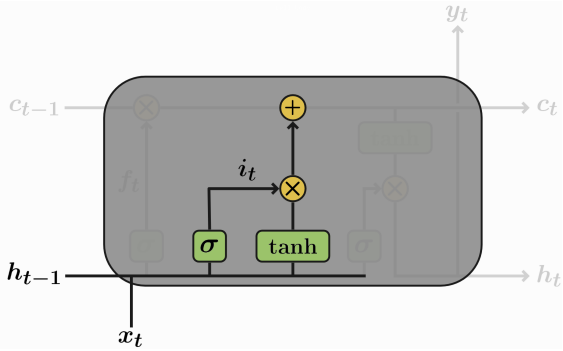
The first step is to forget irrelevant parts of the previous state, and this is achieved by taking the previous state and passing through one of the sigmoid gates. This is done by *forget gate*.



$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

Figure 3.11: forget gate

Where $\mathbf{W}_f$ and $\mathbf{b}_f$ are the weighs and bias related to the forget gate.
The next step is to determine which part of the new information is relevant and store this in the cell state. This is done by *input gate* and has two parts. First, a sigmoid layer decides which values must be updated ($\mathbf{i}_t$). Next, a tanh layer creates a vector of new candidate values, $_t$, which could be added to the state. In the next step, these two values are combined to create an update to the state.

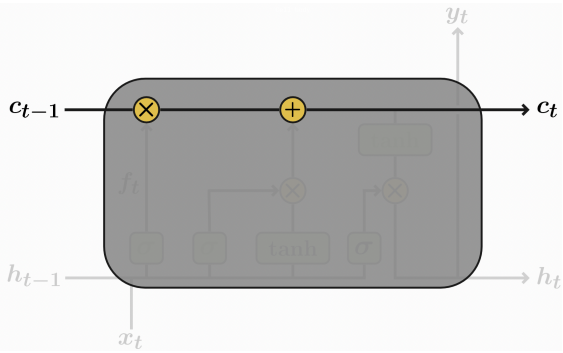


$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C)$$

Figure 3.12: input gate

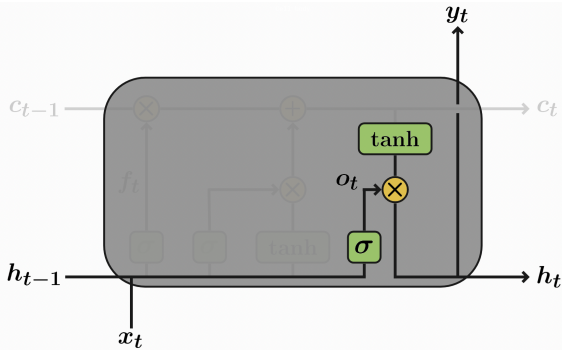
Where $\mathbf{W}_i$ and $\mathbf{W}_C$ refer to the weight correlated with the previous input and the candidate cell state, while $\mathbf{b}_i$ and $\mathbf{b}_C$ are the corresponding biases.
The old cell state $\mathbf{C}_{t-1}$ is then updated to the new cell state $\mathbf{C}_t$. $\mathbf{C}_{t-1}$ is multiplied by $\mathbf{f}_t$, forgetting the information that the network decided to forget earlier. Then it is added by $\mathbf{i}_t \times \tilde{\mathbf{C}}_t$



$$\mathbf{C}_t = \mathbf{f}_t \times \mathbf{C}_{t-1} + \mathbf{i}_t \times \tilde{\mathbf{C}}_t$$

Figure 3.13: cell state update

Finally, the output of the LSTM can be returned. The *output gate* control of the information encoded in the cell state is outputted and sent to the network as input in the next time step.



$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

$$\mathbf{h}_t = \mathbf{y}_t = \mathbf{o}_t \times \tanh(\mathbf{C}_t)$$

Figure 3.14: output gate

where $\mathbf{W}_o$ and $\mathbf{b}_o$ are the weights and bias related to the output gate.

In conclusion, it can be said that LSTM can regulate the flow of information, forgetting irrelevant information from the past and keeping the relevant part from the current input. By doing this, they can handle long-term dependencies in a better way, overcoming the problem of vanishing gradients.

3.1.5 Evaluation metrics

Once the train phase is over, it is necessary to evaluate the performance of the model. The evaluation phase is precisely the process by which the quality of the network's predictions is quantified. The goodness of the network's predictions is measured against new, never-before-seen observations, using different evaluation metrics.
Since in a regression problem the network tries to predict continuous numerical values instead of discrete numbers of classes as in classification problems, the evaluation metrics for regression models are quite different from those used in classification tasks.

When dealing with regression problems, *Mean square error* (*MSE*) or *Root MSE* (*RMSE*), *Mean absolute error* (*MAE*) and *Mean absolute percentage error* (*MAPE*) are a widely adopted family of measures that evaluate the quality of fit in terms of distance of the regressor to the actual training points [24].

Mean square error

Mean squared error measures the average of the squares of errors, that is, the average of squared differences between the predicted output and the true output. Its mathematical formulation is as follows:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.10}$$

Where y_i is the i-th observed value, $\hat{y}_i$ is the predicted value corresponding, and n is the number of observations.

MSE indicates how close a regression line is to a set of points. It does this by taking the distances from the points to the regression line (these distances are the *errors*) and squaring them. Squaring is necessary to remove any negative signs. As the data points fall closer to the regression line, the model has less error, decreasing the MSE. A model with fewer errors produces more precise predictions.
MSE can be used if there are outliers that need to be detected. In fact, MSE is great for attributing larger weights to such points, thanks to the L2 norm.

Root mean square error

Root Mean Square Error [25] is the square root of the Mean Square Error. It is one of the most widely used measures for evaluating the quality of predictions, and shows how far the predictions fall from the true measured values using the Euclidean distance.
Its mathematical formulation is as follows:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{3.11}$$

In the absence of better information, the *mean value* of the target variable can be considered to be the simplest estimate of the values of the target variable, whether in trying to model existing data or in trying to predict future values. A standard way to measure the average error, in this case, is the *standard deviation*, since it measures how far away an actual value is from the mean:

$$SD = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y}_i)^2} \tag{3.12}$$

The RMSE is quite similar to the standard deviation, but instead of measuring how far an actual value is from the mean, it measures how far an actual value is from the model's prediction for that value. On average, a good model should have better predictions than the naïve estimate of the mean for all predictions, so the RMSE of a good model should be smaller than its standard deviation.

Mean absolute error

Mean absolute error represents the amount of error in the measurements. It is the difference between the measured value and the true value. The difference between MAE and MSE lies in the evaluation metric, respectively, linear L1 or quadratic L2.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3.13}$$

Mean absolute percentage error

Mean absolute percentage error measures the accuracy of a forecast system. It is one of the most widely used measures of forecast accuracy, due to its advantages of scale independence and interpretability.
Its mathematical formulation is as follows:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{3.14}$$

Due to its definition, its use is recommended in tasks where it is more important to be sensitive to relative variations than to absolute variations.

3.2 Word embeddings

Word embeddings [26] are real-valued word representations capable of capturing lexical semantics and sometimes also syntactic relationships between words, so that words that are closer in the vector space are expected to have similar meaning.
In general, it has been found to be useful to represent them as vectors, which have an intuitive interpretation and can be the subject of useful operations such as addition, subtraction, distance measures, and lend themselves well to be used in many Machine Learning algorithms [27]. There are many ways to represent words as a fixed-length vector. Two main approaches to compute word embeddings can be identified in the literature [28]:

- *Count-based models*: methods that represent a target word by words that co-occur with that target word in various contexts using some co-occurrence measure. They use global information, generally corpus-wide statistics such as word counts and frequencies, to learn semantic similarity between words. This method views a target word by the nature of words that co-occur with that word in multiple contexts, so the meaning of a word is given by the words that co-occur with that word in various scenarios. In general, the co-occurrence frequency is represented as word-context matrices. Taking into account a fixed word dictionary D and a set of words W to embed, the co-occurrence matrix C is of size $|W|x|D|$. C is then dictionary size dependent, and its size can lead to a high memory usage. Dimensionality reduction techniques are usually used to make such structures more efficient in terms of memory usage and to make the process more robust by capturing more useful information and eliminating less significant information. On the other hand, this technique is easily parallelisable, so it is possible to train it over more data and get a more accurate model.

- *Prediction-based models*: methods that update fixed-dimensional word vectors (possibly randomly initialized) so that it is possible to accurately predict the words that appear in a target word in a given context. This predictive ability is improved by minimizing the loss between the target word and the context word. In this way, prediction-based models can produce low-dimensional and dense word representations. Unfortunately, these embeddings are difficult to interpret compared to the sparse and high-dimensional representations produced by counting-based methods, where each dimension can be explicitly identified with a context word [29].

3.2.1 FastText

FastText [30] is a library for text classification and representation. It transforms text into continuous vectors that can be used later in any language-related task. In particular, it is a prediction-based model based on the skipgram model, where each word is represented as a bag of character n-grams.

Briefly, in the skipgram model, the goal is to learn a vectorial representation for each word w in a vocabulary of size W. Given a large training corpus represented as a sequence of T words $w_1, ..., w_T$, the objective of the skipgram model is to maximize the following log-likelihood:

$$\sum_{t=1}^{T} \sum_{c \in C_t} \log p(w_c|w_t) \tag{3.15}$$

where context C_t is the set of indices of words that surround the word w_t and $p(w_c|w_t)$ is the probability of observing a context word w_c given w_t. The problem of predicting context words can be seen as an independent prediction of the presence (or absence) of context words. For the word at the position t consider all context words as positive examples and sample negatives at random from the dictionary. For a chosen context position c, the probability is defined as:

$$p(w_c|w_t) = \log(1 + \exp(-s(w_t, w_c))) + \sum_{n \in \mathcal{N}_{t,c}} \log(1 + \exp(s(w_t, n))) \tag{3.16}$$

where s is a scoring function that maps pairs of (word, context) to the scores in $\mathbb{R}$ and takes into account two different vectors, one for the word w_t and one for the word w_c.

Using a distinct vector representation for each word, the skipgram model ignores the internal structure of the words, so the FastText intuition was to use a different score function to take into account this information. Each word w is represented as a bag of character n-gram, including the w word itself and adding the special characters < and > at the start and end of the n-gram.
So, for example, taking the word *where* and n = 3, it will be represented by the character n-grams:

<wh, whe, her, ere, re>

and

<where> .

Assuming to build a dictionary of n-grams of size G. Given a word w, let us denote

$G_w \subset \{1, .., G\}$ the set of n-grams appearing in w. We associate a vector representation z_g to each n-gram g. By representing a word by the sum of the vector representations of its n-grams, it is possible to obtain the scoring function:

$$s(w, c) = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g^T \mathbf{v}_c \tag{3.17}$$

This model allows learning word representations by taking into account subword information.

One of the key features of FastText word representation is its ability to produce vectors for any word, even made-up ones. In fact, FastText word vectors are built from vectors of substrings of characters contained in it. This allows to build vectors even for misspelled words or concatenation of words.

3.3 Outlier handling

An outlier is an observation of the dataset that deviates from the rest of the data distribution.
As an example, Figure 3.15 illustrates outliers in a 2-dimensional dataset. As we can see, most observations lie in two regions, $R1$ and $R2$, while it is possible to notice some points that are far away from these regions, namely $o1$ and $o2$. These points are called *outliers*.

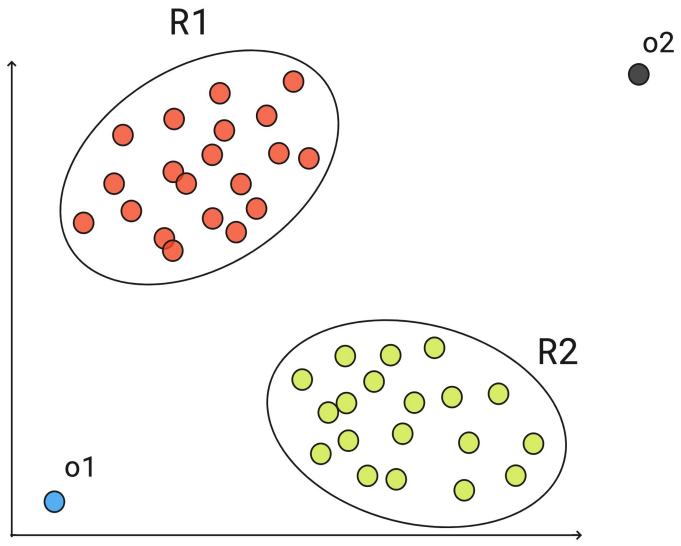


Figure 3.15: Outliers example

In this scenario, outlier detection aims to find patterns in the data that do not conform to expected behavior [31].
Outliers could be introduced in the data for a variety of reasons, such as malicious activity, credit card fraud, and cyber attacks. In all these cases, they represent interesting cases to analyze, as they can help to discover the attacker and his attack patterns.

In other cases, outliers are seen as noise in the dataset. Noise can be defined as a phenomenon in the data that is not of interest to the analyst, but acts as an obstacle. Noise removal is driven by the need to remove unwanted objects before any data analysis is performed on the data [31].

Typically, the steps followed in the outlier detection phase are to identify the observations belonging to the normal regions, i.e. those in which there are the most data, and to consider the rest as outliers.
This approach may seem simple, but in reality it can be highly challenging, due to the difficulty in defining *normal behavior* of the data. Among other causes, this may be due to:

- Understand any possible normal behaviour;
- Imprecise boundary between normal and outlier behavior;
- In many domains normal behavior keeps evolving.

Moreover, most of the existing outlier detection techniques solve a specific problem formulation, which is induced by various factors such as nature of the data, availability of labeled data, type of outliers to be detected, etc.

These factors are often determined by the application domain in which outliers need to be detected, and therefore, in most cases, ad hoc techniques have to be implemented, taking into account the specific application domain.

3.4 Standardization

Standardization [32] is a feature scaling technique used to handle quantities of widely varying values, or to compare measurements that have different units. Variables measured on different scales do not contribute equally to the analysis and might end up creating bias. To address this problem, it is necessary to apply feature rescaling techniques to independent variables or features of the data during the preprocessing step.

Standardizing a dataset involves centering the values around their mean with a unit standard deviation, following the formula:

$$x^{'} = \frac{x - \mu}{\sigma} \tag{3.18}$$

where x is the value that needs to be standardized, μ and σ are, respectively, the estimated mean and standard deviation of the values.

Chapter 4

Related works

The purpose of this chapter is to present the current state-of-the-art, describing the most significant works related to the topics discussed in this thesis.

Getting a representation of the source code that preserves the semantic meaning of the program is a very complex topic that many have tried to address.

One of the most important contributions to this field was made by Uri Alon et al. in *code2vec: learning distributed representations of code* [33]. This article presents a framework capable of representing a code snippet in a way that it can be used as input to any machine learning model and use this representation to train a neural network to predict program properties. In particular, the authors have taken into account the program's abstract syntax tree (AST) to capture regularities that often are common in a program. The information derived from each path is then aggregated into a single vector using an attention mechanism that captures information about the entire code snippet.

Nghi D. Q. Bui et al., in *InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees* [34], identify a lack of generalizability in code representation such as code2vec. Since these code representations and program models are trained in a (semi)supervised learning paradigm, they are built with respect to specific tasks. In summary, they do not perform well when used for different tasks. The solution they present has the purpose to overcome this limitation by developing a model trainable without any manual human labeling, flexible in producing embeddings for any code unit that can be parsed into syntax trees, and general enough so that its trained representations for code can perform well for various downstream tasks. Learning about source code representation starts with the transformation of the code snippet into an intermediate representation (AST). Then, for each AST, a set of subtrees is generated, and all of them are accumulated into a vocabulary of subtrees. The original SAT is fed into a neural network, and the generated vector is used to predict the previous subtrees in a self-supervised approach. After training the encoder, it can be used as a pre-trained model for downstream tasks.

D. Zügner et al. approached this problem in the article *Language-agnostic representation learning of source code from structure and context* [35]. Their solution involves AST

and a sequence of tokens that describe the context of the program, summarizing the features with the attention mechanism with the purpose of learning an agnostic representation of the code.

This thesis addresses a similar problem, but the starting point is not the source code, but rather the sequences of assembly instructions generated by the execution of various programs. The main problem in analyzing such sequences is that the order in which instructions are executed may be different from the order in which they are written due to compiler optimization, which may alter the order of instructions. Additionally, all proposed solutions analyze the code statically, while a dynamic approach is necessary. Therefore, a different solution is needed.

On the main topic of this thesis, namely the creation of a neural network model capable of predicting the overhead caused by the application of certain obfuscation transformations, to the best of the author's knowledge, few works have focused on performance degradation, while almost all have approached the problem by attempting to identify complexity metrics or features capable of influencing the application of obfuscation.

In the article titled *The performance cost of software obfuscation for Android applications*, Yan Zhuang [36] analyzes the performance penalty imposed by obfuscation of Android applications to optimize the selection of the obfuscation transformation to apply to the application. In particular, he measures the degree of complexity of obfuscation transformations using software complexity metrics and performance loss in terms of CPU cycles.

Other articles address different aspects of applying obfuscation transformations. Sebastian Banescu in *Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning* [37], proposes a general framework for selecting program features that are relevant to predict the resilience of software protection against automated attacks. These features are used to build a regressor model capable of predicting the resilience (defined as a function of time spent by the deobfuscator to run a successful attack and the time spent by the programmer in building the deobfuscator) of several code-obfuscating techniques against an attack based on symbolic execution.

Chapter 5

Problem statement

Today, software obfuscation is certainly considered one of the most effective countermeasures against MATE-type cyber attacks. There are also many other fields in software engineering and computer security where program obfuscation is very useful, such as defending against viruses or protecting software watermarks and fingerprints. On the other hand, obfuscation is also used for malicious intent, such as malware obfuscation.
Although these and many other applications have led to the development of new and increasingly sophisticated obfuscation techniques aimed at improving the robustness and effectiveness of protection, the loss of performance caused by the application of such transformations remains one of the most significant side effects that must be taken into account.

In fact, a well-protected but extremely slow software would be of no practical use and would only worsen the user experience.

However, tolerance to the performance penalty introduced by an obfuscator may vary depending on the context of use and type of software. In some cases, like mobile games or real-time applications, performance, low latency, and high frame rate are the priority, and performance loss must be minimized.

In some other services, such as banking, the security of the whole infrastructure, as well as the data owned, is the priority, and consequently, a drop in performance, within certain limits, is acceptable. In such cases, it is completely understandable to sacrifice performance in exchange for more effective protection.

There are many ways to define the cost of software performance, each of which may have advantages and disadvantages.

This thesis aims to focus on a practical notion of performance that can be linked to user experience. In particular, this work considers the drop in performance due to the increase in execution time of the obfuscated software, and aims to build a deep learning model capable of understanding the relationship between instruction sequences in the *vanilla* code and the overhead caused by the application of obfuscation.

In a few words, the main objective of this thesis is to predict how much overhead will be added to a new observed function without applying any kind of obfuscation transformation to the function itself.

The starting point will be a set of files that contain all the information collected during

the execution of various open-source programs. All of this information will be processed to construct a complete dataset suitable for machine learning tasks. In particular, it will contain, for each non-obfuscated program, the list of all the instructions executed by all its functions, when a specific set of input is used, with appropriately calculated overhead values at the end.

The deep learning model will receive as input the traces thus constructed, as well as the performance overhead between the execution of the vanilla code and the execution of the same when obfuscation is applied, for each obfuscation technique chosen.

At evaluation time, an unseen application will be fed into the network, and the expected output will be precisely the performance drop that the application of obfuscation on that particular application would have caused.

Getting an estimate of the overhead can be extremely useful in all the situations in which there are limited resources or in which there are strict constraints on the application performance. Since under these conditions, a high overhead could slow down the application in a not negligible way, thanks to this preliminary information and without the need to actually apply the obfuscation, the user could choose other obfuscation transformations that eventually will have better performances.

Chapter 6

Design

This chapter aims to provide an overall view of how the overhead prediction problem was addressed and its resolution strategy.

6.1 The relevance of data

In any work related to the field of Data Science, the relevance of obtaining a dataset that contains valuable and consistent data, without useless junk, is one of the most important points during the development of a solution focused on the use of artificial intelligence: meaningless data can lead to equally meaningless predictions.
Although in the era of big data it is possible to have access to terabytes and terabytes of information, if they cannot be correctly interpreted, it will not be possible to build any model capable of extracting useful information from the data and generating meaningful predictions. For this reason, data creation and manipulation are crucial aspects of the machine learning process.

The attempt to collect data on the execution of programs represents the first fundamental step towards the construction of a complete dataset, capable of properly training a neural network with the aim of predicting the overhead caused by the application of obfuscation.

This crucial phase, as described in detail in Section 2.3, started with the selection of a pool of applications. To obtain realistic and heterogeneous data, it is important to record various realistic runs of each application; therefore, a set of test cases for each application is generated, with the help of a particular type of fuzzer capable of receiving feedback after running the application, to understand whether the input caused the execution of a new path in the application.
The generated test suites were first used to collect information about the execution of each application by running each of it with these input, then to measure their execution time.

Subsequently, each application was obfuscated with the chosen obfuscation techniques, and then the obfuscated applications can be executed with the same generated test cases

as input while measuring the time needed for their execution.

Eventually, all the information concerning the execution and the time measurements of the various applications was stored, and they represent the starting point for the work proposed in this thesis.

6.1.1 Dataset analysis

The following part focuses on analyzing and preparing these data to create a complete dataset. At this stage, in fact, the data are still too raw and contain a lot of useless information that could be misunderstood or misinterpreted by the model and could lead to an incorrect overhead prediction.
Figure 6.1 gives a general overview of the different phases necessary to analyze the traces. The remainder of this paragraph will be spent explaining each of these steps.

The *overhead computation* phase was performed in parallel to the actual analysis of the traces. In particular, since 100 runs were made for each trace and each input, this step consists of grouping all the values of the different runs of the same traces, choosing a single value between them as the reference values for the trace, and compute the overhead as a ratio between the execution time of the obfuscated and the vanilla version of the same application, running with the same test case as input. Different obfuscation transformations were chosen and, consequently, different overhead values were generated for the same application.

The dataset, so far, contains one directory with all execution traces and one directory with all the overhead information needed.
In particular, each *trace* represents a complete execution of the vanilla version of the program under analysis, run with a particular set of inputs found by the fuzzer. Consequently, for each program, there are as many execution traces as the set of inputs generated by the fuzzer.

A single trace consists of multiple information from the vanilla execution of the program. Indeed, in a single line, it contains the following:

- The name of the executed function;
- the offset from the beginning of the function;
- the opcode of the actual executed instruction.

There are obviously as many lines as there are instructions executed by the program.

It must be said that among all the functions in the traces, there are no calls to standard libraries; this is to avoid including unnecessary information and to focus only on the functions contained in the program.

The line containing the last executed instruction does not determine the end of the trace. In particular, there is other information concerning statistics of the entire execution, in terms of the number of instructions present for each function.

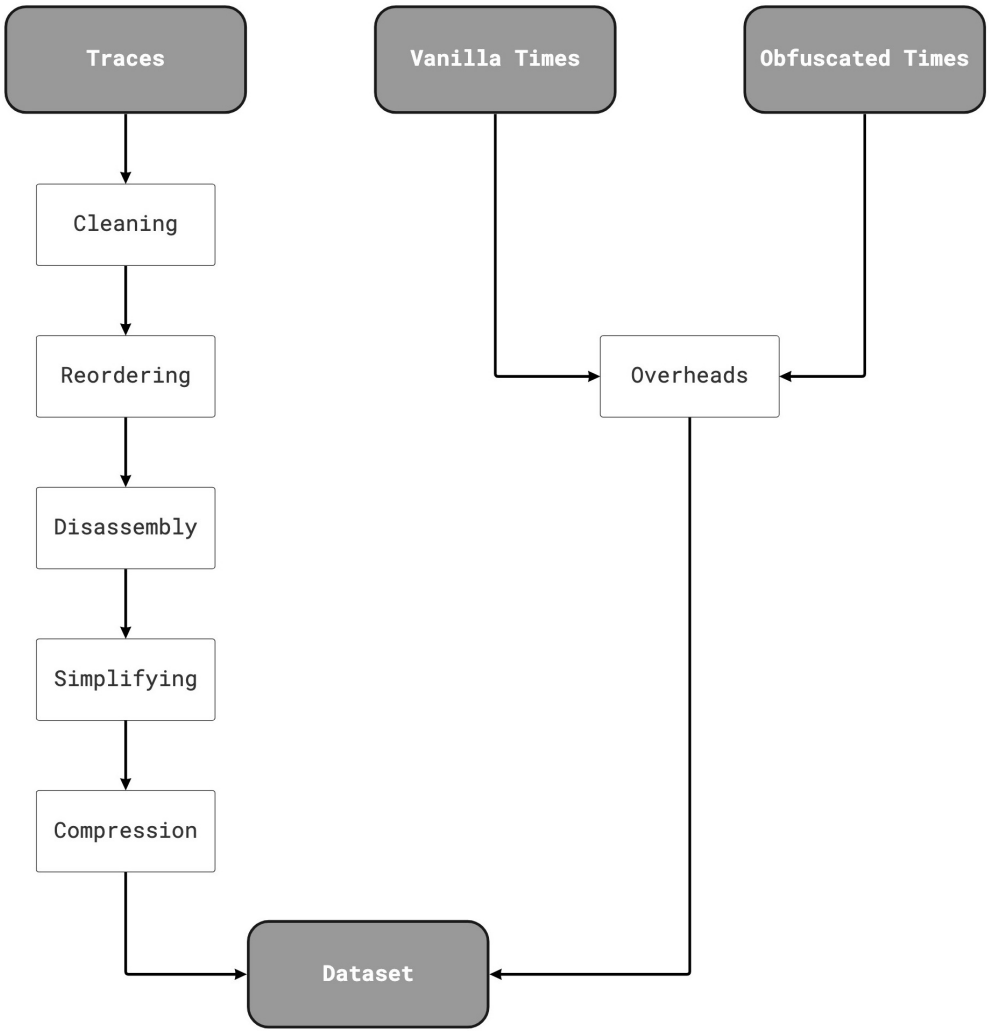


Figure 6.1: Overall workflow in trace analysis

Therefore, in the *cleaning* phase, only useful information is extracted from the traces. In particular, all that is needed are the lines that contain the name of the function, the offset, and the opcodes of the instruction. All remaining information on the statistics was not taken into account, as it was not considered meaningful or essential for the purposes of this thesis.

Since the Tigress obfuscation tool is able to add obfuscation only to one single function at a time, the aim of the entire work will be to predict the overhead at the function level. Therefore, it would not make sense to consider a trace as the set of all the instructions

executed in the program: it is not usable directly for our purpose.

To approach this problem, a single trace must be seen as a set of functions called during the execution of the program under analysis.
In the *reconstruction* phase, information about the name of the function and the offset was used to group all instructions belonging to a specific function, and this is done for every function in the trace and, for all the traces in a program, and for all the programs. From now on, a trace will be considered as the set of instructions executed by a function.

In the following steps, the traces will be manipulated to obtain a more compact version that can be best used by an LSTM network.
Indeed, as discussed in Section 3.1.4, although LSTM nets were introduced to solve the vanishing problem that affects all RNN networks, they are not able to handle really long ones anyway. Since the lengths of the traces can also exceed thousands of instructions, a way to handle very long sequences is needed.

The first step requires instructions *disassembly*. In fact, when the execution data was extracted, the instructions were saved as machine code. Disassembling an instruction means converting it into some form of assembler language so that it is readable by a human. Since this work analyzes this problem for the x86 architecture, the instructions are converted to x86 assembly.
As an example, the sequence $b800000000$ is converted to $mov\ eax,0x0$.
This will make it easier to determine a further compression strategy.

Next, the *simplification* phase represents a crucial step in achieving subsequently an acceptable compression level.
Normally, the structure of a generic instruction is of type *mnemonic + operands* (in some cases, the operands are zero). Since the set of all mnemonics and operands is quite large, their various combinations can lead to situations where there are semantically similar but syntactically different instructions.
From this assumption, the idea behind the simplification phase is to find a way to transform the instruction so that semantically similar instructions have the same syntax. It is possible to divide this phase into two different sub-phases:

- *mnemonic simplification*: the aim is to identify a minimum subset of different instructions so that each set consists of all instructions that share the same behavior. In this way, it will be possible to transform an instruction into just a string that represents a set of similar instructions. For example, instructions add, adc, inc can be grouped into a unique set that represents a general *add* operation.

- *operand simplification*: each register, namely *eax, ecx, edx, ebx, esp, ebp, esi, edi* considering 32-bit registers, will be converted to $\_R$, while each memory address will be converted to $\_M$. The idea is that it is not necessary to know specifically to which register or memory cell is being referred. All that is important to know is that the access time to a memory cell will always be longer than to a register. Therefore, if at least one of the operands in an instruction is a memory access, the other (register or immediate value) will be overlooked: the final output will be $\_M$. On the other hand,

if among all the operands there is no memory access, it will mean that the longest access time will be relative to the register: the final output will be $\_R$.

Some examples may help clarify ideas:
$cmov\ [var], edx \rightarrow mov\_M$
$adc\ eax, 0x15 \rightarrow add\_R$

Compression is the last phase. The simplifications made in the previous step will be used to apply the lossless data compression algorithm *Run-Length Encoding*. The key idea behind the RLE algorithm is that, in a sequence that presents redundant values, instead of storing the entire sequence, only a single value will be stored that represents the repeated value and how many times it appears in the sequence.
RLE is a good compression method when data have many repetitions. The simplification of the instructions previously done has served just this purpose: in this way, instructions that were previously only similar in semantics are now identical and can therefore be grouped together.
Of course, when dealing with instruction compression, what is needed is not character-level compression, so a custom implementation will be required. This approach allows for the identification of an entire instruction, so that when two or more identical instructions are encountered, there will be grouped into unique ones, preceded by the counting of their appearances.
For example:
$mov\_M\ mov\_M \rightarrow 2\ mov\_M$
$add\_R\ add\_R\ add\_R \rightarrow 3\ add\_R$

Finally, information on *overhead* previously computed was added at the end of each trace.

In summary, the main aspects of analyzing a trace are the following:

- *overhead computation*: obfuscated and vanilla time executions were compared to obtain a measure of performance overhead;
- *cleaning*: information regarding function name, offset and instruction is retained;
- *reconstruction*: name and offset information was used to reconstruct the function call tree to group all functions within each trace;
- *disassemble*: the machine code of all instructions was translated into x86 assembly language;
- *simplification*: instructions have been converted into a simpler form;
- *compression*: RLE compression was performed;
- *overhead addition*: the overhead values were added at the end of each trace.

The same procedure was repeated for every trace within an application and for all applications in the dataset.

All the steps described so far have led to the creation of a complete dataset containing much more meaningful and compact information. However, before being used to train the network, other considerations must be taken into account.

6.1.2 Data preprocessing

Once the data have been organized in a more appropriate way, a single program will generate as many files as there are functions executed within it. A trace will consist of all the instructions of a single function call, with at the end the overhead values given by the application of the chosen obfuscation techniques on that function.
Therefore, the overall dataset will contain, for each execution of a given program with a given set of inputs, a list of files each containing a function.

Although the analysis just carried out was indispensable to structure the dataset so that it could be used to train a deep learning model, the information contained within it is still raw and not very interpretable by the network: the quality of the data directly affects the ability of the model to learn.
From this point, a further manipulation phase will be necessary to select only useful traces and discard useless ones.
Figure 6.2 shows the principal aspects of data preprocessing.

Each of these steps will be explained in the remainder of this paragraph.

As explained in the previous paragraph, RLE compression was implemented to reduce the sequence length. RLE compression works well when there are many repetitions, which in this case would mean having a situation where there are many identical instructions repeated one after the other.
Unfortunately, there is no guarantee that this will happen when the code is executed. Furthermore, the compiler may arbitrarily move instructions to different parts of the code for optimization reasons. Although the application of this technique has led to an overall reduction in the length of the tracks, there are still some traces that are too long and exceeded 1000 instructions, which were therefore too long to be fed into the network.
For these reasons, in the *filtering* phase, the choice was to remove all traces that were still too long after compression. The elimination threshold was set at 1000: a length ≥ 1000 results in the trace being deleted from the dataset.
Then the overhead values were checked and the traces with an overhead ≤ 0 were filtered.

Outlier handling was one of the most challenging parts of this thesis work.
Although at this point all traces have a correct structure, the dataset presents traces whose overhead values deviate greatly from the rest of the distribution. This may be due to measurement errors or an unfortunate execution of the code that generates inconsistent values. In addition to this problem, there are many identical traces with different overhead values. Since the execution time depends on many factors, such as the location of the data on the different memory levels or whether the pipeline is full or empty at that particular time, different execution times generated by the execution of the same code are actually possible. Although it may not be a mistake to have identical traces in terms of executed instructions with different overhead values, this could lead the network to a wrong prediction, as it

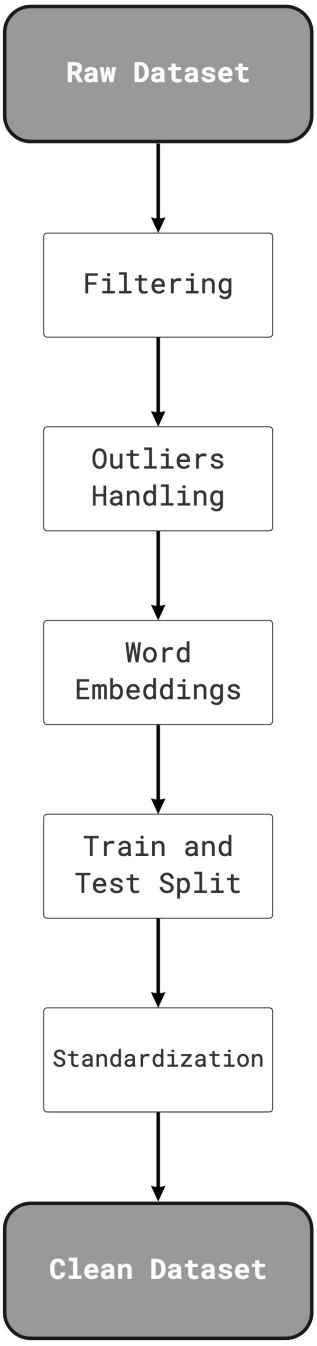


Figure 6.2: Data preprocessing

would not know which of the many values to associate with the given trace.

The proposed solution groups all identical traces, generating a hash-table-like[1] structure in which the assembly is treated as the unique index, and the corresponding list of overheads are the values.
Subsequently , for each group:

- the median of the respective overhead values is computed;
- each value within the group is compared with respect to the median, and all values that are $\geq (median + 0.5 \cdot median)$ or $\leq (median - 0.5 \cdot median)$ were initially counted;
- if this counter exceeds half the length of the overhead values list, then the trace is considered an outlier and discarded from the dataset, as most of its values deviate considerably from the median;
- if the counter is less than half the length of the group, the key value of the group, that is, the assembly, is included in the dataset with overhead value given by the median computed in the first step.

Now, all traces in the dataset are different, and the overhead associated with each of them represents a typical execution of its instructions.
These steps eliminated possible fluctuations in the overhead values within the traces, but the overall distribution of these values is still unbalanced, presenting values so high and far from the distribution as to suggest some possible calculation error or some unfortunate program run in which the CPU triggered some interrupt. For this reason, all these problematic overheads were eliminated.

Although with these changes the size of the dataset was reduced, the consistency of the data was promoted with respect to its quantity.

In the *word embedding* step, an additional problem was addressed. So far, the dataset presents both numeric and categorical features. The numerical values are the number preceding the instruction, which was generated by the RLE algorithm and which indicates the total number of times the given instruction was repeated in succession (i.e. 2 in 2 *mov_M*) and of course the overhead values, while the categorical ones are the strings representing the instructions themselves.
The latter introduce a challenge for many machine learning algorithms that do not support text values and must therefore be translated into numerical values.
For this purpose, a continuous word embedding representation is chosen, using a deep learning model called FastTex [38]. This powerful library allows words to be represented as vectors of real values. This vector representation is capable of capturing hidden information about a language, such as word analogies and semantics.

The dataset is now ready to be splitted into *Train set* and *Test set*: this is a common practice in machine learning tasks.

[1]Hash Table is a data structure which stores data in an associative manner. In a hash table, data are stored in an array format, where each data value has its own unique index value

The key idea is that since the main goal of supervised learning is to build a model that performs well on new data, it is possible to simulate new observations by starting from the initial dataset and dividing it into two subsets. The first subset will be used to fit the model, while in the second one, the input element will be provided to the model as if it were data never seen before, then predictions will be made and compared to the expected values.
The final aim will be to estimate the performance of the machine learning model on new data.

The last preprocessing step involves *standardization* of the data. Standardizing the features around the center and 0 with a standard deviation of 1 is important when comparing measurements that have different units. Variables measured on different scales do not contribute equally to the analysis and could end up creating bias. It is important to specify that all standardization and normalization techniques should be applied after splitting the data between the training and the test set, using only the data from the training set. This is because the test set plays the role of fresh unseen data, so it is not supposed to be accessible during the training phase.

In summary, the main aspects of data preprocessing are as follows:

- *filtering*: data with overhead ≤ 0 and length ≥ 1000 were discarded from the dataset;
- *outliers handling*: outliers were detected and discarded from the dataset;
- *word embedding*: categorical values were translated into vectors of real numbers;
- *train and test split*: dataset was divided into train and test set;
- *standardization*: standardization was applied to the dataset.

6.2 Overhead prediction

Overhead prediction represents the last phases of this thesis work.
The sequential nature of the data in the dataset, as discussed in Sections 3.1.3 and 3.1.4, led to the choice of a deep learning model based on Recurrent Neural Network. In particular, *Long Short-Term Memory* was developed to handle even the longest sequences.

With the help of specially designed Python libraries[2] for deep learning, the structure of the LSTM network was defined. In particular, it consists of an LSTM layer followed by a fully connected layer with the ReLU activation function at the end. The next step was to implement the forward pass, which consists of passing a trace through the network.
In particular, within this function, first of all, the vectors representing the hidden state $\mathbf{h}_0$ and the internal state $\mathbf{c}_0$ are set to zero, which will then be passed along with the trace through the first layer defined above, i.e., the LSTM layer.

[2] https://pytorch.org/

Subsequently, the result of this operation will be passed through the fully connected layer, and finally the activation function will be applied.

The main structure of the network is now complete, but before training the model, some necessary hyperparameters had to be defined:

- *input size*: the number of expected features;
- *hidden size*: the number of features that the LSTM should create, i.e., how many LSTM cells are in the hidden layer and how many outputs the first layer will have;
- *num layers*: the number of staked LSTM layers;
- *num classes*: the number of output classes, i.e., how many values the network has to predict;
- *num ephocs*: the number of iterations through the dataset;
- *learning rate*: parameter that influences both learning time and convergence (see Section 3.1.2).

Once the network and hyperparameters have been defined, it is possible to move on to *train phase*.
This step is crucial because, when the network is created, the weights are initialized with random values, which causes the network to make incorrect predictions. Therefore, during training, the aim will be to iteratively improve these weights by defining an appropriate loss function that will have to be minimized to obtain a lower error by the network from epoch to epoch. At first, the network will make mistakes, but by adjusting the weights, the loss function will decrease accordingly and there will be fewer and fewer errors.
In *evaluation phase*, the network will iterate through the unseen data and the optimized weights previously found will be used to make predictions.

Chapter 7

Implementation

In this chapter all the main topics presented in this thesis will be addressed in detail, focusing on the solutions implemented and the adopted technologies.
Figure 7.1 shows an aggregated view of the main steps taken to build a deep learning model capable of predicting the worsening in performance of a function to which an obfuscation transformation has been applied.

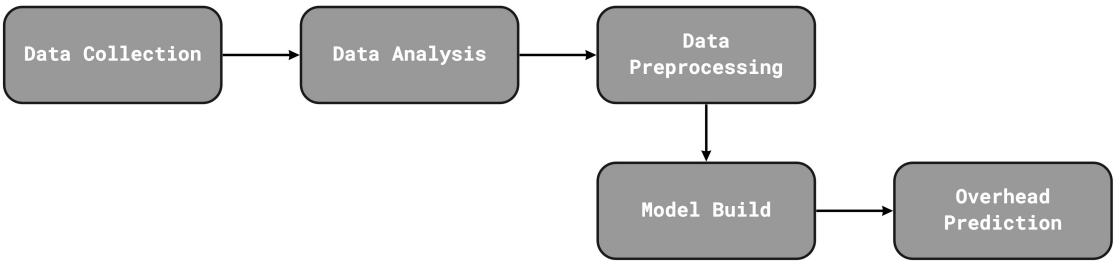


Figure 7.1: Aggregate view of the workflow

After a brief recap of Stefano's thesis work on the various steps involved in data collection, the remainder of the chapter will be spent analyzing in much more detail the steps shown in the figure, expanding what was already seen in Chapter 6.

7.1 Data collection

This section aims to sum up the results reached by Stefano Alberto in his thesis work [4], in which he collected data regarding the execution of various programs, tracking their execution times, and applying various obfuscation techniques in order to gather information regarding the performance loss caused by them. These data will be crucial for the construction of a complete dataset.
Figure 7.2 summarizes all essential parts of the data collection.

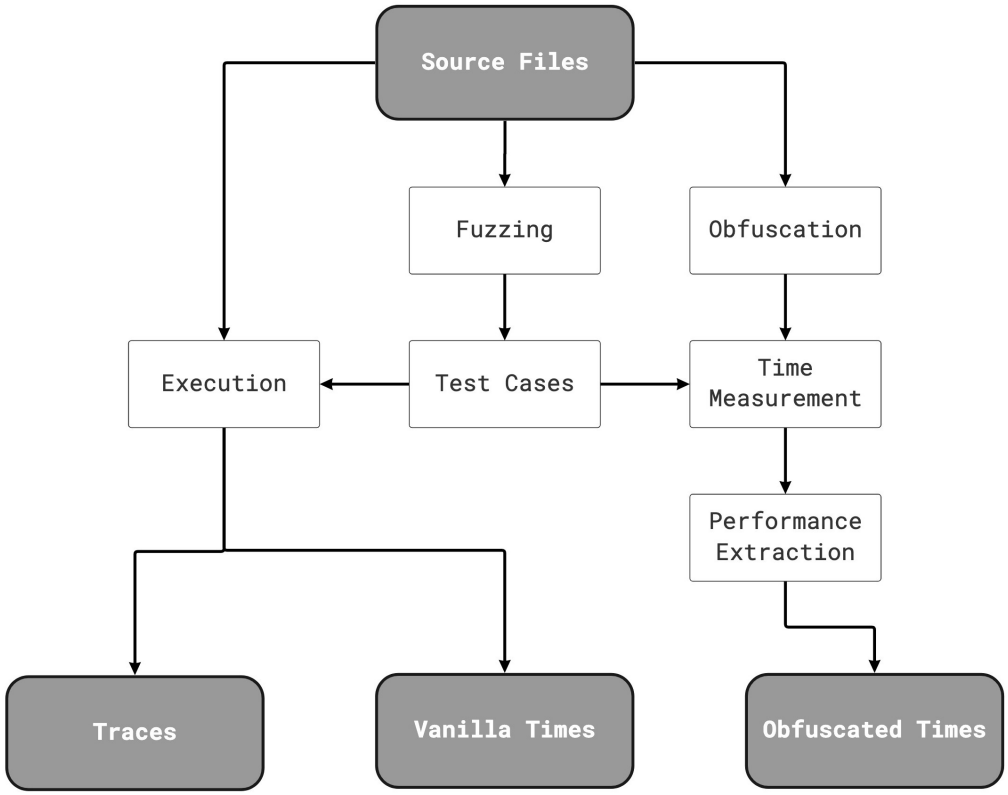


Figure 7.2: General workflow of data collection

The starting point was the selection of a pool of applications to be tested to extract realistic information about software executions. The chosen applications must satisfy certain constraints:

- software must be open source to apply code obfuscation without breaking the license terms of the application;
- software must be written in C language, since the obfuscation tool used is Tigress[1], which offers various obfuscation techniques, but only applicable to C code;
- the execution of the application must be deterministic and repeatable, without depending on external factors.

Once the applications have been chosen, to extract meaningful information from them, it is necessary to run these programs using a proper set of inputs. Since each application is different, each requires different input data. A good set of inputs leads to the execution of different paths, and consequently, the information extracted will make more sense. On the other hand, running an application with random data as input usually causes the application to crash or execute the same path in its code many times. Since finding a sufficient set

[1] https://tigress.wtf/index.html

of inputs for all applications would have been very time-consuming and resource-intensive, the proposed solution was to automatically generate it, involving the use of a feedback-based *fuzzer*, as seen in Section 2.2.
Although the standard use of a fuzzer is to find anomalies in the code and report them to the user, in this case it was used to trace different execution paths and save a new input each times a new paths was found.
It is now possible to collect information about the execution of each applications by running them many times with the generated inputs.

Finally, the last part to be addressed to obtain a detailed description of a program and thus to complete the collection of data, is to trace the execution time of both vanilla and obfuscated applications. First, the unprotected applications were executed 100 times for each set of inputs found by the fuzzer, as explained in Section 2.3.6.
Then, the execution times were recorded and saved in special files. Subsequently , the selected obfuscation techniques (*code flattening* and *opaque predicate*, as explained in Section 2.1), have been applied to the applications. The obfuscated source code is now ready to be executed to measure execution times.
In fact, to collect data on performance drop, each obfuscated version of all applications must be re-executed, using as input the same data found by the fuzzer in the previous step, in order to track their execution times. Also in this phase, the obfuscated applications were executed 100 times for each set of inputs and the execution times obtained were saved in different files.

7.2 Data analysis

In this section, the data collected by Stefano will be analyzed and processed to create a complete dataset that will contain coherent data with a sequential structure that can subsequently be processed by an LSTM-type neural network.

Eight different applications were examined to collect information on their performance, and, so far, the total amount of data collected is mainly divided into 4 parts:

1. a folder containing, for each program, all the traces representing, each, the execution of the entire program with one of the test cases found by the fuzzer. In particular, Table. 7.1 shows the number of traces for each program;

2. a folder containing, for each vanilla program and for each trace, a file with the time measurements of each function executed;

3. a folder containing the same time computation for the flatten obfuscated version of the programs;

4. a folder containing the same time computation for the opaque obfuscated version of the programs.

Figure 7.3 shows the directory tree structure of the data.

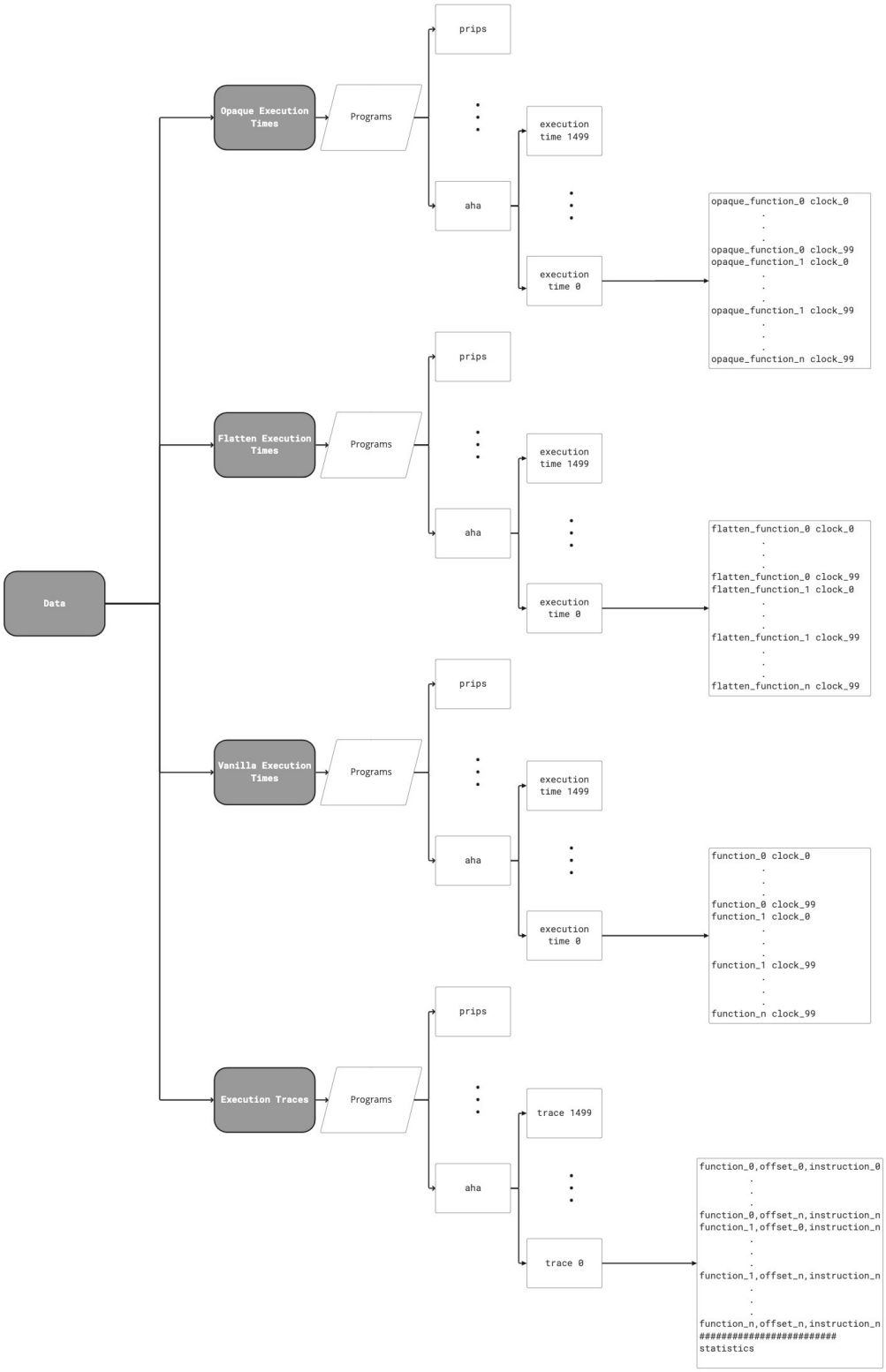


Figure 7.3: Data directory tree

Program Name	*Number of Traces*
aha	1500
ascii	1000
colorize	1548
d48	1000
d52	1000
dz80	1000
id3ren	1000
prips	1000

Table 7.1: Number of traces

The following steps aim both at calculating the overhead for each function of each trace and at analyzing and selecting only the relevant parts of the traces.
The end of this section will determine the complete construction of the dataset.

7.2.1 Overhead computation

This section wants to analyze the execution times of both the vanilla and obfuscated programs. Figure 7.4 shows the workflow.

The starting point here is all the execution time measurements taken during the data collection phase.
As explained above, for each program and for each set of inputs generated by the fuzzer, the clock cycles of each executed function were measured by running a particular function whose purpose was to record the clock cycles at the start and end of the function to be analyzed. In this scenario, all calls to external library functions were not taken into account, as they will not be taken into consideration by the obfuscator at the time of transformation.

Furthermore, to minimize possible fluctuations in these measurements, which are mainly due to the CPU scheduler and the state of the various caches during memory accesses, all measurements were performed 100 times. This means that, for example, if a trace consists of a 10 function call, for each of them there will be 100 different measures, each representing the clock cycles of an execution. The next step is to extract a single value that can summarize in some way all 100 executions of the function under analysis.

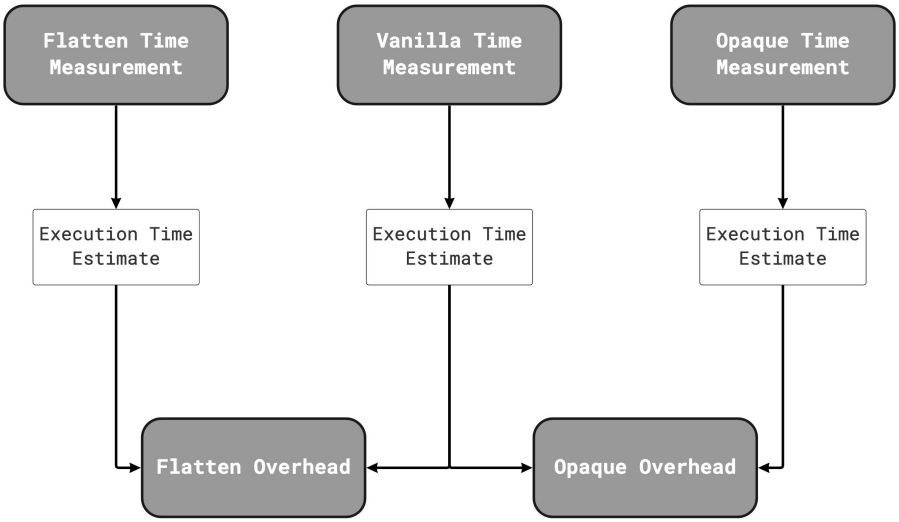


Figure 7.4: Overhead computation

The main problem addressed in this part was that the distribution of these values was not very balanced, as there were both extremely high and very small values. Taking the mean as a reference measure would have led to a result strongly influenced by these outliers.
A more robust representation of these data is certainly provided by the *median*, which represents the central value of the ordinal distribution of these data. Consequently, for each of the functions, the median between the 100 executions was computed, and a new file was created, for each execution trace, containing all function-median pairs.
Figure 7.5 shows the old file and the new one, which contains only the list of all functions executed by a program with the median value beside it. This computation was performed for both the vanilla and obfuscated programs.

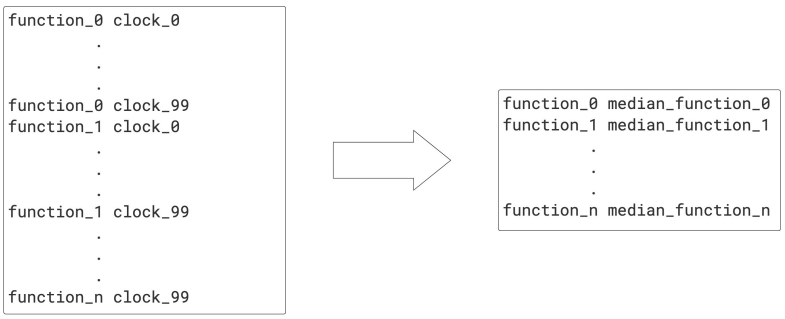


Figure 7.5: Old vs. new clock file

The relative pseudocode is shown below (the algorithm performs the same steps for both datasets, so for simplicity's sake a general version is given):

```
1  foreach program ∈ programs do
2      foreach time_traces ∈ program do
3          foreach trace ∈ time_traces do
4              create output_file
5              foreach ex_times, function_name ∈ trace do
6                  clock_cycle_list ← []
7                  i ← 0
8                  foreach clock_cycle ∈ ex_times do
9                      /* save the 100 execution times in a list */
                       clock_cycle_list[i] ← clock_cycle
10                     i ← i + 1
11                 end
12                 clock_cycle_median ← median(clock_cycle_list)
13                 output_file ← (function_name, clock_cycle_median)
14             end
15         end
16     end
17 end
```

It should be noted that, by construction, the functions are written in order of termination, from the first to the *main*, which is the last to terminate. This detail will be very useful when reconstructing the call tree as an additional verification of correctness.

The last step involves the computation of the overhead values.
Once a single value summarizing all the various executions has been associated with each function, this function will have one execution time for its vanilla version and two different for the two obfuscated versions (flatten and opaque). It is now possible to compute the overhead, given simply by the ratio:

$$overhead = \frac{obfuscated\_time}{vanilla\_time} - 1 \tag{7.1}$$

Thus calculated, the overhead will assume the following values:

- $= 0$: both the obfuscated and the vanilla functions take the same amount of time to execute;
- < 0: the obfuscated function takes less time to execute than the vanilla, probably due to some error (this case will be handled later);
- > 0: the obfuscated function takes longer to execute than the vanilla.

As before, a file was created for each obfuscation technique, program, and execution trace, where the function-overhead pairs were listed. The order in which the functions were written in the file is the same as previously explained, i.e. from first to last to terminate.

7.2.2 Trace analysis

The purpose of this section is to examine the execution traces in depth, attempting to select only the essential information considered essential for the purpose of this thesis.

To construct these new traces, not only the overhead values computed in the previous phase will be used, but also information concerning the execution order of the functions, to determine and reconstruct the function call tree of a trace, as will be explained later.

Since the steps will be the same for all the traces, for simplicity's sake, all the reasoning that follows will be based on a single trace. The detailed workflow followed is presented in Figure 7.6.

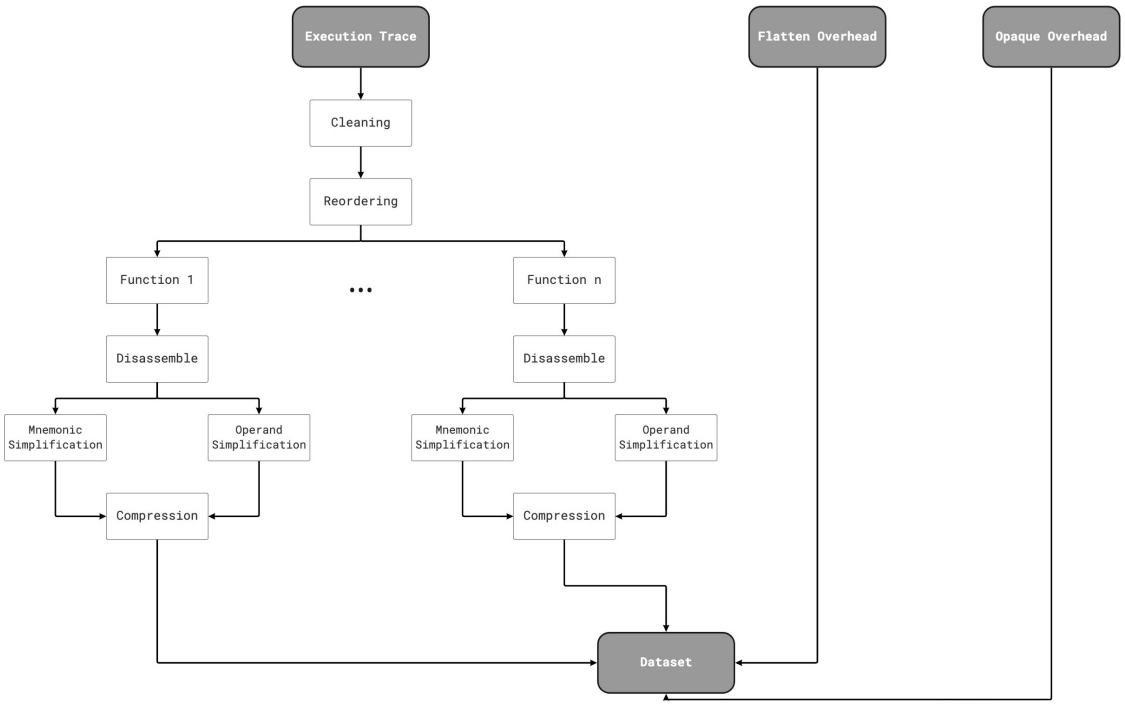


Figure 7.6: Trace analysis detailed

Data cleaning

The first step consists in understanding the structure of a trace.
Its first lines summarize the execution of one of the vanilla programs with an input one of the test cases generated by the fuzzer, and lists, in order of execution, all the instruction for all the functions run by the program. In particular, a line consists of:

- the name of the executed *function*;
- an *offset* value that represents a counter that starts when a new function is called and is incremented for each instruction executed within the function itself;
- the actual executed instruction, in the form of a *opcode*, specify in machine language the operation to be performed and zero or more operands with which it is performed.

Once the last instruction of the last executed function has been reported (which in the case of correct execution should be the termination of the *main*), further statistics on the

execution of the program are reported, such as, for example, the number of times a given instruction appears within a function.

Although this additional information could have its own importance, for the purpose of the thesis, they do not add anything significant, so the proposed solution is to cut these additions from the trace, leaving only the information concerning the execution. For a better understanding of how a trace is composed, Figure 7.7 shows part of a real execution trace.

It can be seen that the information on execution is divided from the statistic part by the string *#########*; therefore, during *cleaning phase* everything above this separator was retained and passed on to the next stages, while the rest was discarded.

```
main,+0,f30f1efa
main,+4,55
main,+5,4889e5
main,+8,53
...
parseArgs,+2058,48895818
parseArgs,+2062,488b4dd0
parseArgs,+2066,488b5dd8
parseArgs,+2070,48894820
...
main,+5942,488b5df8
main,+5946,c9
main,+5947,c3
##########
deleteParse
divide
getNextChar
main
    endbr64 = 1
    push = 2
    mov = 102
    sub = 2
    xor = 1
    lea = 27
    callq = 15
...
    pop = 1
    retq = 1
swapColors
##########
executed 5/11 functions
```

program execution

separator

program statistics

Figure 7.7: Example of trace structure

Data reordering

After this cleaning, the trace contains only information about the functions and instructions performed. As explained above, the aim of this thesis concerns the prediction of the overhead caused by applying a certain obfuscation technique to a certain piece of code.

However, a not negligible constraint imposed by the Tigress obfuscator is that it works by obfuscating only one function at a time. This is done to avoid the obfuscation of non-critical code and to reduce the performance impact of the protection on the final application.

In this scenario, it would therefore not be correct to directly use a trace with the current structure, i.e. containing the entire execution of a program.
To be consistent with what Tigress offers, such data must therefore be divided: the main purpose is therefore to isolate individual functions.

Therefore, it is necessary to group all the instructions for each function executed in the trace. Since the instructions are in the order they were executed, it is possible to reconstruct the call tree of the program.

Although at first glance this part may seem trivial, there are many aspects to take into consideration and several issues were encountered while approaching this problem.
A first approach was based on scanning instructions by determining a new function call when an opcode belonging to the *call* function family was encountered (namely when the first two characters of the opcode took on one of the following values: *e8*, *ff*, or *9a*) and the termination of a function when the opcode belonged to the *ret* function family (namely when the first two characters of the opcode took on one of the following values: *c3*, *cb*, *c2*, or *ca*).

This solution seemed solid until the first results were examined, which showed inconsistencies in the various subdivisions of the functions. In fact, many instructions belonging to different functions were grouped together.
It turns out that the basic idea of the overall structure of a function can be wrong in real compiled applications. Indeed, the compiler applies many optimizations to avoid the overhead of the classic call instructions, especially when the called functions are small. One of these optimizations consists of using lighter *jump* instructions to enter and exit a function. This kind of optimization can confuse the distinction between a standard *jump* instruction and the one used as a function call or return from a function when analyzing the execution trace.

However, sometimes the compiler may reorder the execution of instructions to improve their efficiency, while respecting dependencies. In these cases, the order in which the instructions are listed in the trace may not correspond with the actual order of execution, and therefore reconstruction of the call tree is no longer possible. The programs containing such optimizations were discarded and the applications chosen in Table 2.1 were specially selected to avoid such problems.

The proposed solution to reconstruct the call tree involves the use of a temporary queue (*fun_list*) in which all instructions of not yet terminated functions are stored in the form of function name-list of instructions. When a function terminates, all these previously collected information were moved from this queue to a final one (*complete_fun_list*) which stored all completed functions in termination order.

At the beginning, the trace is read and, for each line, information about *function name*, *offset* and *instruction opcode* was retrieved. Being the first iteration, a new entry was

created for *fun_list*, containing the function name and a list with only one element, the first instruction.
In the next iterations:

- if the current instruction is a *call* one, it will be stored in the current position in *fun_list* and the next line will be examined in detail:
  - if the next function name is different from the current one, it means that a new function will be called: in the next iteration a new entry for *fun_list* will be created in the same way as in the first iteration;
  - if the next function name is equal to the current one, it means that the current function has called itself: there was a recursion call. In this case, all the following instructions will continue to be added to the current entry of *fun_list*, since it is basically always the execution of the same function;
- if the current instruction is a *ret* one, it will be stored in the current position in *fun_list*. Since it is not yet possible to determine a priori whether the function is terminated or not, as it may be in the middle of a recursion, this function is not currently included in *complete_fun_list*. For this purpose, the following line is examined:
  - if the next function name is different from the current one, the function is really terminated: the corresponding information about the function name and the list of its instructions will be moved from *fun_list* to *complete_fun_list*. In testing this algorithm in reality, there were cases in which the terminated function did not return the control to its caller but went further up the call tree, even returning the control to previous functions: in this case, probably due to some compiler optimization, all the 'skipped' functions by this *ret* function will be considered terminated, added to *complete_fun_list*, and deleted from *fun_list*;
  - if the next function name is equal to the current one, this means that this function is in the middle of a recursion: it is not yet terminated, so, also in this case, all the following instructions will be added to the current entry of *fun_list*.
- if the current instruction is a *jump* one, it will be stored in the current position in *fun_list*. For the above-mentioned reasons, this instruction can also be used to call other functions or to return from them. Therefore, also in this case, the following line will be examined:
  - if the next function name is different from the current one, it is the case in which the *jump* is used to change function. To understand in which of the two cases we are (call or ret), the offset is examined:
    - if it is 0, it means that the *jump* is used as a *call* instruction: a new function was called, so in the next iteration a new entry in *fun_list* will be created;
    - if it is different from 0, the *jump* is used as a *ret* instruction: the corresponding information about the function name and the list of its instructions will be moved from *fun_list* to *complete_fun_list*, as well as all the other 'skipped' functions, if any, as explained above.

– if the next function name is equal to the current one, it means that the *jump* instruction is used as usual to jump to other parts of the code, but always in the context of the current function: it will be added to the corresponding entry in *fun_list*. (N.B. this cover also the case in which the jump is used to call a recursion)

- in all the other cases, the instructions are simply part of the current function, so they are added to the corresponding entry in *fun_list*.

Figure 7.8 shows the trace-reordering loop.

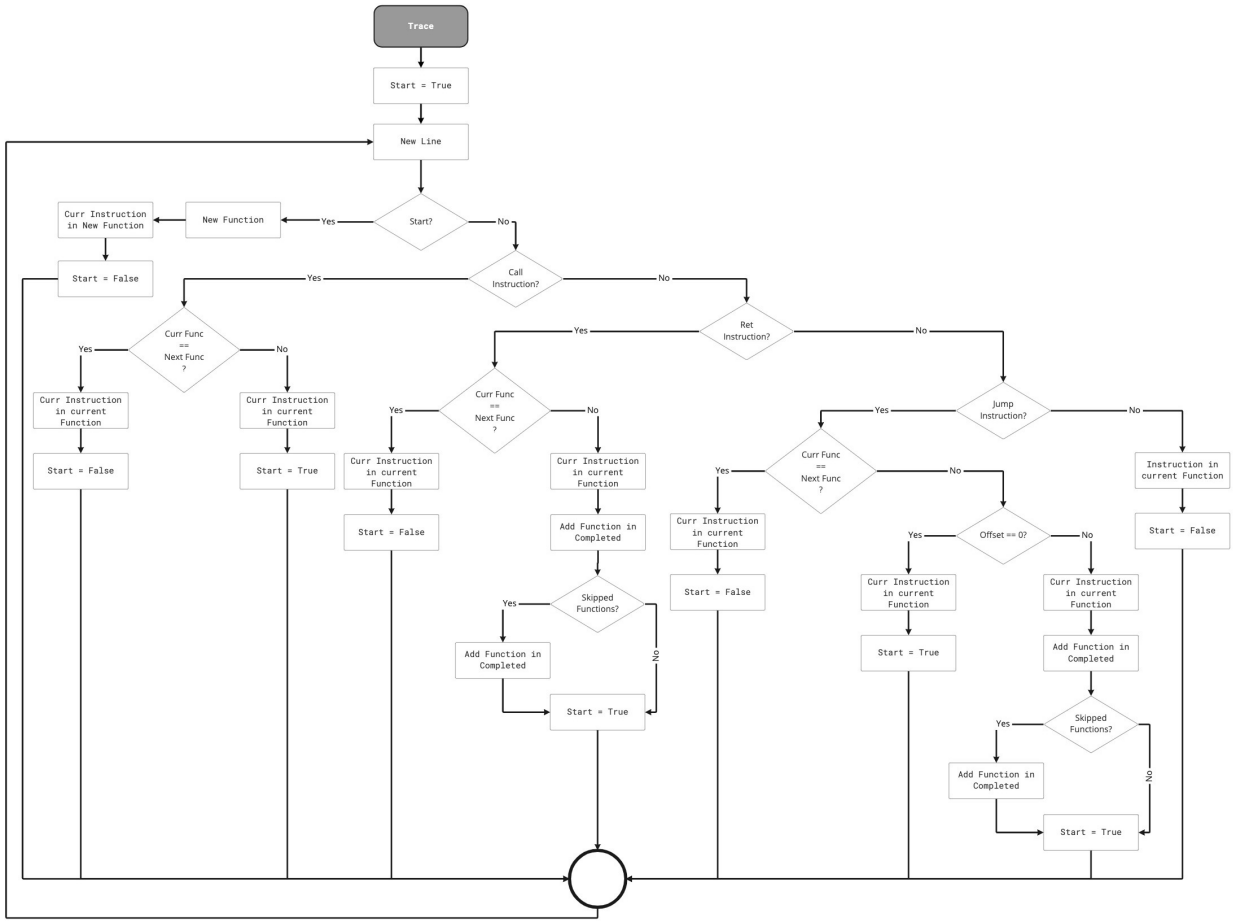


Figure 7.8: Trace reordering loop

It should also be noted that the call to external libraries was also handled implicitly, as their execution begins and ends with their call. The next function will therefore be the same as the current one and in this way fall back to the cases where recursion was handled, saving the instructions in the current function.
The correct termination of this algorithm implies that in *complete_fun_list* there is, for each function executed in the trace, the pair (function, instruction list) and that each of them appears in order of termination. As a further check, the list *fun_list* must be empty,

which ensures that all functions have been moved correctly in *complete_fun_list*.
Finally, for each function, a file was created containing all the instructions executed by the function itself.
From now on, we will consider a trace as the set of instructions executed by a function.

Code disassembly

After the data reordering phase, each function executed by one program is grouped in a different file, each containing only the instructions it executed.
The length of the individual functions will certainly be shorter than the entire initial trace, but it is not possible to have an a priori estimate of its length, as it depends on the instructions executed. In the case of very complex functions containing many or long loops, or recursions, this length may not be negligible and must be handled appropriately.
As explained above, in fact, the neural network model we are going to use (LSTM), is capable of handling longer sequences than a normal RNN, but still has difficulty with excessively long ones.
Therefore, all subsequent phases attempt to analyze the individual function in such a way as to reduce its length while trying to minimize the loss of semantics.

The first step involves translating the opcodes into an equivalent format in the assembly language. This is usually done by tools called *disassemblers*, which are used to convert machine code into a more readable format, allowing the programmer to read and understand the otherwise incomprehensible code.
In the case under analysis, disassembly will be essential to discover the structure and data on which the various instructions operate, and this will allow, in subsequent phases, to implement strategies aimed at simplifying and compressing instructions.

To achieve this result, the Capstone[2] disassembly framework was used. As explained in Section 2.5.1, this tool offers the possibility of disassembling the code by exploiting various architectures and modes of usage. Among them, the one used was *CS_ARCH_X86* and the mode *CS_MODE_32*, which represents the x86 architecture with 32-bit registers.
In a straightforward way, for each function, the opcodes for the executed instructions were placed in a buffer and analyzed by Capstone. As output, these instructions were translated in x86 assembly code.

Simplification

Next, a simplification phase is necessary in order to find a representation as compact as possible for the sequence of instructions executed by a function.
So far, a trace is nothing more than a list of all the instructions that are executed and disassembled.
A generic instruction is composed of:

[2] https://www.capstone-engine.org/

- *mnemonic*: a special string that describes the operation to be performed, such as add, move, sub, mul, etc.

- *operands*: data on which the mnemonic performs its operation. An instruction can generally have from 0 to 3 operands, which can be only registers, memory locations, or immediate values.

As a consequence, all the possible combinations of mnemonics and operands admitted by the language, would make it very difficult to find identical instructions executed one after the other in sequence.
In this context, trying to find recurring patterns to condense a trace and make it shorter can be very difficult, if not impossible.
Therefore, the proposed solution consists of trying to simplify the instruction as much as possible in order to obtain a representation that can preserve its original semantics and at the same time minimize the variety of elements of which it is composed.
The final objective will be to obtain syntactically identical instructions in the presence of semantically similar instructions.

in order to implement this solution, both the simplification of mnemonics and operands will be discussed.

Mnemonic simplification The first step is to identify equivalence classes that can relate a subset of all possible mnemonics that share a similar behaviour, with a given name representing that subset. For example, we can search for all instructions that have the common behaviour of moving/copying data from a source to a destination, which are the instructions mov, movd, movz, etc, and relate them to the string *mov*. In this way, whenever one of the instructions belonging to this subset is found in the code, it will be replaced with the string *mov*.
This reasoning was extended to all the instructions that appeared most frequently within the code, and the results are shown in Table 7.2.

Some other instructions, although common enough to appear in almost every track, such as *or*, *neg*, did not appear frequently enough to justify their simplification and were therefore left as they appeared.

Operand simplification Similar to mnemonics, operands can also be simplified. In this case, the reasoning is based on the access time to the resources.
The access time to different registers has an impact that can be considered negligible. Following this reasoning, for example, the access to *eax* will be indistinguishable from the access to *ebx*, so they were all treated as the same entity. The same reasoning can be applied for immediate values, whereas for other memory access, a remark must be made.
Any memory access (not involving registers) is theoretically highly dependent on the type of memory being accessed: reading/writing a value from a cache has far less impact than performing the same operation on disk. However, keeping track of such information is extremely difficult in a complex context such as that of a program execution: the choice of accessing one type of memory rather than another depends on many factors, related to the state of the memory during the execution of each instruction.

Simplified Instruction	*Equivalent Instructions*
mov	mov, movd, movz, movs, movzx, movsx, cmove, cmovne, movsd, cmovg, cmovbe, cmovge, cmovs, lea, lds, les
add	add, adc, inc, addsd
sub	sub, sbb, dec, cmp, subsd
mul	mul, imul, mulsd
div	div, idiv, divsd
and	and, test
shift	sal, shl, sar, shr
rot	rol, ror, rcl, rcr
c_jmp	je, jz, jne, jnz, js, jns, jg, jnle, jge, jnl, jl, jnge, jle, jng, ja, jnbe, jae, jnb, jb, jnae, jbe, jna, jo, jno, jp, jnp, jcxz, jecxz, loop, loopz, loopnz, loope, loopne

Table 7.2: Equivalence class for mnemonics

Keeping track of this information would have led to an over-complication of the model, which would have been totally unmanageable. For this reason, the type of memory accessed by an instruction was not taken into account during the data collection phase.
Since this distinction is therefore not present within the traces, the proposed solution takes into account all memory accesses as if they were equal.
Therefore, the proposed solution consists of:

- simplify all registers, which are eax, ecx, edx, ebx, esp, ebp, esi, edi, (considering the 32-bit register architecture) by replacing them with $\_R$;
- simplify all immediate values by replacing them with $\_N$.
- simplify all memory accesses by replacing them with $\_M$;

Another problem that could make all the efforts made so far useless concerns the number of operands in an instruction. Their number, in fact, can vary, but in most cases they are between 0 and 2, in rare cases 3.
Since, on average, an instruction operates on 2 operands, all their possible combinations, even if they have previously been simplified, could still lead to having semantically the same but sintattically different instructions.
As an example, we can think of two mov-type operations that operate one between register

and memory and the other between memory and register.
in this case the simplification would result in:

$$mov\_R\_M,\ mov\_M\_R$$

These two instructions have practically the same behaviour, and also the access time to their operands, on the whole, is the same; the inverted order of the operands, however, makes them different.
Therefore, the proposed solution provides, in the case of 2 or more operands, to a further simplification, in which they are reduced to one, more precisely the one whose access time weighs the most on the entire execution. In this way:

- one memory access will always prevail over register and immediate value access;
- a register access will prevail over an immediate value access;
- immediate value access will always have the worst, as it is considered the fastest, assuming that there is no instruction operating exclusively on immediate values.

The algorithm that makes this decision is called every time an instruction has more than one operand, and its pseudocode is the following:

```
1 foreach operand ∈ instruction.operands do
2 |   if operand == 'M' then
3 |   |   return 'M'
4 |   end
5 end
6 return 'R'
```

Figure 7.9 shows a final example in which a trace appears before and after simplification.

Compression

The compression step represents the last part in which the trace is manipulated, before being inserted into the dataset.
The previous steps, which involved the disassembly and simplification of instructions, were intended to make the trace readable and understandable and subsequently modify the syntax to ensure that semantically similar instructions had the same syntax.
The sole purpose of these steps was to prepare the trace for the compression step, which will allow the length of the traces to be reduced.

The compression algorithm chosen is *Run-Length Encoding*. As explained in Section 2.4.1, *RLE* it is a common lossless data compression technique which compresses data by reducing repetitive and consecutive data.
The basic version of this algorithm scans the data looking for individual repeating characters. For example, the string *ABBCCCD* will be compressed into *1A2B3C1D*.

Applying such a version to the traces would result in the decomposition of each instruction into its character-level compressed version, which would lead not only to a complete loss of the semantics of the instruction itself, but in most cases also to an unnecessary increase in the size of the compressed file, since a single instruction rarely contains repeated

characters. In fact, taking the *mov_M* instruction as an example, its compression would result in *1m1or1v1_1M*.

In this scenario, it is necessary to develop a customized version of the RLE algorithm. Such a version will have to handle each instruction in the trace ad hoc, in order to compress at instruction level. This new version will reduce repetitive and consecutive instructions by scanning each one and recording the number of times an it appears, followed by the instruction itself. For example, in the case of a trace containing *mov_M mov_M*, compression would result in *2 mov_M*

The revisited algorithm involves the use of a counter *cnt* and two indices i and j in two nested cycles that scan the trace to be compressed.
The outermost loop scans the trace taking into account the i-th instruction and initializes $cnt = 1$.
The inner loop searches in the same trace for all occurrences of the i-th instruction, starting with index $j = i + 1$ and incrementing the counter each time it encounters an equal one.
As soon as a different instruction is found, the inner loop is stopped, and the counter-instruction pair is saved as a result.

The pseudocode is as follows:

```
1  i ← 0
2  compressed_trace ← []
3  foreach instruction1 ∈ trace do
4  |   cnt ← 1
5  |   j ← 0
6  |   foreach instruction2 ∈ trace do
7  |   |   if j > i then
8  |   |   |   if instruction1 == instruction2 then
9  |   |   |   |   cnt ← cnt + 1
10 |   |   |   else
11 |   |   |   |   break
12 |   |   |   end
13 |   |   else
14 |   |   |   j ← j + 1
15 |   |   end
16 |   end
17 |   i ← i + cnt
18 |   compressed_trace ← (cnt instruction1)
19 end
```

Figure 7.10 shows a final example in which the same simplified trace shown in Figure 7.9 appears before and after compression.

Dataset creation

The last step consists in inserting at the end of the trace the information regarding the overhead given by the application of the two previously chosen obfuscation transformations

(code flattening and opaque predicate) to the function itself. These two values are taken directly from the files previously created in Section 7.2.1, where, as already explained, each file contains all functions executed by a program with a specific set of inputs, with its overhead.

Figure 7.11 shows a final example in which the same simplified trace shown in Figure 7.10 appears before and after the addition of overheads.

In this phase, the information previously mentioned regarding the order in which the various functions are written in these files is finally used. In fact, in Section 7.2.1 it was explained that by construction the functions are written in order of termination, from the first to the main, which is the last to terminate.
This information was very useful as a further check to verify the correctness and completeness of the results obtained. The verification is twofold:

- *Correctness*: it has been verified that the order in which the different functions were generated from the initial trace during the reordering phase is equal to the order present in the overhead files. In this way, it was checked that the function call tree had been correctly reconstructed, respecting the termination order of each function;

- *Completeness*: it was verified that all the generated functions were present in the overhead files.

Only if these checks were correct for all the functions of all the traces of the program, then the traces will be inserted into the dataset. In fact, if even one of these checks fail for a single function, the entire algorithm would crash and the error would be reported.

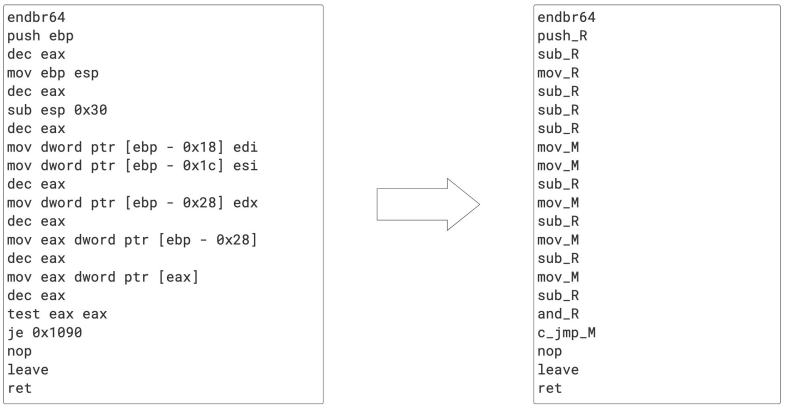


Figure 7.9: Trace before and after simplification

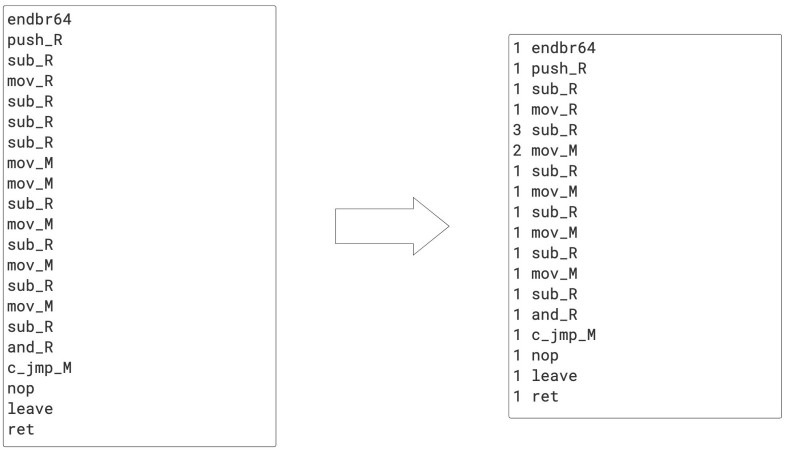


Figure 7.10: Trace before and after compression

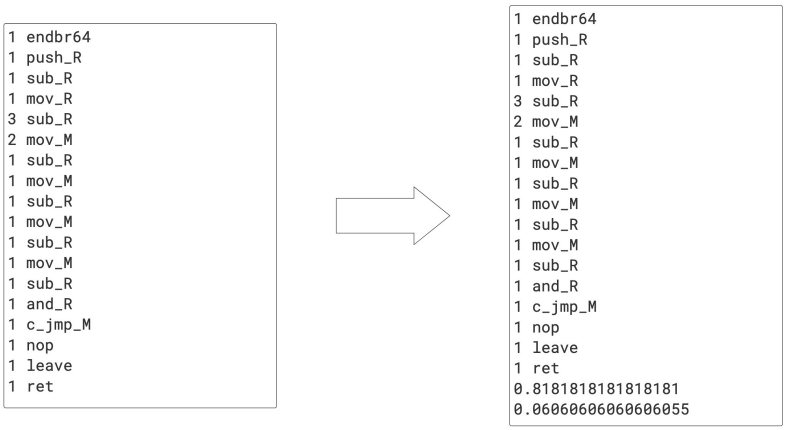


Figure 7.11: Trace before and after the addition of the overheads

7.3 Data preprocessing

The data collected and analyzed in the previous phase were used to populate the dataset, which now has a suitable structure to be processed by a sequential neural network. Each trace, in fact, represents a sequence of information linked by a temporal relationship. The goal of the network will be to predict the last two values, i.e. the overheads.

During the phases that involve the creation of the dataset, the data was analyzed exclusively from the point of view of its form. In fact, the previous phases had the sole and precise purpose of analyzing certain programs and sampling their execution, generating traces that had a structure suited to the type of architecture taken into consideration during the initial phase, in which the main purpose of this thesis work and its possible resolution were defined.

Although the dataset now has a correct structure, no emphasis has yet been placed on the quality of the observed data: so far, the data are still too raw and the presence of inconsistent data, incorrect values and outliers could produce meaningless predictions.
For these reasons, a step is required, in which the quality of the data will be examined, applying techniques to analyze, filter, transform, and encode the data, so that the machine learning algorithm can be fed correctly.
This phase is called *Data Preprocessing*, and is common to all real-world applications of artificial intelligence, as data extracted from real-world scenarios always present missing or inconsistent values.

The detailed workflow followed is presented in Figure 7.12.
The remainder of this section will be spent discussing all the steps shown in the figure.

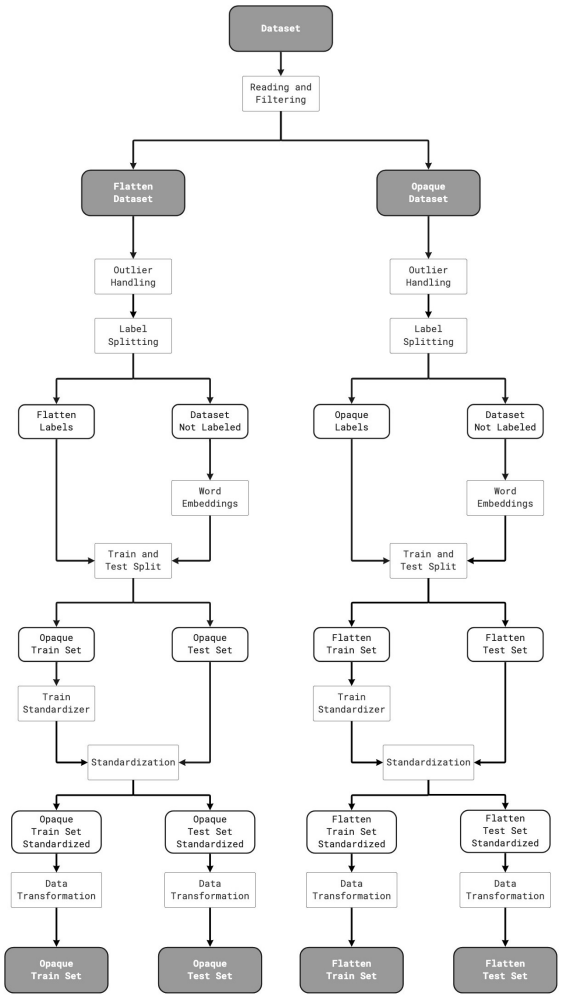


Figure 7.12: Data preprocessing detailed

Dataset reading and filtering

Since the initial dataset contains, within each track, the overhead values related to both the flattening and opaque transformations, the choice of splitting the dataset, creating one relating only to the flatten transformation and one to the opaque transformation, seemed the most logical solution. In fact, in this way, the same trace will be present in both datasets, and each will contain only one overhead value.
It will then be possible to analyze the two datasets separately, being able to make individual considerations and taking into account a single overhead value at a time.

Taking, for example, a trace that has a negative overhead value for the flatten transformation and a positive overhead value for the opaque one, without this division, the only possible solution would be to discard the trace, as a negative overhead value is meaningless. However, this would also lose the correct value present, which would have been instead.
Considering instead two separate datasets, this trace would have one overhead (the negative one) within the flatten dataset, and another (the positive one) within the opaque dataset. Consequently, the same trace would be discarded from the first dataset and kept in the second.

Parallel to the reading of the dataset, the filtering phase was also carried out to avoid a second reading of the data.
Basically, two different types of filtering were carried out: one based on overhead values and one on traces length.

As explained in Section 7.2.1, the overhead value is computed by applying the formula 7.1. The case where a trace presents overhead < 0 means that the vanilla function takes longer to execute than its obfuscated version. Since, by construction, the chosen transformations worsen the performance once applied to the program, there can be no cases where the obfuscated application is faster than the original. Such results are probably due to accidental slowdowns caused by the operating system, such as interrupts triggered during the execution of the vanilla function.
Traces with overhead values $= 0$ were also disregarded, as the case where the vanilla time is equal to that of the obfuscated version is of little interest.
For this reason, all overhead values ≤ 0 were discarded.

The other type of filtering concerns the length of the traces.
Although the traces were manipulated and compressed appropriately in order to reduce their length, unfortunately there are still traces that are too long to be processed. Therefore, the adopted solution is to discard all traces with a length > 1000.

Applying such filtering to the datasets, From the initial 12846554 traces, 4535822 were selected for the flatten dataset, and 4361729 for the opaque one.

Outlier handling

The detection and handling of outliers represent a crucial parts of the data preprocessing phase.

As explained in Section 3.3, an outlier is an observation in the dataset that deviates from the rest of the data distribution.
Although in some contexts outliers may be cases of interest because they could be introduced into the dataset for malicious purposes, in our the context, such observations will be handled as noise, i.e a phenomenon in the data that is not of interest and that can hinder the statistical analysis and training process of a machine learning algorithm, leading to a deterioration in performance. For these reasons, they must be removed before carrying out any analysis of the data.

All steps in the detection and removal of outliers are shown in Figure 7.13.
The image shows the various steps performed on a single dataset, as the steps are exactly the same for the two examined datasets.

Figure 7.13: Outlier handling

Data grouping Extremely inconsistent data are present in the dataset at this time. The extreme variability with which the execution of a trace can be slowed down by factors that are outside the direct control of the programmer and that depend mainly on the operating system, leads to cases in which the same program, executed several times, has totally different execution times. This obviously has the consequence of the variability of the corresponding overhead values.
The final result is that there are many traces in the dataset that have the same sequence of instructions and thus represent the same execution, but with completely different overheads. This extreme variability could lead the machine learning algorithm to make very inaccurate predictions, as it would not know which of the many values to associate with the particular execution trace.

For this reason, the solution proposed in this first phase aims to group all traces having an identical assembly code into a hash-table like structure, *data_grouped*, in which the assembly is stored as the key, and the corresponding list of overheads are the values.

The results obtained indicate that in the flatten dataset there are 1720 unique traces, whereas in the opaque dataset there are 1194 unique traces.

Outlier removal In this step, using the data structure we have just created, outliers will be detected and removed.

However, first of all, it is necessary to define what an outlier is meant by in the domain of our dataset.

As explained in the previous step, in *data_grouped* there is, for each unique trace, the corresponding list of overheads. These values represent the relationship between various

executions of the same vanilla application and its obfuscated counterpart. Very different overhead values mean that the execution times of the trace are not stable and that there is a lot of variability in the trace itself. To define a measure that could, in some way, give a total indication of the overhead of such a trace, the average of the list was initially used. During the analysis of these values, however, it was realized that there were extremely high overheads, which led this statistic to become skewed towards these values.
For this reason, a more robust measure was chosen, namely the *median*, which is the central value of an ordered distribution and thus indicates a typical realization of the distribution.

Once a measure representing the typical overhead value for the trace under consideration has been defined, the next step is to find out whether this track has overhead similar to this value or not. For this purpose, all values within the overhead list that are far from their median are counted. In particular, a counter is incremented each time a value of overhead $\geq (median + 0.5 \cdot median)$ or $\leq (median - 0.5 \cdot median)$ is observed. This threshold was chosen empirically on the basis of the observed values. Subsequently, if at the end the counter is greater than half the length of the list, the trace under consideration will be considered as an outlier. This is because it would mean that more than half of the observations are distant from the median by a non-negligible value, so it follows that the variability of the trace is too high to consider it as quality data.

If, on the other hand, the counter is smaller than the chosen threshold, then only one instance of the trace will be included in the dataset, with the median calculated above as the overhead value.

The pseudocode is as follows:

```
1  foreach group ∈ data_grouped do
2      cnt ← 0
3      assembly ← group.key
4      overhead_list ← group.value
5      median ← median(overhead_list)
6      foreach overhead_value ∈ overhead_list do
7          if overhead_value ≥ (median + 0.5 · median) or
             overhead_value ≤ (median − 0.5 · median) then
8              cnt ← cnt + 1
9          end
10     end
11     if cnt < length(overhead_list / 2) then dataset ← (assembly, median)
12
13 end
```

Figure 7.14 shows the outlier-removal loop.

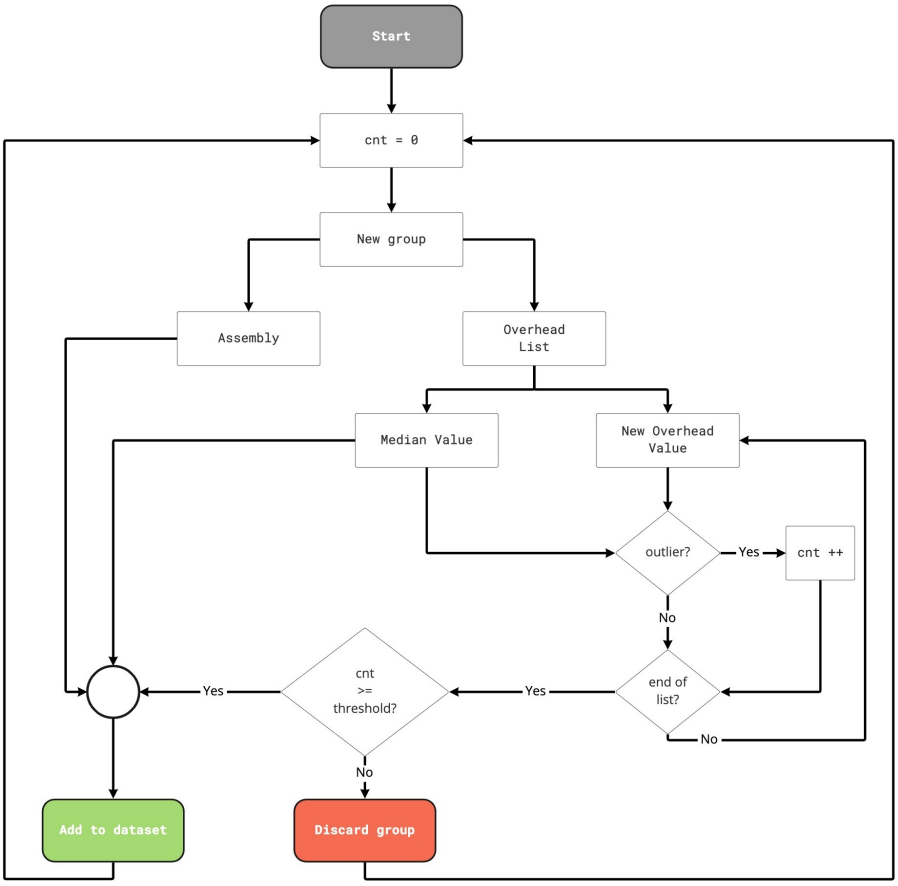


Figure 7.14: Outlier removal loop

in particular, applying this algorithm on the two datasets, 53 outliers were found in the flatten dataset and 86 in the opaque dataset.

Noise removal So far, the dataset contains traces with a unique sequence of assembly instructions and an overhead value given by the median of all values that were previously associated with the trace.
All the steps performed so far eliminated possible fluctuations in overhead values within the traces, but the overall distribution of these values is still unbalanced, presenting few values so high and far from the distribution that it is impossible to think they could be the result of smooth executions. Such high values suggest some possible measurement error or some unfortunate code run in which the CPU triggered some interrupt.
For this reason, all these problematic traces were discarded.

In this case, the standard deviation was used, whose formula is given by

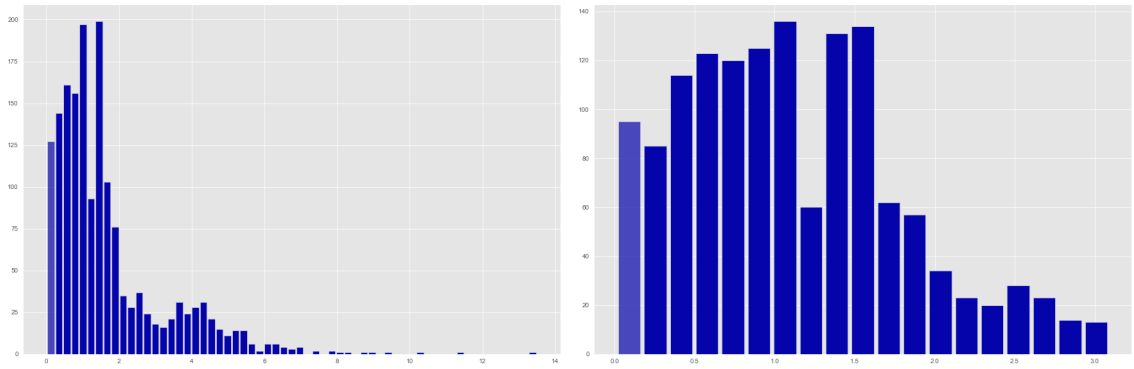
$$\sigma = \sqrt{\sum_{i=1}^{N} \frac{(x_i - \mu)^2}{N}}$$

and represent a measure of how far the data deviates from the mean, i.e. it measures the dispersion of a distribution. In this case, since the distribution of the two datasets is extremely different, as the opaque dataset is much more unbalanced than the flatten dataset, it was decided to cut off different portions of the distributions from the different datasets. In the case of the flattening dataset, only the values that fell in the $[-2\sigma, +2\sigma]$ range were considered, covering almost the entire range of the starting observations.
On the other hand, for the opaque dataset, a sharper cut was required, opting to retain only the values that fall between $[-\sigma, \sigma]$.

Figures 7.15 and 7.16 show the distribution of the two datasets before and after application of this technique.

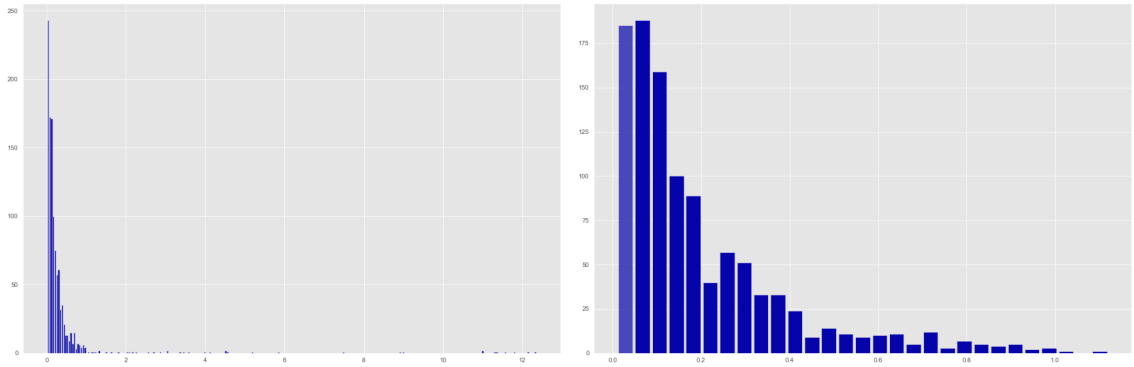
This final step resulted in the cutting of additional 270 tracks from the flatten dataset and 37 from the opaque dataset. The final size of the two datasets is 1397 traces for the flatten dataset and 1071 traces for the opaque dataset.

In parallel with the removal of the outliers, the labels, i.e. the overhead values, were also split from the dataset.



(a) Before noise removal (b) After noise removal

Figure 7.15: Outlier removal for flatten dataset



(a) Before noise removal (b) After noise removal

Figure 7.16: Outlier removal for opaque dataset

Word embeddings

So far, the constructed dataset consists of program execution traces, each containing a different sequence of instructions.
Instead, what all traces have in common is their structure. In fact, each of them has two fields for each row containing:

- an integer value representing the frequency with which a given instruction appeared in sequence;
- a string representing the instruction performed.

Since most machine learning algorithms only work with numeric values, the categorical features, i.e. the instructions, represent a problem, and they need to be transformed into numerical ones.
For this purpose, the proposed solution was to represent instructions as vectors of continuous values, a technique known as word embeddings.

Word embeddings are, in fact, a class of techniques in which individual words are represented as real-valued vectors in a predefined vector space. Each word is mapped into one vector, and its values are learned in a way that resembles a neural network.

The distributed representation is learned based on the usage of words. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning.

The main problem faced when choosing the best embedding solution was the need to represent words that are not commonly used. In fact, most word embedding models work with pre-computed word dictionaries and do not accept words that are not present. Although such structures usually contain an immense amount of words, in our case the strings used to represent instructions are so specific that none of them is present in any dictionary.
The answer to this problem, as explained in Section 3.2.1, was found in *FastText*, a deep

learning model capable of learning word representations while taking into account their morphology. To model morphology, it considers subword units, and represents words by a sum of its character n-grams. By using a distinct vector representation for each word, the model ignores the internal structure of words. Therefore, it is capable of building word vectors for words that do not appear in the training set [30]. The practical implementation provides, for each of the two datasets:

- the generation of a custom vocabulary, which in our case was created by scanning all the traces in the dataset and taking all the different instructions;
- The choice of size for the output vector containing the embedding of an instruction, which in our case was set to 10. In this way, the embedding of a single instruction (e.g. *mov_R*) will be equal to a vector of 10 real numbers;
- The addition of 10 initially empty columns to our dataset;
- The training of the FastText model on the generated vocabulary, so as to create a vector for each word within it;
- The filling of the previously added columns with the model's output values;
- The deletion of the column that previously contained the assembly instructions in string form;

The final result is that each row of a trace consists of a column containing an integer value representing the frequency at which the given instruction appeared and 10 columns representing the instruction.

Train and test split

This section represents a fairly standard but not less important part to apply to the dataset and is part of the steps before training the neural network.
In fact, it is a model validation procedure that allows the model performance to be simulated on data never seen before. In fact, the main purpose of any supervised learning algorithm is to build a model that will perform accurately on new data.

To do this, it is possible to split the original dataset into *training set* and *test set*, random sampling without replacing about 80% of the data, placing them in the training set, and placing the remaining 20% in the test set.

Standardization and transformation

the final step, which precedes the definition of the deep learning model, is to apply certain transformations to the data in order to make them suitable for the network.

The first step is to *standardize* the data. Since the range of the variables may differ a lot, using the original scale may give more weight to the variables with large values. In fact, variables that are measured on different scales do not contribute equally to the analysis and could end up creating bias. To address this problem, all data are centered, causing them to have mean 0 and standard deviation 1, so that we are dealing with the same scale of possible values.

It should be noted that, at the implementation level, a model will be instantiated that will be trained on the training set only, computing its mean and standard deviation, and then uses these values to apply standardization on both train and test set. This is because the test set represents new observations and theoretically we cannot have access to its information, such as mean and standard deviation.

Eventually, since the network model expects inputs as 3-dimensional tensors[3], the dataset was converted specifically to respect this constraint.

7.4 Model build and overhead prediction

Thanks to all the previous steps, it was possible to construct a complete and coherent dataset, both in terms of structure and content.
Each trace represents an execution of a single function, so as to be consistent with the way Tigress works, and contains the list of all the instructions executed, appropriately analyzed, and processed in such a way as to be as short as possible but still preserve their semantics. Each trace is associated with an overhead value calculated in such a way as to represent a typical execution of that function, appropriately calculated among all functions that presented the same instructions, thus avoiding taking values of unfortunate executions that could lead the network to create bias.
The final step is to construct a neural network model capable of handling this dataset appropriately.

Model build

As discussed in Sections 3.1.3 and 3.1.4, the sequential nature of the data in the dataset led to the choice of a deep learning model based on Recurrent Neural Network. In particular, *Long Short-Term Memory* was developed to deal with the problem *vanishing gradient* and thus handle even the longer sequences.

In particular, the practical definition our network passes through the definition of some layers:

- an *LSTM* layer, which applies a multi-layer long short-term memory (LSTM) to an input sequence. For each element in the input sequences, it computes the functions i_t, f_t, g_t, o_t, c_t, and h_t, as explained in Section 3.1.4;
- a *Fully Connected* layer, which applies a linear transformation to the incoming data;
- *ReLu* layer, which applies the rectified linear unit function for each element.

The next step involves the implementation of the forward pass, in which, each time, an entire sequence will be passed through the network.
It will perform the following steps:

[3] a tensor is a multi-dimensional matrix containing elements of a single data type.

1. set the initial vectors representing hidden state and cell states h_0 and c_0 to 0;
2. pass h_0 and c_0 to the LSTM layer, together with the input at the current timestamp t, x_t;
3. new hidden state h_n, current state c_n and the *output* are returned by the LSTM layer;
4. reshape the output to Fully Connected layer shape;
5. apply the ReLu activation function on the output of the LSTM layer;
6. pass the output through the fully connected layer;
7. return the output, which represents the prediction made by the network.

with the definition of all the layers we are interested in, and the forward pass, the model is complete.

Train

In the training phase, the following hyperparameters will be defined:

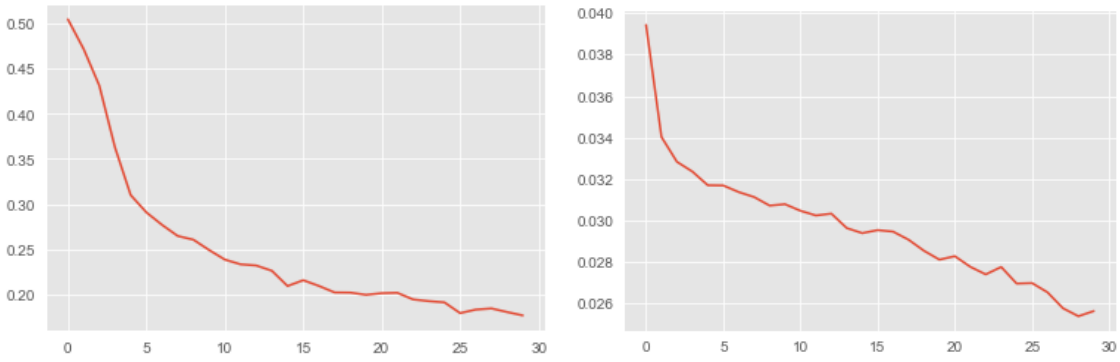
- *input_size*: the number of expected features in the input x, i.e. the columns in the dataset, which in our case will be 11, since the first represents the frequency with which the instruction was repeated in succession, while the remaining 10 represent the embedding of the instruction;
- *hidden_size*: the number of features in the hidden state h that the LSTM should create, i.e., how many LSTM cells are in the hidden layer and how many outputs the first layer will have;
- *num_layers*: the number of staked LSTM layers. For example, setting $num\_layers = 2$ would mean stacking two LSTMs together to form a stacked LSTM. In our case, only 1 LSTM layer will be created;
- *num_classes*: the number of output classes, i.e., how many values the network has to predict. In our case, only 1 value will have to be predicted;
- *num_ephocs*: the number of iterations through the dataset;
- *learning_rate*: parameter that influences both learning time and convergence (see Section 3.1.2).

Two different instances of the network were defined, passing each such hyperparameters: the first will be trained on the flatten dataset while the second on the opaque one.
At the beginning of the training, the loss function *MSE* (see Section 3.1.2) will be defined, which creates a criterion that measures the *mean squared error* (squared L2 norm) between each element in the input x and the target y.
Subsequently, the *Adam* optimizer (see Section 6) will be defined, which will hold the current state and will update the parameters based on the computed gradients.

The training loop consists of the following steps:

```
1  criterion ← MSELoss() /* loss definition */
2  optimizer ← Adam() /* optimizer definition */
3  epoch_loss ← []
4  i ← 0
5  foreach epoch do
6  |   overall_loss ← 0
7  |   foreach trace ∈ dataset do
8  |   |   output ← forward_pass(trace[X])
9  |   |   optimizer.zero_grad() /* gradient computation */
10 |   |   loss ← criterion(output, trace[y]) /* loss computation */
11 |   |   loss.backward() /* backpropagation of the loss */
12 |   |   optimizer.step()
13 |   |   overall_loss ← overall_loss + loss
14 |   end
15 |   overall_loss ← overall_loss / length(dataset)
16 |   epoch_loss[i] ← overall_loss
17 |   i ← i + 1
18 end
```

Both networks were trained many different times to test various combinations of hyperparameters and find the set that would give the best result.
After trying many combinations, it was observed that after about 30 epochs the results for both datasets remained constant, so 30 was chosen as the final number of epochs. A good compromise between training time and final results is given by a $hidden\_size = 512$ for both datasets.
Considering learning rate, for the flatten dataset the best result was obtained with 0.001, and for the opaque data set with 0.0001. The training losses are shown in Figure 7.17



(a) Flatten dataset
$lr = 0.001$

(b) Opaque dataset
$lr = 0.0001$

Figure 7.17: Training losses

Evaluation

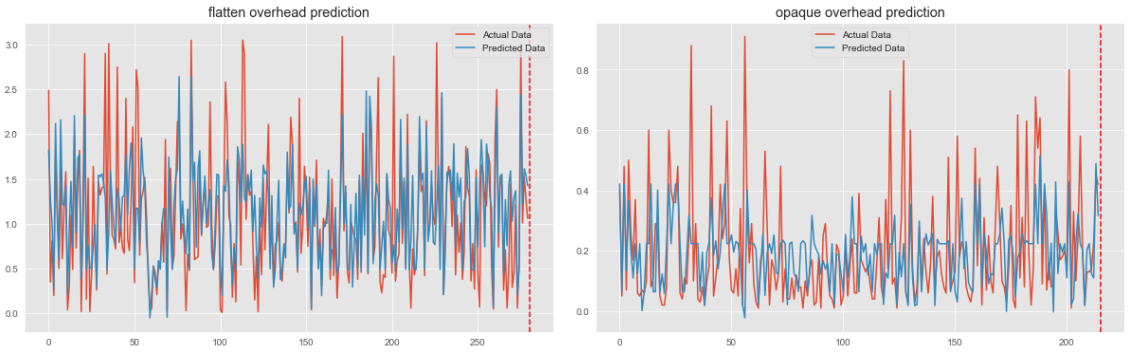
In the evaluation phase, the performance of the two different networks will be tested on new, never-before-seen observations coming from the test set.
The same weights found by the previously best-performing model will be used to make the predictions.
For each trace coming from the test set, its prediction will be calculated simply by performing the forward step. Subsequently, it is added to a list of predictions.

Eventually, *RMSE* and *MAPE* (see Section 3.1.5) will be computed, to give an idea of how the network behaved in the face of new date. Table 7.3 shows their values.

	STDEV	*RMSE*	*MAPE*
flatten dataset	0.72	0.50	98.62
opaque dataset	0.19	0.17	125.73

Table 7.3: RMSE and MAPE scores obtained

As explained in Section 3.1.5, the standard deviation can be used as a measure of how the simplest model, i.e. the one which uses the simple mean to make prediction, should perform. In fact, it measures how far the true value is from the mean.
The RMSE is quite similar to the standard deviation, but it measures how far the true value is from the prediction of the model for that value.
In both cases, the models performed slightly better than their naive counterparts, so they were able to learn from the data and make smarter predictions than the mean values.
Figure 7.18 shows the predictions compared to the real values.



(a) Flatten dataset (b) Opaque dataset

Figure 7.18: Overhead predictions vs. real values

As can be seen, the model trained on the opaque dataset achieved the worst results, both in terms of RMSE and MAPE. From the figures representing the overhead predictions, we can indeed see that it has a great difficulty in predicting values much higher than the average. These large differences between the true value and the incorrect prediction

could be the cause of these low scores, since the evaluation metrics are proportional to the difference between the true value and the predicted value. On the other hand, the model trained on the flatten dataset can better handle high-value predictions, although it still makes some errors.

The results obtained show that there are some limitations in predicting overhead, probably due to the low correlation between instructions and overhead values.
In fact, as explained in Section 2.3.6, the execution times of individual instructions depend on many factors, mostly independent of the programmer's will, but should be taken into account in some way. CPU performance has a major impact on program execution speed, and depends on:

- *interrupts*: interrupt calls require the processor to suspend the current program, save its state, execute the interrupt handling function, and then reload the program state and resume execution. Interrupt handling is one of the most important causes of program delays, and although in our case many interrupts have been intentionally disabled in order not to delay execution, some kernel interrupts cannot be disabled, therefore, the operating system will continue to block the execution of the current program in order to satisfy them;
- The presence of various *cache levels* and their size: these very fast but small memories are used by the processor to save information that will most likely be used again after a short time. If an instruction is in the cache at the time it is to be executed, then its fetch-decode-execution will take a few clock cycles; otherwise, in the case of cache misses, the time will increase considerably. Since modern processors can have several cache levels, the execution time of an instruction therefore depends heavily on where it is located.

Another very important factor to take into account is the presence of *Hardware-Based Speculation* [39] in modern processors, which allows out-of-order execution of instructions as soon as its operands are available, to optimize the pipeline management.
Instructions are broken up into micro-instructions and executed in a non-arbitrary order, also based on the prediction of conditional jumps. Once executed, these instructions are reordered due to *reorder buffer* and committed in order, to give the impression to the programmer that they were executed in order.

In conclusion, the combination of all these factors results in extreme variability in the instructions. Determining the execution time of a single instruction, as well as of an entire trace, becomes very difficult in this context, and this would demonstrate the little correlation present between the trace itself and its execution time (and consequently its overhead).
However, this work is intended as a further step towards a complete solution to the problem of performance prediction in obfuscated programs.

Chapter 8

Conclusions and future works

This thesis presents the work done to build a complete dataset with which to train a deep learning model that aims to predict the computational overhead that obfuscation adds to a program. The huge initial collection of data contains information on the execution and obfuscation of programs, and, in particular, each program, executed with one of the possible pre-computed input sets, is described in terms of:

- execution traces containing all the instructions executed for each function, together with other various statistics concerning the execution of the program;
- execution time of the non-obfuscated application in terms of clock cycles;
- execution time of the obfuscated application in terms of clock cycles.

Data on execution times were appropriately manipulated to define a measure of overhead to quantify the performance loss due to obfuscation.
Then the traces, after being cleaned of unnecessary information, were analyzed to recompose the function call tree, so that a file could be created for each executed function, each with its own instructions.
Subsequent steps focused on reducing the length of each trace. After disassembling the opcodes, a simplification was made in order to generalize an instruction as much as possible, resulting in instructions with the same syntax for similar semantics.
This simplification was used to apply compression, revisiting the classic Run-Length Encoding algorithm to take into account successive repetitions of instructions.
Finally, each trace was filled with the corresponding overhead values.

Subsequent preprocessing steps focused on ensuring better data quality, first filtering out traces that were found to be too long even after compression, and then also those with incorrect overhead.
Traces with identical instructions were grouped together, and between all the corresponding overhead values, one, the most significant, was taken to represent a typical execution. This value was used to determine whether the trace in question could be added to the dataset or was to be considered an outlier and consequently discarded.
A further cut of traces with overheads that were too far from the distribution was made,

which favors the consistency and quality of the data with respect to its quantity.
The categorical features present in the dataset required an encoding solution based on word embedding, in which an instruction was represented with a vector of real numbers of fixed size.
The sequential nature of the data led to the choice of a different neural network model from the standard one, capable of handling data that have a precise temporal relation.
In conclusion, part of the created dataset was used to train a neural network model based on LSTM, while the remainder was used to evaluate the performance of the model.

This thesis aims to get closer to a complete solution to the problem of performance prediction in obfuscated code, and could be an inspiration for possible new solutions that focus on the use of different or better implementation technologies to improve the results obtained.
Different manipulations of the data could lead to the creation of a more balanced dataset, while the use of other compression techniques could lead to a further reduction in the length of the traces. In addition, the choice of different types of machine learning model could also be considered.

Future developments can also be achieved by improving the quality and completeness of the dataset.
By taking advantage of the development environment previously created specifically for the extraction of information on the execution and obfuscation of code, it would be possible to select new programs and new obfuscation transformations, and thus expand the size of the dataset.
Considering different factors that could somehow ensure less variability between assembly instructions and improve the measurement of execution time by making it less dependent on external factors, would lead the dataset to be more consistent.

Appendix A

User manual

A.1 Requirements

- tested in a macOS and Linux environment. Should also work on Windows
- Python $\geq$ 3.8.2
- tqdm: https://pypi.org/project/tqdm/
- capstone: https://www.capstone-engine.org/
- FastText: https://fasttext.cc/
- Pytorch: https://pytorch.org/
- numpy: https://numpy.org/
- pandas: https://pandas.pydata.org/
- sklearn: https://scikit-learn.org/stable/index.html
- matplotlib: https://matplotlib.org/

A.2 Environment setup

A.2.1 Python libraries installation

Installation command for tqdm, capstone, gensim, torch, numpy, pandas, scikit-sklearn and matplotlib.

```
pip3 install tqdm capstone gensim torch numpy pandas sklearn matplotlib
```

A.3 Scripts usage

The following scripts must be launched in the following order:

1. clock_computation
2. trace_analysis
3. overhead_prediction

A.3.1 clock_computation

This script, for each function executed within each program, has the task of computing the median between the 100 execution times for both the vanilla and obfuscated versions. It then computes the overhead between the two versions of the same function using the just computed values.

```
python3 clock_computation.py <program_clock_path> <clock_out_dir>
    <overhead_out_dir> <analyzed_program_path>
```

Parameter	*Description*
program_clock_path	folder where the files containing the individual program executions are located
clock_out_dir	output folder where the computed clock values will be saved
overhead_out_dir	output folder where the computed overhead values will be saved
analyzed_program_path	check to select only the correctly analyzed programs

A.3.2 trace_analysis

This script is responsible for all trace analysis. It cleans, reconstructs the call tree, disassembles the machine code, simplifies the instructions and compresses them, and then finally adds the overhead.

```
python3 trace_analysis.py <base_path>
```

Where *base_path* is the folder where all the traces are located and where the output will be saved.

A.3.3 overhead_prediction

This script reads and processes all the traces generated by the *trace_analysis* script. It takes care of the preprocessing phase in which erroneous traces are filtered out and handles outliers. It then defines the LSTM model and performs training and evaluation on these data

```
python3 overhead_prediction.py <base_path>
```

Where *base_path* is the folder where all previously analyzed traces are located.

Appendix B

Developer manual

This manual is intended as a reference for future developments and improvements of the work presented in this thesis.

B.1 Trace analysis

It is possible to change the behavior of the *trace_analysis* script through appropriate flags defined as global variables at the start of the script.
In particular:

- RLE: data compression using the Run-Length Encoding algorithm occurs only if the value of this flag is *True*; otherwise, the traces will not be compressed. In this way, it will be possible to develop and apply new compression techniques in a simple manner, or decide not to compress the data at all, only by setting the flag to *False* and appropriately implementing new compression techniques;
- SIMPLIFY: if *True*, simplification to the instructions of which the trace is composed will be applied. It is essential if you want to apply a subsequent compression using RLE, otherwise it can be set to *False*. In this way, a single instruction will be in the form of a mnemonic + list of operands (if presents);
- CLOCK: if *True*, the information on execution time, given in clock cycles, will be added to the end of the trace instead of the overhead values. Otherwise, if *False*, the two overhead values given by applying the two obfuscation techniques to the trace will be added.

B.2 Overhead prediction

The *overhead_prediction* Jupiter notebook needs certain values, defined as global variables in the second cell, which must be appropriately initialized before being executed.
In particular:

- vector_size: integer number that defines the size of the embedding vectors. It is used by the FastText model to convert the categorical values into vectors of real numbers;

- RLE: if in *trace_analysis* script the traces have been compressed using the RLE algorithm, this flag must be set to *True* and, for each trace, an additional column will be added to the dataset containing the integer values that precede each instruction and indicate the number of times it has appeared in sequence;
- FLATTEN: this flag applies the data preprocessing, creates the dataset and uses it to train an LSTM network considering only the overhead values given by the code flattening transformation;
- OPAQUE: this flag applies the data preprocessing, creates the dataset and uses it to train an LSTM network considering only the overhead values given by the opaque predicate transformation.

The *outlier_detection* function is responsible for determining whether or not a trace is considered an outlier. This is done initially by defining two specular thresholds ($(median + 0.5 \cdot median)$ and $(median - 0.5 \cdot median)$) that indicate whether the individual overhead value is considered valid or not. Subsequently, if the count of all invalid values exceeds a predefined value (given by half the size of the overhead list relative to the trace), the trace is discarded.
It is possible to change the behavior of this function by setting different thresholds or applying new outlier detection strategies.
The *outlier_removing* function not only removes the outliers selected by the previous function, subsequently also has the task of eliminating values that are still too far from the distribution. In this case, also, a threshold is set according to which a trace is to be eliminated or not, and again this threshold can be appropriately changed or eliminated if considered appropriate.

It is possible to change the value of the network hyperparameters in the cell concerning the LSTM *parameters definition*, where the values of *hidden_size*, *num_layers*, *num_classes*, *num_epochs* and *learning_rate* are defined.
In the next cell, the LSTM network is defined. This network will be instantiated later, before the train phase, and two different networks will be created for the flatten and opaque dataset. It is possible to use different network models by only changing this cell and consequently its definition in the train phase.

Finally, in the *evaluation* function, it is possible to use different evaluation metrics, in addition to the MAPE and RMSE already used.

Bibliography

[1] S. Banescu, «Characterizing the strength of software obfuscation against automated attacks», Ph.D. dissertation, Jul. 2017. DOI: 10.13140/RG.2.2.36593.79202.

[2] C. Collberg, C. Thomborson, and D. Low, *A taxonomy of obfuscating transformations*, Jan. 1997.

[3] K. Heffner and C. Collberg, «The obfuscation executive», in *Information Security*, K. Zhang and Y. Zheng, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 428–440, ISBN: 978-3-540-30144-8. DOI: 10.1007/978-3-540-30144-8_36.

[4] S. Alberto, *Towards the prediction of performance degradation of obfuscated code*, Politecnico di Torino, 2021.

[5] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, «Protecting software through obfuscation», *ACM Computing Surveys*, vol. 49, pp. 1–37, Apr. 2016. DOI: 10.1145/2886012.

[6] J. Li, B. Zhao, and C. Zhang, «Fuzzing: A survey», *Cybersecurity*, vol. 1, Dec. 2018. DOI: 10.1186/s42400-018-0002-y.

[7] D. Lelewer and D. Hirschberg, «Data compression», *ACM Computing Surveys*, vol. 19, no. 3, pp. 261–296, Sep. 1987. DOI: 10.1145/45072.45074.

[8] U. Jayasankar, V. Thirumal, and D. Ponnurangam, «A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications», *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 2, pp. 119–140, 2021, ISSN: 1319-1578. DOI: 10.1016/j.jksuci.2018.05.006.

[9] J. Capon, «A probabilistic model for run-length coding of pictures», *IRE Transactions on Information Theory*, vol. 5, no. 4, pp. 157–163, 1959, cited By 78. DOI: 10.1109/TIT.1959.1057512.

[10] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, «Static disassembly of obfuscated binaries», in *13th USENIX Security Symposium (USENIX Security 04)*, vol. 13, San Diego, CA: USENIX Association, Aug. 2004, pp. 18–18. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/static-disassembly-obfuscated-binaries.

[11] M. Popa, «Binary code disassembly for reverse engineering», *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, pp. 233–248, 2012.

[12] B. Schwarz, S. Debray, and G. Andrews, «Disassembly of executable code revisited», in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, IEEE, 2002, pp. 45–54. DOI: 10.1109/WCRE.2002.1173063.

[13] Y. LeCun, Y. Bengio, and G. Hinton, «Deep learning», *nature*, vol. 521, no. 7553, pp. 436–444, 2015. DOI: 10.1038/nature14539.

[14] L. N. Kanal, «Perceptron», in *Encyclopedia of Computer Science*. GBR: John Wiley and Sons Ltd., 2003, pp. 1383–1385, ISBN: 0470864125.

[15] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, *Activation functions: Comparison of trends in practice and research for deep learning*, 2018. DOI: 10.48550/ARXIV.1811.03378.

[16] M. Gardner and S. Dorling, «Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences», *Atmospheric Environment*, vol. 32, no. 14, pp. 2627–2636, 1998, ISSN: 1352-2310. DOI: 10.1016/S1352-2310(97)00447-0.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[18] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. DOI: 10.48550/ARXIV.1412.6980.

[19] H. Jabbar and R. Z. Khan, «Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)», *Computer Science, Communication and Instrumentation Devices*, vol. 70, 2015. DOI: 10.3850/978-981-09-5247-1_017.

[20] S. Varma and S. Das, *Deep Learning*. Apr. 2018. [Online]. Available: https://srdas.github.io/DLBook/ImprovingModelGeneralization.html.

[21] J. L. Elman, «Finding structure in time», *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. DOI: 10.1207/s15516709cog1402_1.

[22] A. Sherstinsky, «Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network», *Physica D: Nonlinear Phenomena*, vol. 404, p. 132 306, 2020, ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306.

[23] S. Hochreiter and J. Schmidhuber, «Long short-term memory», *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: 10.1162/neco.1997.9.8.1735.

[24] D. Chicco, M. J. Warrens, and G. Jurman, «The coefficient of determination r-squared is more informative than smape, mae, mape, mse and rmse in regression analysis evaluation», *PeerJ Computer Science*, vol. 7, e623, 2021. DOI: 10.7717/peerj-cs.623.

[25] J. Nevitt and G. R. Hancock, «Improving the root mean square error of approximation for nonnormal conditions in structural equation modeling», *The Journal of Experimental Education*, vol. 68, no. 3, pp. 251–268, 2000. DOI: 10.1080/00220970009600095.

[26] A. Bakarov, *A survey of word embeddings evaluation methods*, 2018. DOI: 10.48550/ARXIV.1801.09536.

[27] F. Almeida and G. Xexéo, *Word embeddings: A survey*, 2019. DOI: 10.48550/ARXIV.1901.09069.

[28] M. Baroni, G. Dinu, and G. Kruszewski, «Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors», in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Baltimore, Maryland: Association for Computational Linguistics, Jun. 2014, pp. 238–247. DOI: 10.3115/v1/P14-1023.

[29] D. Bollegala, K. Hayashi, and K.-i. Kawarabayashi, «Learning linear transformations between counting-based and prediction-based word embeddings», *PLOS ONE*, vol. 12, pp. 1–21, Sep. 2017. DOI: 10.1371/journal.pone.0184544.

[30] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin, *Advances in pre-training distributed word representations*, 2017. DOI: 10.48550/ARXIV.1712.09405.

[31] K. Singh and S. Upadhyaya, «Outlier detection: Applications and techniques», *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 1, p. 307, 2012.

[32] S. Vinay, «Standardization in machine learning», Mar. 2021.

[33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, «Code2vec: Learning distributed representations of code», *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. DOI: 10.1145/3290353.

[34] N. D. Q. Bui, Y. Yu, and L. Jiang, *Infercode: Self-supervised learning of code representations by predicting subtrees*, 2020. DOI: 10.48550/ARXIV.2012.07023.

[35] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, «Language-agnostic representation learning of source code from structure and context», *arXiv preprint arXiv:2103.11318*, 2021.

[36] Y. Zhuang, «The performance cost of software obfuscation for android applications», *Computers & Security*, vol. 73, pp. 57–72, 2018, ISSN: 0167-4048. DOI: 10.1016/j.cose.2017.10.004.

[37] S. Banescu, C. Collberg, and A. Pretschner, «Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning», in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 661–678, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/banescu.

[38] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, *Enriching word vectors with subword information*, 2016. DOI: 10.48550/ARXIV.1607.04606.

[39] S. Wallace, N. Dagli, and N. Bagherzadeh, «Design and implementation of a 100 mhz reorder buffer», in *Proceedings of 1994 37th Midwest Symposium on Circuits and Systems*, vol. 1, 1994, 42–45 vol.1. DOI: 10.1109/MWSCAS.1994.519186.