Program Similarity Analysis for Malware Classification and its Pitfalls

Niccolò Marastoni

Supervisor: Prof. Mila Dalla Preda

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Department of Computer Science
University of Verona, Italy
May 2021

Abstract

Malware classification, specifically the task of grouping malware samples into families according to their behaviour, is vital in order to understand the threat they pose and how to protect against them. Recognizing whether one program shares behaviors with another is a task that requires semantic reasoning, meaning that it needs to consider what a program actually does. This is a famously uncomputable problem, due to Rice's theorem. As there is no one-size-fits-all solution, determining program similarity in the context of malware classification requires different tools and methods depending on what is available to the malware defender.

When the malware source code is readily available (or at least, easy to retrieve), most approaches employ semantic "abstractions", which are computable approximations of the semantics of the program. We consider this the first scenario for this thesis: malware classification using semantic abstractions extracted from the source code in an open system. Structural features, such as the control flow graphs of programs, can be used to classify malware reasonably well. To demonstrate this, we build a tool for malware analysis, R.E.H.A. which targets the Android system and leverages its openness to extract a structural feature from the source code of malware samples. This tool is first successfully evaluated against a state of the art malware dataset and then on a newly collected dataset. We show that R.E.H.A. is able to classify the new samples into their respective families, often outperforming commercial antivirus software. However, abstractions have limitations by virtue of being approximations. We show that by increasing the granularity of the abstractions used to produce more fine-grained features, we can improve the accuracy of the results as in our second tool, StranDroid, which generates fewer false positives on the same datasets.

The source code of malware samples is not often available or easily retrievable. For this reason, we introduce a second scenario in which the classification must be carried out with only the compiled binaries of malware samples on hand. Program similarity in this context cannot be done using semantic abstractions as before, since it is difficult to create meaningful abstractions from zeros and ones. Instead, by treating the compiled programs as raw data, we transform them into images and build upon common image classification algorithms using machine learning. This led us to develop novel deep learning models, a convolutional neural network and a long short-term memory, to classify the samples into their respective families. To overcome the usual obstacle of deep learning of lacking sufficiently large and balanced datasets,

we utilize obfuscations as a data augmentation tool to generate semantically equivalent variants of existing samples and expand the dataset as needed. Finally, to lower the computational cost of the training process, we use transfer learning and show that a model trained on one dataset can be used to successfully classify samples in different malware datasets.

The third scenario explored in this thesis assumes that even the binary itself cannot be accessed for analysis, but it can be executed, and the execution traces can then be used to extract semantic properties. However, dynamic analysis lacks the formal tools and frameworks that exist in static analysis to allow proving the effectiveness of obfuscations. For this reason, the focus shifts to building a novel formal framework that is able to assess the potency of obfuscations against dynamic analysis. We validate the new framework by using it to encode known analyses and obfuscations, and show how these obfuscations actually hinder the dynamic analysis process.

Acknowledgments

I have to start by thanking my wife Patience for staying with me and lending me strength through the most stressful experience of my life. The levels of frustration and anxiety reached during this PhD have definitely soured me as a person at many points, but she held on strong and comforted me. She also helped me directly by editing this thesis, so it is thanks to her that it flows well .

I would then like to thank my supervisor, Mila Dalla Preda, for all her help even before the PhD. Without her I would not have considered this career path and you would not be reading about it in this thesis. With her I need to thank my co-supervisor, Roberto Giacobazzi, for inspiring me during my years at the University of Verona and for the eye-opening discussions.

Throughout these last 3 years of my life I was lucky enough to work in a lab with people that I cherish deeply and that helped me greatly. Not directly with my research, but being able to relate to the frustration of others is a must and I greatly missed it during the months away from the lab.

My parents also deserve a special mention here, I never had to stress too much about life as their support has held me together ever since I can remember. This achievement is in good part theirs and I hope they can be proud of it.

Had any of the aforementioned people not been there during the last few years I have little faith that this thesis, this monumental task, would have been completed. Thank you all.

As an aside, I would like to mention that this is the first thing I started writing on this thesis. I started from the acknowledgments because I am hopeful that remembering how lucky I have been to receive so much support will ease the burden of filling in the rest of this manuscript.

Contents

1 Introduction

Malware, or **mal**icious soft**ware**, is a term that encompasses all programs that intentionally exhibit a malicious behaviour. Two things are worth investigating: 1) what makes a program behaviour "malicious" and 2) what it means for a program to "intentionally" exhibit a malicious behaviour.

In general, a malicious behaviour in a program can be anything that the end user did not approve of. Any program action that causes breach of privacy, denial of service, loss of data, etc., without the user's explicit approval, is an example of malicious behavior. It is an acceptable behaviour when a program sends the private data of the user to an end system if the user is fully aware and is okay with what is happening. Conversely, if the program sends private data without the user's express permission, or worse, without their knowledge, that is considered to be a malicious behavior.

Due to the various types of malicious behaviours and their wildly diverse effects, there are many categories in which to classify malware. Some of the most common types are:

- *adware* → malware that infects systems so that advertisements are injected into programs that would not otherwise have them
- *scareware* → these programs make users think something bad happened through the use of scare tactics, in order to compel them to act against their best interest
- *ransomware* → areas of the filesystem are encrypted by the malware and will be decrypted only after the user pays a hefty amount of money
- *spyware* → the private information of the user is mined without permission to be later sold or used directly by the malware vendors

This is by no means a complete list, and most malware would not fit neatly into just one of the types outlined above. Most malicious programs, in fact, tend to have more than one malicious behaviour, and thus can be grouped in multiple categories.

It is more interesting, then, to consider malware as belonging to different "families", which are more specific groups than the categories above. A malware family is a class of malware that have the same behaviours and usually descend from one common initial sample. For example, a single malware family might include all the samples that are modified from an

initial malware sample that first locks the device of the user (denial of service), then steals their information (spyware), and finally encrypts the device and asks for a ransom to decrypt it (ransomware).

The fight against malware is a continuous one: when security experts find a way to fight a particular malware family, malware developers will inevitably generate variants of the existing samples or come up with an entirely different type of malware that is impervious to the new anti-malware techniques. From the security angle of this work, malware developers will be often called "attackers" while "defenders" will refer to the agents identifying and analyzing the malware.

Malware Classification

Malware classification is the task of categorizing malware samples into specific groups. The simplest type of malware classification has two classes, malware and goodware (benign programs), and it consists of simply determining whether a program is malware or not. Familiar classification is a more complex malware classification problem, where we classify malware samples according to their "family", i.e. a group of malware that behaves similarly or, in other words, that exhibits the same malicious behaviours.

If malware can be successfully grouped according to their behaviors, the reverse engineering process can be expedited to find a "cure". A successful familiar classification of a malware sample can also allow the systems to apply whatever correction is needed to fight the malware, provided that the malware family is a known threat for which a solution has been already found.

For this reason (and possibly many others that will not be considered in this thesis), understanding when two programs have the same behaviour is an important task in software security. In order to classify malware according to their behaviours, we first need to consider what they actually do. In other words, we need to be able to group them according to their semantic similarity. This, as will be proven later in the chapter, is an undecidable problem.

The problem being undecidable does not mean that it cannot be solved ever, just that there is no algorithm that will work correctly for every instance of the problem. The key word here is "correctly", as one way to approach an undecidable problem is to voluntarily lose accuracy and thus allow for errors in the results, as long as the errors can be accounted for.

Scenario 1

One way to approximate the task of determining program similarity is to use abstractions of the programs, rather than the actual programs, and then to build a similarity measure between

these approximations. Clearly, this will result in a loss of accuracy in the results. These semantic abstractions have to be built from program code, which means that the defenders either have to possess the original source code or they need to be able to retrieve a suitable representation of it (i.e. through reverse engineering efforts or disassembly). In the scope of this work we call this "scenario 1", and it represents all situations where the source code is readily available for analysis or can be easily retrieved.

Building a semantic approximation of the program and then defining a similarity function between the approximations has the clear advantage of making the results readily interpretable. The control given to the analyzers (the defenders) in generating the program representations translates into their ability to know what these representations mean, and what their flaws are. This is a double edged sword, as the loss of accuracy allows the problem to be approximately solved but also opens it up to be exploited by the attackers. In fact, one of the drawbacks of this approach is that there are clear and defined limits to the expressivity of the approximations (and/or their similarity functions).

On another note, scenario 1 is not necessarily the most common one in the wild, especially in malware analysis.

Scenario 2

It is very common for malware (especially on desktop systems) to be distributed as compiled binaries, and as a result, it is not always easy to obtain the original source code through reverse engineering or disassembly of these binaries. For this reason we consider "scenario 2", where the malware is in a binary form and it is neither feasible nor convenient to rebuild its source code.

Working with the binary representation of a program is wildly different from the previous approach, since compiled programs do not easily lend themselves to the extraction of semantic abstractions. We could attempt to create an approximation of the binary, but that does not address the issue of the disassembly process not completing or failing altogether. Furthermore, this strategy leads to the same drawbacks as in the previous scenario.

Since it is harder to apply domain expertise and build easily interpretable program approximations and similarity functions, it might be a good idea to let them be discovered by learning algorithms instead. The malware binaries can be treated as raw data, as there are many learning algorithms that work well with raw, unstructured data. These algorithms often fall under the umbrella term of "deep learning" and are doubly apt for the task since many of them can build their own internal representations (approximations or sets of features) of the programs.

Scenario 3

So far we have assumed that malware classification has to rely on statically obtained information, namely, information retrieved either from the source code or the compiled binary. This has some advantages, as static analysis is a very well understood field that has been explored for decades and has produced many theoretical and practical results. These allow us the expertise needed to focus our efforts in the right direction. But one of the downsides is that static analysis is inherently imprecise, and there is always a way for the attacker to fool a system that relies solely on static analysis, be it hard coded or learned from raw binaries.

One way to skirt the imprecision issues of static analysis is to adopt dynamic analysis, which consists of running the programs under analysis and then analyzing the individual execution traces. We call this "scenario 3", which can be associated with any real situation where even the binary cannot be analyzed but it can be executed. The main difference here is that whatever is observed during a dynamic analysis is "true", meaning that the system cannot be "fooled" into imprecision by the attackers.

Dynamic analysis comes with its own problems. For example, since the execution traces being analyzed must be of finite length, the program under execution has to be stopped early in the case of long execution times, thus possibly losing information. There can also only be a finite number of these execution traces, which gives rise to a further loss of information.

In Section 1.3 we expand on the three scenarios in the context of malware classification.

Goal

The goal of this thesis is to explore the intricacies and the pitfalls of program similarity analysis in the context of malware classification. In order to place our work into the right context, we identify three real life scenarios and analyze how to minimize the impact of their unique pitfalls. The scenarios can be clearly divided by the amount of "openness" of the system in question and by the amount of domain expertise specific to program analysis that can be utilized to solve the problem in each of the scenarios.

This first chapter serves as a general introduction to the concept of program similarity and its use mostly in malware detection and malware classification. We will start with an intuitive distinction between syntactic similarity and semantic similarity between programs, subsequently delving into why this is a hard problem to tackle. The last section of this chapter outlines the contributions of this thesis and serves as general introduction to the next three chapters that will detail the development of three case scenarios.

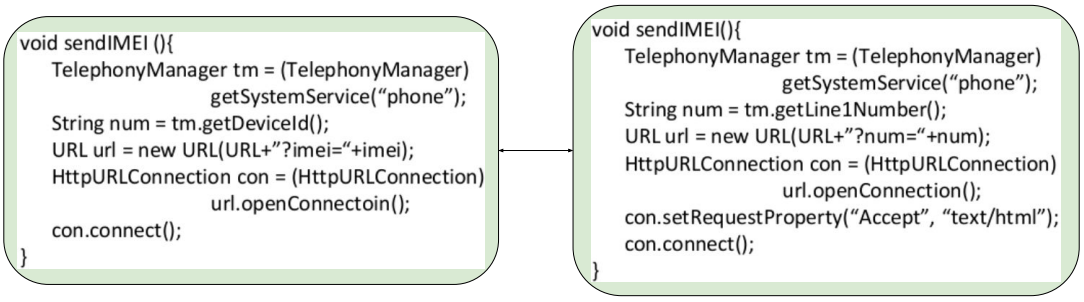

Figure 1.1: Semantically similar methods in two Android malware samples

1.1 What is Program Similarity and Why it is Useful

The first important distinction that has to be made is between program "similarity" and program "equivalence". When two malware samples are equivalent, they will invariantly exhibit the same malicious behaviours, so this condition is sufficient to classify them in the same family. It is not a necessary condition however, since two programs do not need to be equivalent in order to exhibit the same behaviours. The behaviours themselves need not be exactly equivalent between malware samples belonging to the same family, rather, they just need to be means to the same goal.

For example, if two malware samples are designed to steal the personal information of a user and then upload it to some remote server, they do not necessarily need to go through the same steps. This makes them not equal, but still similar in their semantics. Figure 1.1 shows an example of this exact scenario found in two Android malware samples.

In other words, equivalence is a stronger concept that always implies similarity. Similarity is weaker, but, as we argue at the end of this section, it is more appropriate when dealing with malware.

A second important distinction is between "semantic" and "syntactic" program similarity.

Syntactic Similarity

Take two programs P_1 and P_2. They are syntactically similar if they are written similarly, for example P_1:

```
x = int(input())
for i in range(x):
    x = x * x
print(x)
```

And P_2:

```
x = int(input())
for i in range(x):
    x = x + x
print(x)
```

These two programs are almost identical, the only difference being the operator used in the for loop, making them syntactically similar. While sharing most of their code, the two programs compute wildly different functions, namely $\prod_{i=1}^{i=x} x$ and $\sum_{i=1}^{i=x} x$.

In broad terms, when we are interested in how a program is written we refer to its syntax. When the focus is on what the program computes we refer to its semantics. In this light, P_1 and P_2 are syntactically similar but semantically different.

Syntactic similarity between programs is far from useless. Often enough, syntactic similarity implies semantic similarity. It is also useful to look for syntactically similar code when hunting for bugs in big codebases, since bugs are often the result of implementation mistakes which are inherently syntactic.

Semantic Similarity

Semantic similarity deals with what the programs actually compute and forgoes how they are written altogether. Even within the scope of semantic similarity, there are many different types of semantics that can be considered.

For example, when two programs compute the same function they have the same denotational semantics. This is referred to as functional equivalence or input-output equivalence and it is what we generally mean when we say that two programs are semantically equivalent. A stricter type of semantics, for example, is operational semantics which deals with more fine-grained information (especially small-step semantics). Two programs are equivalent according to their operational semantics if they compute the same function and they go through the same states while doing so. Operational semantic equivalence implies denotational semantic equivalence but the opposite does not hold.

Malware Similarity

It is unclear which type of similarity is more correct in the context of malware classification. If two samples are semantically equivalent, then they definitely belong to the same family. If they are syntactically equivalent, then that implies semantic equivalence and they again belong to the same family. If they are semantically similar, they belong to the same family when the shared behaviours are the defining behaviours of the family. If they are syntactically (sometimes

stylistically) similar, then they might have been written by the same developer/group or share the same code, thus possibly belonging to the same family.

This last type of classification is very valuable, as it makes it easier to trace the malware samples back to the same author [119]. Generally, knowing which features characterize a malware developer makes it easier to identify their source and stop their spread. After all, defeating the malware once it has already spread is only half the battle.

In this work, we will never consider stylistic similarity, as our goal is to group malware according to their behaviours. In order to do so, we will not use semantic equivalence but rather semantic similarity, as many malware variants tend to be equivalent only in the malicious part and semantically divergent in the rest of the program. Chapter 2 deals with this aspect more in detail.

Notation

For the remainder of this work we will alternate between intuitive notions and more formal language. In the latter, every program is usually represented by the letter P, while two or more programs will be distinguished by their subscripts as P_1 and P_2. Every program P belongs to the set of all programs $\mathbb{P}$, for which we can also write $P \in \mathbb{P}$. When two programs are syntactically different we can write $P_1 \neq P_2$, but this is technically redundant as $\forall P_n, P_m \in \mathbb{P}.n = m \leftrightarrow P_n = P_m$. If P_1 and P_2 are functionally equivalent we write $[[P_1]] = [[P_2]]$, where $[[P]]$ represents the semantic function $[[\,]]$ applied to the program P. In order to express subprograms, or parts of a program, we will indicate them as a subset of the original program with lowercase letters as in $p_1 \subset P_1$. The set of all programs semantically equivalent to a given program P is the equivalence class $P^{[[\,]]}$ defined as $P^{[[\,]]} := \{P_i \in \mathbb{P} \mid [[P]] = [[P_i]]\}$.

Programs can be transformed syntactically and this transformation generates a new program, such that a code transformation $\mathcal{T} \in \mathbb{T}$, can be defined as $\mathcal{T} : \mathbb{P} \rightarrow \mathbb{P}$, where $\mathbb{T}$ is the set of all code transformations.

A program similarity algorithm represents a function that ideally takes two programs as input and generates a measure of similarity between them: $\mathcal{S} : \mathbb{P} \times \mathbb{P} \rightarrow \{0, 1\}$. Instead of considering the whole program, many program similarity algorithms generate an abstract representation $P^{\mathcal{A}}$ that approximates the semantics of the program. Ideally the abstract representations and the similarity measure would work against program transformations so that $\mathcal{S}(\mathcal{T}_1(P_1)^{\mathcal{A}}, \mathcal{T}_2(P_1)^{\mathcal{A}}) = 1$.

1.2 Why Program Similarity is Hard

There are various degrees in which a problem can be "hard". For a human, calculating the result of $3\,459\,123 * 123\,234$ is a hard task. For a computer, this is a simple mathematical operation that can be solved in O(1), which does not mean that it can be solved instantly, but only that its difficulty does not change with a change in input size (in fact, there is no input to this problem at all).

A computer might instead struggle with an instance of sub-graph isomorphism, since it belongs to the class of NP-complete problems, that is, the class of problems for which it is not clear whether a polynomial solution exists at all. But even for how hard NP-complete problems can be, a solution to them always exists (possibly by using brute force) that will halt in finite time. This is not always the best reassurance since the heat death of the universe will also occur in finite time.

Even so, there is an entire class of problems for which we do not have even this small guarantee, which are computationally unsolvable (or uncomputable) problems. The most famous uncomputable problem is the Halting Problem: find an algorithm that will decide whether a computer program halts or not. It is unsolvable because it can be proved that no such algorithm exists.

The problem that we are set to tackle in this thesis is also an unsolvable problem, and its reason for being unsolvable is deeply connected to the halting problem itself. In this section we make a case for why program semantic similarity is such a hard problem, starting from an intuitive view of the matter and later exploring its formal foundations.

1.2.1 Intuition: Infinite Syntactic Variants Exist for the Same Program

As the challenge in determining semantic similarity between programs is finding an algorithm for it, all discussion of this problem in this work will be from the point of view of automated processes. We will not discuss how hard it is for a human to understand when two programs compute the same function.

In general, it is hard to check whether two programs compute the same function because there are infinite semantics-preserving syntactic variants for any program. This means that given any program P_1, there is an infinite class of program transformations $\mathcal{T}_1, \mathcal{T}_2, \ldots \in \mathbb{T}$ for which it is true that $\forall i \in \mathbb{N}. [\![\mathcal{T}_i(P_1)]\!] = [\![\mathcal{T}_{i+1}(P_1)]\!]$.

This is relatively easy to prove, even in an intuitive way, by using the padding technique. In most programming languages there is a statement that corresponds to `no-op`, or an instruction that does nothing at all. In assembly this is famously the `NOP` command, but even in languages

that do not have an explicit way to write an empty command we can build it by generating idempotent statements like:

```
x := x + 1;
x := x - 1;
```

By defining a function $\mathcal{T} : \mathbb{P} \to \mathbb{P}$ that adds a no-op statement anywhere in the code we create an algorithm that generates syntactic variants of programs while maintaining their semantics. Applying this function i times to a program $P \in \mathbb{P}$ can be written as $\mathcal{T}^i(P)$, i.e. $\mathcal{T}^3(P) := \mathcal{T}(\mathcal{T}(\mathcal{T}(P)))$. Every application of the function adds a no-op statement to the code, so is clearly true that $\forall i, j \in \mathbb{N}. i \neq j \to \mathcal{T}^i(P) \neq \mathcal{T}^j(P)$. At the same time, since adding a no-op does not influence the semantics of the program, we have that $\forall i, j \in \mathbb{N}. [\![\mathcal{T}^i(P)]\!] = [\![\mathcal{T}^j(P)]\!]$. Since programs are finite strings, adding one statement produces another finite string, thus the function $\mathcal{T}$ can be applied a countable infinite number of times. This generates $P^{\mathcal{T}}$, an infinite set of semantically equivalent programs that are syntactic variants of P. Considering $P^{[\![]\!]}$, the set of all programs semantically equivalent to P, it can be shown easily that $P^{\mathcal{T}} \subset P^{[\![]\!]}$ (for example, by applying the idempotent operations shown above generates variants that are not in $P^{\mathcal{T}}$). It is then proven that there exist infinite syntactic variants of any program.

The fact that infinite syntactic variants of the same program exist tells us that checking syntactic similarity alone will not cut it. Comparing the syntax of two programs is generally not enough when trying to gauge their semantic similarity. At the root of the problem is the simple fact that the semantics of a program is an inherently infinite object, while the syntax is finite. Take the following program P:

```
x = int(input())
return x + 1
```

P is a very short program at only 2 lines long. Its semantics is the semantics of the successor function, which is a map of every integer to its successor. The size of the semantics of this program is $|\mathbb{N} \times \mathbb{N}|$. From this we gather two insights:

1. The syntax of a program is easy to work with but it is often not enough for determining program similarity

2. The semantics is impossible to work with

The intuitive notions explored above do not ultimately prove that we are dealing with an uncomputable problem, but hopefully they have laid an intuitive groundwork that can be completed by the formal proof offered in the following subsection.

1.2.2 Rice's Theorem

In order to prove formally that semantic program equivalence is an uncomputable problem, we will now introduce Rice's theorem and explain intuitively why it is connected to program similarity analysis. We start by giving an overview of recursively enumerable, recursive and non-recursively enumerable sets. Note that we will only consider subsets of $\mathbb{P}$ for this section, but the definitions are the same for subsets of $\mathbb{N}$. In fact, there is an easy bijection from $\mathbb{P}$ to $\mathbb{N}$ ($\mathbb{P}$ is a set of finite strings that can be easily ordered), which means we could consider either set, but $\mathbb{P}$ was chosen for clarity.

We start by defining the concept of extensionality, which is a key part of Rice's theorem.

Extensionality

A property of programs is any subset of the set of all programs $\mathbb{P}$. For example, the property of all programs that halt when given an even number as input clearly defines a subset of $\mathbb{P}$. The sets $\emptyset$ (empty set) and $\mathbb{P}$ are, of course, properties of $\mathbb{P}$ and they are referred to as "trivial". Intuitively, it is very easy to check whether a program belongs to either of these trivial sets (it either is a program or it is not).

In order for a property Π, a subset of $\mathbb{P}$, to not be trivial there needs to be at least a program $P_1 \in \Pi$ and a program $P_2 \in \mathbb{P}/\Pi$. Some of these non-trivial sets are recursively enumerable sets, and they belong to the superset RE. If a set $A \subset \mathbb{P}$ is in RE, then there exists an algorithm which can decide whether an element $P \in \mathbb{P}$ belongs in $\mathbb{P}$ in finite time. If it is also possible to check in finite time whether an element $P \in \mathbb{P}$ does not belong in $A \subset \mathbb{P}$, then A is a recursive set. These belong to the superset R. Within R lay all decidable problems about programs.

For example, there is an algorithm that checks whether a program $P \in \mathbb{P}$ belongs to the property of "all programs that are more than 5 lines long" and one to check whether P does not belong to such a set. This makes the set recursive and the problem decidable.

There is also an algorithm that can tell in finite time whether a program $P \in \mathbb{P}$ halts when given itself as an input ($[[P]](P)\downarrow$). Intuitively, the algorithm could just run P until it eventually halts and then answer TRUE. However, an algorithm that can tell whether P will not halt on itself ($[[P]](P)\uparrow$) does not exist. This is an example of undecidable problem (also referred to as uncomputable). It describes a recursively enumerable set that is not recursive, the property K, which is famously the set of programs that halt when given themselves as inputs. It is proven that $K \in RE \setminus R$.

We can now define extensionality. A property $\Pi \subset \mathbb{P}$ is extensional if:

$$P_1 \in \Pi \wedge [[P_1]] = [[P_2]] \rightarrow P_2 \in \Pi \tag{1.1}$$

Recall that $[[P_1]] = [[P_2]]$ means that P_1 and P_2 compute the same function (denotational equivalence). In other words, a property Π is extensional if every program therein included shares the property with all its semantically equivalent counterparts. It is easy to see that both trivial properties defined earlier ($\mathbb{P}$ and $\emptyset$) are extensional properties.

It can also be shown that the empty set is the only finite extensional property. As we have proven in the previous section, there are always infinite syntactic variants to a program. Now assume that a program P_1 is in the property Π. In order for Π to be extensional, then $\forall P_2 \in \mathbb{P}.[[P_1]] = [[P_2]] \rightarrow P_2 \in \Pi$. This implies that all the syntactic variants of P_1 will also belong to Π, making it an infinite set.

Rice's Theorem

The theorem can now be stated:

Theorem 1. *Given an extensional property Π, then $\Pi \in R \leftrightarrow (\Pi = \emptyset \vee \Pi = \mathbb{P})$*

In other words, the only decidable extensional properties are whether a program is a program ($P \in \mathbb{P}$) or whether it is not ($\emptyset$). Any other extensional property then represents an undecidable problem. Not all undecidable problems are extensional. For example, the set K mentioned above is not recursive but it is also not an extensional property. We will not prove Rice's Theorem in this thesis as it needs some more background notions and it is generally out of scope.

It is interesting to note how this theorem ties in with program similarity analysis. Checking whether two programs $P_1, P_2 \in \mathbb{P}$ are semantically equivalent means proving that $[[P_1]] = [[P_2]]$. Given a program P we can define its semantic equivalence class $P^{[[]]}$ (the set of programs semantically equivalent to P) as $P^{[[]]} := \{P_i \in \mathbb{P} \mid [[P]] = [[P_i]]\}$. We can now use Rice's theorem to prove that this is not a recursive set and thus it describes an undecidable problem. The first step is to prove that $P^{[[]]}$ is not "trivial", which means proving $P^{[[]]} \neq \emptyset \wedge P^{[[]]} \neq \mathbb{P}$. We start with the most basic assumption that $P \in P^{[[]]}$, since a program has to belong to the set of programs semantically equivalent to itself. The second simple assumption is $\exists P_2 \in \mathbb{P}.[[P_2]] \neq [[P_2]]$, since there exists at least one program that is not semantically equivalent to P.

Proof. $P \in P^{[[]]} \rightarrow \exists P_1.P_1 \in P^{[[]]}$
$\exists P_2 \in \mathbb{P}.[[P_2]] \neq [[P]] \rightarrow \exists P_2.P_2 \in \mathbb{P} \setminus P^{[[]]}$
$\exists P_1.P_1 \in P^{[[]]} \wedge \exists P_2.P_2 \in \mathbb{P} \setminus P^{[[]]} \rightarrow P^{[[]]} \neq \emptyset \wedge P^{[[]]} \neq \mathbb{P}$ □

Now we have to show that $P^{[[]]}$ is an extensional property. For this proof the simplest assumption is that, if a program P_1 belongs to $P^{[[]]}$, then all programs semantically equivalent to P_1 must also belong to the same set. After all, $P^{[[]]}$ is a semantic equivalence class.

Proof. $\forall P_1 \in \mathbb{P}.P_1 \in P^{[[]]} \rightarrow \forall P_2 \in \mathbb{P}.[[[P_1]] = [[P_2]] \rightarrow P_2 \in P^{[[]]}]$
$\forall P_1, P_2 \in \mathbb{P}.\{P_1 \in P^{[[]]} \rightarrow [[[P_1]] = [[P_2]] \rightarrow P_2 \in P^{[[]]}]\}$
$P_1 \in P^{[[]]} \rightarrow [[[P_1]] = [[P_2]] \rightarrow P_2 \in P^{[[]]}]$
$P_1 \in P^{[[]]} \rightarrow [[[P_1]] \neq [[P_2]] \vee P_2 \in P^{[[]]}]$
$P_1 \notin P^{[[]]} \vee [[P_1]] \neq [[P_2]] \vee P_2 \in P^{[[]]}$
$P_1 \in P^{[[]]} \wedge [[P_1]] = [[P_2]] \rightarrow P_2 \in P^{[[]]}$ □

Since $P^{[[]]}$ is an extensional property and it is not "trivial", then according to Rice's theorem, it is not a recursive property and thus it represents an undecidable problem.

Now we can tie it neatly to our problem setting. Assume we have an algorithm $\mathcal{S}$ that in finite time can tell when two generic programs P_1 and P_2 are semantically equivalent (and when they are not). Then this algorithm can be applied to P and any other program P_i, meaning it would be possible to decide in finite time whether P_i belongs in $P^{[[]]}$ or not. This is absurd, as it would imply that $P^{[[]]}$ is a recursive set, thus such an algorithm does not exist. A key aspect of this result is that it does not matter how small or simple the initial P is, there is no algorithm that will always halt and correctly decide whether it is equivalent to any other program.

Now that the reader is hopefully convinced of the uncomputable nature of program semantic equivalence, we argue that all is not lost. In fact, the general problem is unsolvable, but it still needs to be tackled for all the reasons we outlined in Section 1.1.

As mentioned earlier, in order to tackle uncomputable problems we need to sacrifice something. We need to sacrifice generality, as in: our solution will not be applicable to all problems in this set. This means that what we end up losing in the end is precision, as any methodology we come up with in order to solve an uncomputable problem will not be accurate. It is a small price to pay, especially if we can somehow pinpoint exactly where our solution falls short and why.

This aspect will be expanded upon in the following subsections.

1.2.3 Obfuscations

As shown in previous sections, programs can be modified in (countably) infinite ways and still maintain their input-output semantics. More formally, we can define a semantics-preserving syntactic program transformation $\mathcal{T} : \mathbb{P} \rightarrow \mathbb{P}$ as:

$$\forall P \in \mathbb{P}.P \neq \mathcal{T}(P) \wedge [[P]] = [[\mathcal{T}(P)]] \tag{1.2}$$

The set of all possible semantics-preserving code transformation is $\mathbb{T}$. Many of these possible modifications entail adding blank statements or no-op statements to the code, as it is the simplest form of syntactic transformation that does not interfere with the semantics by design. Equivalently, we have shown how idempotent statements can be inserted anywhere in the program, for example `x := x + 1` followed immediately by `x := x - 1`. These types of transformations tend to be avoided especially in compiled languages, since most compilers are equipped with enough heuristics for optimizations that will remove such meaningless statements.

A more interesting class of program transformations considers only those that transform the original program into another "which is much more difficult [...] to understand" [32]. These are called "obfuscations".

It should be clear that this definition of obfuscation is rather vague. What does it mean for a program to be "much more difficult to understand"? By whom is it understood? This is generally dependent on the problem setting, but for this thesis, obfuscations will try to fool automated analyzers, meaning algorithms. The problem of fooling human analysts is an interesting one, but it can be argued that humans and computers have such a different skill set that many of the obfuscations used to fool humans would probably not work on automated approaches, and vice versa.

Take program P_1 as an example:

```
x = input()
if x > 0:
    if x % 2 == 0:
        return 0
    else:
        return 1
return x
```

It is a very simple program that, with any input greater than 0, will return either 0 or 1 (depending on whether the input is even or odd), while with negative integers as input, it will return the input unchanged. The program can be modified to a semantically equivalent version that is slightly harder to understand, $\mathcal{T}(P_1)$:

```
x = input()
while x > 1:
    x = x - 2
return x
```

The obfuscation applied above has a structural effect on the control flow of the program, as shown in Figure 1.2.

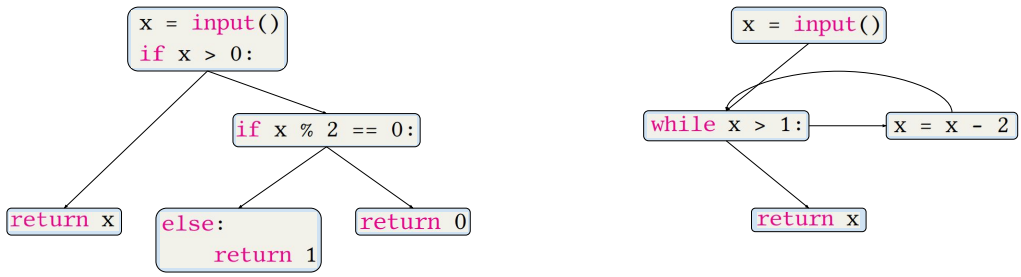

Figure 1.2: The CFG for P_1 (left) and $\mathcal{T}(P_1)$ (right)

In order to further confuse the syntax, from the modified program $\mathcal{T}(P_1)$ we can easily apply another obfuscation and obtain $\mathcal{T}_1(\mathcal{T}(P_1))$:

```
x = input()
z = -x
while z < -1:
   z = (z * 6)/3 - 4
   z = z / 2
return -z
```

The obfuscation shown above uses simple idempotent arithmetic operations in order to complicate the syntax of the programs. The CFG of the program is structurally identical to what it was before the transformation. All the code above is in Python so any skeptics can easily check that the three programs do indeed compute the same function. Of course, there are many more types of obfuscations other than the ones that affect the control flow or arithmetic operations, some of which are explored in Chapter 3.

It is interesting now to enquire whether it is always possible to modify a program such that the resulting version is much more difficult to understand. To this end, there is a very interesting impossibility result reached by Barak et al. [10] that seems to have put a definite pin on the possibility of general obfuscation. The truth is that the aforementioned study simply finds a class of functions that cannot be obfuscated with the "black-box" obfuscation requirement. Thus this result, while groundbreaking, does not apply to other models for obfuscations. Among these there is the “indistinguishability obfuscation” model [12], for which there are no impossibility results. Moreover, a candidate construction of an indistinguishability obfuscator for general circuits has been proposed recently by Garg et al. [59].

For the purpose of this study, we assume that obfuscation is possible for any program, since the “black-box” requirement is unnecessarily strict and there are no impossibility proofs for other models.

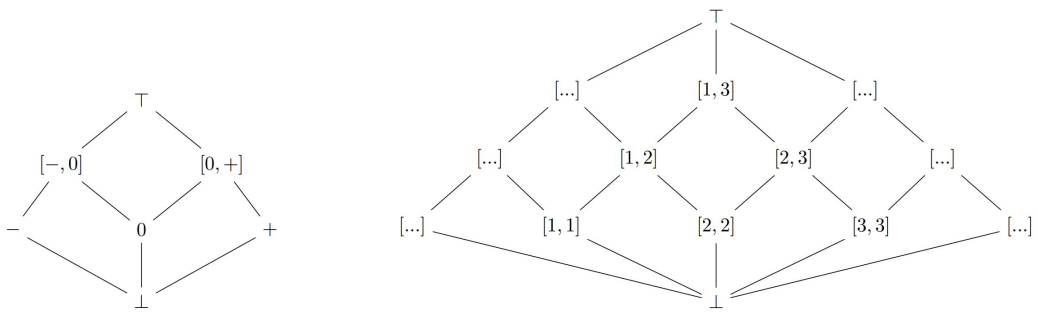

Figure 1.3: The *sign* abstract domain (left) and the *intervals* domain (right)

1.2.4 Static Analysis

Static analysis encompasses all types of analysis that operate on the source code, or on any static representation of the program, be it a disassembled version or the compiled binary. We mentioned in the previous section how, with static analysis, it is possible to "sacrifice" precision in order to regain computability. The key advantage of static analysis is that, unlike dynamic analysis, it allows one to reason about infinite computations. Another advantage of static analysis is that it can be built to be sound, meaning that it will either return results that are correct for the program under analysis or return "I do not know". Throughout this thesis we will not worry about the soundness of our analyses but it is an important aspect that is worth mentioning.

Abstract Interpretation

Abstract interpretation is a well known mathematical framework for static analysis developed in the 70's by Radhia and Patrick Cousot [37]. Intuitively, abstract interpretation allows to formalize most (if not all) static analyses by substituting the concrete domain on which the program is evaluated with an abstract domain. Then, the concrete semantics is replaced by an abstract semantics that correctly approximates the semantics of the program on the abstract domain. One of the advantages of abstract interpretation is that it allows to build static analyses that are sound by design.

In this thesis we use abstract interpretation only for examples in this chapter and in chapter 4, so we will not delve too deep into the formalisms.

The denotational semantics of a program is a function in the computational domain $\mathbb{D}$,

P_1 :

```
x = 10
while x > 1:
    x = x - 2
return x
```

P_2 :

```
x = 15
while x > 2:
    x = x - 3
return x
```

P_3 :

```
x = -9
while x < 0:
    x = x + 2
return x
```

P_4 :

```
x = -9
while x < 1:
    x = x + 2
return x
```

Figure 1.4: A motivating example

defined as $[[]] : \mathbb{P} \to (\mathbb{D} \to \mathbb{D})$. The semantics specifies how a program evolves from the starting state to the end state and is needed in order to know what the program computes. For simplicity, in our examples the domain of the programs is $\mathbb{N}$. In abstract interpretation, an abstraction $\mathcal{A}$ is a quadruple $\langle A, \leq, \alpha, \gamma \rangle$ where A is a set of abstract elements a that are in direct correspondence with elements c in a concrete set C via the monotone functions α and γ:

$$\begin{aligned} \alpha &: C \to A \\ \gamma &: A \to C \end{aligned} \tag{1.3}$$

Since the elements in the concrete domain and the abstract domain are a partially ordered set, this quadruple describes a Galois connection. The abstract elements of A are less or equally as precise as the elements in D. For example, since our computational domain is $\mathbb{N}$, some possible abstract domains are *sign* and *intervals*, pictured in Figure 1.3.

Given an abstraction $\mathcal{A} = \langle A, \leq, \gamma, \alpha \rangle$, the corresponding abstract semantics can be defined as:

$$[[]]^{\mathcal{A}} : \mathbb{P} \to (A \to A) \tag{1.4}$$

The abstract semantics of a program P represent a function between abstractions $a \in A$. Assuming a simple concrete semantics in which $\mathbb{D} = \mathbb{N}$, if $\mathcal{A}$ is an abstraction of the stores, then the resulting abstract semantics is a function between elements of *sign* or *intervals*:

$$[[P]]^{\mathcal{A}} : A \to A \tag{1.5}$$

The abstract semantics is the best approximation of the concrete semantics for a program P when we have that $[[P]]^{\mathcal{A}} = \alpha([[P]])$. This is the best case scenario and it corresponds to the concept of completeness in abstract interpretation [66]. Nonetheless, it is not a common scenario for most program analysis tasks.

We can use abstract interpretation to generate semantic abstractions of programs and show how imprecise these abstractions are compared to the concrete semantics.

Taking the program P_1 in Figure 1.4 as an example, the value of the variable x at the return point is 0 so the concrete semantics of the program is $\forall n \in \mathbb{N}.[\![P_1]\!](n) = 0$. Since the result does not depend on an input value we can write $[\![P_1]\!] = 0$. When the *intervals* abstraction is considered, then the abstract semantics of the program actually return $[0, 1]$. Let us unravel the abstract computations step by step in order to gauge how the process works, at least intuitively. We monitor the value of the variable x (which is the return variable) in the *intervals* domain, starting from its initial value of 10, which corresponds to $[10, 10]$ in the abstract domain of choice. The program points are characterized by the letter l and the number of the line in which the corresponding instruction appears.

$$l_1 : [10, 10] \rightarrow l_2 : [10, 10] \rightarrow l_3 : [8, 10] \rightarrow l_2 : [8, 10] \rightarrow l_3 : [6, 10] \rightarrow l_2 : [6, 10] \rightarrow \tag{1.6}$$

$$\rightarrow l_3 : [4, 10] \rightarrow l_2 : [4, 10] \rightarrow l_3 : [2, 10] \rightarrow l_2 : [2, 10] \rightarrow l_3 : [0, 10] \rightarrow l_2 : [0, 10] \rightarrow l_4 : [0, 1] \tag{1.7}$$

What happens in the first 3 lines of the program is relatively simple to explain, the abstraction keeps track of all possible values of x and thus only the lower bound is decreased. In order to exit the while loop, the exit condition has to be met ($x < 1$), thus the value at l_4 is obtained by solving $x \in [0, 10] \wedge x < 1$, which yields $x \in [0, 1]$. Clearly this abstraction is not ideal for P_1, since approximating the result of the concrete semantics yields a more precise result than the one just obtained. In other words, $\alpha([\![P_1]\!]) = [0, 0]$ thus $\alpha([\![P_1]\!]) \subset [\![P_1]\!]^{\mathcal{A}}$.

Moving to the program P_2 in Figure 1.4, its semantics is simply $[\![P_2]\!] = 0$, exactly like P_1 thus we have that the two programs are semantically equivalent ($[\![P_1]\!] = [\![P_2]\!]$). Following the same steps as above, the abstract semantics of P_2 computed on the *intervals* domain is $[\![P_2]\!]^{\mathcal{A}} = [0, 2]$. Not only is this result not precise, meaning that $\alpha([\![P_2]\!]) \subset [\![P_2]\!]^{\mathcal{A}}$, the abstraction for this program is even less precise than for P_1 ($[\![P_1]\!]^{\mathcal{A}} \subset [\![P_2]\!]^{\mathcal{A}}$). The programs P_1 and P_2 represent two semantically equivalent programs that would have evaded detection if the similarity function simply equated their abstract semantic representations. In other words, they represent a false negative in the program similarity problem.

On the other hand, program P_3 represents a program that is semantically divergent from P_1 and P_2, since $[\![P_3]\!] = 1$. To illustrate the imprecision of static analysis, P_3 has been crafted so that its semantics on the *intervals* domain is equivalent to the one for P_1: $[\![P_3]\!]^{\mathcal{A}} = [0, 1] = [\![P_1]\!]^{\mathcal{A}}$. Their equivalence in the abstract representation can be seen as a false positive for a program similarity analysis that considers these abstractions as program representations.

Finally, P_4 is a program that is semantically equivalent to P_3, thus semantically divergent from both P_1 and P_2. Applying the *intervals* abstraction once again we obtain $[\![P_4]\!]^{\mathcal{A}} = [1, 1]$.

The abstract semantics for this program is precise as $[[P_3]]^{\mathcal{A}} = \alpha([[P_3]])$. More importantly, comparing the abstractions finally leads to a true negative result for P_1 and P_4: $[[P_1]] \neq [[P_4]] \wedge [[P_1]]^{\mathcal{A}} \neq [[P_4]]^{\mathcal{A}}$.

Interestingly enough, it is possible to generate infinite functions semantically equivalent to P_1 that are never equivalent to each other in their abstract representations. The following program, P_n, is a generic template for these functions, which also preserves the number of executed while loops as an invariant.

```
x := 5 + (5 * n)
while (x > n):
    x := x - (n + 1)
```

Assuming n is an input variable, it is relatively easy to see that $\forall i \in \mathbb{N}.[[P_n]](i) = 0$. At the same time it is also true that $\forall i \in \mathbb{N}.[[P_n]]^{\mathcal{A}}(\alpha(i)) \subset [[P_n]]^{\mathcal{A}}(\alpha(i+1))$.

The goal of these examples is to show what kind of imprecision can be added by semantic abstractions, and consequently, static analysis as a whole. This is not done to discourage the use of static analysis or semantic approximations of programs but merely to give an intuitive vision on what the pitfalls of these approaches can be. It should be clear by now that comparing the syntax of programs to understand whether they are semantically similar is not the right way to go, and thus the most sensible choice is still to approximate the program semantics.

1.3 Malware Analysis

Hopefully the previous sections will have convincingly outlined why program similarity is an interesting, albeit hard, problem to tackle. This section inserts program similarity in the context of malware classification and better introduces the three scenarios mentioned earlier.

Malware Families

Knowing when to classify a malware sample into a specific family can be a hard problem. If it is syntactically equivalent to other samples already successfully classified in the same family, then the process is simplified. Recall that syntactic equivalence implies semantic equivalence and if two malware samples M_1 and M_2 are semantically equivalent ($[[M_1]] = [[M_2]]$), then they naturally belong to the same family. But two malware samples do not necessarily need to be semantically equivalent in order to belong to the same family. This is true especially with mobile malware.

In general, a malware sample is not simply a program that only affect a systems negatively. In order to spread and hide among “legitimate” programs, most malware samples masquerade

as benevolent programs and conceal their malicious nature as much as possible. This means that, given two malware samples M_1 and M_2 belonging to the same family, it is generally unwise to expect $[[M_1]] = [[M_2]]$. Of course, if that happens, then the two samples do indeed belong to the same family and they are either the exact same program, or a simple syntactic variant of one another.

A more complex situation is where two malware samples share only their malicious behaviors, while differing in their benevolent facades. In this case, we would need to isolate the malevolent parts and check their semantic similarity, as in $m_1 \subset M_1, m_2 \subset M_1.[[m_1]] = [[m_2]]$. Then M_1 and M_2 would belong to the same malware family, provided that m_1 and m_2 are in fact their malicious behaviours.

Knowing for certain that two programs M_1 and M_2 are semantically equivalent is hard enough (generally impossible, in fact), and having to account for the situation in which only a subset of their semantics to be equivalent is evidently harder. A possibly even harder problem is detecting clearly what constitutes a "malicious" behaviour.

Presenting solutions to these problems is out of scope for this section, but they need to be introduced in order to define the scenarios in which we position ourselves.

Three Scenarios

We mentioned that, not unlike other uncomputable problems, the problem of malware classification can be (partially) solved in many different ways.

In order to apply static analysis approaches we first need to possess a static representation of the programs, be it source code, assembly or compiled binary. If the source code of the malware samples is available, then the best choice might be to develop a program similarity tool that explicitly makes use of known program representations and their respective similarity measures. This of course is only possible with the presence of source code or disassembled code.

Earlier in this chapter, we mentioned how obtaining the source code of malware or correctly disassembling them is not always feasible, thus leading us to identify the three scenarios. These scenarios can be thought of as being defined by the inherent "openness" of the system under analysis. For example, we will see how Android is an open system that allows the easy recovery of the source code or a very close representation of it. Windows on the other hand, is much more closed, and recovering the source code from compiled programs is much harder. The openness of the systems describes the amount of information about the program that can be gathered for the analysis. The information can then be used in order to make informed decisions and possibly apply domain expertise. The three scenarios can be then qualified as:

- Scenario 1: open system and ideal scenario where all the information about the program is available. We can work directly with the source code.
- Scenario 2: semi-closed system where the source code is not available and the decompilation process can be problematic. We can only work with the compiled program.
- Scenario 3: closed system. The program is not available even in its compiled form but it can be executed. Dynamic analysis becomes the only choice.

The next three chapters will explore the main contributions of this thesis, all of them in the context of a different scenario.

1.3.1 First Scenario (Open System) - Static Analysis of the Source Code

For scenario 1 we focus on Android malware, since Android is a very open system that allows us to obtain a pretty faithful representation of the source code from almost any application. The openness of Android indeed simplifies the analysis process, but it also makes it much easier for malware developers to insert malicious behaviours into existing apps and then releasing them into the stores.

The goal of this work is to build a program similarity tool for Android programs (R.E.H.A.) that correctly groups malware samples with other samples that share the same malicious behaviours. In order to focus on the malicious behaviours we identify a relatively short list of Android APIs that can be used to negatively affect the users. Then, as a first step of the analysis, we filter out every method in the code that does not contain at least one call to these "risky" APIs.

After this initial screening, we generate the smali representation of Android programs and from this extract the control flow graph of every method as a program representation to be used in our program similarity algorithm. From the CFG, a new vectorial feature is generated in order to further approximate the program representation and allow some structural leeway in the similarity function.

Losing precision twice during this process, once by focusing on the CFG and then by approximating it, means that the precision of the tool suffers as it generates many false positives. On the same note, attackers can easily exploit the flaws in the abstractions by using targeted obfuscations. Thankfully, since we are in scenario 1 and we are generating our own representation and similarity measure, we have full domain knowledge and know exactly what program transformations can negatively affect our approach.

With this knowledge, we introduce a second feature with its own similarity measure which disregards control flow structure altogether. Using backward slices, we isolate instruction

sequences that are confined in single basic blocks, in a feature called "strand". With this we build a second tool, called StranDroid. Again, from every method in the dataset that has passed the initial risky APIs screening, we generate a set of strands. The similarity algorithm then matches each method of every program in the dataset against the others using the Jaccard Index as the similarity measure.

We test both tools against newly-collected ransomware and malware datasets and verify that the advantages of StranDroid are confined to its precision, while R.E.H.A. has the best performance and can analyze bigger datasets in less time.

Some of the main advantages of this approach are:

1. *interpretability* → the abstractions and the similarity functions can be built with full domain expertise, thus yielding results that are readily interpretable.

2. *precision-cost tradeoff* → it is possible to fine-tune the amount of precision needed for a specific analysis, thus tweaking the computational complexity of the algorithms.

There are also many flaws with this approach, among which:

1. *uncommon scenario* → abstractions are built from code, which means we assume complete access to the source code of malware. This is not a common scenario.

2. *static analysis weaknesses* → as we leverage abstractions, we open up holes in our approach for attackers to potentially exploit.

1.3.2 Second Scenario (Semi-closed System) - Learning on Binaries

Scenario 2 assumes a more closed system, where the source code of the malware is not available and their disassembly is not feasible. In this scenario, the focus is on classifying malware by their compiled binaries, mainly on the Windows platform.

The lack of source code makes it harder to use domain expertise in order to build our own program representations and similarity measure. One of the methods to overcome this limitation is to hand over these tasks to a learning algorithm, treating the binaries as sources of raw data to be fed to a model. At this point the information that is not obtained from domain expertise has to be supplemented by well structured datasets with a robust ground truth.

The first flaw that we can see in this approach is indeed the dataset generation process. Collecting malware in the wild often generates unbalanced datasets, meaning that some families tend to be more represented than others, sometimes to the detriment of the learning process. Another problem is often the lack of a properly sized dataset, especially with approaches that expect huge quantities of data, such as deep learning models. Smaller datasets often result in

over-fitting, a process in which the network learns the training dataset perfectly and does not generalize well to unseen data.

These problems are easily fixed in other learning contexts. For example, in image recognition and image classification, both unbalanced datasets and undersized datasets can be fixed with data augmentation techniques. These techniques usually entail processing the images with some transformation that changes the general look of the image but preserve their content (their semantics). Some examples of these transformations are rotation, zoom in and out, tilt, flip, etc.

It is clear that none of these transformations can be applied to binaries while maintaining their semantics. In fact, most of these would not return a working program to begin with.

We solve this problem by showing that it is possible to use semantics-preserving obfuscations on the programs of a dataset as a data augmentation technique. After generating a dataset of programs in 47 semantic equivalence classes, we train two deep networks on it, a convolutional network and a long short-term memory.

Another way to solve the problems generated by the lack of a proper dataset is to adopt transfer learning. This process entails training a deep network on a big dataset and then using the features learnt in order to classify a new smaller dataset. Lacking a big enough malware dataset we show that it is possible to generate a custom dataset with the data augmentation technique explored above and then train a deep network. The features learnt from this custom dataset can then be used to classify a new smaller malware dataset with transfer learning.

These approaches have a few advantages:

1. *hard to attack* → since the representations and the similarity function are learnt automatically, it is harder to find an attack vector.
2. *scalable* → after the initial training process, the classification of new malware is computationally inexpensive.

Possible flaws are:

1. *no interpretability* → letting the networks learn the features means that they often do not have an easily interpretable meaning.
2. *unknown cost-precision tradeoff* → with previous approaches we could tweak the precision of the analysis to gain computational advantages. With deep neural networks there is less control on this matter.
3. *imprecision* → this is still a static analysis approach, so a precise algorithm for malware classification cannot be found.

1.3.3 Third Scenario (Closed System) - Dynamic Analysis

Scenario 3 represents a worst case scenario of sorts. It is characterized by the complete lack of static information on the programs.

The only choice is then to execute the programs and gather what is possible from the execution traces. This process is called dynamic analysis and is often used to circumvent the lack of precision of static analysis. Using this type of analysis, the execution traces contain information that is not spurious and is not added by the approximations of static analysis.

Even so, dynamic analysis presents many flaws, not least of them the fact that it can only consider a finite number of finite traces. This means that it can only see a small subset of the actual behaviours. Dynamic analysis lacks the formal treatment that static analysis has in literature. While it is always possible to understand what effects obfuscations have on static analysis, it is not clear how to prove formally when an obfuscation works on dynamic analysis.

For these reasons in this scenario we do not work on malware per se, but rather we explore the theoretical hole that seems apparent in this field. In a departure from the other two main chapters of this thesis, we perform a purely theoretical investigation on dynamic analysis.

In order to prove the effect of some obfuscations on dynamic analysis, we build a mathematical framework with which we can formalize both the analysis and the obfuscations. We then show how to use the framework to prove the strength of certain obfuscations against dynamic analysis found in literature.

1.4 Contributions and Structure of the Thesis

Within this section we provide the structure of the thesis and detail which works have been published for each chapter.

1.4.1 Program Similarity for Android Malware

Chapter 2 explores the first scenario, where the system is open and we can apply static analysis solutions with full domain expertise to solve the problem of malware classification. Most of the content in this chapter has been published in two papers:

1. **GroupDroid: Automatically grouping mobile malware by extracting code similarities**[94]
   Niccolò Marastoni, Roberto Giacobazzi, Mila Dalla Preda
   Published in the Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop [**2017**]

Abstract: As shown in previous work, malware authors often reuse portions of code in the development of their samples. Especially in the mobile scenario, there exists a phenomena, called piggybacking, that describes the act of embedding malicious code inside benign apps. In this paper, we leverage such observations to analyze mobile malware by looking at its similarities. In practice, we propose a novel approach that identifies and extracts code similarities in mobile apps. Our approach is based on static analysis and works by computing the Control Flow Graph of each method and encoding it in a feature vector used to measure similarities. We implemented our approach in a tool, GroupDroid, able to group mobile apps together according to their code similarities. Armed with GroupDroid, we then analyzed modern mobile malware samples. Our experiments show that GroupDroid is able to correctly and accurately distinguish different malware variants, and to provide useful and detailed information about the similar portions of malicious code.

2. **Revealing Similarities in Android Malware by Dissecting their Methods**[109]
Michele Pasetto, Niccolò Marastoni, Mila Dalla Preda
IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) [**2020**]

   Abstract: One of the most challenging problems in the fight against Android malware is finding a way to classify them according to their behavior, in order to be able to utilize previously gathered knowledge in analysis and prevention.In this paper we introduce a novel technique that discovers similarities between Android malware samples by comparing fragments of executed traces (strands) generated from their most suspect methods. This way we can accurately pinpoint which (possibly) malicious behaviors are shared between these different samples, allowing for easier analysis and classification.We implement this approach in a tool, StrAndroid, that we evaluate on a few dataset of malware and ransomware samples, comparing its results to an existing similarity tool.

1.4.2 Deep Learning on Compiled Programs

Chapter 3 assumes that the malware samples in our possession are compiled binaries. Most of the content in this chapter has been published in two papers:

1. **A deep learning approach to program similarity**[95]
Niccolò Marastoni, Roberto Giacobazzi, Mila Dalla Preda Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis [**2018**]

   Abstract: In this work we tackle the problem of binary code similarity by using deep learning applied to binary code visualization techniques. Our idea is to represent binaries

as images and then to investigate whether it is possible to recognize similar binaries by applying deep learning algorithms for image classification. In particular, we apply the proposed deep learning framework to a dataset of binary code variants obtained through code obfuscation. These binary variants exhibit similar behaviours while being syntactically different. Our results show that the problem of binary code recognition is strictly separated from simple image recognition problems. Moreover, the analysis of the results of the experiments conducted in this work lead us to the identification of interesting research challenges. For example, in order to use image recognition approaches to recognize similar binary code samples it is important to further investigate how to build a suitable mapping from executables to images.

2. **Data augmentation and transfer learning to classify malware images in a deep learning context**[96]
Niccolò Marastoni, Roberto Giacobazzi, Mila Dalla Preda Journal of Computer Virology and Hacking Techniques, 1-19 [**2021**]

Abstract: In the past few years, malware classification techniques have shifted from shallow traditional machine learning models to deeper neural network architectures. The main benefit of some of these is the ability to work with raw data, guaranteed by their automatic feature extraction capabilities. This results in less technical expertise needed while building the models, thus less initial pre-processing resources. Nevertheless, such advantage comes with its drawbacks, since deep learning models require huge quantities of data in order to generate a model that generalizes well. The amount of data required to train a deep network without overfitting is often unobtainable for malware analysts. We take inspiration from image-based data augmentation techniques and apply a sequence of semantics-preserving syntactic code transformations (obfuscations) to a small dataset of programs to generate a larger dataset. We then design two learning models, a convolutional neural network and a bi-directional long short-term memory, and we train them on images extracted from compiled binaries of the newly generated dataset. Through transfer learning we then take the features learned from the obfuscated binaries and train the models against two state of the art malware datasets, each containing around 10 000 samples. Our models easily achieve up to 98.5% accuracy on the test set, which is on par or better than the present state of the art approaches, thus validating the approach.

1.4.3 A Formal Approach for Dynamic Analysis

Chapter 4 explores the dynamic analysis angle, proposing a new formal framework in order to evaluate obfuscations in this scenario. Most of the content in this chapter has been published in the following paper:

1. **Formal framework for reasoning about the precision of dynamic analysis**[45] *Mila Dalla Preda, Roberto Giacobazzi, Niccolò Marastoni* International Static Analysis Symposium [**2020**]

   Abstract: Dynamic program analysis is extremely successful both in code debugging and in malicious code attacks. Fuzzing, concolic, and monkey testing are instances of the more general problem of analysing programs by dynamically executing their code with selected inputs. While static program analysis has a beautiful and well established theoretical foundation in abstract interpretation, dynamic analysis still lacks such a foundation. In this paper, we introduce a formal model for understanding the notion of precision in dynamic program analysis. It is known that in sound-by-construction static program analysis the precision amounts to completeness. In dynamic analysis, which is inherently unsound, precision boils down to a notion of coverage of execution traces with respect to what the observer (attacker or debugger) can effectively observe about the computation. We introduce a topological characterisation of the notion of coverage relatively to a given (fixed) observation for dynamic program analysis and we show how this coverage can be changed by semantic preserving code transformations. Once again, as well as in the case of static program analysis and abstract interpretation, also for dynamic analysis we can morph the precision of the analysis by transforming the code. In this context, we validate our model on well established code obfuscation and watermarking techniques. We confirm the efficiency of existing methods for preventing control-flow-graph extraction and data exploit by dynamic analysis, including a validation of the potency of fully homomorphic data encodings in code obfuscation.

1.4.4 Conclusions

Chapter 5 contains summaries for all the preceding chapters, combined with the conclusions and possible future works that might stem from each of them.

2 Program Similarity for Android Malware

The goal of this chapter is to explore behavioral classification of malware samples in the context of scenario 1, that is, an almost completely open system.

Android is the perfect environment for such a study, as Android applications are packaged using the Android Package file format (.apk) and can be easily de-constructed back to their original source code. Of course, this is not perfect since exact decompilation is a generally impossible task as a lot of information is lost in the compilation process, either due to optimizations or simple heuristics such as variable renaming.

Nonetheless, the open nature of the system allows us to gauge domain expertise and thus craft custom code representations and similarity measures to group malware samples according to their behaviour. On the same note, thanks to the openness of the system, we can challenge the ground truth and design our tools with clustering (grouping) in mind.

The rest of the section outlines the problem posed by malware on Android devices, starting with the usual arguments on why it is a worthwhile endeavour to study this platform and its attack vectors. We then outline the ideas behind our approaches and specify the contributions of this chapter.

2.0.1 Why it is Important to Study Malware on Android

At the time of writing, Android is the most widespread mobile operating system in the world with an estimated 85% market share as of 2020 [73]. Combined with the fact that around 3.5 billion people nowadays own a smartphone [120], that results in close to 3 billion active Android users and many more Android devices. These numbers are helpful in showing why Android is a highly coveted attack vector for malware developers.

Even though the security model of Android is complex [54] and new versions of the system tend to keep up with new threats, it is rare for users to consistently have the new version installed on their devices [90]. Securing the app distribution system is also not always a sound solution for the security of the ecosystem, as users from all over the world regularly access third-party stores that are not known for their thorough app vetting process [146]. All of these factors combined result in an ecosystem that presents many potentially lucrative attack vectors.

Recent work on Android malware focuses on developing new intelligent and adaptive

methods for malware detection and classification [138, 153], generally by adopting machine learning models. Within these works, the models are presented with a specific view of the malware samples, usually features extracted by experts, and they learn a way to put each sample in the correct class. In other ways, what they learn is a measure of similarity between malware samples. These academic endeavors seldomly reflect in the actual usage by anti-malware vendors, where the main techniques used for the classification of malware is still signature-based [77] and thus easily circumvented by simple code modifications [46]. It is indeed necessary to find better ways to discover the similarities between malware samples.

A recent report published by MalwareBytes [93] shows that Android malware is getting "stealthier and more aggressive", which should result in research focused on more precision and accuracy. The approaches that we present in this paper is meant as a step towards applying more sophisticated methods to the analysis of Android malware, possibly leading to more precision in the analysis results.

In this chapter, we illustrate two novel techniques to identify code similarities among Android apps, with the intent of recognizing and extracting code that produces similar behaviors. Our approaches are based on static analysis and work at the method level.

Reverse Engineering Helper for Android [R.E.H.A.]

As an initial step, we extract the Control Flow Graph (CFG) of every method and encode each CFG in a vector of features that we then use to measure the similarity. Chen et al. introduced the concept of 3D-CFG and its relative centroid to build a scalable method for app clone detection in [27], focusing on applications that share most of the code, or at least the core functionality. One of their main motivations for working on clone detection is the fact that malware prefers to use app clones as "carriers" for propagation, while our focus is almost the opposite. We look for similarities in potentially small portions of malicious code, those that are deemed interesting for our particular analysis.

We implemented our approach in a tool called Reverse Engineering Helper for Android, or R.E.H.A. for short. The system is able to classify Android malware on the basis of its code similarities and to extract the portions of similar code, providing useful and detailed feedback of the classification and helping in the reverse engineering process. We evaluated R.E.H.A. against 4,211 malicious Android apps, showing that it is able to successfully classify different families. Our experiments showed that R.E.H.A. is not only able to group together malware samples of the same family, but it can also distinguish slightly different variants by identifying differences in the similarities.

In our paper "GroupDroid: Automatically grouping mobile malware by extracting code similarities" [94] we referred to the tool as GroupDroid, following the common trend of using

the post-fix (sometimes pre-fix) "droid" in the tool name [29, 157, 2, 150, 8, 53, 5]. GroupDroid has since become a subsystem of the bigger R.E.H.A., namely comprising only the clustering part.

StranDroid

At the end of the chapter we will discuss some of the limitations to this approach and describe a second system, StranDroid, which sacrifices some computational power in order to gain precision. To develop this system we take direct inspiration from a brilliant work by Yaniv David, Nimrod Patush and Eran Yahav [48]. In that work, the authors apply similarity by composition to binaries in order to discover their similarity even when compiled with different toolchains and optimizations. The idea behind similarity by composition comes from a work in the field of image recognition [20] and leverages the fact that two images are similar if they are made of similar components. Thus finding the similarities between the components is a key part of establishing whether the two original images are also similar. The goal of [48] was to find pieces of code that were semantically similar to a query fragment mainly to allow the search of bugs or known exploits in a benign scenario where binary samples are generated with different compilers or different versions of the same compiler. This use case is, of course, different from malware analysis where we need to take into consideration the use of malicious code modifications explicitly designed to prevent analysis. For this reason, the methodology differs in some key areas that we will expand upon in Section 2.6.

We believe the similarity by composition approach works for Android malware because malicious behaviors are often just different combinations of the same few base components. For example, there are approaches that aim at finding Android clones that are similar wrt their behaviors [100] . For this purpose, Object Based Actions (OBAs) are defined as all the API calls that can be grouped into a common semantic group: i.e. HTTP-based actions, TelephonyManager-based actions, SMSManager-based actions, etc. A malicious behavior can then be summarized by the combinations of these OBAs into common patterns. For example, if the goal of the malware is to steal the user data and send it to a remote server then the corresponding behavior can be described with TelephonyManager-based + HTTP-based actions, while if the data was sent through SMS texts then we would have TelephonyManager-based + SMSManager-based actions. These are malicious behaviors that can help classify the malware into different classes and they are all combinations of smaller components.

With the similarity by composition approach of [48], we can give a more precise characterization of the behaviors hidden in Android methods.

The contributions of this chapter are:

- Implementation of R.E.H.A., a static analysis tool for behavioral similarity of Android apps based on structural similarity
- Implementation of StranDroid, another tool for Android app similarity based on similarity by composition
- Evaluation of both tools against the GENOME dataset and two custom datasets with malware and ransomware

The rest of the chapter is thus structured:

- Section 2.1 will give a thorough background on some aspects of static analysis, plus a primer on the Android system
- Some related works are presented in Section 2.8 while the main motivation for the study can be found in Section 2.2
- Our static analysis-based approach to identify code that produce similar behaviors is discussed in section 2.3
- R.E.H.A., our tool that finds similarities in malicious Android apps and classify them accordingly, is described in section 2.4
- Section 2.5 contains the evaluation of R.E.H.A. on a dataset of 4,211 Android malware samples.
- In Section 2.6 we introduce StranDroid and explain its significance in the scope of this work
- StranDroid is evaluated in Section 2.7 in order to gauge its effectiveness, especially when compared to R.E.H.A.

2.1 Background

In this section we will give a brief rundown of the main concepts needed to understand our methodology and the problem it sets out to solve. For readers familiar with the static analysis of Android malware we suggest skipping to Section 2.3.

2.1.1 Android Environment

Android is an open source operating system meant for mobile devices that adopts a very strict security model [54]. Every app installed in an Android device is packaged in an Android Package (APK) file, Android's standard packaging file format, and must conform to the model in order to interact with the most vulnerable parts of the system, and this is enforced through the use of *permissions* [57] and specific system *APIs*. Throughout the thesis we will use the terms *app* and *APK* interchangeably.

A rough structure of the Android Environment can be seen in Figure 2.1.

APIs and Permissions

Android offers a comprehensive list of APIs to its developers in order to better control the behavior of apps wrt the system itself. We analyzed a number of Android malware samples and created a list of APIs that can be used to execute malicious attacks on the device, which we call *risky APIs*. These are functions that range from network APIs, necessary to send packets through network protocols, to device-specific APIs, which are used to access private information about the user of the device. This is a small sample of our *risky APIs* list:

```
android.telephony.TelephonyManager
android.provider.Contacts
java.net.ServerSocket
org.apache.http.impl.DefaultHttpClient
```

Other works share our view that some APIs have more weight than others, for example, [150] calls them "sensitive" APIs and uses them as meaningful features to automatically characterize Android malware.

The usage of system APIs in Android is regulated by requiring the use of each API to be accompanied by a request for permission to the user. These permissions are stored in an XML document (the *manifest*) at the root of the application and are often used in malware analysis to spot which APKs could contain malicious behaviors [86, 2, 134]. Naturally, applications that do not request permission to use any *risky API* will not be able to affect the behavior of the device nor infringe on the privacy of the user in any way. It is hard to draw conclusions from the sole inclusion of permissions in the *manifest*, since app developers tend to be overzealous and request more permissions than necessary [57].

For this reason we focus on the API calls encountered in the code without checking the permissions, as we believe this gives us an edge when it comes to the precision of the analysis.

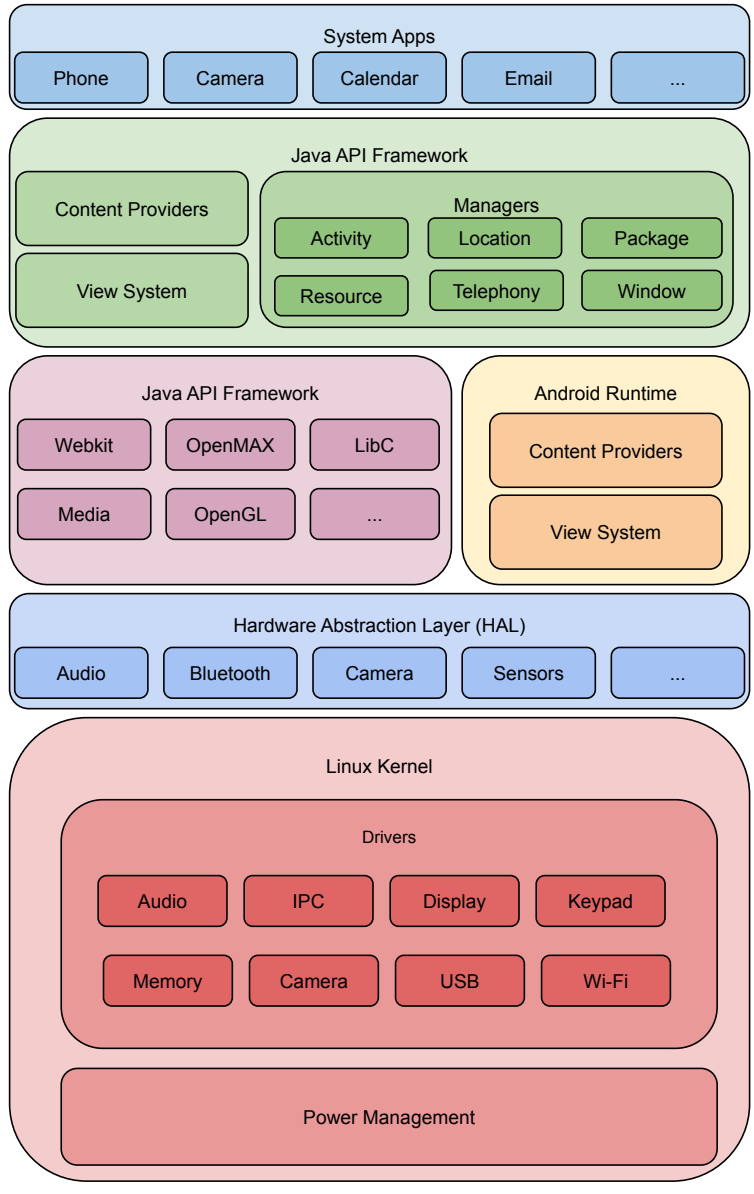

Figure 2.1: Structure of the Android environment

Smali

Before being run by Dalvik, the JVM implementation of Android, every app is compiled into *dex* bytecode.

```
01000000 90000000 00000000 00000000 02000000 9C000000
01000000 AC000000 14010000 CC000000 E4000000 EC000000
07010000 2C010000 2F010000 01000000 02000000 03000000
03000000 02000000 00000000 00000000 00000000 01000000
00000000 01000000 01000000 00000000 00000000 FFFFFFFF
00000000 57010000 00000000 01000100 01000000 00000000
04000000 70100000 00000E00 063C696E 69743E00 194C616E
64726F69 642F6170 702F4170 706C6963 6174696F 6E3B0023
4C636F6D 2F627567 736E6167 2F646578 6578616D 706C652F
42756773 6E616741 70703B00 01560026 7E7E4438 7B226D69
6E2D6170 69223A32 362C2276 65727369 6F6E223A 2276302E
312E3134 227D0000 00010001 818004CC 01000000 0A000000
00000000 01000000 00000000 01000000 05000000 70000000
00200000 01000000 57010000 00100000 01000000 64010000
```

This format is not very user-friendly, and for this reason most static analysis on Android code is done on the *smali* format. As an example we show a simplified `onReceive()` method found in a ransomware sample, generating the following smali method m_1:

```
    const-string v7, "PrintStream"
    const-string v9, "\n"
    invoke-interface {v8}, Landroid/text/Editable;->toString()Ljava/lang/String;
    move-result-object v8
    invoke-virtual {v8, v9}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
    move-result v8
    if-eqz v8, :cond_0
    iget-object v8, v8, Lcom/sssp/s$100000001;->this$0:Lcom/sssp/s;
    invoke-virtual {v8}, Lcom/sssp/s;->stopSelf()V
    :cond_0
    invoke-virtual {v7}, Lcom/sssp/s;->toPrint()V
    return-void
```

Listing 2.1: sample method in smali

R.E.H.A. uses *APKTool* [143] in order to translate the native bytecode into smali files, which will then be parsed and analyzed by our tool.

A detailed description of the parsing process and our analysis methodology is in Section 2.3.

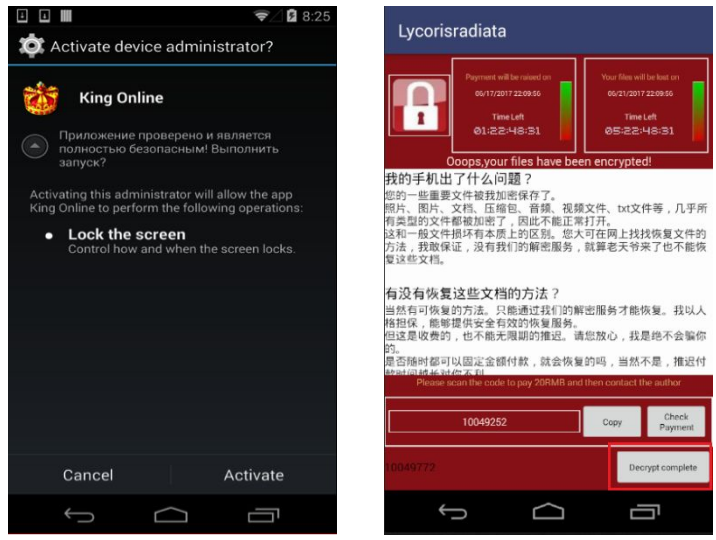

Figure 2.2: Two malware samples

2.1.2 Android Malware

There are many types of Android malware found in the various markets, and they can be grouped by behavior [80] as:

- Spyware, which is malware that allows an external entity to acquire private information about the unsuspecting user
- Ransomware, encrypting the private data of the user and requiring a payment (ransom) to decrypt it
- Adware, showing unwanted ads to the user

Of course many other types of malware exist, but in this chapter we target a small dataset of ransomware samples in order to gauge the effectiveness of R.E.H.A..

In Figure 2.2 we show the screenshots of two popular malware samples.

2.1.3 Static Analysis

Program analysis is mainly divided into static and dynamic analysis, where the former studies the properties of programs extracted from the code itself, while the latter executes the program

and analyzes the execution traces. The approach described in this chapter is static, as we analyze the smali code obtained from the APKs. We now give a brief description of the basic components of static analysis that are required to understand the inner workings of R.E.H.A..

Basic Blocks

Basic blocks are fragments of code uninterrupted by control flow instructions. They usually are represented as nodes in the control flow graph.

As an example, we take the previous smali code example in Listing 2.1. From this method we can extract 3 basic blocks:

1) lines 1 to 6 belong to the first basic block, b_1, then the instruction on line 6 generates a split in the control flow of the method:

```
1     const-string v7, "PrintStream"
2     const-string v9, "\n"
3     invoke-interface {v8}, Landroid/text/Editable;->toString()Ljava/lang/String;
4     move-result-object v8
5     invoke-virtual {v8, v9}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
6     move-result v8
7     if-eqz v8, :cond_0
```

Listing 2.2: first basic block

2) the second basic block, b_2 is formed by lines 7 and 8, representing the "else" branch of the conditional:

```
1     iget-object v8, v8, Lcom/sssp/s$100000001;->this$0:Lcom/sssp/s;
2     invoke-virtual {v8}, Lcom/sssp/s;->stopSelf()V
```

Listing 2.3: second basic block

3) finally lines 9 and 10 form the third and last basic block in this simple method, b_3:

```
1     :cond_0
2     invoke-virtual {v7}, Lcom/sssp/s;->toPrint()V
3     return-void
```

Listing 2.4: third basic block

Control Flow Graph

The control flow graph is generated by connecting the basic blocks to each other according to their jump instructions. Following the previous example with method m_1, we can connect the basic blocks extracted above in order to form the CFG of m_1.

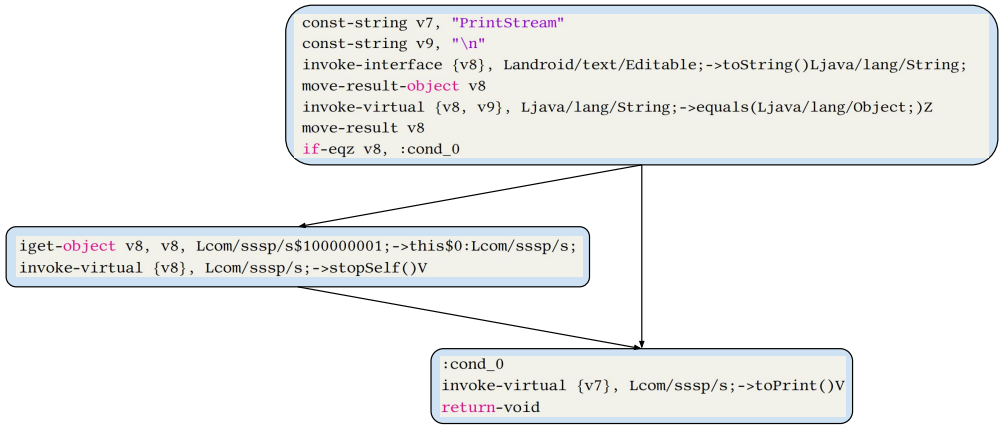

Figure 2.3: Example CFG from code sample in Listing 2.1

Clearly, the graph will start with b_1 as it is the beginning of the method, then b_1 will have two outgoing edges connecting it to b_2 and b_3. If the variable $v8$ is equal to zero, control will flow into b_3 as it is the target of the jump, while it will flow into b_2 if the condition is not met. One final edge has to be added between b_2 and b_3.

The CFG we just described can be seen in Figure 2.3. For the rest of the chapter, the CFGs pictured will not contain the smali code in the basic blocks as that can easily lead to unnecessarily big graphs.

Program Slicing

Program slicing is a static analysis procedure that allows the isolation of code fragments (slices) containing only instructions that are directly related to the slicing criterion through data flow and control flow [141]. The slicing criterion can be a variable in a specific location, a list of variables or an instruction that contains at least one variable.

There are two types of program slicing: forward slicing and backward slicing. As suggested by their names, they differ only in the direction that the slicing takes. In this chapter, we will focus on backward slicing as our preferred technique to extract strands from the APKs, meaning that we identify a program location of interest (slicing criterion) and perform a backward scan of the code, saving a list of instructions that are correlated to at least one of the variables in the slicing criterion.

For example, the result of backward slicing on the sample code in Listing 2.1 starting from

the variable $v8$ on its last appearance on line 8 is simply all the code from line 2 to 8. The reason why the code on line 1 is ignored is that the variable $v7$ does not interact with any computation that reads or writes on $v8$, which is our slicing criterion.

Strands

The concept of strand was first introduced in [48] (section 3.2) and simply defines a basic block-level program slice. Isolating slices in a basic block means that all the variables therein contained are connected with only data flow (as basic blocks do not contain any control flow by definition). This allows for a less precise representation of the code itself but a more fine-grained kind of analysis, where it is easier to discard unimportant elements. Section 2.3 goes more in depth about our specific strands implementation, while two examples of strands can be seen in Figure 2.16.

For completion we show the strands of the code in Listing 2.1. As before, the slicing criterion is the variable $v8$ on line 8. The first strand is:

```
1     const-string v9, "\n"
2     invoke-interface {v8}, Landroid/text/Editable;->toString()Ljava/lang/String;
3     move-result-object v8
4     invoke-virtual {v8, v9}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z
5     move-result v8
6     if-eqz v8, :cond_0
```

Listing 2.5: first slice

While this is the second strand:

```
1     iget-object v8, v8, Lcom/sssp/s$100000001;->this$0:Lcom/sssp/s;
2     invoke-virtual {v8}, Lcom/sssp/s;->stopSelf()V
```

Listing 2.6: second slice

From this example it is clear how the strands represent a refinement of both slicing and the basic blocks subdivision, allowing for finer control on what is shown to the analysis.

2.2 Motivation

Malware lives in a complex ecosystem that, similarly to an industrial environment, includes malware developers, managers, maintenance, and business strategies. This ecosystem is, of course, stimulated by the financial incentives that revolve around it. Common trends in the mobile scenario include: stealing and selling user information, stealing user credentials,

premium-rate calls, SMS spam, ransoms, advertising click frauds, and in-app billing frauds. Armin [6] studied the mobile underground market finding an alive and thriving ecosystem that benefits from the existence of an established modus operandi for desktop malware, which is well-structured and successful. Such a market is based on a crime-as-a-service model, in which resources, such as customizable malware, are sold and rented online. For instance, a Trojan called "Exo Android Bot" was heavily advertised in forums in 2016. For $400 per week or $3,000 per year, the author promised Android malware that could intercept SMS, use screen overlays, and had 24/7 support [144]. Unfortunately, sometimes malicious apps manage to evade detection and appear on official stores. For example, in February 2017, an `Android.Fakebank.B` variant masked as a weather app called "Good Weather" was published on the official Google Play Store and was downloaded by approximately 5,000 users [144].

One of the distinctive aspects of malware, especially on mobile, is its evident need to fit in among legitimate apps, to entice users and enable faster spread. Since its appearance needs to resemble goodware, a large part of the app is dedicated to behaviors that are not malicious at all, and often the benevolent part of the app varies among samples of the same malware family. A malware family is thus defined by the only component that is maintained constant among every sample: the malicious payload (Figure 2.4). This phenomena, known as *piggybacking*, in which cybercriminals embed malicious code into benign apps, has been observed and studied by researchers in previous works [155, 154, 87]. Moreover, as we previously described, malware authors re-use (part of) their malicious code across different malware versions and variants, which are constantly released on the underground market.

On the basis of such observations, if we find a good and efficient way to analyze code similarities between many apps, we can potentially isolate the malicious behaviors shared among the different malware samples, and classify apps according to these similarities.

Previous works proposed approaches to classifying apps by looking at their similarities [50, 152, 70]. However all these approaches mainly provide black-box tools, which do not generate detailed information about the behaviors the apps share. Instead, we aim at proposing a novel approach that allows to obtain practical and detailed feedback of the classification.

2.3 Methodology

R.E.H.A. at its core takes as input smali files from Android apps and filters the individual methods according to some fixed thresholds that allow us to be selective in regards to the behaviours we want to observe. Then the code is parsed to extract some static features, 3D-CFG centroids and API vectors, which will then be used to compare apps at the method level, to check for similarities. Once R.E.H.A. completes all the comparisons, it then groups the apps

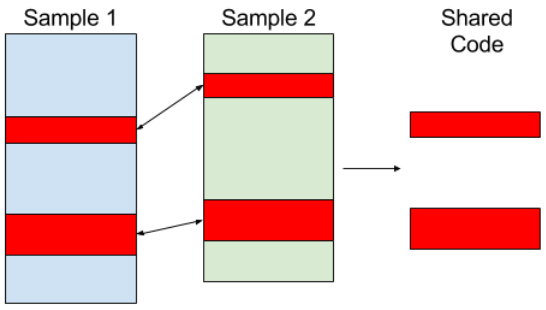

Figure 2.4: Isolating Malicious Behaviors.

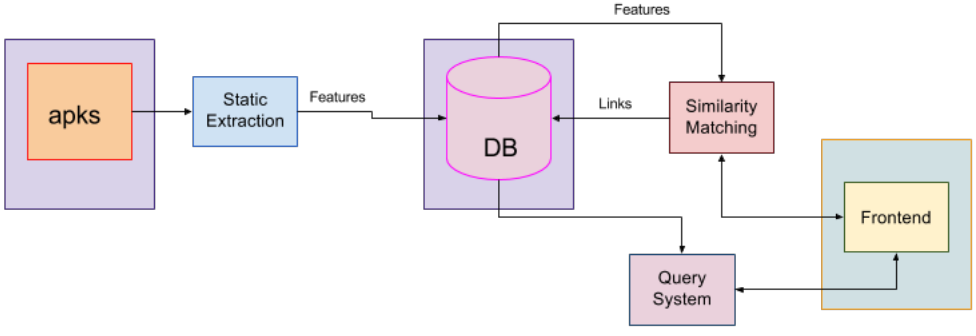

Figure 2.5: Workflow of R.E.H.A..

together according to how much code they share. In the next sections, we describe the phases of the approach of R.E.H.A., following the workflow depicted in Figure 2.5.

2.3.1 Filter

As a first step, apps are unpacked using apktool and the smali files are parsed to extract methods. The filtering phase considers some thresholds that are variable and can be tinkered with in the "settings" section. The first one is the *min_method_size*, which counts the minimum number of instructions in a method for it to be extracted. This tells the parser to ignore those methods that do not contain enough statements. The filter can be adjusted at each analysis, but it is

usually set at 6 to weed out some methods that are prevalent in every Android app, mainly "init" methods. As these methods usually just initiate a couple of variables and invoke one additional method, they do not contain any pertinent control flow information. If not filtered out, they become the main source for false positives and usually make the results of our analysis less interesting.

The second filter checks if a method invokes any of the "Risky APIs". If it does not, we do not consider it since it cannot possibly have any interesting behavior. "Risky APIs" is just a collection of all the APIs offered to Android developers to interact with the phone, and they range wildly between APIs that allow apps to write SMS and others that grant access to the phone's filesystem. With the way the Android OS is structured, apps just do not have any way to do anything harmful without using these APIs. We collected the APIs from various sources on the internet.

2.3.2 Feature Extraction

We consider similarity at the method level, so we can encode every method and then compare it to others. Since our similarity computation needs to be both fast and resilient to code transformations (to a certain degree), we decided to first consider the structure of the code. We extract the CFG of every method and encode each CFG in a vector of features that we use for the final similarity measure.

The first 4 features are the 3D-CFG's weighted centroid of the method, as first defined in [27]. The idea behind 3D-CFG centroids is borrowed directly from physics, to be more specific it is a reinterpretation of the "center of mass" of an ensemble of rigid bodies.

3D-CFG. Each method in a sample app gets transformed into its CFG, which represents the rigid structure, and every node (basic block) in the CFG is treated like an object with mass, connected to the other objects via weightless sticks (the edges of the CFG). At this point an "end" node will be added, and every basic block with a return statement will naturally flow into it, providing a single exit node.

Before assigning any mass to the nodes of the CFG, this needs to be transformed even further to make it fit into a 3D space. First, each node needs to have $< x, y, z, >$ coordinates, where:

- x = sequence number
  The choice of the sequence number depends on the order in which every basic block in the CFG will be executed. This is not particularly easy to judge using static analysis, but the goal of the 3D-CFG conversion is to provide a 1-to-1 conversion between code and

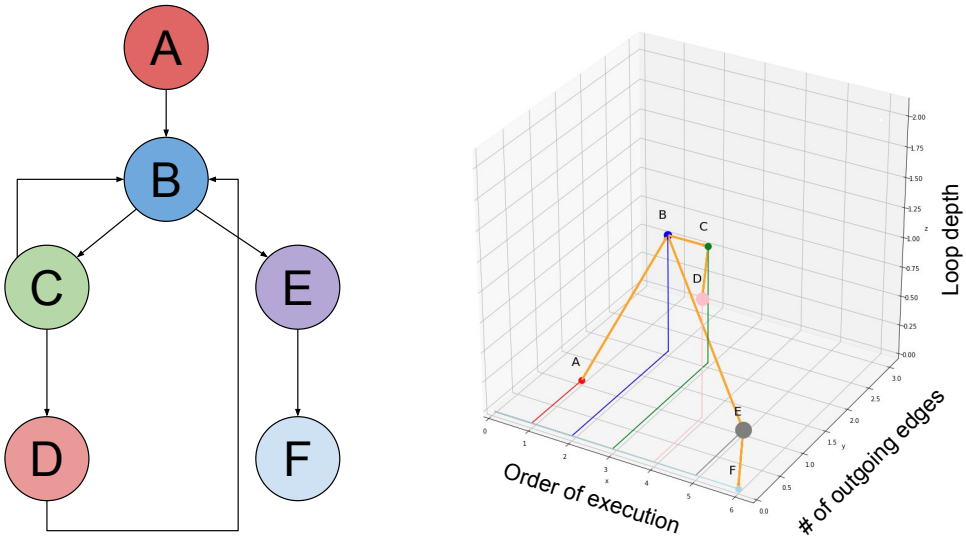

Figure 2.6: a CFG and its corresponding 3D-CFG

3D-CFG, thus it can rely on simple heuristics that will assure the same conversion every time copies of the same method will be fed to the algorithm.

- y = number of outgoing edges

- z = loop depth
  The nesting level of the basic block.

In Figure 2.6 we show an example of a CFG (on the left) and its corresponding 3D-CFG (on the right).

3D-CFG centroids. Once the method is successfully converted into its 3D-CFG representation, we can calculate its centroids. The weight w_i of each node is given by the number of statements in it.

A centroid is a vector $< c_x, c_y, c_z, \overline{w} >$ where:

$$c_x = \frac{\sum_{e(p,q) \in 3D-CFG} (\overline{w}_p x_p + \overline{w}_q x_q)}{\overline{w}} \tag{2.1}$$

$$c_y = \frac{\sum_{e(p,q)\in 3D-CFG} (\overline{w_p} y_p + \overline{w_q} y_q)}{\overline{w}} \tag{2.2}$$

$$c_z = \frac{\sum_{e(p,q)\in 3D-CFG} (\overline{w_p} z_p + \overline{w_q} z_q)}{\overline{w}} \tag{2.3}$$

$$\overline{w} = \sum_{e(p,q)\in 3D-CFG} (\overline{w_p} + \overline{w_q}) \tag{2.4}$$

Modified Centroid. Because of the nature of methods in Android apps and their reliance on invocations of the framework APIs, a second type of centroid is introduced, where the weights of each node are the sum of the number of statements and the number of invocations. This allows for better distinction between possibly cloned methods, as it enhances the underlying differences, and does not increase the complexity of the calculation (both centroids can be calculated at the same time with no added overhead).

Given the number of invocations as n_i, the new weighted centroid can be defined as a vector $< c_x, c_y, c_z, \overline{w}' >$ where $\overline{w}' = \overline{w} + n_i$.

Centroid Difference Degree. Once the methods have been reduced to simple 3D vectors (technically 4D, as we have the weight in addition to the 3 dimensions), we can define a simple distance measure between the centroids.

$$CDD(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1_x} - c_{2_x}|}{c_{1_x} + c_{2_x}}, \frac{|c_{1_y} - c_{2_y}|}{c_{1_y} + c_{2_y}}, \frac{|c_{1_z} - c_{2_z}|}{c_{1_z} + c_{2_z}}, \frac{|\overline{w_1} - \overline{w_2}|}{\overline{w_1} + \overline{w_2}}) \tag{2.5}$$

So the first 4 features are the modified centroid, as we experimented a bit with it and found that it has a way better performance than the normal one or a combination of the two.

The 3D-CFG centroid is a structural feature, and not a very precise one, so it does lose much of the information contained in the method. This, coupled with the tendency of Android method to be rather simple, gives rise to a lot of false positives. In the original paper, the authors added a statement type vector, but after extensive testing with our datasets it became clear that it did not solve most of the false positives, if any.

Our solution is to look at the APIs that are called during the method execution: if two pieces of code are similar, they need to exhibit some of the same behaviors. To check this, we extract the API invocations during the first pass of the smali code, annotating every method with a 5th feature.

API Vectors. This 5th feature in our vector is a binary encoding of the APIs with a value between 0 and $(2^{23} - 1)$. We isolated 23 "risky" APIs mostly from literature and our own experience and saved them in a file, then every time a method invokes an API in the list, the 5th feature (which is initialized to 0) will be added to 2^a (where a is the index of the API). So an invocation to the first API will add 1 (2^0), to the 4th will add 8 (2^3) and so on. This is done mostly to provide a new feature that does not slow down the extraction process any further and that still allows us to check in O(1) the new property.

During the extraction of the 5th feature we also save a vector of $len(risky\_APIs)$ where we store at each index how many times the given API is invoked in the method's body. This is our last group of features, at this time it's a vector of 23 elements with each element being an integer in the range 0+. This will be used in the last step of our similarity function.

2.3.3 Code Similarity

This is the algorithm for our code similarity measure, given 2 methods m_1 and m_2:
$s(m_1, m_2) = CDD(m_1.centroid, m_2.centroid) < cThreshold$
$\wedge\, BVE(m_1.apiBool, m_2.apiBool)$
$\wedge\, VDD(m_1.apiVector, m_2.apiVector) < vThreshold$

Where CDD is the Centroid Difference Degree:

$$CDD(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1_x} - c_{2_x}|}{c_{1_x} + c_{2_x}}, \frac{|c_{1_y} - c_{2_y}|}{c_{1_y} + c_{2_y}}, \frac{|c_{1_z} - c_{2_z}|}{c_{1_z} + c_{2_z}}, \frac{|\overline{w_1} - \overline{w_2}|}{\overline{w_1} + \overline{w_2}}) \tag{2.6}$$

This is a very fast operation that lets us filter methods that share at least some common structure. It returns a weighted distance value between 0 and 1.0, where 0 means complete similarity and 1.0 no similarity at all. The threshold for the final similarity measure can be changed at will, but the recommended setting is 0.4 to allow for structural dissimilarities introduced by code transformations. The next steps of the algorithm will catch any discrepancies produced by this lax approach.

BVE is the Boolean Vector Evaluation:

$$BVE(bv_1, bv_2) = bv_1 \& bv_2 \tag{2.7}$$

This is an $O(1)$ function, incredibly simple and designed to act as a rough filter to avoid analyzing methods that do not share any API invocations. As previously explained, our 5th feature is a number between 0 and $2^{23} - 1$ that is mined without any additional overhead during the parsing of the method and encodes succinctly which APIs are called in the body of the method.

Consider this practical example: methods m_1 and m_2 passed the CDD test, which means that either their structure is fairly similar or that the threshold was set too high. Method m_1 calls 2 APIs, which are located at positions 2 and 7 in the risky APIs vector, so its binary feature is the number 10000100 ($1 * 2^7 + 1 * 2^2 = 132$), while method m_2 calls three APIs that can be found at positions 3, 5 and 6, producing the binary feature 1101000 ($1 * 2^6 + 1 * 2^5 + 1 * 2^3 = 104$). At this point the BVE function simply applies a bitwise AND operation to the binary features and discovers that they do not share any API calls (132&104 = 0), meaning that the structural similarity resulted in a false positive. Hence, we can declare the methods to be not similar immediately without further computations.

If the BVE function returns a TRUE value, then the following function is applied to the remaining features. The VDD is the API Vector Distance Degree:
$av_1 = [v_{1,1}...v_{1,23}]$
$av_2 = [v_{2,1}...v_{2,23}]$

$$VVD(av_1, av_2) = max\left\{\frac{|v_{1,i} - v_{2,i}|}{|v_{1,i} + v_{2,i}|} | i \in [0, 22]\right\} \tag{2.8}$$

This is very similar to the CDD function and again outputs a value between 0 and 1.0, where 0 is an exact match between the API vectors, while 1.0 this time means that at least one of the elements in one vector did not have a match in the other. We use this function to allow for future relaxing of the API vector threshold, but for now it's set at a firm 0 (indicating we want an exact match all the time).

If all 3 functions return True then the two methods are deemed similar. Figure 2.7 depicts the workflow of the process, with every extracted feature associated with its similarity measure.

2.3.4 App Similarity

The app similarity score is calculated as the ratio between the number of shared methods and the number of methods in the first app. The similarity score is asymmetrical to account for the difference in size between different apps, if an app shares 20% of its code with a second app, the second app does not necessarily share 20% of its code with the first app.

$$score(app_1, app_2) = \frac{|\{(m_1, m_2) | m_1 == m_2 \wedge m_1 \in app_1 \wedge m_2 \in app_2\}|}{|\{m_1 | m_1 \in app_1\}|} \tag{2.9}$$

Two apps are considered not similar at all when their score is exactly 0, which means that they share no methods. Two apps are considered equal when their score is 1, which only happens when every method from the first app has a match in the other app.

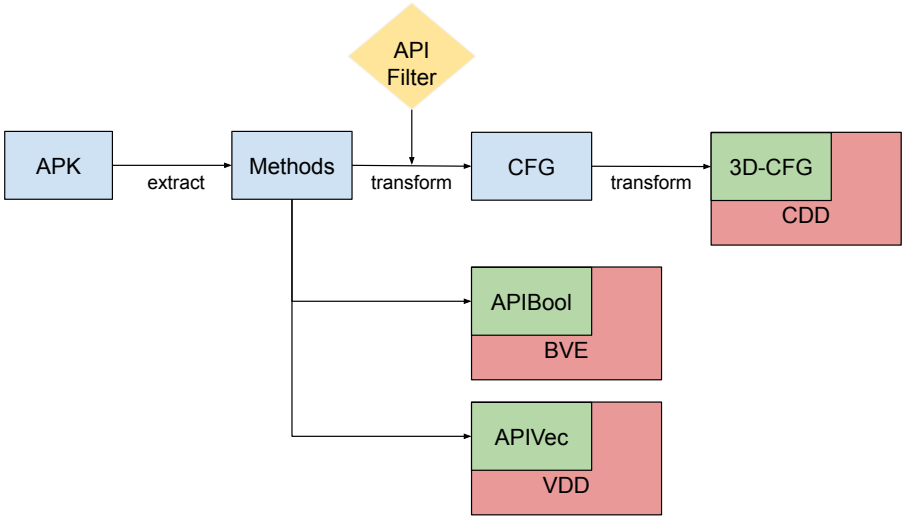

Figure 2.7: Workflow of the feature extraction phase, with each feature associated with its similarity function

2.3.5 Grouping

The grouping phase of R.E.H.A. consists of a general clustering algorithm that uses the app similarity score as a distance measure. It starts by creating a cluster for the first analyzed app, then iterating over all other analyzed apps and adding to the cluster all apps with a good similarity score. To account for the asymmetry of the score, we always check for the best one of the pair. The similarity score threshold is one of the parameters that we can play with and can be set to 1.0 if we only want to consider groups of apps that share 100% of the code we care about. This particular value has proved to be very valuable in our analysis, as it gives more concise results when few methods are filtered, but it can also severely impair the grouping accuracy when we are analyzing for more APIs.

Figure 2.8 shows the pseudocode of our grouping algorithm.

2.3.6 Method Query

One of the key features of R.E.H.A. is its ability to instantly highlight the methods that are shared among different apps. However, the grouping algorithm requires the threshold at which apps are deemed similar to be set manually. This has an unfortunate side-effect; certain apps just do not share enough relevant code to meet the criteria in the grouping algorithm, but

```
for sample ∈ analyzed_apps do
  for group ∈ groupset do
    score ← similarity_score(group, sample)
    if score > threshold then
      group ← group∪sample
    else
      new group
      group ← sample
      groupset ← groupset∪group
    endif
  endfor
endfor
```

Figure 2.8: Grouping algorithm.

still share some critical methods. The method query subsystem was created by leveraging our optimizations when it comes to search space reduction (described in Section 2.4). In short, a method can be inserted as query, which then gets transformed into its centroids and API vectors, and ultimately it is searched in the database.

2.4 System and Implementation

In this section we describe the implementation details for our system, starting with the search space reduction algorithm that allows us to reduce the analysis time by an order of magnitude.

2.4.1 Search Space Reduction

To analyze an entire dataset, R.E.H.A. needs to encode every method of every sample and compare it to the others. Numerically, this means that if the dataset contains n methods, there will be $O(n^2)$ comparisons, which becomes a problem very fast when analyzing big datasets.

For example, the GENOME dataset (which isn't particularly big) has more than 1,000 apps, and each one of them has up to 3,000 methods in it (mostly some members of the DroidKungFu family), which totals at about $3 * 10^6$ methods with $9 * 10^{12}$ comparisons. Our algorithm can compute 10^5 comparisons per second on average, so that would take close to $9 * 10^7$ seconds (slightly less than 3 years). Of course this is not an acceptable running time for any kind of analysis, thus it became necessary to restrict the search space.

Since the first step of the algorithm considers structural similarity between methods by converting the CFG into a 3-dimensional vector (4-dimensional if we consider the weight),

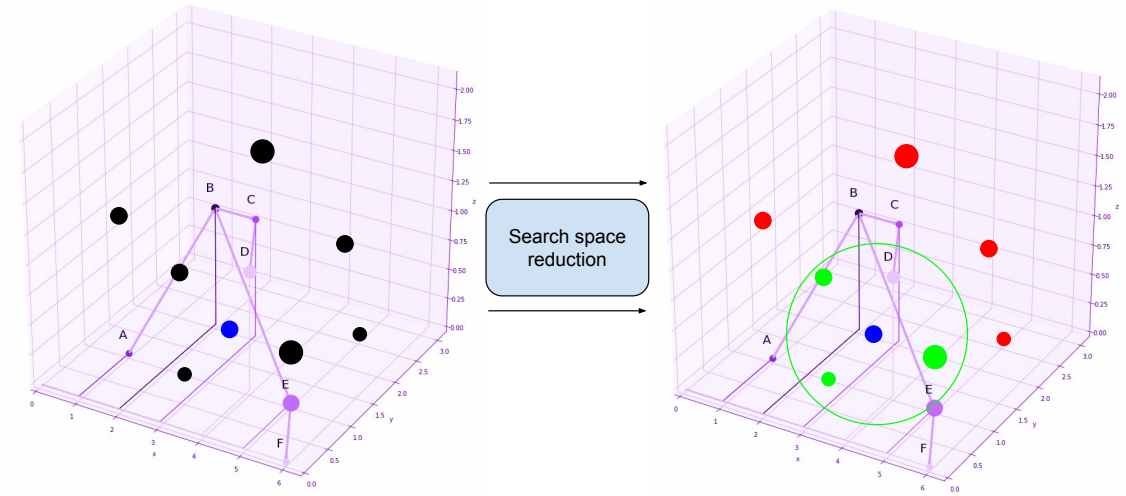

Figure 2.9: Search space reduction: on the left the whole search space, on the right the green centroids in the spheroid are considered while the red ones are discarded

restricting the search space literally means that we may consider only the regions of the 4-dimensional space that contain centroids closer to the input centroid.

The first step of the dimensionality reduction occurs during the preliminary phase. When every method is coded in its centroid, R.E.H.A. updates a nested dictionary-like data structure that will act as a hash-table to allow for fast searching.
The dictionary is thus updated:

```
c = (x, y, z, w)

update_centroid_dict(Centroid c):
dict[floor(x)][floor(y)][floor(z)][w].append(c)
```

This way, a centroid c_1 with the coordinates `[1.342,3.45,8.01,12]` will be added to a list of other centroids in `dict[1][3][8][12]`.

The algorithm to search for a matching centroid is now fairly straightforward, we just need to calculate a valid range of coordinates to check, and then look into their respective lists. The range of coordinates is calculated using the CDD function. For example, given the previous centroid c_1 and the standard thresholds for R.E.H.A., its matching centroids will be searched in the following ranges: $x = (1, 1), y = (2, 4), z = (6, 10), w = (9, 15)$. This gives us 48 possible lists of centroids, with most of them realistically being empty.

For a more practical example, we'll run our algorithm on the first method of the first sample in the GENOME dataset: 86 actual MDD checks and only 9 API checks, in a dataset with about 70k methods. Checking every single method in the dataset against every other method would take more than 13 hours, but by reducing the search space it will take just 18 seconds, assuming that the search space was equally distributed (a pretty bold and unrealistic assumption).

Since there is not any theoretical reason why methods should stack up in a particular spot, we ran some tests and the worst performance by far was in a method that had to be checked against 290 other methods. With this experimental worst case in mind (assuming that every single one of the methods had to find 290 suitable doubles), the execution time of our algorithm would still take just over 3 minutes. Compared to 13 hours, it is still a pretty big improvement.

A simplified illustration of the results of our space search reduction algorithm can be seen in Figure 2.9.

2.4.2 Implementation

We implemented R.E.H.A. in about 3K lines of Python as a web application, to make it easier to deploy on remote servers. This section contains a simple overview of the implementation details concerning its core elements.

For the server side aspects of R.E.H.A., we used the web framework Flask, which allows for faster initial development and launch on a local machine. Different technologies will be considered for an eventual future production deployment.

The system works by leveraging apktool to extract the apps and translate the dex files into the more readable smali format. Then, a simple parser scans through the smali files and generates the features needed for the similarity measure. Each method is transformed into its respective CFG (as better explained in Section 2.3.2) and saved in a dictionary that will remain in RAM for the duration of the analysis. A previous version of R.E.H.A. had everything stored on disk but that created unnecessary bottlenecks. Currently, the only disk I/O operations occur on server startup and after the analysis is completed. This means that the dictionary containing the features and all the reports are saved on disk in plain text using the Python library Pickle, so that R.E.H.A. can reload them in memory at each startup.

The analysis times have dropped significantly thanks to this implementation detail, but the program start-up time has increased significantly and so has the RAM usage. In order to further speed up the analysis we leveraged multi-threading during the method comparison phase.

2.5 Experimental Evaluation

In the following sections, we describe our datasets and the experiments we performed to evaluate R.E.H.A.. More precisely, we first provide the results of the classification we performed using R.E.H.A.. To evaluate the classification accuracy, we leveraged AVClass [125], a malware labeling tool that determines the most likely family of a given sample by clustering the AV labels obtained through VirusTotal. Then, we present some interesting case studies that show how malware samples reuse malicious code. Finally, we assess the runtime performance of our tool.

2.5.1 Datasets

We used four different datasets for our evaluation. First, we got access to 675 ransomware samples from the Heldroid [5] dataset (**Dataset_1**). Then, between July and August 2017, we used the VirusTotal Intelligence API to obtain two datasets: (1) the 500 most recent Android ransomware labeled as ransomware by at least 5 AVs (**Dataset_2**); (2) the 1,000 most recent generic malware labeled as malicious by at least 5 AVs (**Dataset_3**).

Finally, we gained access to 2,036 apps labeled as malicious by at least 25% of the AVs in AndroTotal [89] (**Dataset_4**).

In summary, our datasets total 4,211 malicious apps.

2.5.2 Classificiation Results

Dataset_1 + Dataset_2. For this study, we analyzed a dataset of 1,175 ransomware. The goal of the analysis was to group these samples by highlighting their shared code.

The biggest group in the dataset turned out to comprise 242 samples, with all of them sharing the same methods to encrypt and decrypt files. This group goes well beyond code reuse, as all of them have the same structure and contain only one package with always exactly 22 smali files. The only differences are in the package name and in the file names, as all of the samples seem to contain different permutations of random strings.

The methods found to be most relevant, using the API filter function, are the `encrypt()`, `decrypt()` and `init()` methods, which contain all the code necessary to perform the main action of the ransomware.

The second group has 94 members and is another case of extreme code reuse, where most samples contain exactly the same files, this time even with the same names and with only a couple different packages for most of it. R.E.H.A. was able to identify code reuse even among those few samples that were not cloned exactly. In these samples, as before, the only

C	koler	locker	simplocker	slocker	crosate	svpeng
G1	242	0	0	0	0	0
G2	69	0	0	0	0	0
G3	0	0	0	0	22	39
G4	40	0	0	0	0	0
G5	0	34	0	2	0	0
G6	27	0	0	0	0	0
G7	0	25	0	0	0	0
G8	0	23	0	0	0	0
G9	0	22	0	0	0	0
G10	0	0	18	2	0	0
G11	0	0	7	13	0	0
G12	0	18	0	2	0	0

Table 2.1: Classification of **Dataset_1 + Dataset_2**. On the x axis we have the classification by AVClass, while R.E.H.A.'s classification is on the y axis.

transformation applied were different names for the packages and smali files. However, in these cases, the entire structure of the application was changed.

All in all, R.E.H.A. helped isolate at least 18 groups of apps in the dataset that share their core code. In Table 2.1, we show 12 interesting groups and their relationship with AVClass classes. From top to bottom, it is easy to see that what AVClass labels as **koler** or **locker** are actually 9 different groups of applications, which probably exhibit the same behaviour (they are all ransomware, after all) but do not really share enough code to be considered similar by R.E.H.A.. Another interesting observation can be made by looking at the table from left to right on G11, the families **simplocker** and **slocker** could be easily merged, which may also explain the similarity of their class names. The same observation can be made on the samples in the families **svpeng** and **crosate**, which were both classified by R.E.H.A. into G3. Manual analysis has confirmed that all these samples indeed belong together.

This means that R.E.H.A. gave a more accurate representation of the dataset as a whole by providing more detailed granularity to the generic labels **locker** and **koler** while also grouping together samples unnecessarily classified into different families. This is evident when looking at the graphs in Figure 2.10 and 2.11.

One of the best results was given by the second biggest group isolated by R.E.H.A., all of which was comprised of samples of ransomware downloaded between July and August 2017. The 94 samples were nearly identical clones, and were correctly grouped together into one class by R.E.H.A., while AVClass produced 4 different groups (**jisut, slocker, congur,**

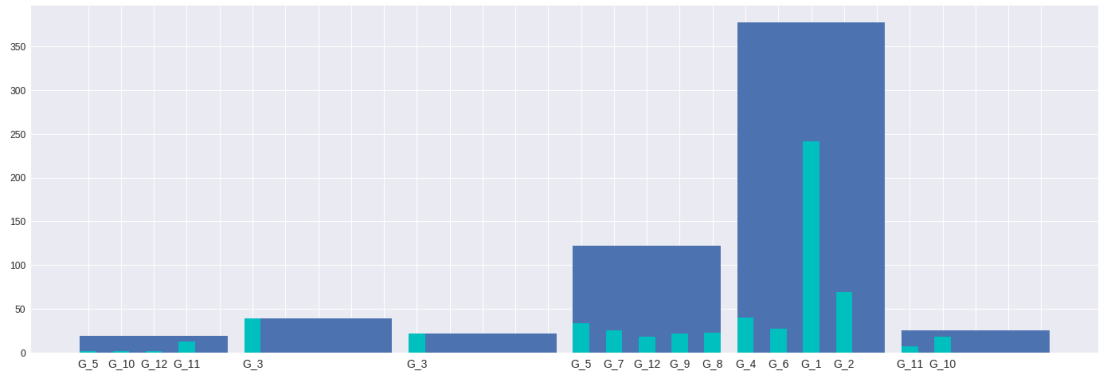

Figure 2.10: AVClass labels (darker shade) are more generic, R.E.H.A. helps refine the results by subdividing them.

C	SmsThief	SmsReg	SmsSpy	smsforw	smsPay
G1	25	0	1	0	0
G2	0	22	0	0	1
G3	0	18	0	0	1
G4	0	4	0	0	2
G5	0	8	0	0	0
G6	0	0	0	9	0
G7	7	0	0	0	0
G8	0	0	0	0	0
G9	0	0	0	6	0

Table 2.2: Classification of **Dataset_3** (AVClass on the x axis, R.E.H.A. on the y axis).

lockscreen), with samples evenly distributed among them. It is very likely that the labeling of new malware is less consistent among different antivirus software, as we can see the opposite trend during our evaluation of the tool with the GENOME dataset.

Dataset_3. Another dataset we worked on was downloaded from VirusTotal and comprises 539 samples. R.E.H.A. identified about 20 unique groups when analyzing for TELEPHONY related behaviours. In Table 2.2, we highlight only some of the most interesting groups. Looking at this table, it is clear that the family **SmsReg** is actually divided into 4 groups and the families **SmsThief** and **smsforw** contain 2 distinct groups each, again making a case for a very weak ground truth. These results also hint at the fact that **SmsThief** and **SmsSpy** could

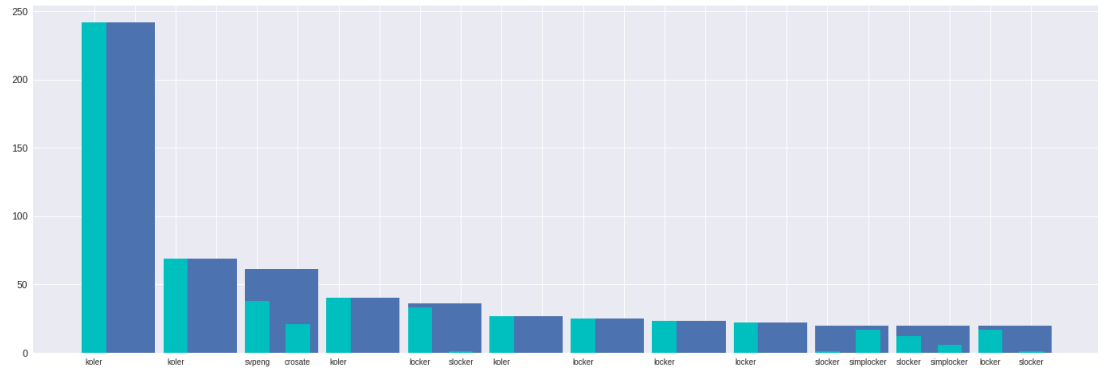

Figure 2.11: R.E.H.A. helps grouping together samples that were divided by AVClass.

C	SmsThief	SmsReg	SmsSpy	lockscreen	singleton
G1	32	0	11	0	0
G2	0	0	0	8	20
G3	0	22	0	0	0

Table 2.3: Dataset_3 with Crypto APIs (AVClass on the x axis, R.E.H.A. on the y axis).

be merged, which becomes clearer when changing the filter to consider cryptographic APIs, as shown in Table 2.3. The most interesting result from this analysis comes from group G2, where 8 samples were classified by AVClass as **lockscreen** and 20 as **singleton**. This means that AVClass did not have a label for them, so they are 20 unseen samples that have been correctly classified by R.E.H.A. as members of the family **lockscreen**. The fact that all members of the family **SmsReg** were classified into G2 in Table 2.2 and then into G3 in Table 2.3 is a good indication that the grouping was indeed correct.

Dataset_4. Our last analysis is a dataset consisting of 1,790 samples, all downloaded from AndroTotal.
Table 2.4 depicts some interesting groups of malware with telephony-related behaviours. The classification in this case seems to be much more consistent between R.E.H.A. and AVClass. One reason for this could be that these families have been around for a while, as they appear in the original GENOME dataset (apart from **Sandr**), so they have been thoroughly studied and analyzed. It is also interesting to note that R.E.H.A. divided the samples in what AVClass decided was the family **DroidDream** into 2 different groups. With the help of R.E.H.A.'s

C	BaseBridge	BeanBot	SMSpy	DroidDream	Sandr
G1	65	0	0	0	0
G2	0	49	0	0	0
G3	0	0	37	0	0
G4	0	0	0	36	0
G5	0	0	0	33	0
G6	0	0	0	0	31

Table 2.4: Classification of **Dataset_4** (AVClass on the x axis, R.E.H.A. on the y axis).

handy method filtering tool, we looked into these samples. We noticed that, while all of the samples in the 2 groups mined the IMEI, IMSI and device ID from the unsuspecting user, they went about it in a completely different way: the samples in G4 divided each action in 3 different methods, while those in G5 did it in one single method. But this is not the only difference, as the samples in G4 can send SMS messages to steal the user's IDs, while those in G5 do not have this capability and just send everything through the internet. So, in a way, all 69 samples have some consistent behaviours, but they implement them in very different ways.

2.5.3 Classification Accuracy

R.E.H.A. was first tested using the very well known GENOME dataset [156]. It contains 1200 malware samples of Android malware, categorized into 49 families, all of which were collected between 2010 and 2011.

This dataset has already been the subject of many studies, all of which have expanded the information gathered on the families and addressed the various problems that can be solved by classifying malware correctly ([2], [152], [18]). The fact that the dataset is already divided into malware families and the huge corpora of existing studies on it means that R.E.H.A.'s accuracy can be assessed easily.

As the samples in this dataset have been gathered in 2010 and 2011, that implies that they do not perfectly reflect modern malware, however the dataset overall seems like a suitable candidate for initial testing.

Of the 49 families in the dataset, 16 of these contain only one sample each. Since R.E.H.A. can classify samples by grouping them with regard to their similarity, these 16 families cannot possibly be analyzed and so were taken out of the dataset. The rest of the dataset was grouped and we checked manually to see if the labels were consistent and the results for the remaining 33 families can be seen in Table 2.5. In the end, our tool achieved an accuracy of 92%.

All results are averaged throughout every family. The precision score is skewed due to the

	Precision	Recall	Accuracy
CCD	0.7812	0.9578	0.8605
BVE	0.8741	0.9487	0.9099
VDD	0.9057	0.9442	0.9246

Table 2.5: Classification accuracy for the GENOME dataset.

dataset containing 5 variants of DroidKungFu that are often grouped together, thus inflating the number of false positives. On the other hand, recall does not fluctuate much because the false negatives remain fairly consistent. It is easy to see that accuracy of the classification is improved by the addition of the BVE and VDD formulas to check for API vector similarity.

2.5.4 Case Studies

Most of the code reuse we found only involve simple app cloning, sometimes with package and filename renaming, probably to avoid detection by simpler tools. In this section, we describe one of our most interesting findings: inter-group code reuse.

Code Removal. The group G2 of **Dataset_1 + Dataset_2** is one of the biggest groups, containing 94 samples. All of its members share the methods `encrypt()`, `decrypt()`, `getKey()`, `getMD5string()`, and `init()` that are deemed as interesting by R.E.H.A. when filtering for cryptographic methods. However, when the classification is extended to other functions (such as display functions), there is another group that shares a lot of code with samples from group 2. In fact, they share not only the code, but most of the application's structure as well (as shown in Figure 2.12). The reason they do not belong in the same group according to R.E.H.A. is that they are missing all the classes that deal with encryption, making each sample in the group more like a "scareware" as they lack the ability to actually do any harm to the filesystem. This highlights one of the challenges when trying to classify entire applications with regards to their code similarity, as we need to manually set how much shared code constitutes similarity to begin with.

Code Transformations. Many samples encountered in our analysis are clones. For instance, all members of group G2 in **Dataset_1 + Dataset_2** share the same exact code and file structure, apart from 7 samples that are just a Chinese variant of the malware (they still share the core code, but with a changed file structure of the app and additional classes). Other families change some basic properties to avoid detection. For example, group G1 in **Dataset_1 + Dataset_2** keeps all the code and method names intact, while file names differ across samples.

```
smali
    com
        sssp
            BAH.smali
            R$string.smali
            R$xml.smali
            R$id.smali
            R$drawable.smali
            bbb.smali
            R$attr.smali
            R$raw.smali
            R.smali
            R$color.smali
            s$100000000.smali
            s.smali
            s$100000001.smali
            MyAdmin.smali
            R$style.smali
            R$layout.smali
            MD5Util.smali
            M.smali
            BuildConfig.smali
            R$anim.smali
            DU.smali
    adrt
        ADRTSender.smali
        ADRTLogCatReader.smali
```

```
smali
    com
        bangbangtang
            lock
                R$string.smali
                R$id.smali
                R$drawable.smali
                R$attr.smali
                b.smali
                R.smali
                c.smali
                a.smali
                R$style.smali
                R$layout.smali
                BuildConfig.smali
                b$100000000.smali
    adrt
        ADRTSender.smali
        ADRTLogCatReader.smali
```

Figure 2.12: Case study, Code Removal (Section 2.5.4). The main instructions that have been removed are highlighted in red.

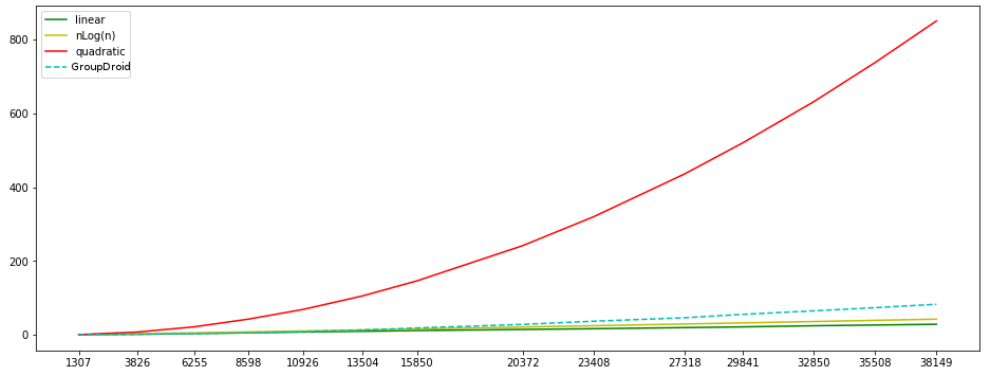

Figure 2.13: Performance of R.E.H.A.'s optimized comparison algorithm.

The most interesting samples try to avoid detection by obfuscating everything, from the file structure to their code, as is the case of group G3 in **Dataset_4**, where every smali file is littered with dozens of methods (with encrypted names) that do not do anything but call each other, thus obfuscating the CFG of the whole app. Control flow obfuscation is also applied to the individual methods, making it a challenge for most similarity detection techniques. Despite this, R.E.H.A. was able to successfully group these apps together thanks to its relaxed CDD threshold (which allows for greater control flow manipulation) and to the use of the API vector checks.

2.5.5 Performance

The performance of R.E.H.A. varies wildly in relation to the size of the dataset and the size of the samples in the dataset. We conducted an empiric study, adding few APKs at a time from the GENOME dataset and timing how much it took for the system to analyze them. Figure 2.13 details how much R.E.H.A. scales in comparison to a thorough pairwise comparison (which would incur in a quadratic slowdown) and both linear and nlogn functions (ideals) in the worst case. The total to run the similarity check for 38.149 methods was around 80 seconds, while it would have taken more than 800 seconds (circa 14 minutes) without search space reduction.

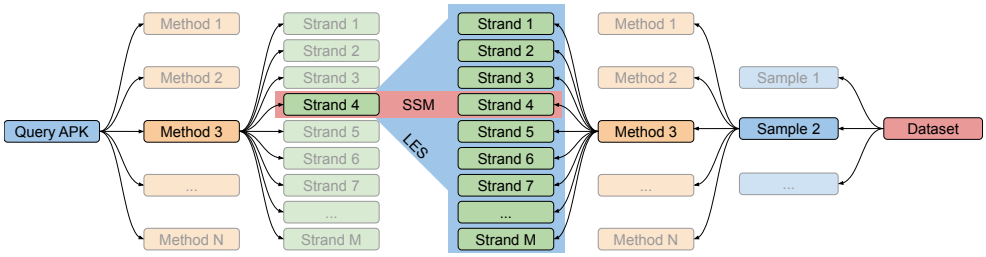

Figure 2.14: Workflow of StranDroid.

2.6 Methodology of the Compositional Approach

In this section we will describe our approach in detail, including some of the methodology in [48] that initially inspired this work and delving deeper in the differences between our work and [48].

A detailed workflow can be seen in Fig 2.14. A query APK is compared to the rest of the dataset by first extracting all its methods and dividing them into basic blocks. Then, from every method we generate the strands (see Section 2.1 and [48]) which are placed into buckets. This is done for every sample in the dataset (only once, then it is saved for future runs). Every strand from the query APK is compared with all the strands coming from each method of the dataset samples using the *Strand Similarity Measure*. The *Local Evidence Score* is then computed to gauge how significant the strand matching really is. Finally, the *Global Evidence Score* will simply sum all the LES from the various strands in the methods and generate a score that indicates how similar two methods are. In the rest of the section we go into detail for each of the steps described above.

Use Case

StranDroid is a tool focused on the analysis and reverse engineering of malware. In order to use the tool at its fullest, we first need a dataset of Android malware samples. Then, we can input a query sample q_s into the system and StranDroid will show which sample best matches q_s wrt its potentially malicious behaviors.

This comparison is done at the method level: the sample that shares the highest number of methods with q_s is selected as the result of the analysis. Along with this, StranDroid will show a comprehensive list of all the common methods between the two samples and their relative similarity scores, allowing the analysts to focus the reverse engineering efforts on these methods.

Method Filtering and Collection

As a first step in the analysis, StranDroid builds an internal representation of all the APKs in the dataset by collecting their methods. We filter out the methods that do not meet two basic requirements: 1) a minimum length, and 2) presence of calls to *Risky APIs* (see Section 2.1 and [94] for a brief introduction).

The minimum length of the methods is a variable parameter that can be set at the beginning of the analysis and is meant to filter out common methods that are used in most APKs (such as *init* methods) and would muddy the detection of actual malicious behaviours. In order to further speed up the analysis, StranDroid will not consider methods that do not contain at least one invocation to a *risky API*, since these methods only contain the internal logic of the app and cannot exhibit malicious behaviors targeted towards either the user or the device itself. Both of these heuristics have been adopted due to our previous experience with R.E.H.A. [94] and they have been successfully tested empirically. During this initial scan of the methods in the dataset, the code is parsed by StranDroid in order to speed up the next part of the analysis.

Parsing

Program slicing requires isolating instructions that are correlated to the slicing criterion through data flow [141] (control flow can be ignored since strands are extracted from basic blocks), thus the first component of the tool is a parser for Smali. In order to generate def-use chains it is essential to know which variables are used and defined in each line. For this purpose, our smali code parser extracts these variables using the pattern of the particular opcode: we define 6 procedures covering all the possible Def-Use behaviors of the instructions and apply the correct routine to each opcode.

Strand Extraction and Normalization

Once all the interesting methods have been collected and parsed for every APK in the dataset, they are split into basic blocks by a simple heuristic. At this point, strands are extracted from the basic blocks. Strands are static slices of the code contained in a basic block, obtained via a simple backward slicing algorithm (see pseudocode at Figure 2.15). These strands then go through a *normalization* process, where every variable encountered is renamed wrt its order of appearance in the strand.

As an example, we show this on a snippet from a simplified method (its statements have been shortened):

```
strands = []
for inst in reverse(BB):
    s <- new Strand
    s <- s + inst
    d <- defined_vars(inst)
    u <- used_vars(inst)
    BB <- BB - inst
    for inst_1 in reverse(BB):
        d_1 <- defined_vars(inst_1)
        u_1 <- used_vars(inst_1)
        for v in u:
            if v in d_1:
                u <- u + u_1
                s <- s + inst_1
                BB <- BB - inst_1
    strands <- strands + reverse(s)
return strands
```

Figure 2.15: Strand Extraction Algorithm

```
move-object v4, v2
const/high16 v5, 0x10000000
invoke-virtual {v4, v5}, setFlags(I)
move-result-object v4
move-object v6, v1
move-object v7, v4
invoke-virtual {v6}, LstartService()
move-result-object v6
```

Assuming that the last statement is the slicing criterion, starting from it, the algorithm walks backwards and collects all the statements that may affect the value of $v6$, which is the only variable in the slicing criterion. This is the extracted strand:

```
move-object v6, v1
invoke-virtual {v6}, LstartService()
move-result-object v6
```

Clearly all the statements ignored do not contain a data-flow connection with $v6$. At this point we normalize the strand by renaming the variables:

```
move-object v1, v2
invoke-virtual {v1}, LstartService()
move-result-object v1
```

```
iget-object v5, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-interface {v5}, Lorg/apache/http/HttpResponse;->getEntity()Lorg/apache/http/
    HttpEntity;
move-result-object v1
.local v1, "entity":Lorg/apache/http/HttpEntity;
invoke-interface {v1}, Lorg/apache/http/HttpEntity;->getContent()Ljava/io/InputStream;
move-result-object v5
invoke-virtual {p0, v5}, Lcom/xxx/yyy/BBBB;->generateString(Ljava/io/InputStream;)
    Ljava/lang/String;
```

```
iget-object v9, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-interface {v9}, Lorg/apache/http/HttpResponse;->getEntity()Lorg/apache/http/
    HttpEntity;
move-result-object v2
.local v2, "entity":Lorg/apache/http/HttpEntity;
iget-object v9, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-virtual {p0, v9}, Lcom/xxx/yyy/BBBB;->getContentCharset(Lorg/apache/http/
    HttpResponse;)
move-result-object v0
.local v0, "charset":Ljava/lang/String;
invoke-interface {v2}, Lorg/apache/http/HttpEntity;->getContent()Ljava/io/InputStream;
move-result-object v9
invoke-virtual {p0, v9, v0}, Lcom/xxx/yyy/BBBB;->generateString(Ljava/io/InputStream;)
```

Figure 2.16: Two semantically equivalent strands from method HppGet

This last step is done in order to thwart the all too common occurrence of statement reordering, which happens both due to obfuscation attempts and the decompilation process. In our tests, strand normalization has proven to be a necessary step, since the similarity measure is very syntactic.

Strand Similarity Measure (*SSM*)

In [48] the similarity between two strands was computed via a program verifier that checks for input-output equivalence between the strands while pairing each variable from a strand with the corresponding one in the other. This is done by lifting the binary procedure into BoogieIVL [13] by first going through IDA pro, then LLVM IR [82] via BAP [22] and lastly SMACK [114] is used to translate LLVM IR into BoogieIVL.

Lacking such a peculiar toolchain for our Android use case, we opted to simplify the strand similarity measure and adopt the very common Jaccard index. Two strands s_1 and s_2 are compared wrt the Jaccard index of the instructions they contain. Let us recall the mathematical definition of the Jaccard index between two sets:

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} = \frac{|A \bigcap B|}{|A| + |B| - |A \bigcap B|} \tag{2.10}$$

Then our strand similarity measure (*SSM*) can be stated as:

$$SSM(s_1, s_2) = J(lines(s_1), lines(s_2)) \tag{2.11}$$

Where $lines(s_1)$ and $lines(s_s)$ denote the set of statements contained in the normalized strands s_1 and s_2 respectively. For example, given the normalized strand in the previous section ($strand_A$) we can have a similar strand extracted from a different method and then normalized ($strand_B$):

```
move-object v1, v2
move-object v2, v1
invoke-virtual {v1}, LstartService()
move-result-object v1
```

This strand is almost equivalent to the previous but has an added line *move-object v2, v1* that does not change the semantics of the strand (we found these types of meaningless insertions in our manual investigation). Calculating the Jaccard index between these strands we have 0.75, since:

$$\frac{|strand_A \bigcap strand_B|}{|strand_A \bigcup strand_B|} = \frac{3}{4} = 0.75 \tag{2.12}$$

This can be thought of as the likelihood of the two strands being similar. While 75% likelihood might seem too low a value, it is mainly dictated by the reduced length of this example, as the Jaccard index of two similar strands is usually higher for longer strands.

The design choice of using the Jaccard index as similarity between two strands comes from the need for an efficient method to compute the function, as it must be computed for every strand in every method. It is also convenient that the Jaccard index produces a value between 0 and 1, giving a sort of likelihood to the similarity of two strands. We will discuss possible problems with this approach in Section 2.9.

Method Similarity Measure

When every strand is extracted both in the dataset and in the query sample q_s, StranDroid checks every method in q_s against every method of the dataset with the following algorithm. First we define a function that, given a query strand s_q and a target method t, gives us the

likelihood of finding a strand $s_t \in strands(t)$ such that s_q and s_r are semantically similar. This is equal to the maximum of the *SSM* between the query strand s_q and every strand contained in t:

$$\mathcal{P}(s_q|t) = max_{s_t \in t}(SSM(s_q, s_t)) \tag{2.13}$$

Continuing with the methodology first described in [48] we implement a function that measures the statistical significance of a strand wrt the entire dataset. This is done in order to give more weight to strands that are not common in the dataset, and should result in a similarity measure that focuses on the more unique parts of the code. Given a query strand s_q and all the strands in the dataset $s_t \in T$, we define:

$$\mathcal{P}(s_q, T) = \frac{\sum_{s_t \in T} SSM(s_q|s_t)}{|T|} \tag{2.14}$$

A lower value of $\mathcal{P}(s_q, T)$ represents a higher statistical relevance, as it means that s_q has few semantically similar strands in the dataset.

Following [48] we can now define the Local Evidence Score (*LES*) between a strand and a method as:

$$\begin{aligned} LES(s_q, t) &= Log(\frac{\mathcal{P}(s_q, t)}{\mathcal{P}(s_q, T)}) \\ &= Log(\frac{max_{s_t \in t}(SSM(s_q, s_t))}{\mathcal{P}(s_q, T)}) \end{aligned} \tag{2.15}$$

The last function that we need to define in order to obtain a similarity measure between methods is the Global Evidence Score (*GES*). Given a query method q contained in the query sample and a target method t extracted from one of the samples in the database we have:

$$GES(q|t) = \sum_{s_q \in q} LES(s_q|t) \tag{2.16}$$

The measure of similarity between two methods is given by the sum of the individual values of *LES* for every strand in the query method. This sum is, of course, lower-bounded by 0, but does not have an upper bound, which can induce significant errors that we discuss in Sections 2.7 and 2.9. For each method in the query sample, StranDroid computes the GES for every

method in the dataset and returns only the method that generates the highest score, provided it exceeds the set threshold of 4.

The sample r_s that matches the most methods with the query sample q_s is then returned as the result of the analysis, along with a list of all similar methods between the two samples.

2.7 Evaluation of StranDroid

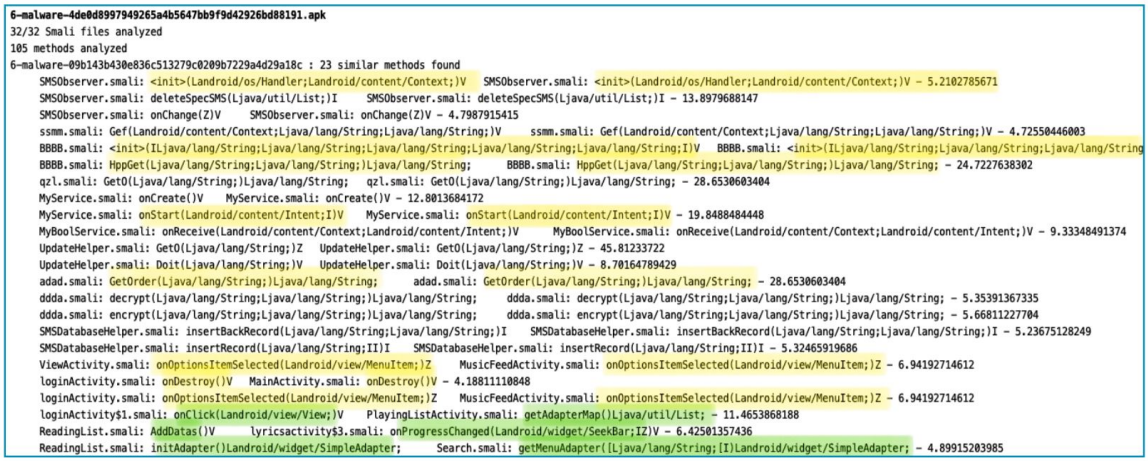

Figure 2.17: A screenshot of StranDroid, the new similar methods that were missed by R.E.H.A. are highlighted in yellow. The only false positives of StranDroid are highlighted in green.

As a first step, StranDroid has been evaluated against the well known GENOME dataset [156] simulating a classification task. Since our tool requires an existing dataset of known malware, this proved to be the easiest test bed to ascertain its efficacy, even though the dataset itself is showing its age. The GENOME dataset contains mostly malware that has not been thoroughly modified, so these results, while encouraging, were not sufficient for a proper evaluation. For this reason, a second evaluation has been conducted on the PraGuard dataset [91], which contains samples with various program transformations such as string encryption, class encryption and reflection.

We then tested StranDroid on samples selected from a dataset of Android ransomware and malware previously used in the evaluation of R.E.H.A. [94], as this allowed us to easily spot any inconsistencies between the two approaches. Unlike our previous approach, StranDroid adopts a more precise similarity algorithm that has virtually zero false positives, thus it easily

refines the results we had with R.E.H.A.. Figure 2.18 shows a comparison between the two tools by comparing the number of similar methods that were found among 8 pairs in the dataset. R.E.H.A. almost always over-approximates and returns more similar methods than StranDroid, except in a few cases where StranDroid actually discovers some methods that eluded detection in R.E.H.A.. An example of the analysis results can be seen in Figure 2.17.

GENOME

We extracted around 600 samples from the GENOME dataset, excluding the families that contained less than 20 samples each. One sample from each family was then selected and removed from the dataset. We then used StranDroid with the removed samples as the query APKs, in order to find similar APKs in the dataset. Our tool paired each query APK with samples of their original family on 100% of the cases, thus validating the approach.

PraGuard

The PraGuard dataset [91] provided a few different code obfuscations applied to the GENOME dataset. We tested StranDroid with these by extracting one sample from each class in the obfuscated database and using it as a query APK against the entire original GENOME dataset. The results on the samples modified using string encryption have been positive, with every sample extracted randomly from each malware class being classified correctly with other samples in the same class. This test proved that the string comparison, used to gauge the equivalence of statements for the *SSM* between strands, does not negatively impact the effectiveness of our tool when used for classification.

The next two obfuscation classes obtained from PraGuard, class encryption and reflection, uncovered a lot of flaws in the approach. Both classes resulted in unsatisfactory classification, with the samples obfuscated with class encryption resulting in zero similar APKs for many of them. This negative result is unavoidable as the APKs are obfuscated with DexGuard [69], which encrypts and compresses (with GZIP) every class in the APK. The content of the classes is thus completely hidden to a static analyzer and is only revealed at runtime. Our approach relies on extracting data flow information from methods statically, which means that the only methods available for analysis were the ones used for runtime decryption.

The samples obfuscated with reflection instead generated many false positives, which is easily explainable by the confined nature of our analysis (every strand comes from a single basic block). In order to correctly calculate the similarity between strands, the method invocation has to be part of the strand itself.

Use Case Dataset

We used a dataset of 20 Android malware and ransomware samples, a reduced version of the one collected in 2017 for [94], where each sample is similar to at least one other sample in the dataset, giving a total of 10 semantically-similar program pairs (or families). By *similar* samples, in this context, we mean that they contain some of the same malicious behaviors, while the rest of the application (usually a piggy-backed legitimate app) is not considered for the similarity. In Figure 2.18 we show a direct comparison between StranDroid and R.E.H.A.. The latter almost always returns more similar methods but this is due to the presence of false positives, while StranDroid is generally more precise for all the classes considered and sometimes yields even fewer false negatives than R.E.H.A. (for example in *1-ransom*).

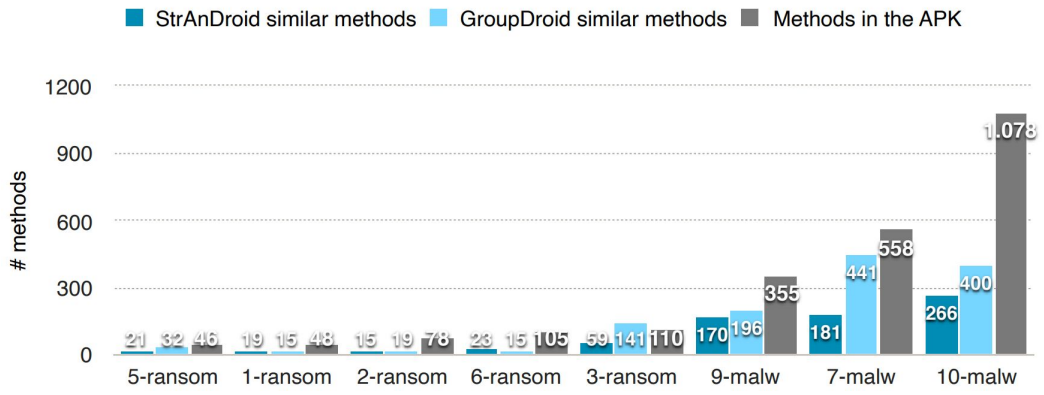

Figure 2.18: Comparing the number of similar methods found in 8 pairs of the dataset. R.E.H.A. often over-approximates and finds many false positives while StranDroid is more precise.

The dataset contains ransomware and malware samples and is fairly small to allow manual verification of the results, since the goal of this evaluation phase is to challenge the ground-truth extracted from an analysis by R.E.H.A.. In Section 2.10 we speculate on some possible improvements of this step.

Precision

The nature of the similarity measure implemented in StranDroid should make it so that the tool is not affected by certain types of code obfuscation such as structural transformations (modifying the CFG of the methods) and dummy code insertion. We verified this in our reduced

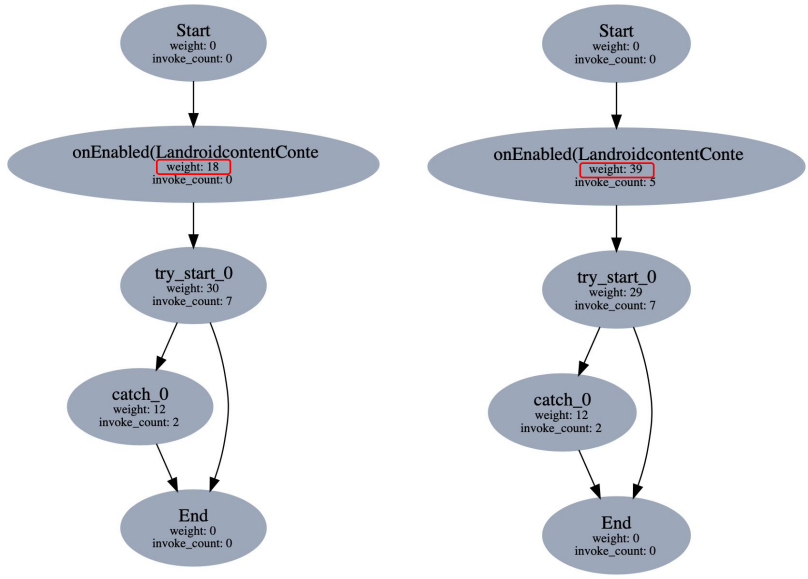

Figure 2.19: CFGs extracted from two semantically equivalent methods with bogus code insertions

dataset by running both R.E.H.A. and StranDroid and manually evaluating the results of the analyses.

The results of our tests are overwhelmingly positive. Using StranDroid , we uncovered the source of some false negatives in the analysis with R.E.H.A., mostly coming from samples employing the two aforementioned modifications.

In Section 2.7.1 we explore two specific examples of these tests.

Normal Code Evolution

Since most malware nowadays consists of modified versions of existing malware, it is possible that some of the transformations that we noticed in the samples are not always the result of an attempt to obfuscate the code, but rather they could simply be the result of updated and refined code for the new versions. Malware developers could add code to their samples not just as dummy filler to fool signature based approaches (although that seems to work well [55]), but also to add new behaviors. It is our opinion that this has to be investigated more, as it is possible to use StranDroid to analyze the evolution of malware in the same family over time (we will expand on this in Section 2.10).

Threshold

As anticipated in Section 2.6, the similarity between two methods is given by their *GES*, which is a summation of all the local *LES* between the strands and the method itself. This causes the similarity measure to assume theoretically unbounded values (since it depends on the number of strands in a methods), which means that setting a predefined threshold Tr that works on all methods is not a trivial task. If Tr is too small, it can cause false positives among methods that are not similar but contain a lot of strands (as we will see in Section 2.9), and false negatives if the methods are indeed similar but too small to reach Tr. We thoroughly experimented with our dataset and reached the conclusion that $Tr = 4$ is a good threshold to decide the similarity between two methods, as we encountered 71 methods in the class $7 - malware$ that are similar between the two samples and returned a *GES* between 4 and 4.5.

2.7.1 Case Studies

The nature of strands, namely that they are confined in basic blocks, allows StranDroid to be very precise when finding similarities between methods even when one of the control flows is modified substantially. Another advantage of using buckets of strands is evident when evaluating the similarity of methods where dummy code has been inserted, as the original code (the code that is semantically relevant) is still present in the form of a composition of strands. We now show two specific examples of these cases and highlight how focusing on strands helped the analysis.

Modified CFG

HppGet() is a method that is present in two similar malware samples ($sample_1$ and $sample_2$) in our reduced dataset and is used to communicate with a remote server with the use of the Apache HTTP API. The peculiarity of the two versions of this method is that the one in $sample_2$ is heavily modified wrt its CFG compared to the version in $sample_1$ (both CFGs can be seen in Figure 2.20).

This proved to be a challenge for the structural similarity approach taken by R.E.H.A., while StranDroid was able to recognize the meaningful strands in the code, ignoring the modifications to the CFG. One of the meaningful strands from each ransomware sample can be seen in Figure 2.16, it is easy to see that they describe the same behavior.

Dummy code insertion

onEnable() is a method extracted from two ransomware samples ($sample_3$ and $sample_4$). Both versions of the method have been manually investigated and evidently perform the same function, but the one in $sample_4$ is almost double the size of its original version in $sample_3$. The CFG of the methods has not been modified, but around 20 lines of dummy code have been inserted. This can be seen in Figure 2.19 where the *weight* parameter in the second basic block shows that the amount of statements therein contained has doubled.

This is usually done in order to fool automatic malware recognition tools that rely on exact file signatures and it also proved to be a challenge for R.E.H.A., since the similarity measure relies heavily on the *weight* of the basic blocks (the number of statements in it). As expected, the strand approach taken with StranDroid works flawlessly with transformations that modify the structure of the CFG.

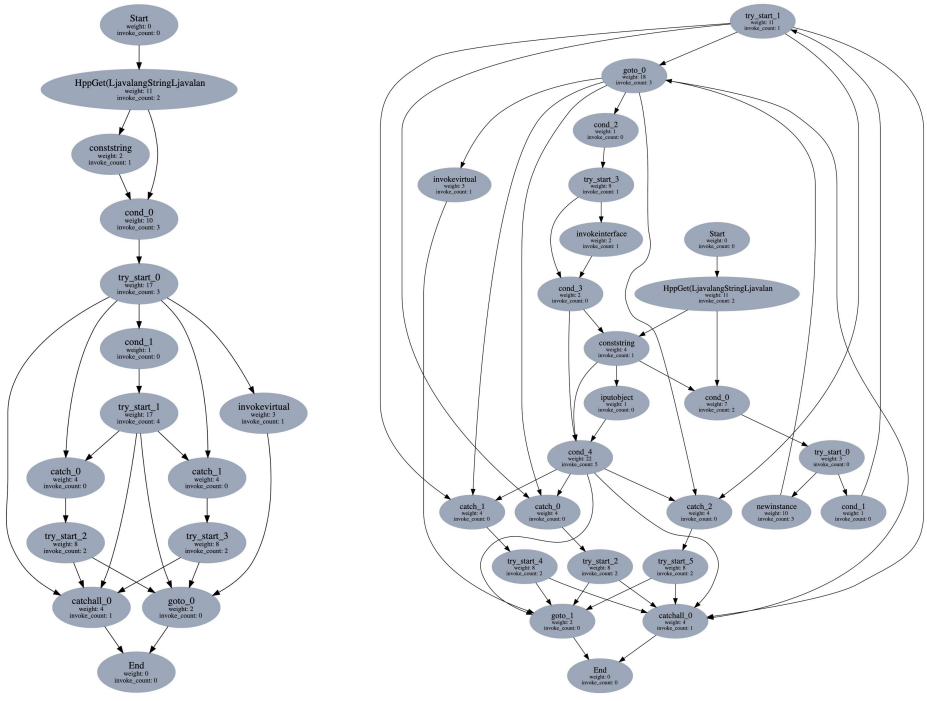

Figure 2.20: CFGs extracted from two semantically equivalent methods, StranDroid deems the methods similar enough

	N. of APKs	N. of Methods	Avg. Time (per method)
TS_1	100	20489	0.9s
TS_2	150	27273	1.32s
TS_3	200	40277	2.15s

Table 2.6: Test set for performance evaluation

2.7.2 Performance

The method parsing and strand generation are relatively simple operations, as they only require one pass for each smali file. Doing this to every sample in the dataset still yields a linear complexity, meaning that even with hundreds of samples the extraction times are fairly small. The true complexity of StranDroid comes from strand comparisons, as each strand from every method in the query APK has to be compared (via *SSM*) with every strand from every method in every sample of the dataset.

To gauge the actual performance of the tool we ran tests on a MacBook Pro with a 2.3 GHz i5 dual-core processor and 8GB of RAM, against a test set composed of 3 different subsets of the GENOME dataset with randomly extracted samples. The specifics of the test set can be seen in Table 2.6, along with the average time it took to analyze a single method in the query sample against every method in the test set. The query samples were also extracted randomly from the original dataset, one for each of the 5 classes *BeanBot*, *DroidDream*, *DroidKungFu*, *Geinimi* and *GoldDream*. The specific execution times for the tests on each sample against the 3 subsets can be observed in Table 2.7.

	TS_1	TS_2	TS_3	Risky methods in query sample
BB43	4m 18s	6m 15s	10m 39s	361
DDL7	3m 35s	5m 18s	8m 50s	234
DKF14	3m 58s	5m 44s	9m 45s	289
GEIN37	2m 50s	4m 23s	6m 58s	163
GD17	2m 24s	3m 58s	5m 55s	166
Avg.	3m 25s	5m 7s	8m 25s	

Table 2.7: Running times of the analysis on the test set

2.8 Related Work

In this section we will cover some existing works that closely relate to our approach.

Previous work on Android systems focused on detecting *clones* and *code reuse* in Android apps, providing ways to cluster APKs, and detect and classify Android malware. Many of the proposed methods employ static analysis to build features that can be used to cluster the malware, and strive to make the analyses scalable while retaining good accuracy results.

Clone Detection and Code Reuse [51] employs *Normalized Compression Distance (NCD)* on each method to find similar methods. Juxtapp [70] builds an application feature matrix and proposes a scalable method to cluster and evaluate similarity between applications. [41, 42] are instead based on *Program Dependency Graph (PDG)*, where the former measures similarity by first filtering methods, and then exploiting subgraph isomorphism to measure similarity. The latter instead exploits *Locality Sensitive Hashing* LSH to find approximate near-neighbors feature vectors. Wang et al. [137] use an approach based on filtering third party libraries, and then building a feature vector using API calls, measuring the similarity by pairwise comparison using the Manhattan distance.

Mobile Malware Classification and Clustering Mobile malware classification has been mainly tackled using machine learning: [50] extracts a signature to represent repackaged malware and tries to cluster malware samples into families. [152] is based instead on a peculiar data structure called *Weighted Contextual API Dependency Graphs(WC-ADG)* to capture the semantics of the methods and use these to build a feature vector. An approach that employs n-gram sequences of dex code is proposed in [88], with an intuition similar to the one in this work: malware samples belonging to the same family share the payload. The most common shared libraries are removed in order to lessen the impact of false positives. The similarity measure between the malware fingerprints generated with the n-grams is the Jaccard distance, making the work in [88] closely related to both our approaches in this thesis. Our approach in StranDroid is more focused on the control structure of the methods so it has a higher-level view of the programs compared to n-grams and it is more efficient (computationally-wise) and arguably more malleable to transformations. At the same time, strands are likely a better choice for semantic similarity since they have semantic significance beyond simple n-grams. Dynamic analysis is also employed in [75], where the authors classify Android malware by first building a "behavior profile" dynamically. After generating this profile for the samples in a specific family, the new samples to be classified undergo the same treatment and the behavior profiles can be checked for matches with a custom similarity function.

Android Malware Analysis Similar ideas have been proposed to tackle malware detection: [7] employs static analysis to extract a set of features that is then classified via linear Support Vector Machines to detect if a sample is malicious. An interesting instance is proposed by [60] and is based on feature space embedding based on call graphs. [2] evaluates how several machine learning algorithms score with an API-based features set. Previous work also explored the use of Markov chains [98] for behavioral models, and *Hidden Markov Models (HMM)* and structural entropy [24] to achieve mobile malware detection. An interesting work that tackles the problem of detecting variants of known malware samples is in [56], where the authors seek to detect variants of existing malware using a similarity digest hashing algorithm to generate a 64 byte static feature set. As stated in their conclusions, some of their false positives come from the original apps (benign) from which the malware authors crafted their malicious samples. This is likely because in the approach there is no applied methodology to distinguish malicious behaviour from normal behaviour, which in our work is done mainly via risky APIs filtering. A slightly different approach is taken by [29], which employs a features set obtained via both static and dynamic analysis. It differs from our work mostly because of the use of dynamic features and machine learning, but we share the view that certain API calls ("sensitive API calls" in their paper) can help distinguish between malicious and benign behaviours.

In [140], the authors highlight how important it is to develop more precise descriptions of the behaviors for existing malware datasets. They conduct a large-scale study where they analyze specific samples in various families that have been classified by existing anti-virus scans. Our approach shares the same goal of achieving a more precise analysis, but it differs in the methodology.

Many Android malware analysis tools have been developed recently using machine learning. There are works such as [150] that we already mentioned in Section 2.1 for sharing our view on risky APIs, and more recent works that employ modern ML techniques such as LSTM and autoencoders in order to better classify Android malware [145, 138]. Even if our approach is static, with no ML influence, we share the goal of finding the best features that can predict the maliciousness of an Android malware.

2.9 Limitations and Future Work

The two approaches described in this chapter suffer from different limitations. We list some of them separately in this section.

2.9.1 R.E.H.A. Limitations

Code Obfuscation. While R.E.H.A. managed to classify apps that were employing CFG obfuscation techniques, it still suffers from code obfuscation. This means that it might not be able to handle advanced obfuscated code. Some features could be better extracted in a dynamic way. For example, our risky_api vector stores the number of times a certain API is written in the method's body, while it would be far more interesting to know how many times it is actually called at runtime.

There are some downsides to extracting dynamic features. First of all, it is a costly endeavour, especially in Android. In order to run Android software in an environment apt to extract dynamic features, it needs to be installed in an emulator. The software itself then needs to be stimulated in order to produce the execution traces. Another reason why dynamic analysis is costly, especially in Android, is that it needs to run the right instructions in order to cover enough behaviours. Chapter 4 expands on some of the weaknesses encountered by dynamic analysis.

All these downsides notwitstanding, the dynamic extraction of the API calls would drastically reduce the number of false positives.

Whitelisting Known Libraries. Many of R.E.H.A.'s false positives come from popular ad libraries (especially admob), since they often use a subset of the risky APIs and are common enough to be found in many samples. The worst cases happen when some parts of an application without ads have been cloned, and the clone has ads. If the parts of the application responsible for ads are more significant than the cloned parts, the application will appear to be more similar to other applications that employ ads than to its own original. For this reason it is usually good practice to whitelist popular ad packages and it is an approach taken by other works that try to pinpoint the payload with a similarity measure such as [88].

Granularity. Right now, our analysis only considers similarity at the method level. However instructions can easily be spread among different methods, or some methods could be grouped, making our approach unsound. A good approach could be to consider the CFG of the entire application and search for common subgraphs. This is of course a problem that is incredibly hard to tackle (NP-hard in fact) and would require new heuristics and solutions to make it viable.

Performance. As almost every algorithm employed in R.E.H.A. is parallelizable, in the future we will adopt a cluster of machines with GPUs to divide the workload and speed up

the application by quite a bit. For now, parts of the method similarity functions have been rewritten with multi-threading in mind, leading to a considerable speed-up especially on systems with at least 8 cores. The experiments and their run times collected in this chapter have been collected without the use of parallelization.

Implementation. As previously explained in Section 2.4.2, every feature and every report is saved on disk in plain text using some serialization libraries. This requires the program to load a huge json file into memory at each start-up and the RAM usage takes quite a hit without running the analysis. We think it could be better to utilize a database driver such as PostgreSQL or MySQL to store the results.

On that note, we re-implemented the tool with MongoDB, a popular NoSQL database, with mixed results. The loading time of the tool has improved consistently but the run time of the analysis has incurred a considerable slowdown. Every method comparison has to query the database at least twice, leading to significant overhead that more than doubles the usual execution time.

All the times and results recorded in this chapter have been done without the MongoDB implementation.

2.9.2 StranDroid Limitations

Our tests with the PraGuard dataset [91] and the manual assessment of the results with our reduced malware/ransomware dataset have unveiled some limitations of our approach.

String Comparison

When calculating the SSM as the Jaccard index between two strands, the union operator considers the strands as sets and the statements therein contained as strings. This means that our algorithm will judge the uniqueness of a statement in the set by using exact string equivalence. This has proven to not be a problem when analyzing most samples in the GENOME dataset [156] but has resulted in a slightly reduced number of equivalent methods found when using samples from PraGuard obfuscated with string encryption.

It might be a good idea to replace the strings with a generic token instead of mining them directly. This would likely solve the aforementioned problems but could introduce new false positives. Such is the nature of static analysis.

Static Thresholds

The value 4 as a threshold for the *GES* between two methods has proven to be a good estimate for method similarity throughout the tests. This value is highly dependent on the size of the methods and on the number of strands, thus it could be useful to have a threshold that adapts to these parameters, or conversely, implement an algorithm that learns the correct threshold given the parameters.

Unbounded Similarity Measure

Related to the previous point, the *GES* is calculated as a summation of the *LES* between all strands. This makes its value theoretically unbounded, which further exacerbates the problem of having a static threshold for the similarity measure. Future evolutions of this work could consider a measure of central tendency such as the arithmetic mean.

Unique Result

As introduced in Section 2.6, StranDroid returns only one method as a result of the similarity analysis for each method in the query APK. This effectively means that, given a dataset containing families of malware and a query APK that belongs to one of the families, the result APK is going to be unique. In other words, no other APKs containing less similar methods (but still similar) is going to be returned. This might be too strict of a design choice, as our similarity measure is in no way perfect.

Improvements

Strand comparisons are independent of each other, which means that the performance of our tool could be increased by a great factor if we employed code parallelization. The tool also suffers from the bare-bones Python implementation, where a great number of string comparisons means a great decrease in performance. We are currently looking into *Cython* [14] in order to leverage the faster C implementation for string comparison.

2.10 Summary and Conclusions

In this chapter, we explored the malware classification and clustering in scenario 1, using Android as the open system from which we could gather the source code of the samples. We developed two different approaches for code similarity in Android starting from the smali representation of the APKs.

Both approaches share the intuition that malware samples by nature need to utilize system APIs that allow them to effectively disrupt the users experience and/or steal data. From this simple intuition, we generated a short list of "risky APIs" and used it to filter out all the methods that do not call any of them. These methods could not do any real damage and are therefore effectively whitelisted.

This simple initial heuristic allows us to filter many methods that would have cluttered the analysis otherwise. The two methodologies developed for this chapter are discussed below.

R.E.H.A.

The first approach utilized a feature called 3D-CFG that approximates the CFG of each method in an APK by synthesizing all its structural information into a single 4-valued vector called 3D-CFG centroid. This feature has the advantage of being relatively easy to extract, as it only requires one whole pass of the source code, while also being reasonably robust. It works especially well against code transformation techniques that do not affect the program control structure too much and of course also works well with programs that have not been transformed at all. Another advantage is the low computational complexity of the resulting similarity measure, since it only involves vector algebra that can be solved in constant time (with very small constants too).

The downsides of the technique stem from the lack of precision introduced by the features. The CFG of a program is an abstraction of the code and by itself does not lead to good results since many methods share the same CFG structure. A few heuristics have been presented in this chapter in order to quell the amount of false positives resulting from the bare analysis. Two additional vectors are collected during the first pass of the code, counting respectively which system APIs have been used in the methods and how many times these APIs have been called, still in the context of a single method.

The methodology has been tested first, and with good results, against the GENOME dataset as it represents a solid ground truth. We then collected two novel datasets with malware and ransomware and ran our tool to discover whether the label assigned by the main antivirus services was correct. The results of this analysis allowed us to discover that many labels were applied incorrectly. Our tool was able to correctly group together malware samples that were split into different families by commercial antivirus software while also separating some samples that were erroneously grouped into the same family.

StranDroid

For the second approach herein presented, we decided that we could forgo some speed in order to gain precision. To this end, we extract a finer semantic feature called "strand" introduced in [48] and applied to Android for the first time in [109]. A strand is just a backward slice limited to the confines of a single basic block. After each method is decomposed into its strands, it is compared to the other methods and their strands. We developed a method similarity measure by treating methods as sets of strands and using the Jaccard index to understand how similar they are to each other.

We tested the methodology with the same process described for the previous tool. This allowed us to directly compare the two approaches and appreciate how StranDroid has a much lower rate of false positives than R.E.H.A. and allows us to be more precise in the semantic categorization of Android malware.

It is only due to the accessibility of the source code that we were able to craft these two custom code similarity techniques. We relied heavily on domain expertise to understand what made the first approach prone to false positives and again to improve this aspect with the second methodology. In the next chapters we will not have the same luxury.

3 Deep Learning on Compiled Programs

In this chapter we consider the malware classification task as described in the second scenario: a semi-closed system where the source code of the malware samples is not available. This section introduces and motivates the malware classification problem, places it into the context of the second scenario, and outlines our contributions.

The Problem

In scenario 2, the only available information about the program is the compiled binary. The binary cannot be reverse-engineered back into its source code (otherwise this would be scenario 1) so it has to be used as is. In order to classify these program samples, as in the previous chapter, two elements are needed:

1. Some representation(s) of each program
2. A similarity measure between the program representations

This time, since the representations need to be generated from the entire compiled binary, they cannot be syntactic code abstractions as in scenario 1. This is not easy to do, as the long sequences of zeros and ones in binaries are encoded to work only on a specific hardware architecture and rarely represent recognizable syntactic or semantic features on their own.

The result is that in this scenario it is not possible to rely on the same expertise as in the previous one. Classifying malware into different families according to their behaviour requires the design of new techniques for program similarity.

For statically extracted code features, it is clear what their expressive power is and which obfuscations can fool them. With features extracted from binaries, there is less information available and thus less expertise can be applied. In the case of malware classification, this can also be seen as a positive factor. In scenario 1, we were able to extract semantic abstractions from the source code and both their expressive power and their weaknesses were known. The attackers (malware developers) had the same knowledge available and could apply their expertise to circumvent the similarity analysis by using the correct type of obfuscations.

In scenario 2 the defenders have lost most of their ability to directly apply their expertise to the problem, but so have the attackers. Since the obfuscations applied in scenario 1 had clear

and understood effects on the source code, they could be used to specifically target an analysis. On compiled programs, however, code transformations have a less controllable effect. As the programs go through the compilation process, they are transformed and often optimized to the point where some obfuscations can be removed outright. This lack of control by the attackers is something that can be exploited. If we find a proper way to defend (classify malware) in scenario 2, then it would be harder for the attackers to circumvent our analysis.

Considering that in this scenario, the malware samples are vectors of zeros and ones with no clear way to measure the similarity between them, the easiest way to approach a classification task would be to utilize machine learning techniques. This comes with some inherent advantages. First of all, machine learning algorithms basically infer the similarity function without a need to specify it. This is of course invaluable in this scenario as it allows to defer our expertise to the machine directly. Many machine learning algorithms still require expertise to extract the program representation for a classification task in a process called feature extraction. This would be a problem in scenario 2, but thankfully certain neural network architectures do not require such a process and instead learn both the program representation and the similarity function.

The fact that the similarity measure and the representations are learnt and not hard-coded means that attacking this type of classification is inherently harder. The attackers not only have less control over the object of our classification (binary vs. source code), they also do not have a deterministic attack vector like the obfuscations against the abstractions in the previous chapter.

Of course there are also some disadvantages to this approach. In order to properly classify malware into different families the first requirement is a dataset of properly labeled malware. There are three main problems with this requirement, as often these datasets tend to be:

1. Too small
2. Imbalanced
3. Mislabeled

When a dataset is too small it makes it hard to apply certain types of deep learning architectures as they tend to overfit easily. If a dataset is imbalanced, that means that some classes are over or under-represented and thus generate a less reliable classifier. A mislabeled dataset is, of course, the worst offender in this list but it is also the hardest to combat since it requires a lot of human expertise and extensive resources.

In this chapter, we explore the classification of malware according to their binary and hypothesize some solutions to the aforementioned problems.

Motivation

The proliferation of malware increases steadily year after year [99]. In a way this phenomenon is due to most households owning one or more devices that can be attacked. In another way, the financial interests of many users are tied to their internet-bound devices, which then become ideal attack vectors. It is thus imperative to find fast and reliable techniques to identify and fight new malware.

Modern-day malware samples are often heavily protected against reverse engineering and many types of program analysis. For example, malware is frequently modified through obfuscations [122]. These are syntactic code transformations that take a program as input and generate a different program that is more difficult to analyze while still maintaining its functionality [32]. The combination of obfuscations with the code optimization algorithms usually embedded in compilers makes the reverse engineering of malware harder, often slowing down or obstructing parts of the disassembly process [4]. For this reason it is imperative to find techniques that deal with the raw binary instead of relying on higher-level features stemming from reverse engineering attempts.

Obfuscations are widely used in malware [149, 104] and they make it harder to classify emerging malware into their specific families. This task is called malware classification, and it is usually achieved with machine learning techniques. These can range from shallow models that require manual feature engineering before the training process, to deep learning models that can work directly on the raw data. The downside of shallow models is that they require specific domain expertise, which means that time and resources are needed to analyze the samples in the dataset before proceeding to the learning phase. On the upside, the input of human-engineered features usually renders the model and the results easier to interpret. With new malware spreading at an alarming pace, the cost of this manual work is simply unfeasible. Deep learning techniques can automatically extract the features from the dataset samples without the need for time consuming feature engineering or specific domain expertise. This advantage makes deep learning the go-to paradigm for malware classification.

Data Augmentation

One of the drawbacks of deep learning techniques, compared to shallower models, is their tendency to overfit when trained with small datasets [129]. This can be a problem in fields like program analysis, and especially in malware classification, as gathering enough samples with the proper ground-truth takes many resources and even more time.

This problem is also common in other fields, such as in image recognition and image classification [111]. The lack of enough training data is easily solvable in the vision context, as

new data points can be generated from existing ones by applying some semantics-preserving transformations to the images such as rotations, translations in space or selective cropping. This process of generating new data from existing samples is known as data augmentation and it is a staple of deep learning.

Transfer Learning

Another way to mitigate the problems that models can find with few data-points is to reuse a part of an already trained model, usually the part dedicated to feature extraction. These models can be trained with millions of data points and then they can be repurposed for a different problem setting by removing the head of the model (dense layer) and re-training a new head while "freezing" the rest of the network. This process is of course less time expensive, but it also does not incur the problem of needing more data to train, as the majority of the weights are "frozen" and thus do not appear as free variables. This technique is called transfer learning [108].

Our Approach

Clearly most (if not all) data augmentation techniques applied in the vision field cannot be reliably applied to the classification of programs. It is thus imperative to find a suitable alternative.

Since obfuscations generate syntactic variants of programs but maintain their semantics, they can be used as data-augmentation transformations specific to code. This is the main intuition of [95], where 47 small programs with different semantics have been transformed iteratively by applying obfuscations and generating 200 variants each, resulting in a final dataset that contains 9 400 samples divided in 47 classes.

Generating the dataset this way allows for fine control on the size of the dataset itself and its class balance (that can often be a problem in real-world datasets [74]). Another upside of this technique is that it uses obfuscations in a novel way. Obfuscations are no longer a shield that prevents the classification of malware, but they become an integral part of the classification task itself. This process allows us to clearly see if there are some obfuscations that modify the programs in such a way that makes them harder to classify. Applying the same data augmentation process to datasets classified with different architectures can show whether certain obfuscations are more powerful or resilient to the particular learning architecture itself.

In this chapter we generate a dataset of 18 800 obfuscated programs with the aforementioned technique, reaching a size that is roughly double than the one generated in [95] with the same technique. We then train two deep neural network models, a convolutional neural network

(CNN) and a bi-directional long short-term memory (LSTM) and achieve an average accuracy around 93% on the generated dataset. We provide an analysis of the classification errors that highlight the strength of certain obfuscations against the classification effort and the weaknesses of the trained models.

To prove that the techniques and models used in this work are suited for real-life scenarios involving malware we train the CNN and the LSTM on two malware datasets heavily used in literature, the dataset from the The Microsoft Malware Classification Challenge hosted on Kaggle [78] (referred to as the MsM2015 dataset from now on) and the MalImg dataset [101]. These datasets will be thoroughly described in Section 3.2.

We then experiment with transfer learning, taking the features from the models trained on the custom dataset and using them to classify the two aforementioned malware datasets. We verify that it is indeed possible to use the features learnt from the classification of a custom-made dataset of binaries in order to classify a real-world dataset. This has the obvious potential of allowing big networks to be trained on huge datasets and then be reused for smaller and newer datasets at a very low cost. In Section 3.4 we show the results of our experiments with transfer learning by training a model on our custom generated dataset and then verifying that the learnt features can be used to classify the other two datasets. The positive results achieved make us believe that this technique has a lot of potential and could help mitigate the problems encountered when applying deep learning techniques to small malware datasets.

The main contributions of this chapter are:

- a novel data augmentation approach for images extracted from binaries
- the design of two neural network models that can classify obfuscated binaries from their images
- thorough comparison of the approaches
- validation of the models on two state of the art malware datasets
- successful transfer learning experiments between models trained with different datasets

The rest of the chapter is thus structured:

- Section 3.1 contains most background knowledge needed to appreciate the chapter. These include a short introduction to obfuscations and more in-depth descriptions of the two learning models used.
- Section 3.2 introduces the 3 different datasets used to train the models. The focus is on the OBF dataset which is custom-generated and represents our first contribution.

- Section 3.3 shows the structure of the learning models in detail. The main design choices are explained in this section, e.g. classification scores and how the dataset is split into training, validation and testing.

- The results obtained from the experiments are described in Section 3.4. The end is focused on the analysis of the classification errors.

- A compendium of related works can be found in Section 3.5. The papers presented here only deal with the problem of malware classification with the binaries.

- Section 3.6 closes the chapter with a summary of what has been discovered and some considerations about possible future work and limitations.

3.1 Background

This section serves as a primer on a few of the key concepts that we use in our study and are unique to this chapter, such as deep learning, data augmentation and transfer learning. Obfuscations have been properly introduced in previous chapters so what follows is only a small reminder. For a more in-depth discussion the reader should see Section 1.2.3 in the first chapter.

3.1.1 Obfuscations

Obfuscations are program transformations that change the syntax of the program without altering its semantics. They are meant to confuse analyzers and reverse engineers, although the amount of *confusion* added cannot yet be reliably measured [23, 26].

Let *Prog* be the set of all programs. An obfuscation is a program transformation $O : Prog \longrightarrow Prog$ that given a program $P \in Prog$ produces a new program $O(P)$ with the same functionality as P but that *is "unintelligible" in some sense* [11].

In this work, obfuscations are used to design a novel data augmentation technique for machine learning tasks involving images extracted from programs. We then apply the technique to generate the first dataset from which we design both our learning models. The programs in the datasets have been created by running the Tigress C obfuscator [1] on simple C programs and the process is explained in more detail in Section 3.2.

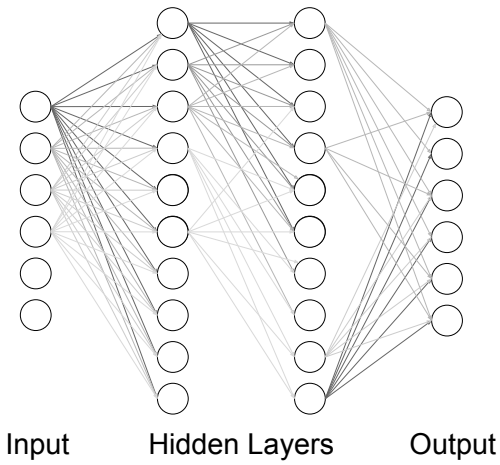

Figure 3.1: Structure of a generic artificial neural network

3.1.2 Artificial Neural Networks and Deep Learning

Artificial neural networks are a class of algorithms for machine learning. Their structure takes heavy inspiration from the way the human brain is configured, or at least the way it was thought to be configured in the 50s when they were first introduced. They are mad up of neurons that communicate with each other by sending "signals" if a predetermined threshold is met. The signals are in the form of real numbers that act as inputs to other connected neurons. Typically the neurons are grouped in "layers", where the first and last layer are reserved for input and output respectively. The layers between the first and last are referred to as "hidden" layers. Figure 3.1 illustrates this configuration.

When a network comprises enough layers it can fall under the umbrella term "Deep Learning". To this day it is unclear how many layers are sufficient for an architecture to be rightfully be considered "deep".

3.1.3 Convolutional Neural Networks [CNN]

CNNs are feed-forward neural network models that take inspiration from the human visual cortex and are widely used in image recognition and classification [115, 83]. Their success in image-related learning tasks has been attributed to their translational invariance [15] which

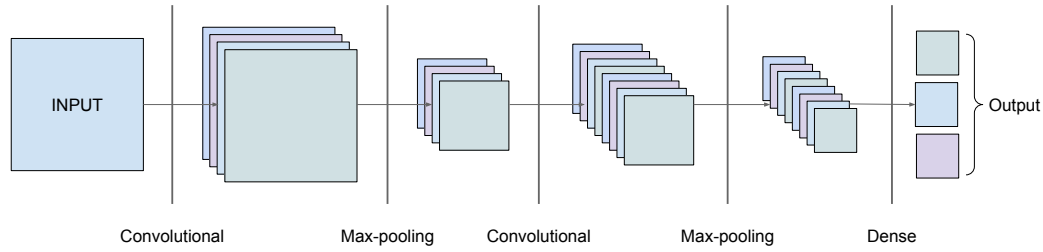

Figure 3.2: Structure of a convolutional neural network

allows them to be used to solve problems such as face recognition [83] or handwritten character recognition [84]. In general, CNNs are very good at extracting spatial features from the data.

At their core, CNNs consist of at least one convolutional layer connected to a dense layer. The convolutional layer operates a convolution on the input in order to isolate the features that the network deems important during the training phase. This process greatly reduces the number of weights needed since the input images are condensed into a smaller feature set. The convolutional network usually is connected to a max pooling layer that contributes further to the reduced size of the features detected by combining the values of multiple neurons into a single value, usually the maximum value (thus max-pooling) or the average value. After a combination of the aforementioned layers the CNN architecture is completed by at least one fully connected (or dense) layer. This layer is responsible for the classification process itself and acts as a multi-layer perceptron that takes as input the features extracted from the previous layers. To prevent overfitting, regularization techniques are commonly used and can be applied anywhere in the network. This reduces the possibility of the network purely memorizing the training data, thus allowing for better generalization [81].

3.1.4 Recurrent Neural Network and Long Short Term Memory [RNN and LSTM]

As stated in the previous subsection, CNNs excel at extracting spatial features from data This is invaluable when classifying or generally working with natural images, but images extracted from code reveal different types of patterns altogether and therefore comprise a different learning problem [95].

The sequential nature of code, and thus of compiled programs, indicates that a learning model that works well with sequential data can be beneficial. Recurrent neural networks

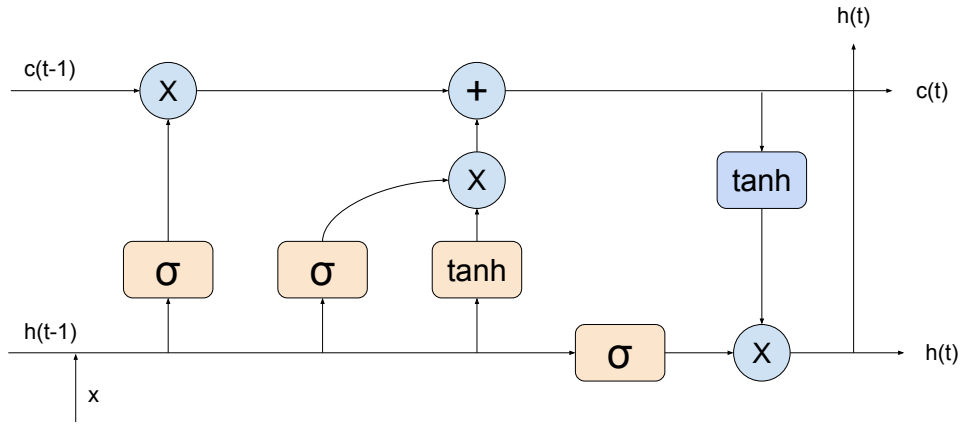

Figure 3.3: Structure of a LSTM cell

(RNNs), which are designed to process sequences of data of arbitrary length from beginning to end, is one example.

Generally, the hidden state h_t of a RNN depends on the output of the previous state h_{t-1} and so on, which means that the state h_t contains a distributed representation of all the tokens observed in the sequence up to the time step t. This way, the network can generate probabilistic dependencies from previously seen data. One common pitfall of this structure is that the dependencies between tokens that are far apart from each other in the sequence are hard to manage. This stems from the nature of the gradient descent algorithm over time steps, which makes the components either decay or grow exponentially [16, 72].

Long short-term memory was developed to mitigate precisely this problem [72]. By storing information at particular timesteps, LSTMs provide a mechanism to specify when to remember certain information, and more importantly, when to forget it. A representation of a single LSTM cell can be seen in Figure 3.3, where σ represents the sigmoid layer, h is the hidden state and c is the memory state. The first sigmoid layer takes in input the previous hidden state of the LSTM combined with the new input read and passes it to the "forget gate" which will modify the memorized state accordingly. The next two layers combine to decide how much information from the current hidden state should be memorized for the next iteration. The vertical arrow with h(t) represents the output of the cell at time t.

The LSTM used in this work is bidirectional, meaning that the input stream is read in both directions at once. This is achieved with two LSTM models, one forward and one backward,

which read the input in the two directions and then combine the resulting vectors to produce a unified result.

In Section 3.3 we outline the specific architecture of our neural networks.

3.1.5 Data Augmentation

Deep neural networks tend to overfit because the number of parameters that are learnt (the weights in the network) can exceed the size of most datasets. This makes it possible for the network to memorize the data and generate a classifier that works perfectly with the training set but that is sub-par with new data. For this reason, another way to avoid overfitting is generally to select huge datasets.

This is not always possible, thus there needs to be a way to generate new samples that fit in the dataset while being different enough from the existing samples. This process is called *data augmentation*.

Let us take image classification as a working example. If we are using a deep neural network to recognize which pictures belong to a specific class but our dataset is too small, we can use many data augmentation techniques in order to enlarge the dataset.

Assume we are trying to recognize faces and classify them with the name of their owner, but the writer of this thesis is shy and only allows one flattering picture to be part of the public dataset. Then there are a number of transformations that can be applied to the image to create variants that would still fit in the right class:

1. rotate
2. zoom in
3. flip
4. add mask

It is imperative that these transformations preserve the meaning of the image (its **semantics**) otherwise we are generating new samples that would not belong to the right class. These transformations can also be applied iteratively and stacked upon each other, which means we can generate many images from a single starting sample. A few examples can be seen in Figure 3.4.

In this work we are indeed working with images but we cannot use the above transformations for data augmentation. In Section 3.2 we explain why.

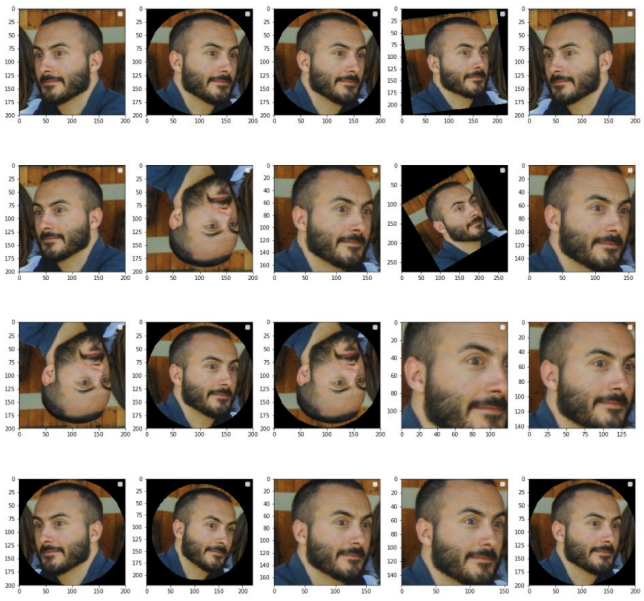

Figure 3.4: Some common data augmentation techniques for images

3.1.6 Transfer Learning

As mentioned in previous sections, a big problem of deep learning tasks is finding datasets that are big enough to allow learning without overfitting. Training deep networks is also very time consuming and requires adequate computing resources. Transfer learning can alleviate both these problems [112], by leveraging the information learnt from one task in order to solve a different task. Some common applications of transfer learning are of course in vision, where huge models trained from the ImageNet dataset [49] can be re-purposed for new image classification tasks.

To illustrate how transfer learning works we can imagine a typical CNN as described earlier, trained on a dataset of images for classification. The convolutional layers are tasked with extracting the features from the inputs while the dense layers towards the output provide the actual classification of the dataset. One way to properly perform transfer learning is to remove the dense layer from the CNN and "freeze" the convolutional layers (meaning, the gradient descent will not modify their weights). The convolutional layers contain information on what features to focus on, for example they allow to detect edges and to combine these edges into a

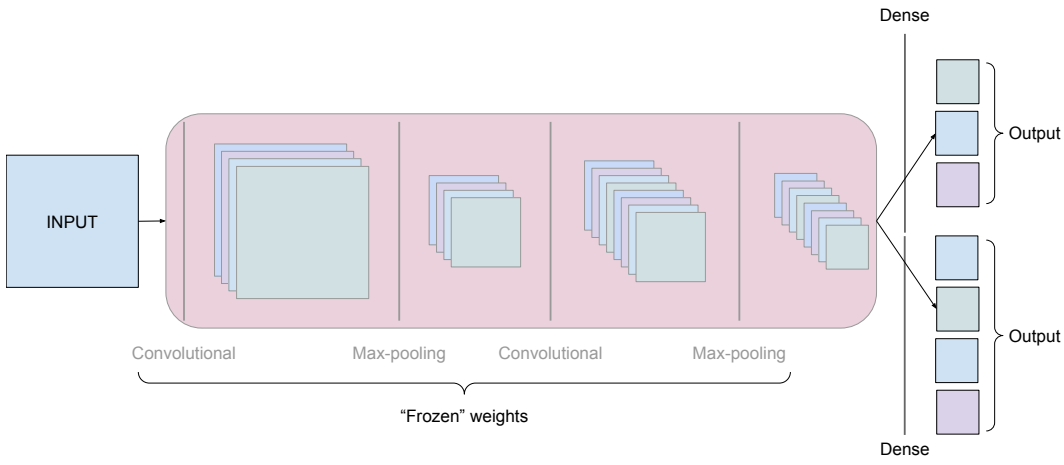

Figure 3.5: Transfer learning on a CNN

higher-level feature. The general structure of transfer learning on a generic CNN can be seen in Figure 3.5.

At this point a new dense layer can be applied with the proper output shape (the number of classes for the new learning problem) and the network can be trained on the new dataset. Since the convolutional layers do not have to be trained again, the learning process is sped up considerably. The size of the dataset is also less important, since the number of free variables (weights) that can be modified is much smaller than those in the original network, thus making overfitting less of a problem. The convolutional layers of the CNN will provide the necessary features for the classification task at virtually no cost as the feed-forward part of the network is relatively inexpensive.

In this work we use transfer learning between different datasets of images generated from compiled programs. This allows us to gauge if the features extracted from one dataset can be used to classify the others. The applications of this can lead to many interesting opportunities that will be discussed in future sections.

3.1.7 Bicubic Interpolation

In order to resize the images in our custom dataset we use bicubic interpolation as implemented in OpenCV [21]. This algorithm infers the intensity of an unknown pixel by applying bicubic interpolation [79] over the neighboring 16 pixels. It is often used in place of its bilinear

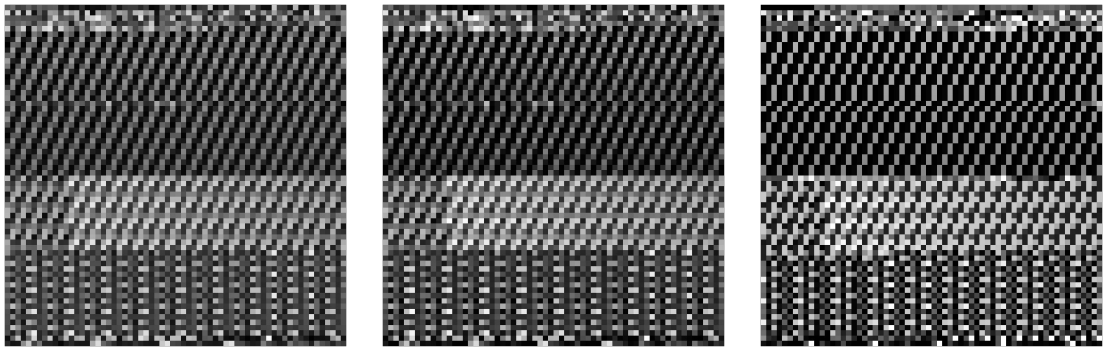

Figure 3.6: Three examples of interpolation in CV2, all on a sample of class theme_park obfuscated with Split, InitImplicitFlow and EncodeLiterals

counterpart when the quality of the resulting image is more important than the computational resources used, which fits our scenario.

In Figure 3.6 we show three different types of interpolation offered by OpenCV. On the left the bicubic interpolation method used in this work, followed by a linear and a nearest neighbor interpolation.

3.2 Datasets

This section describes the datasets used in this work. We start from the novel data augmentation technique developed in this thesis, which we then use to generate of our custom OBF dataset. Afterwards we summarize the peculiarities of the two malware datasets, MalImg and MsM2015.

3.2.1 OBF Dataset

We start with 32 programs that are downloaded from a beginners programming website [113]. These are very simple programs that average 23 lines and they were selected so that the full images extracted from the programs would fit the models in [95]. In Figure 3.7 we show the programs, their names should be well descriptive of what they compute.

Small programs work well enough to illustrate the methodology but bigger programs can lend some validity. For this reason 15 more programs have been added to the original dataset, all taken from solutions of the Google Code Jam. These programs have been selected randomly

1. armstrong_n.c
2. calculator.c
3. char_frequency.c
4. count_digits.c
5. count_vowels.c
6. factorial.c
7. factorial_rec.c
8. factors.c
9. fib_1.c
10. fib_2.c
11. gcd.c
12. gcd_rec.c
13. hello_world.c
14. lcm.c
15. leap_year.c
16. n_is_palindrome.c
17. n_is_prime.c
18. n_is_sum_of_primes.c
19. positive_or_negative.c
20. power_n.c
21. prime_n_intervals.c
22. pyramid.c
23. quotient_remainder.c
24. remove_char.c
25. reverse_integer.c
26. store_struct.c
27. strcat.c
28. strcpy.c
29. stringsort.c
30. strlen.c
31. sum.c
32. times_table.c

Figure 3.7: The initial dataset

and represent a more real-world scenario with source code that spans between 28 lines and 200. The list of programs taken from Google Code Jam is in Figure 3.8.

These programs are all solutions to problems given during Google Code Jam in different rounds of the competition and spanning multiple years. Therefore they each form their own unique semantic equivalence class. Additional attention has been given to the uniqueness of the authors in order to avoid possible code reuse which could potentially inject uncertainty in the classification process.

Data Augmentation for Programs

Naturally, 47 programs do not constitute a dataset that easily lends itself to classification, especially because all 47 belong to different equivalence classes. This is be the perfect scenario in which data augmentation can come in handy.

Since we are classifying images it could be argued that we can apply the same data augmentation techniques listed in Section 3.1.5. This has some downsides. Rotating, tilting and zooming into a picture of a face does not alter the human perception of the image. It is still the same face, although slightly adjusted. Its meaning (its **semantics**) is preserved.

Doing the same to the image extracted from a binary leads to catastrophic results. For

1. alien_language.c
2. bot_trust.c
3. candy_splitting.c
4. fair_warning.c
5. fly_swatter.c
6. magicka.c
7. minimum_scalar_product.c
8. multibase_happiness.c
9. rotate.c
10. saving_the_universe.c
11. snapper_chain.c
12. theme_park.c
13. train_time_table.c
14. watersheds.c
15. welcome_to_codejam.c

Figure 3.8: Google Code Jam solutions

example, zooming into the image will remove some data from the program, rotating the image will re-order the bits and the same can be said when flipping it. Removing bits from a program or rearranging them will inevitably generate a different program, or possibly just a new sequence of zeros and ones that has lost all its meaning as a program. None of the transformation listed in Section 3.1.5 can preserve the semantics of the original program, they all generate a new version that does not fit in the same class as the original sample. This is illustrated in Figure 3.9.

For this reason we need to find a new data augmentation technique that can work on images extracted from programs.

Our Approach

We have introduced obfuscations in Chapter 1, where we explain that they are program transformations that are meant to make the analysis "harder" in some way. In this section obfuscations are not to be feared, as they provide the semantics-preserving transformations that we need in order to enlarge our starting dataset.

The idea is very simple, we select 8 obfuscations and apply them iteratively on the 47 files of the original dataset. In order to apply the obfuscations we leverage the Tigress C obfuscator, which is a tool that operates at the C source code level, leveraging the CIL [102] system for the transformations. The tool offers several syntactic transformations that can be stacked together in order to improve the efficiency of the obfuscation. In our experiments we consider the following obfuscations:

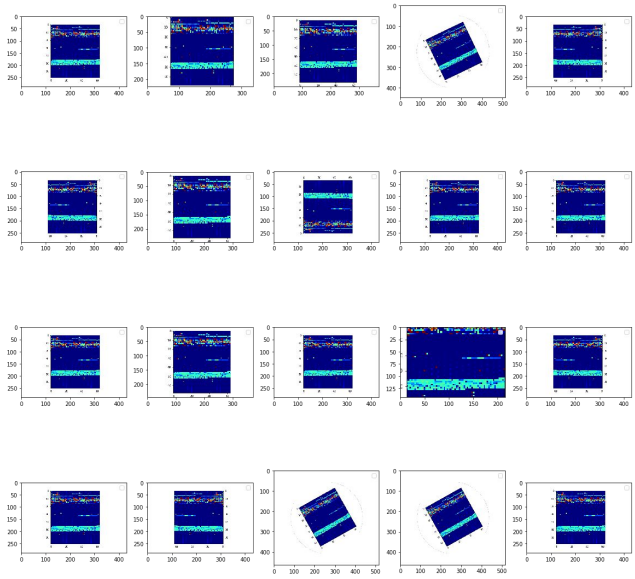

Figure 3.9: Common data augmentation techniques for images applied to a program

- *Flatten*: implements code flattening by completely removing the original control flow structure of the program and replacing it with a switch statement [136]. The switch control variable is then modified dynamically to decide the order in which the basic blocks are executed, which produces a "flattened" control flow. An interesting feature of this transformation is that it allows the basic blocks to be organized randomly in memory.
- *Split*: splits a function into two separated functions that perform the same semantic function when combined
- *RandomFuns*: inserts a random function into the code
- *EncodeArithmetics*: encodes any arithmetic operation into a semantically equivalent but syntactically harder to decipher operation. For example the expression $z = x + y + w$ can be replaced by $z = (((x^y) + ((x \wedge y) << 1)) \vee w) + ((x^y) + ((x \wedge y) << 1)) \wedge w)$. There are many of these transformation (all taken from the book Hacker's Delight [139]) and they are chosen randomly at each run.
- *EncodeLiterals*: initializes literal integers and strings with new functions

- *InitOpaque*: adds specific data structures that can be used to insert opaque predicates
- *InitEntropy*: adds new variables in order to collect entropy
- *InitImplicitFlow*: initializes handlers for implicit flow analysis

These obfuscations have been selected because they represent different types of syntactic transformations. For example, the first 2 deal with structural transformations of the control flow graph while *EncodeArithmetics* and *EncodeLiterals* perform a more symbolic kind of transformation. *RandomFuns* has been added specifically because it adds new functions to the code, preserving the basic semantics of the original program but at the same time augmenting it. This is akin to what happens in some malware, where the payload is consistent between different samples of a family but the surrounding program can have different semantics [50]. On top of these obfuscations, Tigress automatically generates random variable names and function names for every generated variant.x

All 8 obfuscations are applied to every sample in the initial dataset, generating 376 ($47 * 8$) new syntactic variants of the base programs. Then the process is repeated, but this time the new programs are each obfuscated with a new transformation, thus generating 2 632 ($376 * 7$) variants. The reason why only 7 obfuscations can be selected is that we do not allow for repeated obfuscations. After all, most transformations do not result in a new interesting variant when applied twice. For example, *Flatten* applied to a function that has already been flattened would result in exactly the same program, as it is idempotent. We iterate the algorithm 4 times, reaching a final size of 18 800 samples.

After generating the dataset we transform each program into an image. Every sample of the dataset is imported as a raw file in a Python script and then it is converted using the `numpy` package in order to represent it as a list of hexadecimal numbers (16^4 max value). This list is then translated into a matrix with width 64, following the steps taken in [95]. At this point all programs are represented by matrices that have the same width but varying length.

In [95], three different resizing techniques were considered in order to have a dataset of homogenously sized images but all methods either lost too much information (by cropping) or added too much noise (by padding).

For this work all images are resized into two main shapes (64×64 and 256×64) using the standard functions in the OpenCV library [21] with bicubic interpolation over a 4x4 pixel grid. We observed that this method is better suited for this task and the results confirm the observation.

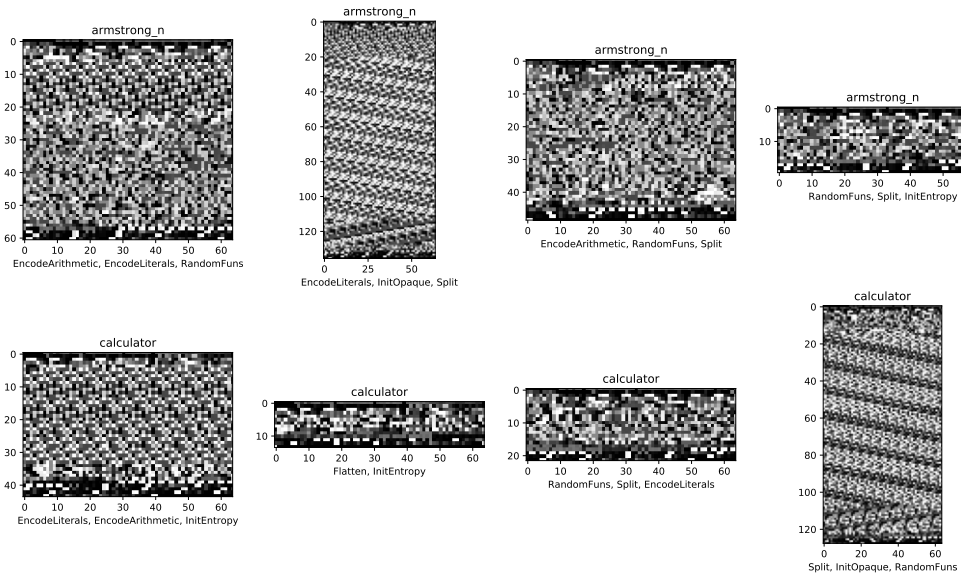

Figure 3.10: 8 samples from the classes armstrong_n and calculator of OBF

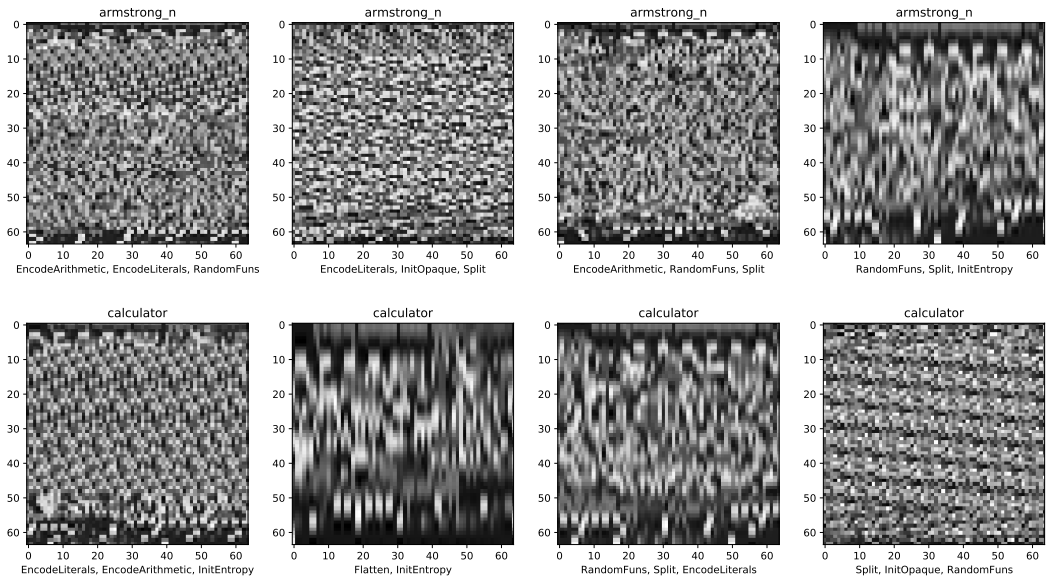

Figure 3.11: 8 samples from the classes ‘armstrong_n’ and ‘calculator’ of OBF, interpolated

Advantages

One of the advantages of generating such a dataset from scratch is that the resulting classes will be as balanced as needed. Some datasets in literature are heavily imbalanced, for example, both the MalImg dataset (used in this chapter) and the GENOME [156] dataset for Android malware (see previous chapter) suffer from this. It is expected to find such imbalance in datasets found in the wild, but this leads to problems in the classification process, or more properly, in the accuracy measurement.

For instance, let us imagine a dataset of malware that contains 25 classes distributed in a balanced way. Given a random sampling of the test set, a simple classifier could always guess one of the classes and get it right 1/25 of the time, thus obtaining around 4% accuracy. This is evidently a bad result and would alert the researchers to the inherent incapability of the learning model. On the other hand, the MalImg dataset contains 9 458 samples divided into 25 classes and the largest class has 2 949 elements, with the second largest one closing in at 1 591. A simple classifier could learn to distinguish between these 2 big classes, then guess all samples to be contained in them and still reach almost 50% accuracy. Of course this is not a good result by itself, but it can trick anyone into thinking that the model is actually learning something from the dataset when that is certainly not the case.

Another advantage of our dataset generation technique is that we can ensure that the obfuscations applied will cause pervasive structural changes in the binaries, generating visually distinctive images that belong in the same class. This is not always the case, especially for datasets that have been collected in the wild, since many code transformations do not act on the global structure of the executable file. This can be easily seen in Figure 3.14, which shows samples from the same classes looking very similar. We will expand on this later.

The dataset generated from the simple programs iteratively obfuscated will be called OBF from now on. In Fig. 3.10 we show 8 programs taken randomly from two classes, '`armstrong_n`' and '`calculator`'. At the bottom of each figure there is a list of all the obfuscations applied to the specific sample. It should be evident that the syntactic transformations of the source code also result in visually distinctive binaries, and thus the images extracted from them also have distinctive features.

The images generated vary in size, from 24x64 for the smallest sample to 9800x64 for the largest one. The application of resizing with bicubic interpolation brings them to one coherent size. In Fig. 3.11 we show the same samples as in Fig. 3.10 but with interpolation applied to them. Even at first glance, the difference in the images caused by the obfuscations is still noticeable, if not more so.

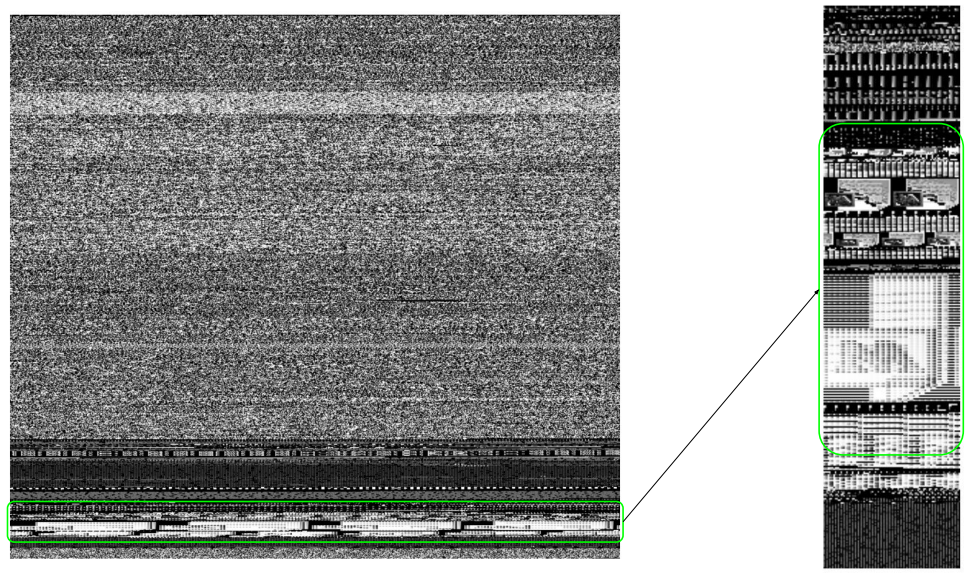

Figure 3.12: Left: the original version of a sample from the family 'Autorun.K' of MalImg. Right: its resized and zoomed in version, showing the embedded icons

3.2.2 Microsoft Malware Dataset [MsM2015]

The MsM2015 dataset consists of more than 20 000 malware samples for Windows, collected by the Microsoft corporation and distributed by Kaggle for a competition in 2015 [118]. Each sample in the dataset belongs to one of 9 known malware families. The dataset provides each datapoint as the hexadecimal representation of its executable and as a collection of metadata generated by IDA pro. The total size of the dataset is around 500GB of data, making it less than nontrivial to work with.

For this study we utilized only the byte representation of the samples, loading it in python using the same technique applied in [95] and representing every sample as an image with a set width of 64.

Each sample is then resized to either 64x64 or 256x64 using bicubic interpolation, generating a more manageable dataset. Since images generated from the MsM2015 dataset are sourced from real-life malware, they come in varying sizes that far surpass those of the OBF dataset. As a result, the interpolation process removes even more detail than it did in the first dataset.

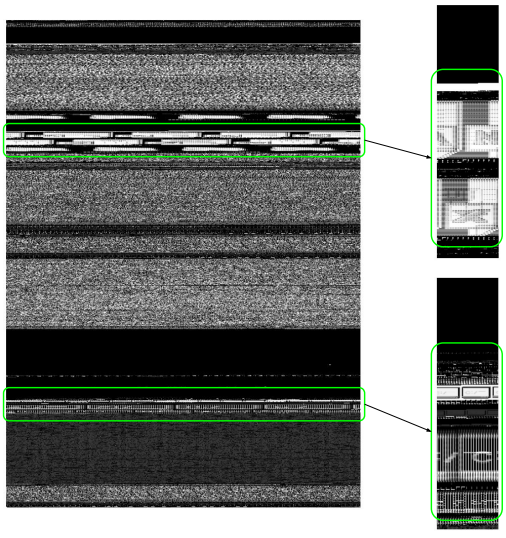

Figure 3.13: Left: the original version of a sample from the family 'VB.AT' of MalImg. Right: its zoomed in version, showing the embedded icons

3.2.3 MalImg Dataset

The MalImg dataset is a collection of 9 458 malware samples divided into 25 families. The main characteristic of this dataset is that the malware samples are not provided directly, but rather as their images as they appear on disk. In a similar way to the work in [95], the bytes of the executable files are trivially mapped to floats, which will then be interpreted as pixel values of grayscale images.

As anticipated, the classes in the dataset are heavily imbalanced: the biggest one ('`Allaple.A`') contains 2 949 samples, while the smallest one contains only 80 samples.

In Figure 3.14 we show 8 random samples taken from two classes of the dataset. It should be readily evident that the images from each class have distinctive patterns that allow us to tell samples from one family apart from samples in the other family. This is again true for samples processed by bicubic interpolation, as shown in Figure 3.15. The observation that different families of malware had a distinctive '`look`' is part of what has driven the original work in [101]. This is not always the case with binaries. Being able to tell at a glance that two executable files belong to the same class is a luxury that we do not have in the OBF dataset. Figure 3.10 shows this clearly, and in fact, the second sample in the first family appears most similar to the fourth sample in the second family.

Windows binaries also have the peculiarity of allowing embedded icons (which are unsupported so far in ELF files), and some families in the MalImg dataset can be easily recognized by the use (and positions) of such icons. However, it is not easy to do so in the original format of the images.

For example, in the leftmost part of Figure 3.13 is shown a sample from the family 'Autorun.K'. Its size has been set to 683×768 by the authors of the original paper and it is based on empirical observations. It is not clear which observations these might be but the result is that most images in the dataset tend to be somewhat square.

At the bottom 10% of the image a clear repeating pattern can be spotted that stands out from the previous white-noise that is typical of executable files. If the image is resized to 5464×96, then it becomes too tall to be printed in its entirety in this thesis. Zooming into the image, roughly showing only the rows between indexes 4800 and 5200, we can generate what is shown in the rightmost part of Figure 3.13. From this new perspective it is evident that the malware was trying to pass as an Office Words file. This simple resizing trick can be done to all samples in the family with roughly the same results.

Another interesting example of this is the family 'VB.AT', whose sample is shown in Figure 3.12. In its original shape it is hard to discern anything except the clear demarcation of different areas full of white noise. When resized with the same technique as above (the width set to 96 is key here) then many patterns suddenly appear in the binary. The top pattern makes it seem like the malware sample is not sure whether it is pretending to be a Words doc or an Excel file, while the bottom pattern recalls a windows command-line.

The fact that these icons are so distinctive within samples of certain families raises the question whether they substantially simplify the classification task. This is certainly not the case with the families in the OBF dataset, which might explain its lower average classification accuracy.

3.3 Experimental Setup

In this section we describe the details of our experiments, from reproducibility and generalizability concerns to model architecture and considerations on how to best present our results.

3.3.1 Preprocessing

Throughout the rest of this chapter the size of the images considered for classification has been set to 64x64 and 256x64. This is done to optimize the training time and the classification

img size	timesteps	features	accuracy (%)	time
32x128	32	128	92.4	279s
128x32	128	32	92.6	743s
64x64	64	64	**93.4**	392s
64x128	64	128	92.8	402s
128x64	128	64	93.2	712s
128x128	128	128	93.1	641s
32x256	32	256	92.9	364s
256x32	256	32	92.4	1 516s
64x256	64	256	92.2	461s
256x64	256	64	92.1	1 251s
128x256	128	256	91.9	673s
256x128	256	128	92.9	1 341s

Table 3.1: Accuracies of the bidirectional LSTM when trained with different timesteps and image shapes

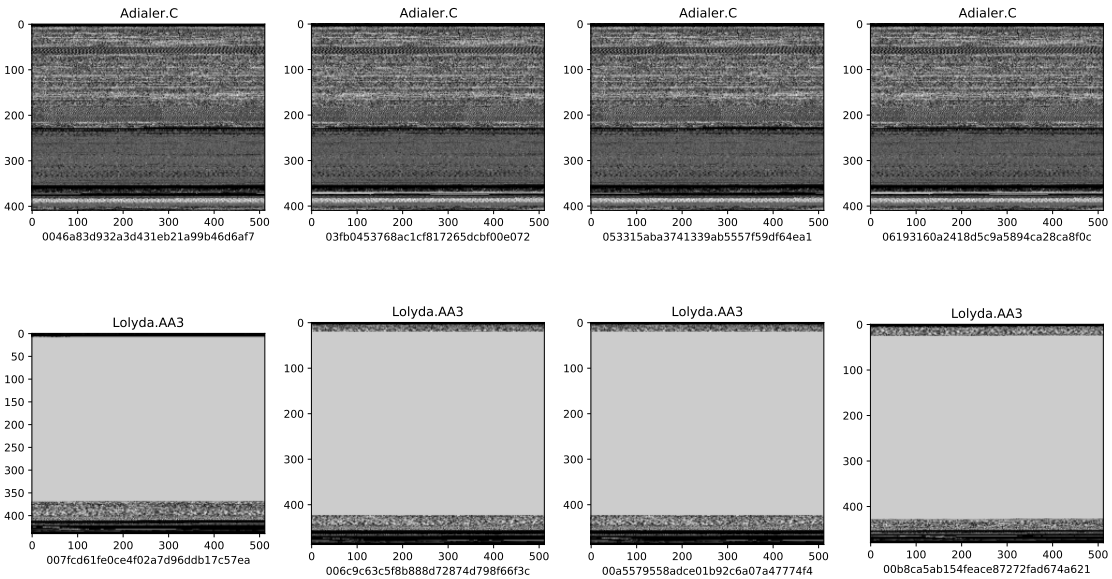

Figure 3.14: 8 samples from the classes '`adialer`' and '`Lolyda`' of MalImg

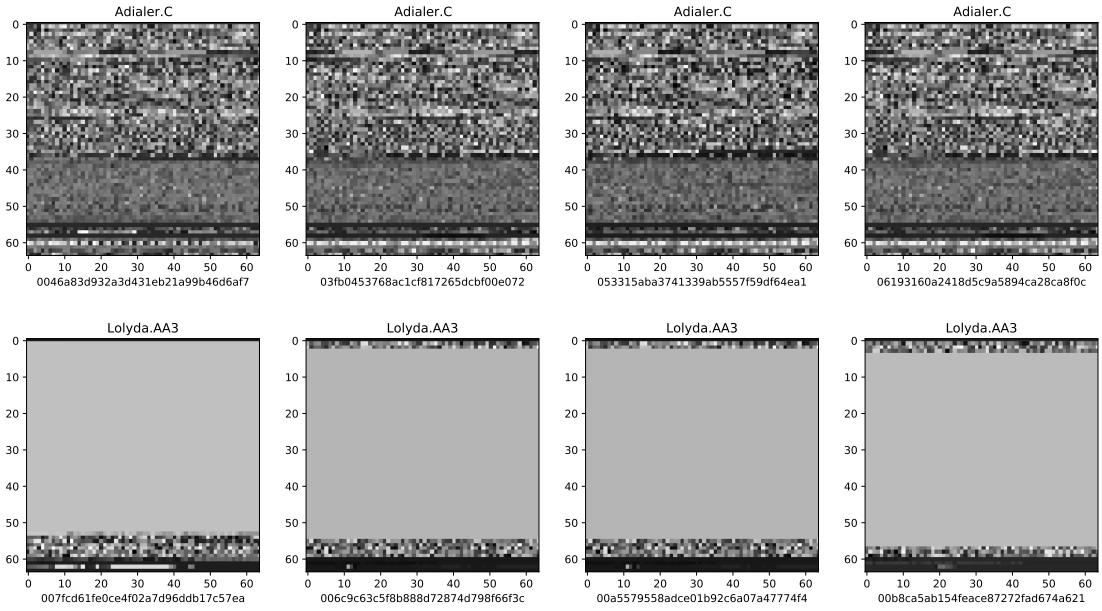

Figure 3.15: 8 samples from the classes '`adialer`' and '`Lolyda`' of MalImg, interpolated

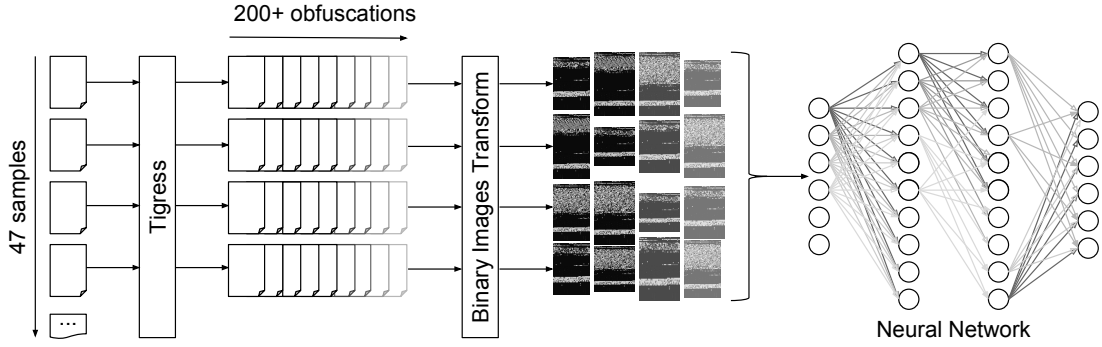

Figure 3.16: Workflow of the OBF dataset generation

accuracy for both models. The training accuracy for various image sizes (resulting in multiple time step values and features) can be seen in Table 3.1. As made evident by the experimental results, the two selected values are among the optimal ones according to classification accuracy. These experiments were done on the OBF dataset but empirical observations led us to keep the same fixed image size for the classification of all the datasets.

Our choice of setting the width to 64 is made mainly because it is a power of 2 and represents the maximum bit size that an instruction can take in a 64 bit system such as the one we work on. This ultimately proved to be a good choice, since this parameter affects the learning process.

In order to investigate whether the width of the images affect the classification accuracy in a simple image recognition problem, we held some experiments with the MNIST dataset [85] in our system. The differences in the prediction scores are not significant and the CNN is able to recognize the digits even if they are scrambled in a way that no human could recognize them anymore (see Table 3.2 and Fig. 3.17).

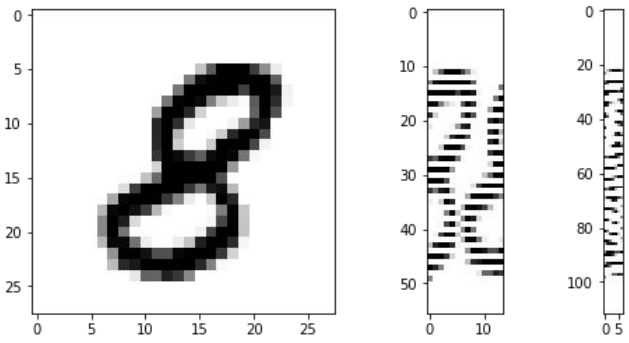

Figure 3.17: A digit from the MNIST dataset with varying widths: 28x28, 56x14 and 112x7

In our initial investigation published in [95] we perform the same tests on our dataset of images extracted from obfuscated binaries, with the distinction that the samples have not been processed with bicubic interpolation and the resizing is done by either padding or slicing. The results reveal a negligible difference in both train and test accuracy only when the width of the binary image is changed from 64 to 32. While when considering other width values (still powers of 2) the classification accuracy drops, and it reaches 0.04 in the test set with a set width of 256, an accuracy that is slightly better than random guessing (see Table 3.3). This is one of the first observations that leads us to believe that our classification problem is vastly more complex than simple digits recognition.

Table 3.2: Width change in MNIST

MNIST	28x28	56x14	112x7	14x56	7x112
train	0.98	0.98	0.96	0.98	0.96
test	0.98	0.97	0.96	0.97	0.96

Table 3.3: Width change in `SimpleObfuscatedDataset`

OBF	?x32	?x64	?x128	?x256
train	0.98	0.97	0.43	0.10
test	0.89	0.91	0.33	0.04

3.3.2 Training, Validation and Testing Sets

In order to guarantee a degree of generalization we split each dataset in training, validation and testing sets. During the learning stages we experimented with different ratios for the splits with the goal of maximizing the fairness of the generalization [116] but also to preserve as many samples as possible for the training stage. The ratio of the hold-out test set for all of the experiments in this chapter is 0.2, meaning that 20% of the dataset is reserved for the testing phase that happens after the training and it is not used to tweak any of the parameters. Of the remaining samples in the dataset, 10% is kept for the validation set, which is used at every training epoch to calculate the validation loss, while the rest is the training set with which the weights of the network will be trained.

All the results from the subsequent sections have been achieved using the hold-out test set, averaged after running the experiments up to 20 times. This should ensure the generalizability of the approach, as any unfortunate division of the dataset should be prevented by the average over multiple random splits.

3.3.3 Models

The two models considered have been fully coded in Python with Keras [107] and deployed on Google Colab notebooks freely available online to simplify the reproducibility of the study. The untrained models for these experiments can be found at [97] but we cannot provide a download link for the MsM2015 and the MalImg datasets as they are made available by each respective owner on their own terms. The OBF dataset is already available in the notebooks, along with methods that can adapt the other two datasets to the models.

In the following we provide a technical description of the models implemented.

classes	precision	recall	f1-score	support
alien_lang	0.99	0.99	0.99	80.0
armstrong_n	0.84	0.84	0.84	80.0
bot_trust	0.98	0.99	0.98	80.0
calculator	0.89	0.94	0.91	80.0
candy_split	0.93	0.82	0.87	80.0
char_freq	0.99	0.96	0.97	80.0
count_digits	0.82	0.88	0.85	80.0
count_vowels	0.90	0.88	0.89	80.0
factorial	0.85	0.90	0.88	78.0
factorial_rec	0.94	0.98	0.96	82.0
factors	0.96	0.88	0.92	80.0
fair_warn	1.00	0.99	0.99	80.0
fib_1	0.75	0.82	0.79	80.0
fib_2	0.79	0.85	0.82	80.0
fly_swatter	0.95	0.96	0.96	80.0
gcd	0.83	0.78	0.80	81.0
gcd_rec	0.87	0.82	0.84	79.0
hello_world	0.99	0.95	0.97	80.0
lcm	0.78	0.82	0.80	80.0
leap_year	0.87	0.84	0.85	80.0
magicka	0.99	1.00	0.99	80.0
min_product	0.97	0.90	0.94	80.0
multibase_hap	1.00	0.98	0.99	80.0
n_palindrome	0.89	0.82	0.86	80.0
n_is_prime	0.84	0.88	0.86	80.0
n_sum_of_p	0.92	0.88	0.90	80.0
pos_or_neg	0.91	0.91	0.91	80.0
power_n	0.70	0.84	0.76	80.0
prime_n	0.84	0.79	0.81	80.0
pyramid	0.87	0.91	0.89	80.0
quot_remainder	0.79	0.85	0.82	80.0
remove_char	0.95	1.00	0.98	80.0
reverse_int	0.91	0.78	0.84	80.0
rotate	0.93	0.95	0.94	80.0
saving_univ	1.00	0.99	0.99	80.0
snapper_chain	0.98	0.99	0.98	80.0
store_struct	0.94	0.96	0.95	80.0
strcat	0.81	0.80	0.81	80.0
strcpy	0.89	0.88	0.88	80.0
stringsort	0.94	0.99	0.96	80.0
strlen	0.91	0.92	0.92	80.0
sum	0.86	0.81	0.83	80.0
theme_park	0.98	1.00	0.99	80.0
times_table	0.86	0.80	0.83	80.0
train_t_tab	0.99	0.99	0.99	80.0
watersheds	1.00	1.00	1.00	80.0
welcome_cjam	0.99	1.00	0.99	80.0

Table 3.4: Classification scores for the LSTM on OBF

CNN We implement a convolutional neural network in Keras, an almost direct translation of the one employed in [95] (which was built on barebones TensorFlow). The network has a base of two convolutional layers, each followed by a max pooling layer. Each convolutional layer has a kernel size of 5 and employs a rectified linear unit (relu) as the activation function, with 32 filters in the first layer and 64 in the second. Both max pooling layers have a size of 2x2 with padding set to '`same`' in order to avoid shrinking the input image. The head of the network is a dense layer followed by a dropout layer that flows in the last dense layer. The dropout layer is a form of regularization which, as anticipated in Section 3.1, prevents the network from memorizing the training set and thus hopefully allows it to better generalize. With a value set to 0.2, the drop out layer randomly selects 20% of the neurons of the previous layer and sets their activation to zero. The network is then trained via gradient descent through the Adam optimizer with default values for both the learning rate and epsilon. The loss measure selected is categorical cross-entropy.

Training is guided by the accuracy measured on the validation set at each epoch. The network keeps training until a set number (*patience*) of epochs have passed since the validation accuracy has improved (*patience* is set to 40). After these epochs have passed without improvement, the network will stop updating its weights and restore the ones that achieved the best validation accuracy 40 epochs earlier. This is generally a good technique to avoid overfitting, since many deep models can train on the training set until they basically commit it to memory. After this point it is almost impossible for the network to generalize on unseen data points. For this reason, once the validation accuracy stops increasing for multiple epochs it is better to halt the learning process, as it is assumed that the network is starting to lose generalization.

This network has also been thoroughly tested on the MNIST dataset of hand-written digits [85], where it reaches accuracy values up to 99%. The purpose of testing it on a dataset with simple images is to show that, while the approach is meant to work on any image recognition task, it can generalize to arguably more difficult tasks. Further, according to the experiments in [95], it is the smallest network that can learn and generalize on the OBF dataset. It is important to contain the size of the network because bigger models tend to overfit and generally underperform unless they are trained with huge datasets [67]. Bigger networks also require far more computing power and RAM, which is one of the reasons why many recent seminal works that highly touted in the image classification field are increasingly hard to reproduce outside the big laboratories where they originate.

We tested with many image sizes, taking the approach shown in [95] but using interpolation to resize the images instead of cropping and zero-padding. Among the various sizes we only show the results with square images of size 64 pixels by 64 pixels and 256 by 64 (64x64 and 256x64 in the tables) as they are the most significant.

	CNN		LSTM	
	64x64	256x64	64x64	256x64
OBF	90.9%	92.3%	**93.4%**	92.6%
MsM2015	90.8%	92%	**93.8%**	90%
MalImg	98.1%	98.3%	**98.5%**	98.2%

Table 3.5: Average classification accuracy

LSTM The long short-term memory has also been implemented in Keras. It consists of two recurrent units, specifically bi-directional LSTM units with 141 and 94 units respectively. In order to quell overfitting the network has been equipped with a patience of 30 epochs, meaning that the model will stop learning 30 epochs after the validation loss has stopped decreasing.

The LSTM model clearly performs better than the CNN in all 3 datasets considered, given the initial input size of 64x64. The advantage of the LSTM model is not only in the improved accuracy but also in its considerably smaller size, although it results in a higher training time. An interesting aspect of the LSTM models is that they do not perform better with bigger images, in fact, the accuracy drops quite a bit. A way to slighly mitigate this effect is to encode the images as 64x256 (short and fat instead of tall and thin). This likely has something to do with the fact that the increase in the height of the image corresponds to an equal increase in the timesteps of the recurrent network. Keeping the number of timesteps to 64 maintains the performance of the network but the improvement on the classification results is not particularly noticeable.

We show the average accuracy for all models and datasets in Table 3.5, while a graph of the validation accuracy during training is shown in Figure 3.18 for the LSTM and CNN trained on images of size 256x64.

3.3.4 Classification Scores

The main results in this chapter are reported as measure of test set accuracy, that being the percentage of samples classified into the right class. Due to the unbalanced nature of the MalImg dataset we also report the precision, recall and f1 measure for each class.

Given the true positives as T_P and the false positives as F_P we can define precision as:

$$Precision = \frac{T_P}{T_P + F_P} \tag{3.1}$$

Figure 3.18: Validation accuracy of the CNN and LSTM models on OBF dataset, with sizes 256x64 pixels

classes	precision	recall	f1-score	support
Obfuscator.ACY	0.92	0.86	0.89	252.0
Simda	0.11	0.12	0.12	8.0
Ramnit	0.85	0.87	0.86	314.0
Vundo	0.88	0.82	0.85	89.0
Gatak	0.86	0.84	0.85	202.0
Lollipop	0.86	0.92	0.89	509.0
Kelihos_ver1	0.98	0.93	0.95	88.0
Tracur	0.69	0.65	0.67	145.0
Kelihos_ver3	1.00	0.99	1.00	566.0

Table 3.6: Classification scores for the CNN on MsM2015

classes	precision	recall	f1-score	support
Obfuscator.ACY	0.95	0.90	0.92	252.0
Simda	0.40	0.25	0.31	8.0
Ramnit	0.90	0.92	0.91	314.0
Vundo	0.92	0.93	0.93	89.0
Gatak	0.91	0.92	0.92	202.0
Lollipop	0.96	0.94	0.95	509.0
Kelihos_ver1	0.93	0.97	0.95	88.0
Tracur	0.84	0.88	0.86	145.0
Kelihos_ver3	1.00	1.00	1.00	566.0

Table 3.7: Classification scores for the LSTM on MsM2015

Intuitively, if a classifier has high precision for a specific class A it means that it guesses correctly when a sample belongs to A and will not classify foreign samples to this class. It is the percentage of correct guesses when the classifier guesses A. However this does not take into account the samples belonging to A that have been wrongly assigned to other classes. Recall, on the other hand, considers these mistakes which are called false negatives (F_N):

$$Recall = \frac{T_P}{T_P + F_N} \tag{3.2}$$

The recall of a classification task for the class A is the percentage of correct guesses made by the model when it should have guessed A.

To better visualize the difference between accuracy and these measures we can add the true negatives (T_N) and then define accuracy as:

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \tag{3.3}$$

Precision and recall are often combined in the $F1$ measure via their harmonic mean:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.4}$$

classes	precision	recall	f1-score	support
alien_lang	0.91	0.99	0.95	80.0
armstrong_n	0.89	0.84	0.86	80.0
bot_trust	0.99	0.95	0.97	80.0
calculator	0.93	0.80	0.86	80.0
candy_split	0.98	0.80	0.88	80.0
char_freq	0.96	0.92	0.94	80.0
count_digits	0.86	0.84	0.85	80.0
count_vowels	0.89	0.85	0.87	80.0
factorial	0.94	0.79	0.86	85.0
factorial_rec	0.97	0.88	0.92	75.0
factors	0.83	0.84	0.83	80.0
fair_warn	0.96	0.98	0.97	80.0
fib_1	0.81	0.79	0.80	80.0
fib_2	0.77	0.82	0.80	80.0
fly_swatter	0.81	0.90	0.85	80.0
gcd	0.92	0.69	0.79	85.0
gcd_rec	0.84	0.92	0.88	75.0
hello_world	0.95	1.00	0.98	80.0
lcm	0.80	0.74	0.77	80.0
leap_year	0.88	0.86	0.87	80.0
magicka	0.97	0.85	0.91	80.0
min_product	0.82	0.90	0.86	80.0
multibase_hap	1.00	0.95	0.97	80.0
n_palindrome	0.78	0.81	0.80	80.0
n_is_prime	0.84	0.86	0.85	80.0
n_sum_of_p	0.75	0.74	0.74	80.0
pos_or_neg	0.83	0.84	0.83	80.0
power_n	0.69	0.79	0.74	80.0
prime_n	0.83	0.81	0.82	80.0
pyramid	0.90	0.95	0.93	80.0
quot_remainder	0.74	0.92	0.82	80.0
remove_char	0.89	0.88	0.88	80.0
reverse_int	0.85	0.89	0.87	80.0
rotate	0.91	0.92	0.92	80.0
saving_univ	0.87	0.94	0.90	80.0
snapper_chain	0.88	0.89	0.88	80.0
store_struct	0.88	0.96	0.92	80.0
strcat	0.89	0.88	0.88	80.0
strcpy	0.87	0.90	0.88	80.0
stringsort	0.91	0.92	0.92	80.0
strlen	0.91	0.94	0.93	80.0
sum	0.89	0.84	0.86	80.0
theme_park	0.99	0.99	0.99	80.0
times_table	0.84	0.91	0.87	80.0
train_t_tab	0.94	0.94	0.94	80.0
watersheds	0.95	0.99	0.97	80.0
welcome_cjam	0.99	0.98	0.98	80.0

Table 3.8: Classification scores for the CNN on OBF

3.4 Results and Analysis

This section is devoted to presenting our results on both architectures presented in the previous section against the datasets presented in Section 3.2. At the end of the section we briefly discuss how the results obtained can be analyzed.

3.4.1 OBF Dataset

CNN: On the OBF dataset the CNN model achieves an average accuracy of 92.3% on the hold-out test set with the input images resized to 256x64. This result is a definite improvement from 88% which was the average accuracy on the same dataset with the CNN in [95], where the images had a bigger height (596 pixels) and were not compressed.

This tells us that the information needed by the CNN for the classification is not reduced by the interpolation process. In fact, interpolating the images results in better classification accuracy than the cropping and padding methods used in [95]. The better performance in the classification of 256x64 images reinforces the observation that bigger size correlates with more information for the model (at least for the CNN) and thus results in an easier classification process. In Table 3.8 we show the classification scores of the CNN.

LSTM: The same consideration cannot be done by looking at the LSTM results, where the accuracy for the OBF dataset oscillates around 92.6% for images of size 256x64, compared to 93.4% for the smaller square images. This result has to be attributed to the difficulty encountered by the LSTM due to the increase of the timesteps, which also incurs a loss of training speed. The nearly tripled training time is noticeable in Figure 3.18, where we show the validation accuracy of the CNN and the LSTM on images of size 256x64. Whatever advantage there might be in having bigger images is then lost to the vanishing information in long recurrent networks. Interestingly enough, the classification accuracy does not improve when we switch the height with the width of the images and learn with the LSTM, while the training time does indeed decrease due to the reduced number of timesteps.

This result is definitely better than the CNN but incurs a processing time overhead. The training process for the LSTM takes about twice as long than the one for the CNN. This is expected, as CNNs are renowned for being very fast models, and furthermore the LSTM is bi-directional so the input must be scanned in 2 directions.

In Table 3.4 we show the classification scores of the LSTM for every family in OBF. It should be noted how the subdivision of the families in this dataset is more balanced than in the others. This is of course due to the fact that the OBF dataset is generated from scratch and thus does not suffer from class imbalance. We also noticed that the classification scores tend to have

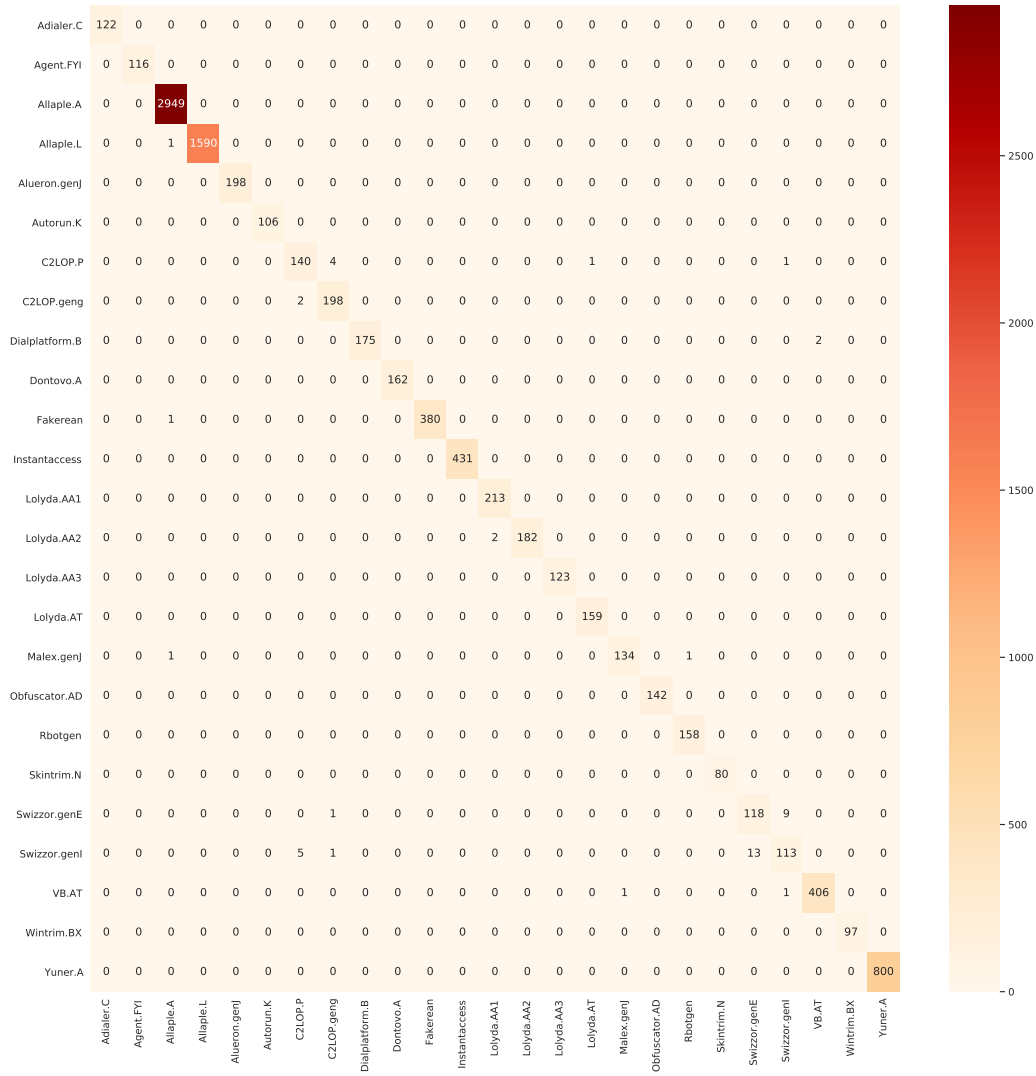

Figure 3.19: Confusion matrix for the CNN on the MalImg dataset

more outliers when the classes are distributed differently between the training set and the test set, thus this is avoided through a simple balancing algorithm that results in the almost uniform distribution clearly shown in Tables 3.4 and 3.8.

3.4.2 MsM2015 and MalImg Datasets

The same architectures described above are used to classify malware samples from the MsM2015 and MalImg datasets.

CNN: The accuracy for the CNN models trained on the two malware datasets is higher than the accuracy achieved on the OBF dataset. Classifying the hold-out test set samples of the MsM2015 dataset yields 92% accuracy on average, while on the MalImg we record results of up to 98.3% average accuracy. The classification scores for the individual classes of CNN on the MsM2015 and MalImg datasets are in Table 3.6 and Table 3.10 respectively.

LSTM The LSTM reaches an average accuracy of 94.2% on the MsM2015 dataset and 98.5% on MalImg. Since the MalImg dataset contains a noticeable class imbalance we provide various classification scores for the single classes with the LSTM in Table 3.9 and with the CNN in Table 3.10. Analyzing these scores it is easy to spot two classes in particular, '`Swizzor.genE`' and '`Swizzor.GenI`', that appear to be difficult to classify for both the LSTM and the CNN. The confusion matrix shown in Figure 3.19 provides a clearer picture on the classification errors for these two classes, where it is clear that the models struggle with deciding whether some samples belong to the '`Swizzor.genE`' class or the '`Swizzor.GenI`' class. Since these malware samples are simple variants of the same family they appear very similar and cannot be reliably distinguished by our models.

In Tables 3.6 and 3.7 we show the scores for the classification of the MsM2015 dataset on the CNN and LSTM respectively. It is easy to notice that the class '`Simda`' is very hard to classify for both models, achieving an F1 score of 0.12 with the CNN and 0.31 with the LSTM. This is easily explained by the support shown as the last column, which is the number of samples against which the classifier has been tested (the samples in the test set for a specific class). The class '`Simda`' is very under-represented, in fact only 40 samples exist of this class in the whole dataset, while the biggest classes feature thousands of samples each. This of course leads to problems with the classification accuracy, as 40 samples is not nearly enough for the class to be relevant in the training process. Some previous works that used this dataset decided to remove the class altogether [].

Both models generate better results for the two malware datasets compared to the OBF dataset. This is probably due to the nature of the obfuscated files of the OBF dataset, where

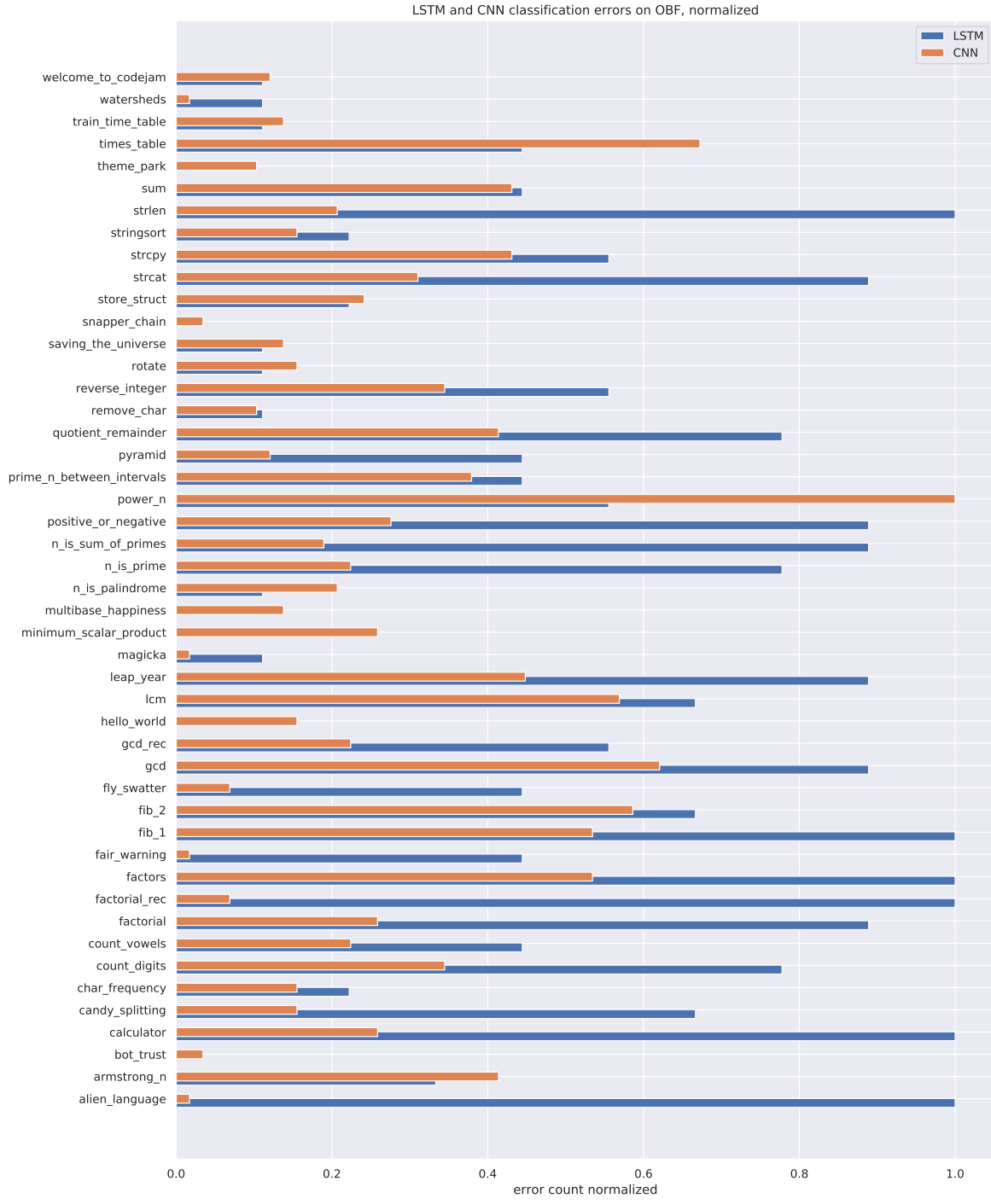

Figure 3.20: Classification errors for the CNN and the LSTM on the OBF dataset, normalized

classes	precision	recall	f1-score	support
Adialer.C	1.00	1.00	1.00	26.0
Agent.FYI	1.00	1.00	1.00	19.0
Allaple.A	1.00	1.00	1.00	604.0
Allaple.L	1.00	1.00	1.00	314.0
Alueron.genJ	1.00	1.00	1.00	40.0
Autorun.K	1.00	1.00	1.00	25.0
C2LOP.P	0.80	0.89	0.84	27.0
C2LOP.geng	0.88	0.92	0.90	38.0
Dialplatform.B	1.00	1.00	1.00	43.0
Dontovo.A	0.97	1.00	0.99	34.0
Fakerean	0.96	1.00	0.98	75.0
Instantaccess	1.00	1.00	1.00	90.0
Lolyda.AA1	1.00	1.00	1.00	45.0
Lolyda.AA2	1.00	1.00	1.00	25.0
Lolyda.AA3	1.00	1.00	1.00	24.0
Lolyda.AT	1.00	1.00	1.00	29.0
Malex.genJ	1.00	0.97	0.99	35.0
Obfuscator.AD	1.00	1.00	1.00	25.0
Rbotgen	1.00	1.00	1.00	28.0
Skintrim.N	1.00	1.00	1.00	21.0
Swizzor.genE	0.67	0.22	0.33	27.0
Swizzor.genI	0.46	0.62	0.52	26.0
VB.AT	1.00	0.99	0.99	78.0
Wintrim.BX	0.87	1.00	0.93	13.0
Yuner.A	1.00	1.00	1.00	156.0

Table 3.9: Classification scores for the LSTM on MalImg

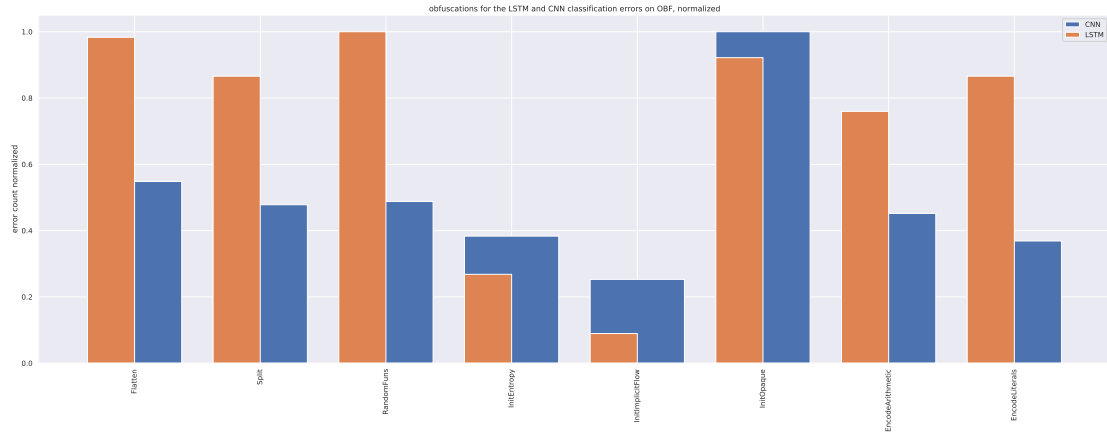

Figure 3.21: Obfuscations frequency in the classification errors for the CNN and LSTM on the OBF dataset

the intra-class similarities and the inter-class differences are not as definite as the samples found in the wild for the other datasets. One advantage of the OBF dataset is the size of the original programs before obfuscation. Since the programs are very small and usually consist of a single function with few auxiliaries, it is easy to change the structure of the programs in a more effective way, generating more pervasive changes, and this reflects on the binaries themselves (and thus on the images).

3.4.3 Transfer Learning

Having trained a CNN on three different datasets we investigate whether the features learnt in one of them can generalize to the other two. In order to do this we employ transfer learning, a technique that is very popular nowadays as it allows to re-use an already trained architecture for a completely different problem.

As explained in 3.1, the convolution layers of the CNN are concerned with feature extraction, along with max pooling and activations. The head of the model, two dense layers and a drop out layer in our case, is tasked with using the features to classify the programs into their respective classes. This of course is also true for the bi-directional LSTM that we trained, removing the dense layer from a model and applying a new dense layer with proper outputs provides with a new model that can use pre-trained features for a new classification problem.

Thanks to this neat subdivision of tasks, it has become good practice to download pre-trained models from either big companies or research labs and re-use them for completely

classes	precision	recall	f1-score	support
Adialer.C	1.00	1.00	1.00	26.0
Agent.FYI	1.00	1.00	1.00	19.0
Allaple.A	1.00	1.00	1.00	604.0
Allaple.L	1.00	1.00	1.00	314.0
Alueron.genJ	1.00	1.00	1.00	40.0
Autorun.K	1.00	1.00	1.00	25.0
C2LOP.P	0.82	0.85	0.84	27.0
C2LOP.geng	0.92	0.87	0.89	38.0
Dialplatform.B	1.00	1.00	1.00	43.0
Dontovo.A	1.00	1.00	1.00	34.0
Fakerean	0.97	1.00	0.99	75.0
Instantaccess	1.00	0.99	0.99	90.0
Lolyda.AA1	1.00	1.00	1.00	45.0
Lolyda.AA2	0.96	1.00	0.98	25.0
Lolyda.AA3	1.00	1.00	1.00	24.0
Lolyda.AT	1.00	1.00	1.00	29.0
Malex.genJ	1.00	0.97	0.99	35.0
Obfuscator.AD	0.96	1.00	0.98	25.0
Rbotgen	0.97	1.00	0.98	28.0
Skintrim.N	1.00	1.00	1.00	21.0
Swizzor.genE	0.70	0.59	0.64	27.0
Swizzor.genI	0.54	0.58	0.56	26.0
VB.AT	1.00	1.00	1.00	78.0
Wintrim.BX	1.00	1.00	1.00	13.0
Yuner.A	1.00	1.00	1.00	156.0

Table 3.10: Classification scores for the CNN on MalImg

new purposes. This allows huge networks such as the ones trained on ImageNet [49] for days (often with very expensive equipment) to be used by people that would otherwise not have the possibility to access such architectures. The hope is that features extracted from a comprehensive image dataset such as ImageNet will hopefully generalize to other image-based learning problems.

In our case we train all the models with images extracted from binaries, thus there could be an interesting intersection in the features as they come from the same problem domain. What needs to be investigated is if the features extracted from one dataset of compiled programs can be used to classify another dataset in the same domain. This is akin to using the networks trained on ImageNet for different (and usually smaller) datasets, since it can save time and resources. At the same time it can open up future possibilities, where big networks are trained on huge datasets of images extracted from compiled programs which can then be repurposed in order to classify smaller and newer datasets.

In the rest of this section we illustrate our experiments with transfer learning with both models on all the datasets.

MsM2015 → MalImg [LSTM]: A bi-directional LSTM model has been trained for 200 epochs on the MsM2015 dataset, achieving around 94% accuracy. After removing the dense layer we set the base as untrainable, thus preventing the optimizer from modifying the weights of the two LSTM units and preserving the features learned from the MsM2015 dataset. We then added a new dense layer with 25 outputs (as opposed to the 9 for the previous classification problem) and we trained on the MalImg dataset for a little over 321 epochs, achieving 98.4% accuracy on the hold-out dataset. This result is on par or even slightly better than most models trained using only the MalImg dataset. The performance of the LSTM on the MalImg dataset is already very good and adding the transfer learning actually increased the learning time from around 100 epochs to more than 400 for the same accuracy.

MsM2015 → MalImg [CNN]: The CNN model performs very well on the MalImg dataset, taking between 45 to 50 epochs to achieve 98.4% accuracy. This is the best result for the CNN so far and the fastest training time.

After downloading a trained model for the MsM2015 dataset (which achieved around 92% accuracy on said dataset), we attach a new dense layer and retrain it for 80 epochs. The final accuracy is 98.2%.

These experiments led us to believe that there is some untapped potential in the process of transfer learning applied to our problem setting. The positive results could stem from the fact that the MsM2015 dataset, while consisting of less classes altogether, contains a comparable

amount of samples. It also certainly helps that the programs from both datasets are for the Windows system, thus possibly sharing many visual features. This hypothesis is tested in the next attempt.

OBF → MalImg [CNN and LSTM] The process described above has been tried with a CNN and an LSTM, both trained on the OBF dataset. The CNN model on the MalImg dataset, with the base of the network set as untrainable, achieved 97% accuracy on the hold out set. This confirms that the features learned by the CNN model to solve the classification problem for the OBF dataset are in fact applicable to the MalImg dataset. An analysis of the classification errors revealed that the same problems persist with the new model. In particular, the classes '`Swizzor.genE`' and '`Swizzor.genI`' are still very hard to tell apart. Since the new network allows updates only on the weights for the dense layers, the training time is also greatly reduced, going from an average 70 seconds to around 18 seconds with only a slight reduction in accuracy. This also confirms that the features learned from the MsM2015 dataset are better suited to classify samples in MalImg. This might be due to their shared architecture or simply because they both contain real programs and not just small samples.

The LSTM model also improves on the training time but the accuracy decreases to around 87%, making it less viable.

OBF → MsM2015 [CNN and LSTM] The models generated with transfer learning for the MsM2015 dataset perform slightly worse than the ones for the MalImg dataset. The average accuracy for the CNN model trained on OBF and then transferred to the classification of MsM2015 is around 78%, a full 14 points lower than its counterpart trained solely on the MsM2015 dataset. The LSTM does not fare better, its accuracy hovering around 70%.

These results come solely from training on images of size 64x64 so the lower accuracy could come from the original model not learning enough features from the OBF dataset. The MsM2015 dataset is also generally harder to learn from when compared to MalImg.

MalImg → MsM2015 [CNN and LSTM] For completeness we tried to apply the models trained on the MalImg dataset to classify samples in the MsM2015 dataset. The accuracy achieved by the CNN model obtained via transfer learning is around 76%, only slightly lower than the previous experiment with the model trained on OBF. Once again the LSTM model achieves a lower score than the CNN with around 70% accuracy.

This experiment suffers from the small size of the MalImg dataset and from the apparent difficulty inherent in classifying the MsM2015 dataset while only looking at the images extracted from its binaries.

3.4.4 Error Analysis

As anticipated, the custom nature of the OBF dataset allows us to monitor the effect of the obfuscations on the classification results. In order to do this we collected all the mistakenly classified samples from every run of both models against the OBF dataset and counted the obfuscations that have been applied to such samples. The obfuscation count is then averaged through the runs (in our case 20 runs) and then finally normalized, so we can have a number between 0 and 1 that is easy to compare between the different models.

In Figure 3.21 we show the result of this study in a bar graph. The CNN results (in the blue, wider bars) clearly show that the network has the most trouble classifying programs that have been transformed by the InitOpaque transformation, while the Flatten transformation, with around half the errors on average, comes in at second place. This big discrepancy between the errors makes it clear that the spatial features extracted by the CNN have trouble when programs are obfuscated with InitOpaque, possibly due to the huge amount of entropy added to the binary (easily seen in Figure 3.10).

The LSTM (in orange, slimmer bars) has a more uniform distribution of the errors wrt the applied obfuscations. InitOpaque is still one of the obfuscations that are harder to classify, but is preceded by RandomFuns and Flatten, while being followed closely by Split and EncodeLiterals. What we can gauge from this analysis is that the LSTM does not have a particular weakness toward specific obfuscations but struggles relatively uniformly on the hardest obfuscations.

Another point of interest is that both models find it very easy to classify programs that are encoded with InitEntropy and InitImplicitFlow. This makes it clear that using a specific model to classify binaries by looking at their images has its drawbacks, highlighted by the general lack of precision of the models trained on OBF. It is also clear that certain obfuscations are more effective than others against this specific classification technique. As with images, the transformation used to fool the classifier can be more or less effective and part of this work is to show that this is the case.

In Figure 3.20 we show the errors with a focus on the classes of the OBF dataset. The errors of the LSTM are again more uniformly distributed while the CNN presents few taller peaks. It is interesting to note that some programs that appear on the higher end of error count for the LSTM are among the easiest to classify for the CNN (i.e. '`factorial_rec`' and '`alien_language`'). This again highlights the difference in strength of the two models.

3.4.5 Comparison with Existing Work

We decided not to directly compare the accuracy values recorded with the two malware datasets in this paper with those achieved in previous works. The main reason for this is that the goal

of this work is not to raise the ever-raising bar of classification accuracy in a specific domain, as that can usually be achieved by simply spending more time over-tuning the parameters or throwing more expensive hardware at the problem. At the same time, the accuracy level that we report is always taken from a hold-out test set that has been extracted randomly from the main dataset. This is a different approach than the one taken in most works we surveyed

3.5 Related Work

In this section we explore the most relevant works that approach the program classification problem using only compiled binaries. We do not include works that use features extracted from the source code of the programs or from the assembly, as our methodology is based on the premise of not possessing either.

It has to be noted that the focus of our study is not only on malware classification but on any obfuscated binary. This is why the two malware datasets have been used mainly for validation. Furthermore, none of the surveyed studies train an LSTM on the images extracted from binaries. This model provides new insights on the classification of images extracted from binaries.

Since the classification of binaries according to their images is done mainly on only two datasets, the MalImg and MsM2015 ones, we will group the related works accordingly.

MalImg papers To our knowledge, the first work that utilizes images extracted from binary files in order to classify them is by Nataraj et al. [101]. The general idea of this paper comes from the consideration that malware samples often appear similar in layout and texture when translated into images. For this reason this study approaches the feature selection with GIST [105], using wavelet decompositions of the images. The classifier used is a simple k-nearest neighbors with Euclidean distance as a measure of similarity. Other than the different models used and the different features used, our approach also does not assume that the samples from the same classes in our database look anything alike. In fact the opposite can be said, since the obfuscations applied generate visually different variants for each class. Nonetheless we use the dataset from this work to show that our approach can generalize to datasets that have not been created ad hoc.

The study by Cui et al. [43] also considers malware classification as as a means to its main goal. The paper argues (rightfully so) that the MalImg dataset is heavily imbalanced, which can potentially lead to less than accurate results. To quell the problem, they perform under-sampling of the dataset (removing samples from specific classes) and demonstrate that the process makes overfitting less of an issue.

Yakura et al. [148] designed a CNN with attention mechanism in order to highlight which parts of the malware image were being considered for the purpose of classifying them into families. Their approach allows the learning process to output specific regions of the image that are being used for classification, thus providing useful information for further manual analysis.

In [117], the authors take the deep residual network architecture with 50 layers from [71] and use it to classify malware from the MalImg dataset. The network is first trained on a typical object detection task, then the last layer is dropped from the model and a new dense model with 25 output nodes is added to classify the malware samples into their respective families. Other recent works such as [28] and [17] have proposed slight variations of the aforementioned approach, always pre-training their model on common image classification tasks.

The intuition of the previous papers is that the malware in the MalImg dataset present specific visual features that make them distinctive to the human eye [117], thus a neural network trained to classify real-life objects could carry enough features to also classify the malware samples.

Our approach starts with the assumption that we can already extract meaningful features from compiled programs, so we need to verify that these features map easily to new datasets. For this reason we use a network that is pre-trained on our custom dataset and a malware dataset in order classify a different malware dataset.

A CNN has been used for malware classification in [74], along with an extreme learning machine (ELM). The main goal of this work is to compare the efficiency of the two models when put to the task of classifying malware images. This study uses the images from the dataset of Nataraj et al. [101] to train their models and the results indicate that ELMs are more suited for the task at hand, being faster and more accurate than CNNs.

MsM2015 The MsM2015 dataset has been extensively used in literature for malware classification tasks. For example, the work by Kang et al. [76] uses word-to-vec approach with an LSTM network to classify the samples in each family. As many other studies on this subject, they do not consider the binary as is but rather generate an assembly file for each sample and collect opcodes and API functions that will then constitute the bulk of the features utilized.

A very interesting work that uses both the MalImg dataset and images extracted from the binaries in the MsM2015 dataset is [135]. The goal of the paper is to evaluate cost-sensitive approaches to malware image classification and for this purpose the authors combine a CNN with various RNN models in order to gauge the effectiveness of the approaches.

3.6 Summary, Discussion and Limitations

In this chapter we explored the second scenario hypothesized in Chapter 1, where the source code of the malware samples is not available. While classifying malware according to their binary it is harder to employ expertise in order to generate both the program representations and the similarity measure. For this reason we decided to extract images from the binaries and employ machine learning techniques to them.

The goal of the chapter was then to make the learning process more effective. In order to improve the process we investigated what are the common pitfalls for this task and found that the datasets employed can either be too small to effectively benefit from deep learning techniques or they can suffer from heavy class imbalance. Both these aspects are due to the nature of malware: certain families are more common than others.

These issues are not uncommon in other areas of machine learning and they can be mitigated in various ways. For example, in an image classification task the lack of data points is often solved with data augmentation techniques appropriate for images. These techniques clearly do not work as well for programs, as they change their semantics. We then hypothesized that it is possible to obtain the same effect as the data augmentation transformations for images by applying semantics-preserving transformations to the programs in our datasets.

We then developed two deep learning models (a CNN and an LSTM) and we have shown how they fare in a classification task on a generated dataset and two real-life malware datasets. The models have been built with an image classification task in mind and have been fine-tuned with a custom dataset generated with our aforementioned data augmentation technique. The results clearly show that the LSTM model performs better in all the classification tasks, reaching 98.5% average accuracy on a hold-out set of the MalImg dataset which is on par or superior to other studies in the state of the art.

Transfer learning is another technique that can aid the learning process when lacking the huge datasets often needed to train deep networks. For this reason, the models trained with the MalImg and MsM2015 datasets have then been subjects of transfer learning experiments, generating new models that have been trained on one dataset, while classifying samples from the other. With the accuracy of both these classifiers generated through transfer learning we verified that the features learned from either dataset can be used to classify malware from the other. This is a great result because it means that, not only can we use the images extracted from executable malware samples in order to classify them into their respective families, we can also transfer the knowledge gathered during such process to classify a new malware dataset. Akin to the results in image recognition, this can potentially save a lot of computation time and resources when dealing with newly released datasets.

Discussion. The promising results listed in this work led us to believe that there is a lot of untapped potential in transfer learning applied to malware classification. Further experiments are needed to have a more comprehensive view of the potentials and limitations of this technique. For example we envision future experiments with a different starting pool of programs to generate a new, bigger dataset to which we can apply the obfuscations. Having more classes in the OBF dataset would allow for more diversity and bigger programs could be included since the size limitations of [95] have been overcome with bicubic interpolation. The pool of obfuscations can also be considerably expanded.

Different visualization techniques could also be needed in the future. It is evident that merely changing the way the images are resized greatly impacts the learning process, this means that the way we extract images from the binaries is important. We wonder if designing new techniques that are not meant for generic images (such as bicubic interpolation) but catered specifically to programs could aid the learning process. On a related note, it would be also interesting to see what features are extracted by the networks when encountering different obfuscations. An attention based network could point out which of these features is important to distinguish the programs in different classes and which ones are merely introduced by the obfuscations.

Limitations. One of the main limitations in applying our data augmentation technique is that it requires the source code of the programs. This requirement obviously cannot be met for most malware distributed, rendering the technique itself limited in its efficacy when not paired with transfer learning.

There are many attacks that can be perpetrated against image classification tasks. Some notorious examples include adversarial learning, where some samples are generated specifically to fool the classifier into misclassification. It is unclear whether the malware classification technique explored in this chapter can be easily fooled by malware developers.

Another major limitation in this work is the lack of a proper deep-learning architecture. All the experiments have been done in Google Colab, whose resources greatly exceed those used in our first foray [95]. Even then, we encountered many difficulties in dealing with RAM and disk space, which is one of the reasons why the size of the images is so small. With proper equipment it is possible that the results of all these experiments would be more enticing.

4 A Formal Approach for Dynamic Analysis

This chapter represents a big change of pace from the rest of the thesis as it presents a strictly theoretical result without a clear practical counterpart. A small background explanation is needed to understand how we decided to establish a formal foundation for dynamic analysis in the context of this thesis.

In scenario 3 (closed system), neither the source code of programs nor their binary is available, but the program can be run in order to obtain the execution traces. These represent all the program states that are reached during the execution and they can be used to analyze what instructions have been executed in order to have a truthful analysis.

There are two immediately evident limitations to this approach, since there is finite time to execute the analysis : 1) the traces extracted have to be of finite length and 2) there can only be a finite number of them.

Many ways of employing dynamic analysis in the program similarity scenario can be hypothesized. In general, we need to leverage some of the advantages of dynamic analysis to offset the glaring limitations. For example, we could extract the CFG of two methods mirroring our approach in Chapter 2, but using dynamic analysis instead of the usual static analysis. This approach surely would not suffer from opaque predicate insertion nor any other static obfuscation meant to add spurious information, making the extracted CFG more faithful to the real one. Dynamic analysis, after all, sees exactly what is executed, without added noise.

Another crude approach for program similarity would be to execute two programs with the same inputs and see if they produce the same outputs. This could be problematic however, as many execution traces that are mined from programs have to be cut short due to the time it often takes to do a program run and their similarity might be apparent only at the end of the complete trace. Another problem is that in order to consider the behaviours of a program we need to take into account many of the intermediate states and system calls that have occurred during the run. For these reasons, considering only the input and output states can be inefficient and insufficient.

A more sophisticated approach found in literature is to find a “semantic alignment” between different execution traces. An example of this can be seen in a brilliant work by Churchill et al. [30] called “Semantic Program Alignment for Equivalence Checking”. In this work, the authors execute two compiled programs and analyze the execution traces to find similar states that

can be used to "align" the traces. Once a suitable alignment is found, the traces are tested for semantic equivalence. The goal of the paper, of course, is not program similarity in the same context as in this thesis, in fact, the technique is meant mostly as an automatic approach to prove semantic program equivalence between different code optimizations found in compilers.

We could hypothesize many more dynamic code similarity techniques, but there is one aspect that is still not well explored. In previous scenarios, we saw how obfuscations can fool static analysis by generating spurious information that impacts the analysis even though it does not actually appear at run time. Certainly dynamic analysis can also be fooled by certain types of obfuscations, but these cannot work the same way as in the static analysis world. There are no false positives in scenario number 3, since everything that is seen by the analysis has actually been executed. In this chapter, we investigate what it means to effectively fool dynamic analysis.

Poisons and Antidotes

The inspiration for this work came after hypothesizing a new type of obfuscation called "poisons and antidotes". We show how the obfuscation works in an intuitive way by following an example. Take a simple program P that computes the square of the input:

```
x = input()
y = x * x
return y
```

Intuitively, a "poison" is just a modification of the value of a variable that will be affected by a random seed, it is introduced by the program transformation $\mathcal{T}_P$. This way, the value of the poisoned variable will change depending on the seed at every execution trace. It makes the most sense to poison a variable that affects the output of the program, for example, the input variable. Assuming our programming language has a function *rand* that accepts two integers a, b as input and outputs i such that $a \leq i \leq b$, we can poison x and generate $\mathcal{T}_P(P)$:

```
i = input()
r = rand(1,4)
x = i + r
y = x * x
return y
```

The output of the function is no longer the input value squared, but rather the square of $x+r$, where $r \in [1, 4]$. Naturally this is not an obfuscation, as it does not preserve the semantics of the original program. In order to cure the poison we need an antidote, which is added right before the function return so that the effect of the poison is maximized. The poisoned program is then modified through the antidote transformation, producing $\mathcal{T}_A(\mathcal{T}_P(P))$:

```
i = input()
r = rand(1,4)
x = i + r
y = x * x
w = r * r
t = r * i
z = t + t
a = w + z
y = y - a
return y
```

Now the program computes the square of the input, just like P. Proving this is simple: starting with the end value for y (the return variable) we can backtrack and substitute each variable with the right-hand value of the statement(s) where it is defined. The assignment statements appear on top of each arrow in brackets:

$$y = y - a \xrightarrow{[a=w+z]} y = y - w - z \xrightarrow{[z=t+t]} y = y - w - 2 * t \xrightarrow{[t=r*i]} \tag{4.1}$$

$$y = y - w - 2 * r * i \xrightarrow{[w=r*r]} y = y - r^2 - 2 * r * i \xrightarrow{[y=x*x]} y = x^2 - r^2 - 2 * r * i \tag{4.2}$$

$$\xrightarrow{[x=i+r]} y = (i + r)^2 - r^2 - 2 * r * i = i^2 + r^2 + 2 * r * i - r^2 - 2 * r * i = i^2 \tag{4.3}$$

At the end of the chapter we go into more detail on poisons and antidotes, which, while not yet totally well defined, provide an interesting window into possible future work.

Through abstract interpretation we can quickly show that poisons and antidotes do work effectively against a static analysis executed with the *intervals* domain introduced in chapter 1. We start by monitoring the value of the variable i with an input value of 2, then at each program point we add the value of the variables that are changed at that specific line. The computation starts with the variable set to the *intervals* representation of 2: $\alpha(2) = [2, 2]$.

$$i = [2, 2] \rightarrow r = [1, 4] \rightarrow x = [3, 6] \rightarrow y = [9, 36] \rightarrow w = [1, 8] \rightarrow z = [2, 8] \rightarrow z = [4, 16] \rightarrow \tag{4.4}$$

$$\rightarrow z = [5, 24] \rightarrow y = [4, 12] \tag{4.5}$$

The result of the abstract interpretation of the program with the *intervals* domain can be written as $[[\mathcal{T}_A(\mathcal{T}_P(P))]]^{\mathcal{A}}([2, 2]) = [4, 12]$. Now we can abstract the result of the concrete execution of the program and pinpoint the exact amount of information that was lost: $\alpha([[\mathcal{T}_A(\mathcal{T}_P(P))]](2)) = \alpha(4) = [4, 4] \subset [4, 12] = [[\mathcal{T}_A(\mathcal{T}_P(P))]]^{\mathcal{A}}(\alpha(2)")$.

This very simple check allows us to prove that the program P, when obfuscated with poisons and antidotes, adds enough spurious information to its semantics such that abstract interpretation with the *interval* domain is fooled. Since poisons affect the values encountered in execution traces, we postulate that they can in fact deter some types of dynamic analysis. For example, the technique described in [30] would probably have a harder time in finding the semantic alignments in the execution traces, since the values vary randomly. Unfortunately, we do not have a framework like abstract interpretation to show the same effect on dynamic analysis, nor a way to formally prove whether it has any effect at all.

The problem identified in this chapter is indeed the lack of a proper formal foundation to prove the effectiveness of obfuscations against dynamic analysis. In order to build this we leave the program similarity path and focus on what it means to fool dynamic similarity in general.

Motivation

Program analysis allows us to infer information about program behaviour, or in other words: the semantics of the program. We showed in the introductory chapter how, due to Rice's theorem, in general it is not possible to decide whether a given program satisfies a semantic property. For this reason analysts resort to approximation either by static or dynamic analysis. Static analysis computes an over-approximation of program semantics while dynamic analysis under-approximates program semantics, meaning that static analysis generally encounters false positives (spurious information) while dynamic analysis generates false negatives (missing information). In both cases, we have a decidable evaluation of the semantic property on an approximation of program semantics.

For this reason, what we can conclude regarding the semantic property of programs has to take into account false positives for static analysis and false negatives for dynamic analysis. Static analysis is precise when it is complete (no false positives) and this relates to the well studied notion of completeness in abstract interpretation [38, 39, 65]. Dynamic analysis is precise when it is sound (no false negatives), which only happens when the execution traces considered by the dynamic analysis exhibit all the behaviours of the program that are relevant wrt the semantic property of interest. The most common way to evaluate the soundness of dynamic analysis (i.e. the absence of false negatives) is by using code coverage [3].

Program analysis has been originally developed for program verification and debugging and researchers have put great effort into developing efficient analysis techniques and tools that reduce the number of both false positives and false negatives. In this setting, the precision of the analysis relates to the ability to identify bugs and vulnerabilities that may lead to unexpected behaviours, or that may be exploited by an adversary for malicious purposes.

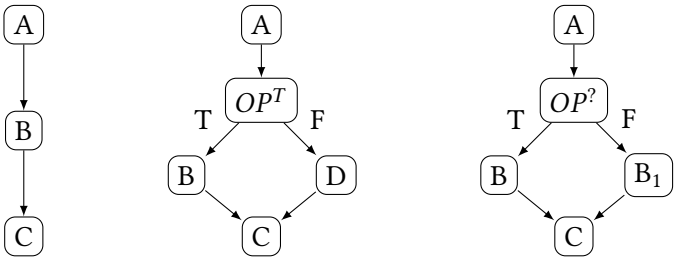

Figure 4.1: Code obfuscation

Software protection is another interesting scenario where program analysis plays a central role, but in a dual way. Indeed, in the software protection scenario, program analysis is used by adversaries to reverse engineer proprietary code and then illicitly reuse portions of the code or tamper with the code in some unauthorised way. Here, the intellectual property and integrity of programs is guaranteed when the analysis is imprecise or very expensive since this complicates the attacks. In this setting, researchers have developed program transformations, called code obfuscations, with the explicit intent of complicating program analysis.

In the last years many different kinds of obfuscation techniques and tools have been proposed [31]. Code obfuscation has proven its efficiency in degrading the results of static program analysis while it seems to be less efficient with respect to dynamic program analysis [123].

For example, consider a program whose control flow graph is depicted on the left of Figure 4.1 where we have three blocks of sequential instructions A, B and C executed in the order specified by the arrows $A \longrightarrow B \longrightarrow C$. A true opaque predicate OP^T is a predicate that always evaluates to *true*, but this invariant behaviour is not known to the attacker who also considers the execution of the false branch as possible [33]. In the middle of Figure 4.1 we can see what happens to the control flow graph when we insert a true opaque predicate, where block D has to be considered in the static analysis of the control flow even if it is never executed at runtime. Thus, $A \longrightarrow OP^T \longrightarrow D \longrightarrow C$ is a false positive path added by obfuscation to static analysis.

On the other hand, a dynamic analysis would execute the program and only hit the trace $A \longrightarrow OP^T \longrightarrow B \longrightarrow C$ over and over again. The obfuscation failed to add any imprecision to the dynamic analysis.

In a way, we could say that dynamic analysis is impervious to opaque predicates. Again, this is intuitively true, but this time we are lacking a formal framework that allows us to prove it.

On the right of Figure 4.1 we have the control flow graph of the program obtained by inserting an unknown opaque predicate. An unknown opaque predicate $OP^?$ is a predicate that sometimes evaluates to *true* and sometimes evaluates to *false*. These predicates are used

to diversify program executions by inserting in the true and false branches sequences of instructions that are different but functionally equivalent (e.g. blocks B and B_1) [33]. This means that at execution time it does not matter which path is taken by the program, since they are semantically equivalent. A static analyzer would obviously not be fooled by the transformation as it would see both paths and correctly assume they are both executed.

Dynamic analysis on the other hand would be affected, since a dynamic analyzer would have to consider more execution traces in order to observe all possible program behaviours. Indeed, if the dynamic analysis observes only traces that follow the original path $A \rightarrow OP^? \rightarrow B \rightarrow C$ it may not be sound as it misses the traces that follow $A \rightarrow OP^? \rightarrow B_1 \rightarrow C$ (false negative).

Code obfuscation hampers static analysis by exploiting its conservative nature, namely by increasing its imprecision (false positives) while preserving the intended program behaviour. The abstract interpretation framework has been used to formally prove the efficiency of code obfuscation in making static analyzers imprecise [44]. It has been observed that adding false positives to the analysis can be formalised in terms of incompleteness in the analysis of the transformed program [44, 47, 62, 64]. In general the imprecision added by these obfuscating transformations to confuse a static analyzer is not able to confuse a dynamic attacker that looks at the real program execution and thus cannot be deceived by false positives. Dynamic analysis observes only paths that are actually executed.

For this reason common deobfuscation approaches often resort to dynamic analysis to understand obfuscated code [19, 34, 127, 147].

It is clear that to hamper dynamic analysis we need to develop obfuscation techniques that exploit the Achilles heel of dynamic analysis and that increase the number of false negatives. In the literature there are defense techniques that focus on hampering dynamic analysis [9, 106, 110, 121].

The central goal of this chapter is to provide a formal framework in which it is possible to prove and discuss the efficiency of these techniques that confuse dynamic analysis, in terms of the imprecision (false negatives) that they introduce in the analysis. This will allow us to better understand the potential and limits of code obfuscation against dynamic program analysis.

We start by providing a formalisation of dynamic analysis and software protection techniques in terms of program semantics and equivalence reactions over semantic domains, and then we characterise when a program transformation hampers a dynamic analysis in terms of topological features.

The contributions of this chapter are:

- a formal specification for dynamic analysis/attacks based on program semantics and equivalence relations

- formal definition of software-based protection transformations against dynamic attacks that induce imprecision in dynamic analysis (false negatives)
- validation of the model on some known software-based defense strategies
- introduction of poisons and antidotes, a new obfuscation meant to disrupt the run-time value of variables

The rest of the chapter is thus structured:

- Section 4.1 establishes the mathematical foundation needed to define our formal framework
- Section 4.2 contains the our formalization of dynamic analysis
- What it means to harm dynamic analysis is introduced in Section 4.3
- In Section 4.4 we show how our framework can be applied to real life obfuscations and provide proof that they work against dynamic analysis
- Poisons and antidotes are explored in Section 4.5
- Section 4.6 lists the works that most closely are connected to this chapter
- Section 4.7 closes the chapter with a summary and some considerations for future work and limitations

4.1 Preliminaries

In this section we introduce the mathematical preliminaries that are used throughout the chapter. Some of the notation used should be very familiar to the readers already acquainted with abstract interpretation. Our framework, after all, aims to be used to verify the strength of dynamic analysis by leveraging some topological properties of the semantics, mirroring the approach of abstract interpretation for static analysis.

Basic lattice and fix-point theory: Given two sets S and T, we denote with $\wp(S)$ the powerset of S, with $S \times T$ the Cartesian product of S and T, with $S \subset T$ strict inclusion, with $S \subseteq T$ inclusion, with $S \subseteq_F T$ the fact that S is a finite set.

$\langle C, \leq, \vee, \wedge, \top, \bot \rangle$ denotes a complete lattice on the set C, with ordering $\leq$, least upper bound (*lub*) $\vee$, greatest lower bound (*glb*) $\wedge$, greatest element (top) $\top$, and least element (bottom) $\bot$.

Let C and D be complete lattices. Then, $C \xrightarrow{m} D$ and $C \xrightarrow{c} D$ denote, respectively, the set and the type of all monotone and (Scott-)continuous functions from C to D. Recall that $f \in C \xrightarrow{c} D$ if and only if f preserves *lub*'s of (nonempty) chains if and only if f preserves *lub*'s of directed subsets.

Let $f : C \to C$ be a function on a complete lattice C, we denote with $lfp(f)$ the least fix-point, when it exists, of function f on C. The well-known Knaster-Tarski's theorem states that any monotone operator $f : C \xrightarrow{m} C$ on a complete lattice C admits a least fix point. It is known that if $f : C \xrightarrow{c} C$ is continuous then $lfp(f) = \vee_{i \in \mathbb{N}} f^i(\bot_C)$, where, for any $i \in \mathbb{N}$ and $x \in C$, the i-th power of f in x is inductively defined as follows: $f^0(x) = x$; $f^{i+1}(x) = f(f^i(x))$.

Given a relation $\mathcal{R} \subseteq C \times D$ between two sets C and D, and two elements $x \in C$ and $y \in D$, then $(x, y) \in \mathcal{R}$ denotes that the pair (x, y) belongs to the relation $\mathcal{R}$. A binary relation $\mathcal{R}$ on a set C, namely $\mathcal{R} \subseteq C \times C$, is an *equivalence relation* if $\mathcal{R}$ is reflexive $\forall x \in C : (x, x) \in \mathcal{R}$, symmetric $\forall x, y \in C : (x, y) \in \mathcal{R} \Rightarrow (y, x) \in \mathcal{R}$ and transitive $\forall x, y, z \in C : (x, y) \in \mathcal{R} \wedge (y, z) \in \mathcal{R} \Rightarrow (x, z) \in \mathcal{R}$. Given a set C equipped with an equivalence relation $\mathcal{R}$, we consider for each element $x \in C$ the subset $[x]_\mathcal{R}$ of C containing all the elements of C in equivalence relation with x, i.e., $[x]_\mathcal{R} = \{y \in C \mid (x, y) \in \mathcal{R}\}$. The sets $[x]_\mathcal{R}$ are called equivalence classes of C wrt relation R. Let $eq(C)$ be the set of equivalence relations over the set C. The equivalence classes of an equivalence relation $\mathcal{R} \in eq(C)$ form a partition of the set C, namely $\forall x, y \in C : [x]_\mathcal{R} = [y]_\mathcal{R} \vee [x]_\mathcal{R} \cap [y]_\mathcal{R} = \emptyset$ and $\cup\{[x]_\mathcal{R} \mid x \in C\} = C$. The partition of C induced by the set of equivalence classes of relation $\mathcal{R}$ is called the quotient set of C and it is denoted by $C/_\mathcal{R}$.

A partition $C/_{\mathcal{R}_1}$ is a refinement of a partition $C/_{\mathcal{R}_2}$, namely $\mathcal{R}_1$ if finer than $\mathcal{R}_2$ or $\mathcal{R}_2$ is coarser than $\mathcal{R}_1$, if every equivalence class in $C/_{\mathcal{R}_1}$ is a subset of some equivalence class in $C/_{\mathcal{R}_2}$. We denote with $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ the fact that the equivalence relation $\mathcal{R}_1$ is finer than the equivalence relation $\mathcal{R}_2$. Given a subset $S \subseteq C$ we denote with $\mathcal{R}(S)$ the set of equivalence classes of the elements of S, namely $\mathcal{R}(S) = \{[x]_\mathcal{R} \mid x \in S\}$, and with $S/_\mathcal{R}$ the partition of the subset S induced by the equivalence relation $\mathcal{R}$, namely $S/_\mathcal{R} = \{[x]_\mathcal{R} \cap S \mid x \in S\}$.

The concepts just explained with subset partitions can be as easily expressed with atomistic lattices. Let us recall the notion of atom in a lattice L. We say that $a \in L$, $a \neq \bot$ is an atom of L if for each $x \in L$, with $x \neq \bot$, we have that $\bot \leq x \leq a$ implies $x = a$. The purpose of building a system that has the same mathematical foundation as abstract interpretation is that one day we might be able to express dynamic and static analysis using the same formal framework.

Program semantics: Let $\mathbb{P}$ be a set of programs ranged over by P.

Let $v \in \mathbb{I}$ denote a possible input and let $\mathbb{I}^*$ denote the set of input sequences ranged over by $\mathcal{I}$, let PP denote the set of program points ranged over by pp, let Com denote the set of program

statements ranged over by C and let Mem denote the set of memory maps that associates values to variables ranged over by $m : Var \to Values$. $\Sigma = \mathbb{I}^* \times PP \times Com \times Mem$ is the set of program states. Thus, a program state $s \in \Sigma$ is a tuple $s = \langle \mathcal{I}, pp, C, m \rangle$ where $\mathcal{I}$ denotes the sequence of inputs that still needs to be consumed to terminate the execution, pp denotes the program point of the next instruction C that has to be executed, and m is the current memory. We denote with $C_1; C_2$ the sequential composition of statements and we refer to *skip* as the identity statement whose execution has no effects on memory. Given a program P we denote with $\mathbb{I}_P \subseteq \mathbb{I}^*$ the set of the initial input sequences for the execution of program P, and with $Init_P = \{s \in \Sigma \mid s = \langle \mathcal{I}, pp, C, m \rangle, \mathcal{I} \in \mathbb{I}_P\}$ the set of its initial states. We use Σ^* to denote the set of all finite and infinite sequences or traces of states, where $\epsilon \in \Sigma^*$ is the empty sequence, $|\sigma|$ the length of sequence $\sigma \in \Sigma^*$. $\Sigma^+ \subset \Sigma^*$ denotes the set of finite sequences of elements of Σ. We denote the concatenation of sequences $\sigma, \nu \in \Sigma^*$ as $\sigma\nu$. Given $\sigma, \nu \in \Sigma^*$, $\nu \preceq \sigma$ means that ν is a subsequence of σ, namely that there exists $\sigma_1, \sigma_2 \in \Sigma^*$ such that $\sigma = \sigma_1 \nu \sigma_2$. Given $s \in \Sigma$ we write $s \in \sigma$ when s is an element occurring in sequence σ, and we denote with $\sigma_0 \in \Sigma$ the first element of sequence σ and with σ_f the final element of the finite sequence $\sigma \in \Sigma^+$. Let $R \subseteq \Sigma \times \Sigma$ denote the transition relation between program states, thus $(s, s') \in R$ means that state s' can be obtained from state s in one computational step. The *(finite) trace semantics* of a program P is defined, as usual, as the least fix-point computation of function $\mathcal{F}_P : \wp(\Sigma^*) \to \wp(\Sigma^*)$ [36]:

$$\mathcal{F}_P(X) \stackrel{\text{def}}{=} Init_P \cup \left\{ \sigma s_i s_{i+1} \,\middle|\, (s_i, s_{i+1}) \in R, \sigma s_i \in X \right\}$$

The trace semantics of P is $[\![P]\!] = lfp(\mathcal{F}_P) = \bigcup_{i \in \mathbb{N}} \mathcal{F}_P^i(\bot_C)$. $Den[\![P]\!]$ denotes the denotational (finite) semantics of program P which abstracts away the history of the computation by observing only the input-output relation of finite traces. Therefore we have $Den[\![P]\!] = \{\sigma \in \Sigma^+ \mid \exists \eta \in [\![P]\!] : \eta_0 = \sigma_0, \eta_f = \sigma_f\}$.

4.2 Topological Characterisation of the Precision of Dynamic Analysis

We start our investigation by considering dynamic analysis that observes features of single execution traces, for example: the order of successive accesses to memory, the order of execution of instructions, the location of the first instruction of a function, the target of jumps, function location, possible data values at certain program points, etc. The extension of the framework to properties of sets of traces (hyper-properties) and relational properties among traces is left as future work.

The simplest way to model properties of single traces is in terms of equivalence relations over program traces. Indeed, an equivalence relation $\mathcal{R} \in eq(\Sigma^*)$ groups together all those execution traces that are equivalent wrt the property used to establish the equivalence for $\mathcal{R}$. In this setting, each equivalence class $[\sigma]_{\mathcal{R}} \subseteq \Sigma^*$ represents the set of execution traces that are equivalent to σ wrt $\mathcal{R}$, namely all those execution traces that $\mathcal{R}$ is not able to distinguish from σ. In general, given a program $P \in \mathbb{P}$ and an equivalence relation $\mathcal{R} \in eq(\Sigma^*)$ it may not be possible to precisely observe property $\mathcal{R}$ of program semantics, namely the set $\mathcal{R}([\![P]\!]) = \{[\sigma]_{\mathcal{R}} \mid \sigma \in [\![P]\!]\}$ may not be precisely observable. This means that it may not be possible to decide whether $\mathcal{R}([\![P]\!]) \subseteq \Pi$, for some $\Pi \in \wp(\Sigma^*/_{\mathcal{R}})$, a set of equivalence classes representing a possible feature of program execution that can be expressed in terms of $\mathcal{R}$. In order to verify these features, analysts resort to approximation either by static or dynamic analysis.

Example 1. Consider function $\iota : \Sigma \longrightarrow \mathbb{I}$ that observes the first input value $v \in \mathbb{I}$ of a program state, namely $\iota(\langle v\mathcal{I}, pp, C, m\rangle) \stackrel{\text{def}}{=} v$. We can define the equivalence relation $\mathcal{R}_\iota$ as $(\sigma, v) \in \mathcal{R}_\iota$ iff $\iota(\sigma_0) = \iota(v_0)$, grouping together traces with the same starting input values. Based on $\mathcal{R}_\iota$ we can define features of program behaviour as for example $\Pi_1, \Pi_2 \in \wp(\Sigma^*/_{\mathcal{R}_\iota})$ where $\Pi_1 = \{[\sigma]_{\mathcal{R}_\iota} \mid \iota(\sigma) \geq 0\}$ observes the equivalence classes of traces whose first input value is positive, and $\Pi_2 = \{[\sigma]_{\mathcal{R}_\iota} \mid \iota(\sigma) \in [l, u]\}$ observes the equivalence classes of traces whose first input value is in the interval $[l, u]$.

We can think about the relation $\mathcal{R}$ as the granularity at which the analysis observes program executions. Given $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ we have that $\mathcal{R}_1$ describes an analysis that is more precise than $\mathcal{R}_2$ in distinguishing program traces, while $\mathcal{R}_2$ describes an analysis that groups together more traces than $\mathcal{R}_1$. The equivalence classes can then be combined to describe properties of programs at different levels of abstraction.

In the literature there exists a formal investigation of the effects of code obfuscation to the precision of static analysis [44, 47, 62, 64]. This has lead to a better understanding of the potential and limits of obfuscation, and it has been useful in the design of obfuscation techniques that target specific program properties [47, 62, 63].

In the following, we apply a similar approach to dynamic analysis. To this end, we formalise the absence of false negatives, namely the precision of dynamic analysis, in terms of topological properties of program trace semantics and of the equivalence relation $\mathcal{R}$ modelling the property to be observed. False negatives occur when dynamic analysis fails to consider some traces that would modify the equivalence classes observed by property $\mathcal{R}$. We show how to transform a program in order to hinder the dynamic analysis of a property $\mathcal{R}$, that is, to make the dynamic analysis of the transformed program not sound.

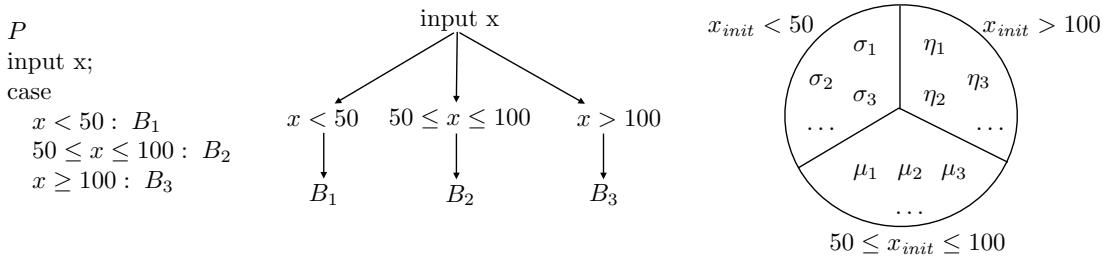

Figure 4.2: Dynamic analysis

4.2.1 Modelling Dynamic Program Analysis

Dynamic analysis observes a finite subset of finite execution traces of a program and tries to draw conclusions on the whole program behaviour from this partial observation.

Definition 1 (Dynamic Execution)**.** *The execution traces of program P with initial states in $T_P \subseteq_F Init_P$ and with time limits $t \in \mathbb{N}$, are defined as:*

$$Exe(P, T_P, t) \stackrel{def}{=} \{\ \sigma \in [[P]] \mid |\sigma| \leq t, \sigma = s_0\sigma', s_0 \in T_P\ \}$$

Note that $Exe(P, T_P, t)$ is a finite set and that each trace in $Exe(P, T_P, t)$ is finite (it has at most t states). This correctly implies that: $Exe(P, T_P, t) \subseteq_F [[P]]$. The goal of dynamic analysis is to derive knowledge of a semantic property of a program by observing a finite subset $Exe(P, T_P, t)$ of its execution traces. Dynamic analysis is therefore specified as the set of observed execution traces $Exe(P, T_P, t)$ and of an equivalence relation on traces $\mathcal{R} \in eq(\Sigma^*)$.

Definition 2 (Dynamic Analysis)**.** *A dynamic analysis of property $\mathcal{R} \in eq(\Sigma^*)$ of program $P \in \mathbb{P}$, is defined as a pair $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$.*

Let us consider program P on the left of Figure 4.2 where the block of code to execute depends on the input value of x. Consider a property of traces $\bar{\mathcal{R}} \in eq(\Sigma^*)$ that observes which block B_1, B_2 or B_3 of program P is executed. On the right of Figure 4.2 we represent the partition of the traces of program P induced by property $\bar{\mathcal{R}}$ where x_{Init} denotes the input value of variable x.

Traces are grouped together according to an equivalence relation $\mathcal{R}$ depicted as red circles around traces, so we have that σ_1 and σ_2 are in the same equivalence class, that σ_3 and σ_4 are in the same equivalence class while σ_5 is in a different equivalence class. In Figure 4.2 we can see three different dynamic analysis of property $\mathcal{R}$ on program P: DA_1, DA_2 and DA_3.

In Figure 4.3 we can see on the left a set of program traces $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5\}$ that represents the trace semantics $[[P]]$ of some program P. The three dynamic analyses pictured give an intuitive view to the concept of soundness in dynamic analysis.

Dynamic analysis $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ can precisely observe property $\mathcal{R}$ of the semantics of P (no false negatives) when $Exe(P, T_P, t)$ contains at least one trace for each one of the equivalence classes of the traces of $[[P]]$.

Definition 3 (Soundness). *Given $P \in \mathbb{P}$ and $\mathcal{R} \in eq(\Sigma^*)$ a dynamic analysis $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ is sound if $\forall x \in [[P]] : [x]_{\mathcal{R}} \in \mathcal{R}(Exe(P, T_P, t))$.*

When a dynamic analysis $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ is sound we have no false negatives, namely $\forall y \in [[P]] : [y]_{\mathcal{R}} \in \mathcal{R}(Exe(P, T_P, t))$. When this happens, all the behaviours of program P that relation $\mathcal{R}$ is able to distinguish are taken into account by the partial observation of program behaviour $Exe(P, T_P, t)$. In the example in Figure 4.2 we have that a dynamic analysis $\langle \bar{\mathcal{R}}, Exe(P, T_P, t)\rangle$ is sound if $Exe(P, T_P, t)$ contains at least one execution trace for each one of the three equivalence classes depicted on the right of Figure 4.2. In the example in Figure 4.3 we can see that DA_1 is not sound since it does not consider any trace in the equivalence class $[\sigma_3]_{\mathcal{R}} = [\sigma_4]_{\mathcal{R}}$ that is a false negative of the analysis. The dynamic analyses DA_2 and DA_3 are both sound since they consider at least one trace for each equivalence class of the traces in $[[P]]$.

To formalise this constraint we introduce the following notion.

Definition 4 (Covers). *Given $P \in \mathbb{P}$, and $\mathcal{R} \in eq(\Sigma^*)$, we say that $S \subseteq [[P]]$ covers P wrt $\mathcal{R}$ when: $\mathcal{R}(S) = \mathcal{R}([[P]])$.*

It is clear that when S covers P wrt $\mathcal{R}$ we have that the partial observation S of the behaviours of P is sound wrt $\mathcal{R}$, since it allows us to observe all the equivalence classes of $\mathcal{R}$ that we would observe by having access to all the traces in $[[P]]$ (no false negatives). Thus, in the example in Figure 4.2 we have that the set of traces $\{\sigma_1, \eta_1\}$ does not cover P wrt $\bar{\mathcal{R}}$, while the set of traces $\{\sigma_1, \eta_1, \eta_2, \mu_2\}$ does.

Similarly, in the example in Figure 4.3 the set of traces observed by DA_1 does not cover $[[P]]$ wrt $\mathcal{R}$, while the sets of traces observed by DA_2 and DA_3 cover $[[P]]$ wrt $\mathcal{R}$.

The following theorem comes straight from the definitions.

Theorem 2. *Given $P \in \mathbb{P}$ and $\mathcal{R} \in eq(\Sigma^*)$, if $Exe(P, T_P, t)$ covers P wrt $\mathcal{R}$ then the dynamic analysis $\langle \mathcal{R}, Exe(P, T_P, t)\rangle$ is sound (no false negatives).*

The goal of dynamic analysis of a property $\mathcal{R}$ on a program P, is to identify the set T_P of inputs, and the length t that induce a partial observation of program semantics that makes the

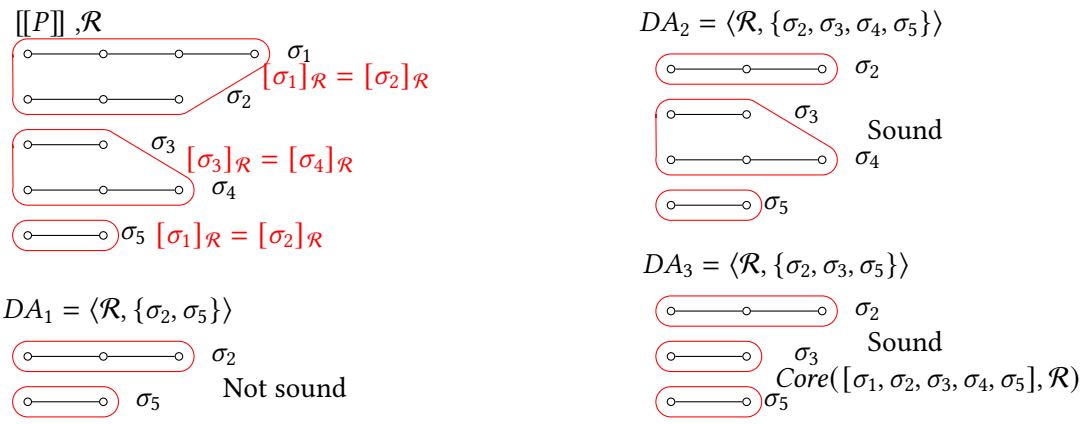

Figure 4.3: Dynamic analysis and soundness

analysis sound (no false negatives) wrt $\mathcal{R}$. Thus, a possible way to hamper dynamic analysis is to transform programs in order to increase the number of traces that is necessary to observe to ensure soundness. In a way, by tying the precision of dynamic analysis to the observation of a wider set of traces (worst case being the observation of all possible traces) we are limiting the advantages of using dynamic analysis.

In order to formalise this idea, in the following section we provide a characterisation of the set of traces that are needed to guarantee the soundness of the dynamic analysis of a program P wrt a semantic property $\mathcal{R}$. We use this characterisation to formalize what it means for a software-based defense transformation to harm dynamic analysis. We validate our model by showing how it naturally relates to the notion of code coverage of dynamic analysis, and by showing how existing techniques for hindering dynamic analysis fit into our framework.

4.3 Harming Dynamic Analysis

In this section we formally characterize what it means to harm dynamic analysis in our framework.

Given an equivalence relation $\mathcal{R} \in eq(\Sigma^*)$ concerning what we can observe and a set of equivalence classes $X \in \wp(\Sigma^*/_{\mathcal{R}})$ we would like to characterise the minimal sets of traces that the relation $\mathcal{R}$ maps to X.

Definition 5 (Core). *Consider* $\mathcal{R} \in eq(\Sigma^*)$ *and* $X \in \wp(\Sigma^*/_{\mathcal{R}})$*:*

$$Core(X,\mathcal{R}) \stackrel{def}{=} \left\{ T = \{\sigma \in \Sigma^* \mid [\sigma]_{\mathcal{R}} \in X\} \;\middle|\; \begin{array}{l} \forall \sigma_1, \sigma_2 \in T, \sigma_1 \neq \sigma_2 \Rightarrow [\sigma_1]_{\mathcal{R}} \neq [\sigma_2]_{\mathcal{R}} \\ \forall [\nu]_{\mathcal{R}} \in X : \exists \sigma \in T : [\sigma]_{\mathcal{R}} = [\nu]_{\mathcal{R}} \end{array} \right\}$$

Theorem 3. *Consider* $\mathcal{R} \in eq(\Sigma^*)$ *and* $X \in \wp(\Sigma^*/_{\mathcal{R}})$*:*

1. *Given* $T \in Core(X,\mathcal{R})$ *we have that:* $\mathcal{R}(T) = X$
2. $\forall S \in \wp(\Sigma^*)$*: If* $\mathcal{R}(S) = X$ *then* $\exists T \in Core(X,\mathcal{R}) : T \subseteq S$

This means that $Core(\mathcal{R}([[P]]),\mathcal{R})$ characterises the minimal sets of execution traces that provide a sound dynamic analysis of property $\mathcal{R}$ for program P. In the example in Figure 4.2 we have that $Core([[P]],\bar{\mathcal{R}})$ identifies those sets of trace that have exactly three traces: one trace with $x_{init} < 50$, one trace with $50 \leq x_{init} \leq 100$ and one trace with $x_{init} > 100$.

Likewise, in the example in Figure 4.3 the set of traces considered by the dynamic analysis DA_3 belongs to $Core([[P]],\mathcal{R})$ while this does not hold for the analyses DA_1 and DA_2. However, dynamic analysis DA_2 is sound since it observes the set of traces $\{\sigma_2,\sigma_3,\sigma_4,\sigma_5\}$ which contains the set $\{\sigma_2,\sigma_3,\sigma_5\} \in Core([[P]],\mathcal{R})$, while DA_1 is not sound since it observes a set of traces that does not contain a set in $Core([[P]],\mathcal{R})$.

Corollary 1. *Given* $P \in \mathbb{P}$ *and* $\mathcal{R} \in eq(\Sigma^*)$ *we have that:*

- $\forall T \in Core(\mathcal{R}([[P]]),\mathcal{R})$ *we have that* T *covers* $[[P]]$ *wrt* $\mathcal{R}$*.*
- *Given* $T_P \subseteq_F Init_P$ *and* $t \in \mathbb{N}$ *the dynamic analysis* $\langle \mathcal{R}, Exe(P,T_P,t)\rangle$ *is sound iff* $\exists T \in Core(\mathcal{R}([[P]]),\mathcal{R})$ *such that* $T \subseteq Exe(P,T_P,t)$*.*
- *For every semantic feature* $\Pi \in \wp(\Sigma^*/_{\mathcal{R}})$ *expressed in terms of equivalence classes of* $\mathcal{R}$*, we have that if* $Exe(P,T_P,t)$ *covers* $[[P]]$ *wrt* $\mathcal{R}$ *then we can precisely evaluate* $[[P]] \subseteq \Pi$ *by evaluating* $Exe(P,T_P,t) \subseteq \Pi$*.*

Thus, a dynamic analysis $\langle \mathcal{R}, Exe(P,T_P,t)\rangle$ is sound if $Exe(P,T_P,t)$ observes at least one execution trace for each one of the equivalence classes of the traces in $[[P]]$ for the relation $\mathcal{R}$.

In the worst case we have a different equivalence class for every execution trace of P.

When this happens, a sound dynamic analysis of property $\mathcal{R}$ on program P has to observe all possible execution traces. Of course, this is unfeasible in the general case. For this reason, if we want to protect a program from a dynamic analysis that is interested in the property $\mathcal{R}$, we have to diversify property $\mathcal{R}$ as much as possible among the execution traces of the program.

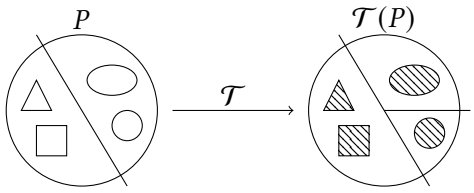

Figure 4.4: Transformation Potency

This allows us to define when a program transformation is *potent* wrt a dynamic analysis, namely when a program transformation forces a dynamic analysis to observe a wider set of traces in order to be sound. See [31] for the general notion of potency of a program transformation, i.e., a program transformation that foils a given attack (in our case a dynamic analysis).

Definition 6 (Potency). *A program transformation $\mathcal{T} : \mathbb{P} \rightarrow \mathbb{P}$ that preserves the denotational semantics of programs is potent for a program $P \in \mathbb{P}$ wrt an observation $\mathcal{R} \in eq(\Sigma^*)$ if the following two conditions hold:*

1. $\forall \sigma_1, \sigma_2 \in [[\mathcal{T}(P)]] : [\sigma_1]_{\mathcal{R}} = [\sigma_2]_{\mathcal{R}}$ *we have that* $\forall \nu_1, \nu_2 \in [[P]] : Den(\nu_1) = Den(\sigma_1) \wedge Den(\nu_2) = Den(\sigma_2)$ *then* $[\nu_1]_{\mathcal{R}} = [\nu_2]_{\mathcal{R}}$
2. $\exists \nu_1, \nu_2 \in [[P]] : [\nu_1]_{\mathcal{R}} = [\nu_2]_{\mathcal{R}}$ *for which* $\exists \sigma_1, \sigma_2 \in [[\mathcal{T}(P)]] : Den(\nu_1) = Den(\sigma_1) \wedge Den(\nu_2) = Den(\sigma_2)$ *such that* $[\sigma_1]_{\mathcal{R}} \neq [\sigma_2]_{\mathcal{R}}$

Figure 4.4 provides a graphical representation of the notion of potency. On the left we have the traces of the original program P partitioned according to the equivalence relation $\mathcal{R}$, while on the right we have the traces of the transformed program $\mathcal{T}(P)$ partitioned according to $\mathcal{R}$. Traces that are denotationally equivalent have the same shape (triangle, square, circle, oval), but are filled differently since they are, in general, different traces. The first condition means that the traces of $\mathcal{T}(P)$ that property $\mathcal{R}$ maps to the same equivalence class (triangle and square), are denotationally equivalent to traces of P that property $\mathcal{R}$ maps to the same equivalence class. This means that what is grouped together by $\mathcal{R}$ on $[[\mathcal{T}(P)]]$ was grouped together by $\mathcal{R}$ on $[[P]]$, modulo the denotational equivalence of traces. The second condition requires that there are traces of P (circle and oval) that property $\mathcal{R}$ maps to the same equivalence class and whose denotationally equivalent traces in $\mathcal{T}(P)$ are mapped by $\mathcal{R}$ to different equivalence classes. This means that a defense technique against dynamic analysis wrt a property $\mathcal{R}$ is successful when it transforms a program into a functionally equivalent one for which property $\mathcal{R}$ is more diversified among execution traces. This implies that it is necessary to collect more

execution traces in order for the analysis to be precise. At the limit we have an optimal defense technique when $\mathcal{R}$ varies at every execution trace.

Example 1. Consider the following programs P and Q that compute the sum of natural numbers from $x \geq 0$ to 49 (we assume that the inputs values for x are natural numbers).

```
P
input x;
sum := 0;
while x < 50
• ≀ X = [0, 49] ≀
   sum := sum + x;
   x := x + 1;
```

```
Q
input x;
n : = select(N,x)
x := x * n;
sum := 0;
while x < 50 * n
• ≀ X = [0, n ∗ 50 − 1] ≀
   sum := sum + x/n;
   x := x + n;
x := x/n;
```

Consider a dynamic analysis that observes the maximal value assumed by x at program point $\bullet$. For every possible execution of program P we have that the maximal value assumed by x at program point $\bullet$ is 49.

Consider a state $s \in \Sigma$ as a tuple $\langle \mathcal{I}, pp, C, [val_x, val_{sum}] \rangle$, where val_x and val_{sum} denote the current values of variables x and *sum* respectively. We define a function $\tau : \Sigma \longrightarrow \mathbb{N}$ that observes the value assumed by x at state s when s refers to program point $\bullet$, and function $Max : \Sigma^* \longrightarrow \mathbb{N}$ that observes the maximal value assumed by x at $\bullet$ along an execution trace:

$$\tau(s) \stackrel{\text{def}}{=} \begin{cases} val_x & \text{if } pp = \bullet \\ \emptyset & \text{otherwise} \end{cases} \qquad Max(\sigma) \stackrel{\text{def}}{=} max(\{\tau(s) \mid s \in \sigma\})$$

This allows us to define the equivalence relation $\mathcal{R}_{Max} \in eq(\Sigma^*)$ that observes traces wrt the maximal value assumed by x at $\bullet$, as $(\sigma, \sigma') \in \mathcal{R}_{Max}$ iff $Max(\sigma) = Max(\sigma')$.

The equivalence classes of $\mathcal{R}_{Max}$ are the sets of traces with the same maximal value assumed by x at $\bullet$. We can observe that all the execution traces of P belong to the same equivalence class of $\mathcal{R}_{Max}$. In this case, a dynamic analysis $\langle \mathcal{R}_{Max}, Exe(P, T_P, t) \rangle$ is sound if $Exe(P, T_P, t)$ contains at least one execution trace of P. This happens because the property that we are looking for is an invariant property of program executions and it can be observed on any execution trace.

Let us now consider program Q. Q is equivalent to P, i.e., $Den[[P]] = Den[[Q]]$, but the value of x is diversified by multiplying it by the parameter n. The guard and the body of the `while` are adjusted in order to preserve the functionality of the program. When observing property

$\mathcal{R}_{Max}$ on Q, we have that the maximal value assumed by x at program point $\bullet$ is determined by the parameter n generated in the considered trace. The statement `n:=select(N,x)` assigns to n a value in the range $[0, N]$ depending on the input value x.

We have that the traces of program Q are grouped by $\mathcal{R}_{Max}$ depending on the value assumed by n. Thus, $\mathcal{R}([[Q]])$ contains an equivalence class for every possible value assumed by n during execution. This means that the transformation that rewrites P into Q is potent according to Definition 6.

Dynamic analysis $\langle \mathcal{R}_{Max}, Exe(Q, T_Q, t)\rangle$ is sound if $Exe(Q, T_Q, t)$ contains at least one execution trace for each of the possible values of n generated during execution.

4.4 Model Validation

In this section, we show how the proposed framework can be used to model existing code obfuscation techniques. In particular, we model the way these transformations deceive dynamic analysis of control flow and data flow properties of programs. We also show how the measures of code coverage used by dynamic analysis tools can be naturally interpreted in the proposed framework.

4.4.1 Control Flow Analysis

The control flow graph CFG of a program P is a graph $CFG_P = (V, E)$ where each node $v \in V$ is a pair (pp, C) denoting a statement C at program point pp in P, and $E \subseteq V \times V$ is the set of edges such that $(v_1, v_2) \in E$ means that the statement in v_2 could be executed after the statement in v_1 when running P. Thus, we define the domain of nodes as $Nodes \stackrel{\text{def}}{=} PP \times Com$, and the domain of edges as $Edges \stackrel{\text{def}}{=} Nodes \times Nodes$.

Dynamic Extraction of the Control Flow Graph.

It is possible to dynamically construct the CFG of a program by observing the commands that are executed and the edges that are traversed when the program runs. Let us define $\eta : \Sigma \longrightarrow Nodes$ that observes the command to be executed together with its program point, namely $\eta(s) = \eta(\langle \mathcal{I}, pp, C, m\rangle) \stackrel{\text{def}}{=} (pp, C)$. By extending this function on traces we obtain function $path : \Sigma^* \longrightarrow Nodes \times Edges$ that extracts the path of the CFG corresponding to the considered execution trace, abstracting from the number of times that an edge is traversed or a node is computed:

$$path(\sigma) \stackrel{\text{def}}{=} (\{\eta(s) \mid s \in \sigma\}, \{(\eta(s), \eta(s')) \mid ss' \preceq \sigma\})$$

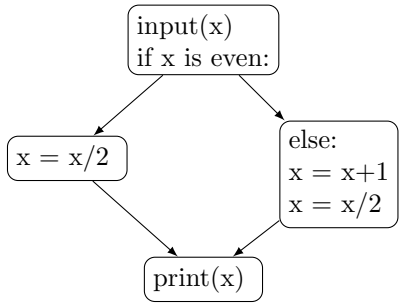

Figure 4.5: CFG of P

where $s \in \sigma$ means that s is a state that appears in trace σ and $ss' \preceq \sigma$ means that s and s' are successive states in σ.

This allows us to define the equivalence relation $\mathcal{R}_{CFG} \in eq(\Sigma^*)$ that observes traces up to the path that they define, as $(\sigma, \sigma') \in \mathcal{R}_{CFG}$ iff $path(\sigma) = path(\sigma')$. Indeed, $\mathcal{R}_{CFG}$ groups together those traces that execute the same set of nodes and traverse the same set of edges, abstracting from the number of times that nodes are executed and edges are traversed.

The CFG of a program P can be defined as the union of the paths of its execution traces, namely $CFG_P = \bigsqcup\{path(\sigma) \mid \sigma \in [\![P]\!]\}$, where the union of graphs is defined as $(V_1, E_1) \sqcup (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$. The dynamic extraction of the CFG of a program P from the observation of a set $X \subseteq_F [\![P]\!]$ of execution traces, is given by $\bigsqcup\{path(\sigma) \mid \sigma \in X\}$. In the general case we have $\bigsqcup\{path(\sigma) \mid \sigma \in X\} \subseteq CFG_P$.

Preventing Dynamic CFG Extraction.

Control code obfuscations are program transformations that modify the program's control flow in order to make it difficult for an adversary to analyze the flow of control of programs [31]. According to Section 4.3, a program transformation $\mathcal{T} : \mathbb{P} \to \mathbb{P}$ is a potent defense against the dynamic extraction of the CFG of a program P when $\mathcal{T}$ diversifies the paths taken by the execution traces of $\mathcal{T}(P)$ wrt the paths taken by the traces of P.

In the following, we show how two known defense techniques for preventing dynamic analysis actually work by diversifying program traces with respect to property $\mathcal{R}_{CFG}$.

We start with a simple program P that computes integer division.

```
1 input(x)
2 if x is even:
```

```
    x = x/2
else:
    x = x+1
    x = x/2
printf(x)
```

Listing 4.1: P

The CFG of P can be seen in Figure 4.5.

Range Dividers: Range Divider (*RD*) is a transformation designed to prevent dynamic symbolic execution and it is an efficient protection against the dynamic extraction of the CFG [9].

RD relies on the existence of n program transformations $\mathcal{T}_i : \mathbb{P} \rightarrow \mathbb{P}$ with $i \in [1, n]$ that:

1. Preserve the denotational semantics of programs:
   $\forall P \in \mathbb{P}, i \in [1, n] : Den[[P]] = Den[[\mathcal{T}_i(P)]]$

2. Modify the paths of the CFG of programs in different ways:
   $\forall P \in \mathbb{P}, \forall i, j \in [1, n]: \ \mathcal{R}_{CFG}([[\mathcal{T}_i(P)]]) = \mathcal{R}_{CFG}([[\mathcal{T}_j(P)]]) \Rightarrow i = j.$

Given a program P, the *RD* transformation works by inserting a `switch` control statement with n cases whose condition depends on program inputs. Every case of the `switch` contains a semantically equivalent version $\mathcal{T}_i(P)$ of P that is specialised wrt the input values. Thus, depending on the input values we would execute one of the diversified programs $\mathcal{T}_1(P), \ldots, \mathcal{T}_n(P)$. Since for each variant $\mathcal{T}_i(P)$ with $i \in [1, n]$ the set of execution traces are mapped by $\mathcal{R}_{CFG}$ into different equivalent classes, we have that property $\mathcal{R}_{CFG}$ has been diversified among the traces of $RD(P)$. Thus, the transformation *RD* is potent wrt $\mathcal{R}_{CFG}$ and harms the dynamic extraction of the CFG.

The graph in Figure 4.6 represents the CFG of program P transformed by *RD*. The CFG of program $RD(P)$ has four different paths depending on the value of the input variable x. Each one of these paths is functionally equivalent to the corresponding path in P (`case 0` and `case 2` are equivalent to the path taken when x is even, while `case 1` and `case 3` are equivalent to the path taken when x is odd). We can easily observe that in this case the paths of $RD(P)$ have been diversified wrt the paths of P. Indeed, a dynamic analysis has to observe two execution traces to precisely build the CFG for P, while four traces are need to precisely build the CFG of $RD(P)$.

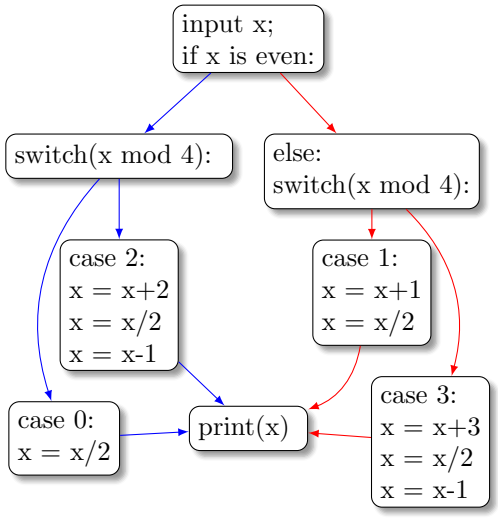

Figure 4.6: CFG of $RD(P)$

Gadget Diversification: In [121] the authors propose a program transformation, denoted $GD : \mathbb{P} \rightarrow \mathbb{P}$ that hinders the dynamic CFG analysis.

GD starts by identifying a sequence Q_{seq} of sequential command (no branches) in program P. Next, GD assumes to have access to a set of diversifying transformations $\mathcal{T}_i : \mathbb{P} \rightarrow \mathbb{P}$ with $i \in [1, n]$ that diversify command sequences while preserving their functionality. These transformations are then applied to portions of Q_{seq} in order to generate a wide set $S_{seq} = \{Q_1..Q_m\}$ of command sequences where each $Q_j \in S_{seq}$ is functionally equivalent to Q_{seq}, while every pair $Q_j, Q_l \in S_{seq}$ are such that $\mathcal{R}_{CFG}([[Q_j]]) \neq \mathcal{R}_{CFG}([[Q_l]])$. This means that each execution trace generated by the run of a sequence in S_{seq} belongs to a different equivalence class wrt relation $\mathcal{R}_{CFG}$, while being denotationally equivalent by definition.

Transformation GD proceeds by adding a `branching function` to the original program P that, depending on the input values, deviates the control flow to one of the sequences of commands in S_{seq}. Thus, depending on the input values, GD diversifies the path that is executed. This makes the transformation GD potent wrt $\mathcal{R}_{CFG}$ according to the proposed framework.

A simple example of GD can be observed in the graph of Figure 4.7, where the original program is transformed to reveal a peculiar CFG structure. The `branch` function is symbolized here as the central block from which all other blocks are called and to which all other blocks return (except for `print(x)` which represents the end of the program). The `branch` function

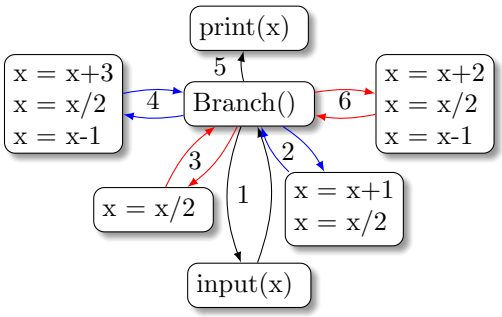

Figure 4.7: CFG of $GD(P)$

will only allow the following sequences of edges:

$$odd(x) \rightarrow \left\{ \begin{array}{l} 1 \rightarrow 2 \rightarrow 5 \\ 1 \rightarrow 4 \rightarrow 5 \end{array} \right\} \qquad even(x) \rightarrow \left\{ \begin{array}{l} 1 \rightarrow 3 \rightarrow 5 \\ 1 \rightarrow 6 \rightarrow 5 \end{array} \right\}$$

We can easily observe that the paths of $GD(P)$ have been diversified wrt the paths of P and while the dynamic construction of the CFG for P requires to observe two execution traces, we need to observe 4 execution traces to precisely build the CFG of $GD(P)$.

4.4.2 Code Coverage

Most dynamic algorithms use code coverage to measure the potential soundness of the analysis [3]. Intuitively, given a program P and a partial observation $Exe(P, T_P, t)$ of its execution traces, code coverage wants to measure the amount of program behaviour considered by $Exe(P, T_P, t)$ wrt the set of all possible behaviours $[[P]]$. We now describe some known code coverage measures.

Statement coverage considers the statements of the program that have been executed by the traces in $Exe(P, T_P, t)$.

This is a function $st : \Sigma^* \rightarrow Nodes$ that collects commands annotated with their program point, that are executed along a considered trace: $st(\sigma) \stackrel{\text{def}}{=} \{\eta(s) \mid s \in \sigma\}$. This allows us to define the equivalence relation $\mathcal{R}_{st} \in eq(\Sigma^*)$ that groups together traces that execute the same set of statements.

Count-Statement coverage considers how many times each statement of the program has been executed by the traces in $Exe(P, T_P, t)$. Thus, it can be formalised in terms of an equivalence

relation $\mathcal{R}^{+}_{st} \in eq(\Sigma^*)$ that groups together traces that execute the same set of statements the same amount of times. It is clear that relation $\mathcal{R}^{+}_{st}$ is finer than relation $\mathcal{R}_{st}$, namely $\mathcal{R}^{+}_{st} \sqsubseteq \mathcal{R}_{st}$.

Path coverage observes the nodes executed and edges traversed by the traces in $Exe(P, T_P, t)$.

This precisely corresponds to the observation of property $\mathcal{R}_{CFG} \in eq(\Sigma^*)$ defined above, where the paths of the CFG are observed by abstracting the number of times that edges are traversed. It is clear that relation $\mathcal{R}_{CFG}$ is finer than relation $\mathcal{R}_{st}$, namely $\mathcal{R}_{CFG} \sqsubseteq \mathcal{R}_{st}$.

Count-Path coverage considers the different paths in $Exe(P, T_P, t)$, where the number of times that edges are traversed in a trace is taken into account. This can be formalised in terms of an equivalence relation $\mathcal{R}^{+}_{CFG} \in eq(\Sigma^*)$ that groups together traces that execute and traverse the same nodes and edges the same number of times. It is clear that relation $\mathcal{R}^{+}_{CFG}$ is finer than relation $\mathcal{R}_{CFG}$, namely $\mathcal{R}^{+}_{CFG} \sqsubseteq \mathcal{R}_{CFG}$.

Trace coverage considers the traces of commands that have been executed abstracting from the memory map. In this case we can define the code coverage in terms of function $trace : \Sigma^* \longrightarrow Com \times PP$ defined as $trace(\epsilon) \stackrel{\text{def}}{=} \epsilon$ and $trace(s\sigma) \stackrel{\text{def}}{=} \eta(s)trace(\sigma)$. The equivalence relation $\mathcal{R}_{trace} \in eq(\Sigma^*)$ is such that $(\sigma, \sigma') \in \mathcal{R}_{trace}$ if $trace(\sigma) = trace(\sigma')$. This equivalence relation is finer than $\mathcal{R}^{+}_{CFG}$ since it keeps track of the order of execution of the edges.

In order to avoid false negatives, dynamic analysis algorithms automatically look for inputs whose execution traces have to exhibit new behaviours with respect to the code coverage metric used (e.g., they have to execute new statements or execute them a different number of times, traverse new edges or change the number of times edges are traversed, or execute nodes in a different order). This can be naturally formalised in our framework. Given a set $Exe(P, T_P, t)$ of observed traces, an automatically generated input increases the code coverage measured as $\mathcal{R}_{st}$ (or $\mathcal{R}^{+}_{st}, \mathcal{R}_{CFG}, \mathcal{R}^{+}_{CFG}, \mathcal{R}_{trace}$) if the execution trace σ generated by the input is mapped in a new equivalence class of $\mathcal{R}_{st}$ (or $\mathcal{R}^{+}_{st}, \mathcal{R}_{CFG}, \mathcal{R}^{+}_{CFG}, \mathcal{R}_{trace}$), namely in an equivalence class that was not observed by traces in $Exe(P, T_P, t)$, namely if $[\sigma]_{\mathcal{R}_{st}} \notin \mathcal{R}_{st}(Exe(P, T_P, t))$ (analogously for $\mathcal{R}^{+}_{st}, \mathcal{R}_{CFG}, \mathcal{R}^{+}_{CFG}, \mathcal{R}_{trace}$). We have seen above that some of the common measures for code coverage can be expressed in terms of semantic program properties with different degrees of precision $id \sqsubseteq \mathcal{R}_{traces} \sqsubseteq \mathcal{R}^{+}_{CFG} \sqsubseteq \mathcal{R}_{CFG} \sqsubseteq \mathcal{R}_{st}$. This means, for example, that automatically generated inputs could add coverage for $\mathcal{R}^{+}_{CFG}$ but not for $\mathcal{R}_{st}$. Indeed, a new input generates a new behaviour depending on the metric used for code coverage.

Fuzzing and dynamic symbolic execution are typical techniques used by dynamic analysis to automatically generate inputs in order to extend code coverage. The metrics that fuzzing and symbolic execution use to measure code coverage are sometimes slight variations of the ones mentioned earlier.

Fuzzing: The term fuzzing refers to a family of automated input generating techniques that are widely used in the industry to find vulnerabilities and bugs in all kinds of software [132]. In general, a fuzzer aims at discovering inputs that generate new behaviors, thus one measure of success for fuzzers is code coverage. Simple statement coverage is rarely a good choice, since crashes do not usually depend on a single program statement, but on a specific sequence of statements [151]. Most fuzzing algorithms choose to define their own code coverage metric. American Fuzzy Lop (AFL) is a state of the art fuzzer that has seen extensive use in the industry in its base form, while new fuzzers are continuously built on top of it [128]. The measure used by AFL for code coverage lays between path and count-path coverage as it approximates the number of times that edges are traversed by specified intervals of natural numbers ([1], [2], [3], $[4-7]$, $[8-15]$, $[16-31]$, $[32-127]$, $[128,\infty]$). Libfuzzer [126] and honggfuzz [133] employ count-statement coverage. To the best of our knowledge trace coverage is never used as it is unfeasible in practice [58].

Dynamic Symbolic Execution: DSE is a well known dynamic analysis technique that combines concrete and symbolic execution [68]. DSE typically starts by executing a program on a random input and then generates branch conditions that take into account the executed branches. When execution ends, DSE looks at the last branch condition generated and uses a theorem prover to solve the negated predicate in order to explore the branch that was not executed. This is akin to symbolic execution, but DSE can use the concrete values obtained in the execution to simplify the job of the theorem prover. The ideal goal of DSE is to reach path coverage, which is always guaranteed if the conditions in the target program only contain linear arithmetics [68]. Thus, the efficacy of DSE in generating new inputs is measured in terms of path coverage formalised as $\mathcal{R}_{CFG}$ in our framework.

Let us denote with $\mathcal{R} \in eq(\Sigma^*)$ the equivalence relation modelling the code coverage metric used either by fuzzing or symbolic execution or any other algorithm for input generation. When $Exe(P, T_P, t)$ covers P wrt $\mathcal{R}$, we have that the fuzzer or symbolic execution algorithm has found all the inputs that allow us to observe the different behaviours of P wrt $\mathcal{R}$. In general, a dynamic analysis may be interested in a property $\mathcal{R}_A \in eq(\Sigma^*)$ that is different from the property $\mathcal{R}$ used to measure code coverage. When $\mathcal{R} \sqsubseteq \mathcal{R}_A$ we have that if $Exe(P, T_P, t)$ covers P wrt $\mathcal{R}$, then $Exe(P, T_P, t)$ covers P also wrt $\mathcal{R}_A$ and this means that the code coverage metric $\mathcal{R}$ can help in limiting the number of false negative of the dynamic analysis $\langle \mathcal{R}_A, Exe(P, T_P, t)\rangle$. When $\mathcal{R} \not\sqsubseteq \mathcal{R}_A$ then a different metric for code coverage should be used (for example $\mathcal{R}_A$ itself).

$\mathcal{T}(P)$
input x;
x := 2*x;
sum := 0;
while x < 2*50
• $X = [x, 2 * 50 - 1]$
  sum := sum + x/2;
  x := x + 2;
x:= x/2;

$\mathcal{T}_n(P)$
input x;
x := n*x;
sum := 0;
while x < n*50
• $X = [x, n * 50 - 1]$
  sum := sum + x/n;
  x := x + n;
x:= x/n;

$\mathcal{T}_H(P)$
input x;
n := select(N,x);
x := H^e(n,x);
sum := H^e(n,0);
while x $<_H$ H^e(n,50)
• $X = [x, H^e(n, 50) - 1]$
  sum := sum $+_H$ x;
  x := x $+_H$ H^e(n,1);
x:= H^d(x);

Figure 4.8: From the left: programs $\mathcal{T}(P)$, $\mathcal{T}_n(P)$ and $\mathcal{T}_H(P)$

4.4.3 Harming Dynamic Data Analysis

Data obfuscation transformations change the representation of data with the aim of hiding both variable content and usage. Usually, data obfuscation requires the program code to be modified, so that the original data representation can be reconstructed at runtime. Data obfuscation is often achieved through data encoding [31, 123].

More specifically, in [52, 92] data encoding for a variable x is formalised as a pair of statements: encoding statement $C_{enc} = x := f(x)$ and decoding statement $C_{dec} = x := g(x)$ for some function f and g, such that $C_{dec}; C_{enc} = skip$. According to [52, 92] a program transformation $\mathcal{T}(P) \stackrel{\text{def}}{=} C_{dec}; t_x(P); C_{enc}$ is a data obfuscation for x where t_x adjusts the computations involving x in order to preserve program's functionality, namely $Den[[P]] = Den[[C_{dec}; t_x(P); C_{enc}]]$. In Fig. 4.8 we provide a simple example of data obfuscation from [52, 92] where $C_{enc} = x := 2 * x$ and $C_{dec} = x := x/2$ and $\mathcal{T}(P) = x := x/2; t_x(P); x := 2 * x$ and program P is the one considered in Example. 1:

```
input x;
sum := 0;
while x < 50
    sum := sum + x;
    x := x + 1;
```

This data transformation induces imprecision in the static analysis of the possible values assumed by x at program point •, right inside the while loop.

It is easy to see how the static analysis of the interval of values of x at program point • in $\mathcal{T}(P)$ is different and wider (it contains spurious values) than the interval of possible values of

x at $\bullet$ in P. However, the dynamic analysis of properties on the values assumed by x during execution at the different program points (e.g., maximal/minimal value, number of possible values, interval of possible values) has not been hardened in $\mathcal{T}(P)$. The values assumed by x at $\bullet$ in $\mathcal{T}(P)$ are different from the values assumed by x at $\bullet$ in P but these properties on the values assumed by x are precisely observable by dynamic analysis on $\mathcal{T}(P)$.

The transformation $\mathcal{T}(P)$ changes the properties of data values wrt P, but it does it in an invariant way: during every execution of $\mathcal{T}(P)$ we have that x is iteratively incremented by 2 and the guard of the loop becomes $x < 2 * 50 - 1$, and this is observable on any execution of $\mathcal{T}(P)$. This means that by dynamic analysis we could learn that the maximal value assumed by x is $99 (= 2 * 50 - 1)$. Thus, transformation $\mathcal{T}$ is not potent wrt properties of data values according to Definition 6 since it does not diversify the properties of values assumed by variables among traces.

In order to hamper dynamic analysis we need to diversify data among traces, thus forcing dynamic analysis to observe more execution traces to be sound. We could do this by making the encoding and decoding statements parametric on some natural number n as described by the second program $\mathcal{T}_n(P) = x := x/n; t_{x,n}(P); x := n * x$ in Fig. 4.8 (which is the same as Q in Example. 1). Indeed, the parametric transformation $\mathcal{T}_n(P)$ is potent wrt properties that observe data values since it diversifies the values assumed by x among different executions thanks to the parameter n. For example, to observe the maximal value assumed by x in $\mathcal{T}_n(P)$ we should observe an execution for every possible value of n.

This confirms what is observed in [123]: existing data obfuscation makes static analysis imprecise but it is less effective against dynamic analysis. Interpreting data obfuscation in our framework allows us to see that, in order to hamper dynamic analysis, data encoding needs to diversify among traces. This can be done by making the existing data encoding techniques parametric.

Homomorphic encryption: As argued above, in order to preserve program functionality the original program code needs to be adapted to the encoding. In general, automatically deriving the modified code $t_x(P)$ for a given encoding on every possible program may be complicated. In this setting, an ideal situation is the one provided by fully homomorphic encryption where any operation on the original data has its respective version for the encrypted data. It has been proven that fully homomorphic encryption is possible on any circuit [61]. Let H^e and H^d be the fully homomorphic encryption and decryption procedures.

We could design a data obfuscation for the variables in P as $H^d; P_H; H^e$ where the program variables are encrypted with H^e, the computation is carried on the encrypted values by using homomorphic operations (denoted with subscript $_H$), and at the end the final values of the

variables are decrypted with H^d. Thus, the original program P and P_H are exactly the same programs where the operations have been replaced by their homomorphic version.

In Fig. 4.8 on the right we show how a homomorphic encoding of program P would work, where the subscript $_H$ denotes the homomorphic operations on the encrypted values. Encryption and decryption procedures have a random nature and use a key (that may be dependent on input values). Thus, the values of encrypted data varies among program traces. Moreover, since successive encryptions of the same values would lead to different encrypted values, we have that re-runs on the same values would generate different encrypted values. This proves that homomorphic encryption could be useful to design a potent data obfuscation against dynamic analysis: as it can diversify the encrypted data values among traces and the original values are retrieved only at the end of the computation.

Abstract Software Watermarking: In [40] the authors propose a sophisticated software watermarking algorithm and prove its resilience against static program analysis. The watermark can be extracted only by analysing the program on a specific congruence domain that acts like a secret key. The authors discuss some possible countermeasures against dynamic analysis that could reveal the existence of the watermark (and then remove it). Interestingly, the common idea behind these countermeasures is diversification of the property of data values that the dynamic analyzer observe.

4.5 Poisons and Antidotes

In this section we introduce a novel class of program transformations that are designed to disrupt the normal flow of values in a program. The general idea is simple: a source of randomness is added to a program and is made to majorly affect its semantics, usually by using it to modify the input value (which hopefully will influence the output of the program). The randomness is then removed with subsequent operations before the program return, making the denotational semantics of the modified program equal to those of the original program. These two phases of the obfuscation are called "poisons" and "antidotes", signifying the addition of the randomness (which negatively affects the semantics of the program) and its subsequent removal before the return point.

The language used in examples is a simplified subset of Python that only considers operations between integers. To this language we add a *rand* function that, given two integers $a.b$ such that $a \leq b$ will return $i \in \mathbb{N}.a \leq i \leq b$, not unlike the existing Python function *randint* in the package *random*. It is easy to see how in the future this could be made to generalize to other value types and operations (e.g. summing numbers can be extended to concatenating strings).

4.5.1 Poisons: Making Programs Run Amok

Poisoning the semantics of a program means introducing randomness to a variable that is directly used to calculate the output. This is done by injecting additional code at a set point in the program, guaranteeing that the output will be affected by the added randomness.

It is important to identify and study different types of poisons in order to diversify the semantics as much as possible and afterwards to craft the correct antidote. In this work, we will limit the transformations to two types of poisons (and subsequent antidotes): additive and multiplicative.

Additive Poison ($\mathcal{T}_P^+$) As the name implies, this poison involves adding a randomly generated value to the poison targets (usually the input variable(s)).

As a simple example we can show the difference between the two types of poisons and antidotes with the same program shown in the introduction to this chapter. Consider the program P:

```
x = input()
y = x * x
return y
```

Listing 4.2: P

This is a program that returns the square of the input value. Using an additive poison means that we add a random n to the initial variable x, making the semantics behave wildly, or, as wild as they can be with the few choices granted by the function *rand*.

```
n := rand(2, 5)
i = input()
x = i + n
y = x * x
return y
```

Listing 4.3: $\mathcal{T}_P^+(P) = P_1$

Naturally the semantics (both denotational and trace semantics) of the program have radically changed as it returns an integer between $(i+2)^2$, $(i+3)^2$, $(i+4)^2$ and $(i+5)^2$. Before showing the related antidote, we introduce the second type of poison.

Multiplicative Poison ($\mathcal{T}_P^*$) As the name implies, this transformation multiplies the input variable by a random value n. Continuing the example of the program P at Listing 4.2, we can easily insert a multiplicative poison:

```
n = rand(2, 5)
x = 2 * n
y = x * x
return y
```

Listing 4.4: $\mathcal{T}_P^*(P) = P_1'$

This resulting program can either return $(i * 2)^2$, $(i * 3)^2$, $(i * 4)^2$, or 10^2.

As with real-world poisons, these transformations will cause trouble if left unchecked. For this reason we define the antidotes in the following section.

4.5.2 Antidotes: Preserving Input-Output Semantics

An antidote undoes the effects of the poison: by restoring the denotational semantics of the original program, it allows a run-time convergence of the semantics into its original output. This transformation can occur at any point after a poison, but is trivial to implement if placed immediately after the poison. If the antidote is applied too soon, it will have a very localized impact on the overall trace semantics of the program. If the poison is introduced at the beginning of the program and the antidote is added right before the end, then the transformation has the best obfuscating impact on the overall semantics of the program.

On the other hand, the longer the wait after the poison, the hardest it is to properly concoct a suitable antidote.

Additive Antidote ($\mathcal{T}_A^+$) Antidotes depend on the poison type, so additive poisons can be "cured" by additive antidotes. A simple example of an additive antidote, when applied to P_1, is as follows:

```
r = rand(2, 5)
i = input()
x = i + r
y = x * x
w = r * r
t = r * i
z = t + t
a = w + z
y = y - a
return y
```

Listing 4.5: $P_2 = \mathcal{T}_A^+(\mathcal{T}_P^+(P)) = \mathcal{T}_A^+(P_1)$

Applying the antidote returns the program to its un-poisoned state and the program resumes computing i^2. Not surprisingly, the additive antidote is always a subtraction and is represented

by the variable a in the code above. The variable involved in the antidote contains all the spurious information that was added to the computation by the poison.

Consider the trace semantics of the program ($\mathcal{S}[\![P]\!]$) as a function that returns all the states that have been traversed during the computation. Then the denotational semantics ($\mathcal{D}[\![P]\!]$) is the input-output semantics, meaning that it only returns the final computed value of the program.

For this example we can then write:

$$\mathcal{D}[\![P]\!] = \mathcal{D}[\![P_2]\!] \wedge \mathcal{S}[\![P]\!] \neq \mathcal{S}[\![P_2]\!] \tag{4.6}$$

Multiplicative Antidote ($\mathcal{T}_A^*$) Taking the program at Listing 4.4 we can apply the multiplicative antidote and construct $\mathcal{T}_A^*(\mathcal{T}_P^*(P))$:

```
r = rand(2, 5)
i = input()
x = i * r
y = x * x
a = r * r
y = y / a
return y
```

Listing 4.6: $P_3 = \mathcal{T}_A^*(\mathcal{T}_P^*(P)) = \mathcal{T}_A^*(P_1')$

Again, the semantic equivalence to P is very easy to prove. Starting from the last value of y and backtracking every instruction of the previous code snippet:

$$y = y/a \xrightarrow{[a=r*r]} y = y/r^2 \xrightarrow{y=x*x} y = x^2/r^2 \xrightarrow{x=i*r} y = i^2 * r^2/r^2 = i^2 \tag{4.7}$$

Adding this to the previous example we have:

$$\mathcal{D}[\![P]\!] = \mathcal{D}[\![P_2]\!] = \mathcal{D}[\![P_3]\!] \wedge \mathcal{S}[\![P]\!] \neq \mathcal{S}[\![P_2]\!] \neq \mathcal{S}[\![P_3]\!] \tag{4.8}$$

Of course, the strength of this obfuscation is not only in the introduction of new trace semantics for the programs affected, but rather the fact that the values observed at each execution can change randomly, leading to a non deterministic trace semantics while maintaining a deterministic denotational semantics.

The previous examples have been constructed using simple programs for illustrative purposes. A better example of poisons and antidotes can be shown in an implementation of the factorial function:

```
def fact(x):
```

```
    z = x
    y = 1
    while z > 0:
        y = y * z
        z = z - 1
    return y
```

Listing 4.7: Factorial function in Python

A version of this function that has been transformed with poisons and antidotes is the following program, which will be called $fact_A^P$:

```
def rand_fact_n(x):
    i = 0
    n = random.randint(1, 4)    #r1 -> probabilistic branching
    z = x * n                   #p1 -> poison
    y = n
    while z > 0:
        y = y * z
        z = z - n
    y = t1(x, y, n)             #t1 -> confluence function
    return y

def t1(x, y, n):
    z = x
    c = n
    while z > 0:
        z = z - 1
        c = c * n
    y = int(y / c)              #a1 -> antidote
    return y
```

Listing 4.8: $fact_A^P = \mathcal{T}_A^*(\mathcal{T}_P^*(fact))$

This is evidently a multiplicative poison followed by a multiplicative antidote. The construction of the antidote has been rendered explicit by defining a confluence function, and the actual antidote application is just the last operation of this function (y = int(y/c)).

Another reason to define a separate confluence function is that it makes it easier to show what happens when we poison a program twice. We explained earlier how it makes sense to poison variables that influence the output of the program, which means that we could locate more than one of such variables. Another important decision has to be made on the location of the poisoning. In this first example, the variable that we have poisoned is the input variable x, which was poisoned right at the start of the function, before the while loop.

We can now perform a more complex poison and antidote transformation to the newly generated program. The following program is $fact_{A^2}^{P^2}$:

```
1  def rand_fact_n_m(x):
2      n = random.randint(1, 4)     #r1 -> first probabilistic branching
3      z = x * n                    #p1 -> first poison (multiplicative)
4      y = n
5      m = random.randint(0, 3)     #r2 -> second probabilistic branching
6      while z > 0:
7          y = y + m                #p2 -> second poison (additive)
8          y = y * z
9          z = z - n
10     y = t2(x, y, n, m)           #t2 -> confluence function for r2
11     y = t1(x, y, n)              #t1 -> confluence function for r1
12     return y
13
14 def t2(x, y, n, m):
15     z = x * n
16     c = 0
17     while z > 0:
18         c = c + m
19         c = c * z
20         z = z - n
21     y = y - c                    #a1 -> first antidote (additive)
22     return y
23
24 def t1(x, y, n):
25     z = x
26     c = n
27     while z > 0:
28         z = z - 1
29         c = c * n
30     y = int(y / c)               #a2 -> second antidote (multiplicative)
31     return y
```

Listing 4.9: $fact_{A^2}^{P^2} = \mathcal{T}_A^+(\mathcal{T}_P^+(fact_A^P))$

This time the poisoned variable is the output variable y, and the poison location is line 7, in the while loop. It should be noted that in this example there are both additive and multiplicative poisons and antidotes. Another interesting thing to notice is that the first antidote applied is for the second poison. This is a common behaviour for this transformation, as the last poison added is always the first one to be cured.

The proof of the correctness for this last example could run for many pages, for the more curious we set up a Google Colab page that can easily be run in the browser in order to check

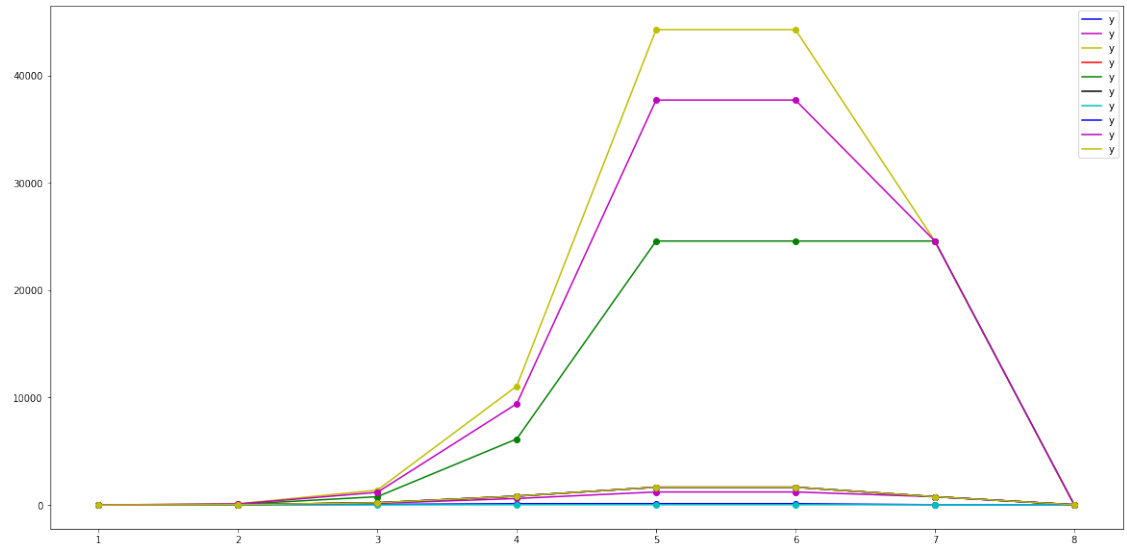

Figure 4.9: Monitoring the value of y in $fact_{A^2}^{P^2}$

the validity of the function, along with more information about the execution traces [97]. In Figure 4.9 we show the value of the variable y in multiple runs of the program we just generated, with 4 as the input value. It can be observed that the values observed converge twice, first partially at 7 and then finally at the end of the function. In the bulk of the program, the values of y vary wildly as per the design of poisons and antidotes.

4.5.3 Poisons and Antidotes Against Dynamic Analysis

We have anticipated in the introduction to this chapter that the discovery of poisons and antidotes has led us to generate the formal framework for dynamic analysis that has become the real focus of this work. The initial motivation was that, while intuitively it is easy to imagine how introducing randomness to the executed traces could disrupt dynamic analysis, in practice, there is no formal way of proving it.

In an anti-climatic twist, our framework cannot prove that poisons and antidotes work against dynamic analysis. Intuitively, the reason is that the transformation does not force the analysis to explore more paths to observe the same property, at least not in a deterministic way. The randomness added to the semantics provide more of a probabilistic type of protection against dynamic analysis. It can be argued that it is *likely* that changing the values at every

run will confuse any analysis that monitors the computed values in execution traces, but it is also possible that the property is revealed in one run with a stroke of luck. This leads us to believe that our framework can be extended to consider probabilistic code protections. The potency of the transformation can then be expressed with the likelihood of it succeeding, or with the expected value of executions needed to have a successful analysis.

Another weakness of poisons and antidotes is that, at least for now, it consists only of a few templates that work with specific operations so they cannot be reliably applied to every program. This is not unlike certain obfuscations, for example, EncodeArithmetics introduced in Chapter 3 cannot be applied to programs that have no clear arithmetic operations. Poisons and antidotes is even more limited than that, and its potency still has to be ascertained against most analyses. The added computational overhead could also be a problem, as can be seen in the last example for the factorial function.

4.6 Related Work

To the best of our knowledge we are the first to propose a formal framework for dynamic analysis efficacy based on semantic properties. Other works have proposed more empirical ways to assess the impact of dynamic analysis.

Evaluating Reverse Engineering. Program comprehension guided by dynamic analysis has been evaluated with specific test cases, quantitative measures and the involvement of human subjects [35].

For example, comparing the effectiveness of static analysis and dynamic analysis on the feature location task has been carried out through experiments involving 2 teams of analysts solving the same problem with a static analysis and a dynamic analysis approach respectively [142].

In order to compare the effectiveness of different reverse engineering techniques (which often employ dynamic analysis), Sim et al. propose the use of common benchmarks [131].

The efficacy of protections against human subjects has been evaluated in a set of experiments by Ceccato et al., finding that program executions are important to understand the behavior of obfuscated code [25]. Our approach characterizes dynamic attacks and protections according to their semantic properties, which is an orthogonal work that can be complemented by more empirical approaches.

Obfuscations Against Dynamic Analysis. One of the first works tackling obfuscations specifically geared towards dynamic analysis is by Schrittwieser and Katzenbeisser [121]. Their approach adopts some principles of software diversification in order to generate additional paths in the CFG that are dependent on program input (i.e. they do not work for other inputs).

Similar to this approach, Pawlowski et al. [110] generate additional branches in the CFG but add non-determinism in order to decide the executed path at runtime. Both of these works empirically evaluate their methodology and classify it with potency and resilience, two metrics introduced by Collberg et al. [33]. Banescu et al. empirically evaluated some obfuscations against dynamic symbolic execution (DSE) [9], finding that DSE does not suffer from the addition of opaque branches since they do not depend on program input. To overcome this limitation they propose the Range Dividers obfuscation that we illustrated in Section 4.4. A recent work by Ollivier et al. refines the evaluation of protections against dynamic symbolic execution with a framework that enables the optimal design of such protections [106].

All these works share with us the intuition that dynamic analysis suffers from insufficient path exploration and they prove this intuition with extensive experimentation. Our work aims at enabling the formal study of these approaches.

Formal Systems. Dynamic taint analysis has been formalized by making the taint information in the program semantics explicit [124].

Their work focuses on writing correct algorithms and shows some possible pitfalls of the various approaches. Ochoa et al. [103] use game theory to quantify and compare the effectiveness of different probabilistic countermeasures with respect to remote attacks that exploit memory-safety vulnerabilities.

In our work we model MATE attacks. Shu et al. introduce a framework that formalizes the detection capability in existing anomaly detection methods [130]. Their approach equates the detection capability to the expressiveness of the language used to characterize normal program traces.

4.7 Conclusions

This work represents the first step towards a formal investigation of the precision of dynamic analysis in relation to dynamic code attack and defenses. The results that we have obtained so far confirm the initial intuition: *diversification of the execution traces is the key for harming dynamic analysis*. Since dynamic analysis generalises what it learns from a partial observation of program behaviour, diversification makes this generalisation less precise (dynamic analysis cannot consider what it has not observed). We think that this work would be the basis for further interesting investigations. Indeed, there are many aspects that still need to be understood for the development of a complete framework for the formal specification of the precision of dynamic analysis (no false negatives), and for the systematic development of program transformations that induce imprecision.

We plan to consider more sophisticated properties than the ones that can be expressed as

equivalence relations. It would be interesting to generalise the proposed framework wrt to any semantic property that can be formalised as a closure operator on trace semantics. The properties that we have considered so far correspond to the set of atomistic closures where the abstract domain is additive. We would like to generalise our framework to properties modelled as abstract domains and where the precision of dynamic analysis is probably characterised in terms of the join-irreducible elements of such domains. A further investigation would probably lead to a classification of the properties usually considered by dynamic analysis: properties of traces, properties of sets of traces, relational properties, hyper-properties, together with a specific characterisation of the precision of the analysis and of the program transformations that can reduce it. This unifying framework would provide a common ground where to interpret and compare the potency of different software protection techniques in harming dynamic analysis.

Another aspect worth studying is to view dynamic analysis as a learner that observes properties of some execution traces (training set) and then generalises what it has observed, where the generalisation process is the identity function. We wonder what would happen if we consider more sophisticated generalisation processes such as the ones used by machine learning. Would it be possible to define what is learnable? Would it be possible to formally define robustness in the adversarial setting? We think that this is an intriguing research direction worth pursuing.

Since the framework cannot prove the efficacy of poisons and antidotes, we believe that it can be extended to work with probabilistic code transformations. With this, the first thing to change would be the transformation potency, as it cannot be calculating exactly how many execution traces are needed to uncover the semantic property under analysis. This exact quantity can be substituted with the likelihood of the analysis discovering the property or with the expected value of the number of traces needed to discover the property.

Overall, the framework presented in this work, along with the challenges uncovered for the formal study of dynamic analysis, is nothing but a first stepping stone towards a very ambitious goal. The next works in this direction will hopefully shed some more light onto the foundational intricacies of dynamic analysis and the code protection techniques that can be used against it.

5 Conclusions

This chapter serves as a central gathering point for all the contributions in the thesis. We start by giving a brief summary of each chapter, focusing on the methodologies and positive results. Afterwards, we postulate the possible future works and the limitations. The last section contains a brief distillation of the conclusions that we believe can be made as a result of the works presented in this thesis.

5.1 Summaries

Each one of the previous chapters is summarized in this section.

5.1.1 The Problem

In chapter 1 we introduced the problem of malware classification and why it is represents a worthwhile endeavor. The unprecedented diffusion of malware in the last few years can be connected to the wider distribution of devices that can be attacked, mainly due to the ubiquity of mobile devices and their increasing complexity.

One of the most important tasks in the fight against malware is collecting them into "families", which are groups of malware samples that exhibit the same malicious behaviours. This allows for quicker responses to attacks, as many defenses are designed to work against all members of the same malware family.

In order to achieve this goal we need to be able to group programs according to their behaviour. More technically, we need to group them according to their semantics, which represents what the programs actually do, and not how they are written. This is an uncomputable problem, which means that a generic algorithm for semantic equivalence of programs does not exist.

Since the problem is uncomputable, the only way to achieve the goal is to allow some imprecision by considering an approximation of the program semantics (also called "abstraction"), rather than the whole program itself. In order to group the malware samples according to their semantic similarity, we can then extract the semantic abstractions (a representation of the program) and find a similarity measure between them.

This type of approach is always possible if there is enough domain expertise on the programming language at hand and if the malware samples can be reverted to their source code or at least to a disassembled version. This is well represented in scenario 1, where the code is readily available to the analyzers and their domain expertise can be used to the fullest extent.

Scenario 1 represents an ideal situation that is not always common in malware analysis. For this reason we introduce scenario 2, where the malware samples have to be analyzed from their compiled binaries since the source code is not available and, we assume, the disassembly process is at least partially impeded. Finding a suitable program representation and a similarity measure then becomes a harder problem due to the lack of domain expertise.

Both scenarios so far allow the use of static analysis to achieve the goal of familial malware classification. Static analysis usually overapproximates the semantics of programs, thus allowing imprecision.

A third scenario is then introduced where the analyzers do not have access even to the malware binaries. This can be reflected in a realistic scenario where an antivirus can only execute a malware sample in a sandbox and then send the execution traces to the analyzer. This scenario presents different challenges from the previous two, mostly because the domain expertise lacks the powerful theoretical tools of static analysis.

In the following we explore the three scenarios in more detail and describe the insights gained during our studies.

5.1.2 Malware Analysis on Android

Chapter 2 is firmly situated in scenario 1, where the source code of the malware samples is readily available (or, at least, a decompiled version of the programs is obtainable). Android is the perfect environment for such a scenario, since the applications can be easily decompiled through the use of an intermediate language called "smali". Malware on Android is also very interesting because of its pervasiveness and its potential to affect many more people than standard viruses.

R.E.H.A.

In order to classify malware on Android according to their behaviour we designed a system that extracts the smali representation of the malware samples and focuses on each method in the code. A simple first observation is made by noticing that, thanks to the security design of Android, apps can only perform malicious actions when they call a specific set of system APIs provided by the Android environment itself. This allows our approach to filter out those

methods that never interact with such APIs ("risky" APIs) and only focus on the methods that are potentially harmful to the user.

After this initial filtering we extract the Control Flow Graph (CFG) of each method and use it as a basic feature (a program representation) for the program similarity. Building a similarity measure between CFGs extracted from methods would likely incur in a computational overhead and a loss of precision with the presence of program obfuscations that aim to disrupt the normal control flow of the programs. In order to combat this we adopt an approximate representation of the CFGs called 3D-CFG, which is a projection of the CFG into a 3D space and results in the methods being represented by a single vector called "centroid". The centroid is very similar to a center of mass for the 3D-CFG and can be computed at basically no cost during the CFG construction. The similarity function between 3D-CFGs is also very light computationally since it consists of three simple vectorial operations.

This approach is implemented in a tool called R.E.H.A. (Reverse Engineering Helper for Android), for which we also design a search space reduction algorithm that further decreases the computational complexity. The tool is then tested against a very popular, albeit dated, dataset of Android malware called GENOME, achieving around 88% accuracy in a classification task. A second round of tests has been conducted on new datasets collected at the time of the first study (2017), containing both general malware and ransomware samples. This new dataset has been first labeled through AVClass, a tool that takes the labels assigned to the malware samples by majority voting of most common antivirus software.

These tests revealed that R.E.H.A. can group together members of the same class that have been misclassified by many antivirus software into different families. At the same time, it can differentiate between malware samples that have the same label according to AVClass by assigning them to separate families when they do not share enough malicious behaviours.

StranDroid

R.E.H.A.'s methodology, of course, is not perfect and allows for many false positives. In order to quell some of this imprecision we introduce a different program representation called "strand" and a new similarity function between them. Strands are easily described as backward slices confined in a basic block. Every method is then represented as a set of strands, each collected from every basic block in the method. Methods that do not call risky APIs are still filtered out, but now, strands that do not flow into an API call are also removed. We then design a similarity function between strands that makes use of the Jaccard Index by treating them effectively as sets.

Strands are a more fine-grained program representation than 3D-CFG centroids and do not suffer from many of their limitations, such as false positives due to coincidental structural

similarity of different methods. We test this with the same ransomware and general malware datasets collected for REHA and notice a significant improvement in the precision of the similarity analysis. At the same time, the feature extraction algorithms and their similarity function lead to a significant increase in computational complexity when compared to our previous effort. This a common tradeoff in static analysis: in order to have precision we need to sacrifice computational resources.

5.1.3 Learning on Malware Binaries

Scenario 2 requires us to find a way to group malware into their true family without their source code or any disassembly attempt. Using solely the compiled binary to classify malware is a task that is well suited to many machine learning techniques, as we can treat the binary as raw data and let the algorithms extract a proper program representation and the similarity function. To this end, we encode each binary as an image by extracting their hexadecimal representation and assigning a pixel intensity to each hex value. Each image is then resized through bicubic interpolation. At this point, we can treat the problem as an image classification task and we can employ common deep learning algorithms such as Convolutional Neural Networks (CNNs) to classify the malware samples.

Naturally, the learning approach is not free from downsides. For example, instead of relying on domain expertise for the design of the program representations and their similarity function, the learning algorithms need to rely on a ground truth given by the labels of the malware dataset. This poses some problems because it requires considerable human effort to correctly classify these samples into the right family and sometimes this results in datasets that are either undersized or unbalanced.

When a dataset is undersized, it is harder to use it to train deep learning algorithms, as the huge volume of free variables (weights) in a deep network require a proportionally larger amount of data point to classify, lest we incur in the all-too-common problem of overfitting. The unbalanced nature of certain datasets is also a detriment to the proper training of deep networks, since the dataset is biased towards a few bigger classes and the resulting classification algorithms will also tend to favor these classes. Both of these problems make for bad generalization of the classifications.

In image classification, it is usually easy to deal with both these problems by using data augmentation techniques. These are simple image transformations, such as rotations, flips, zooms etc. that result in new data points that still represent the same original object as the initial image but are different enough to generate models that generalize better.

The same exact techniques cannot be used for our problem setting, as we are classifying images coming from binaries and rotating them or flipping them would result in new data

points that do not represent the same original object (they probably would not represent a program anymore). For this reason we developed a novel technique for data augmentation that works specifically for program classification.

To this end, we employ a set of obfuscations in order to generate new programs from the original ones before transforming them into images. We develop a proof of concept dataset from 47 sample programs in C taken from programming tutorials and from the solutions to the Google [...]. After the programs have been iteratively obfuscated, we obtain a dataset of 18800 programs divided into 47 semantic equivalence classes.

We then design two deep learning models, a convolutional neural network (CNN) and a long-short term memory (LSTM) to classify the aforementioned dataset. The results are promising, as the accuracies range from 92% for the CNN to 94% for the LSTM.

Since the goal of our work is still malware classification, we train our two models on the MalImg and the Microsoft Malware datasets. The LSTM (our best performing model) achieves 98.6% and 92% accuracy respectively.

Another technique that can be used to combat the lack of properly sized datasets is transfer learning, which consists of learning on a large dataset and then "freezing" the weights of the feature-extraction part of deep learners. The head (the dense layer) of the network can be then swapped with a new one in order to classify a previously unseen (and possibly undersized) dataset. This is a very valuable technique as it can also be used to remedy the lack of proper computational resources. After all, many of these deep networks often require significant monetary investments for the proper hardware.

In order to apply transfer learning in this domain, we leverage the fact that we can generate an arbitrarily-large dataset thanks to our data augmentation technique. We then test transfer learning techniques with both models and both malware datasets, finding that it performs much better on the CNN and the MalImg dataset. We also verify that it is indeed possible to perform transfer learning between the two malware datasets themselves.

5.1.4 A Formal Framework for Dynamic Analysis

After exploring two scenarios with static analysis, we hypothesize a new scenario where we do not possess the source code nor the binary. In scenario 3, we are interested in dynamic analysis alone.

Some of the domain expertise used in previous scenarios can be backed by the wide selection of theoretical results in the static analysis field. These results allow us to know for sure, and prove formally, which obfuscations work against which types of analysis. Generally, they represent a solid foundation on which the pitfalls of static analysis can be framed and properly reasoned about.

Dynamic analysis, on the other hand, cannot draw from the same deep pool of knowledge. Formal frameworks for dynamic analysis are few and they generally do not aim to explain the effect of obfuscations on dynamic analysis. This is an obstacle that we did not face in the scientific efforts previously described when working with static analysis techniques. For this reason, in chapter 4 we build the foundation of a new formal framework which aims to address the problem of describing the efficacy of obfuscating transformations against specific dynamic analysis techniques.

In order to build such framework, we take inspiration from abstract interpretation, a well known mathematical framework for static analysis. From this we define the topological nature of dynamic analysis as partitions of the set of execution traces. Intuitively, we equate the power of dynamic analysis with the concept of code coverage, which is indeed often used to evaluate the effectiveness of dynamic analysis.

Similarly, the power of obfuscations is shown as their ability to add new partitions to the set of execution traces in order to make the code coverage harder. We then describe some obfuscations designed to work against dynamic analysis and show how they can be formalized using our framework to prove that they actually do harm the process of dynamic analysis.

5.2 Limitations and Future Work

Each of the three last chapters so far summarized lends itself to many possible future works.

5.2.1 Malware on Android

The 3D-CFG centroid is an approximation of the CFG extracted from each method in the malware samples. This approximation works very well against program transformations that slightly alter the control flow structure of the method, but it does not bode well against pervasive control modifications. The main problem is that the centroid is very sensitive to changes in the number of statements of the method (which increase the "weight" of the centroid). Due to this, it is relatively easy to add spurious statements to certain basic blocks in order to shift the centroid and render the 3D-CFG effectively unusable for program similarity.

Another problem is that the CFG itself might not be a good choice for the task at hand. With Android especially, it is very common to have methods that have the same exact control structure, often comprising a single basic block with no control statements at all. This leads to many false positives that are only slightly mitigated by the API vectors that we introduced in chapter 2.

The approach that resulted in the StranDroid tool is also not perfect. It suffers from false

positives as well, just in lower numbers, and it requires much more time for each analysis on the same dataset compared to R.E.H.A.. One thing that could help in this respect is the development of a new search space algorithm focused on strands.

5.2.2 Learning on Binaries

One of the advantages of the previous scenario is the ability to directly apply domain expertise and know exactly the expressive power of the program representations. This allows us to properly deal with obfuscations, since it is clear how they impact the analysis.

This is an advantage that we miss in the deep learning context encountered in scenario 2. Letting the algorithms build their own program representation and similarity measure lessens the impact of not being able to directly apply domain expertise to the problem. At the same time, it takes control away from the analyzer and makes it harder to understand the vulnerabilities of the approach.

It is not known to us whether there are some obfuscations that would generate new samples that would be misclassified by our models. On the other hand, this could be seen as an advantage since the attackers also would struggle in order to find such obfuscations. We postulate that adversarial learning would be much harder in this setting. With normal image classification problems, it is relatively trivial to modify an image slightly to maintain its meaning while forcing the models into a misclassification. To apply adversarial learning on this problem, it would not be enough to slightly modify the images in the dataset. The code of the malware samples itself would need to be modified with two requirements: 1) maintain the semantics of the program and 2) generate a binary whose image will force a misclassification. This problem is certainly non-trivial to solve and we would like to investigate it thoroughly.

In general, one of the biggest limitations of our approach mirrors the problems of many deep neural network solutions: the system is not easily interpretable. This is mainly due to the fact that the input to the models is the raw binary and the models have to generate the features on their own. While there are many techniques that can reveal what features are observed by the networks, they tend to be features of the images that hardly can be linked back to semantic features in the original code. This avenue has been explored in the past using networks that employ attention mechanisms but we would like to delve deeper into this problem, as the interpretability of the results is one of the best features of the approach in scenario 1.

In addition, we believe that exploring different types of resizing algorithms could be interesting, as we limited our study to simple bicubic interpolation offered by a common vision library.

5.2.3 Dynamic Analysis

The framework built in chapter 4 has been designed with a mathematical foundation that should be easily connected to abstract interpretation. One of the main reasons for this design decision was to find a way to connect the two frameworks and, hopefully, enable talking about dynamic analysis and static analysis with the same mathematical language. This would make it easier to reason about many types of hybrid analyses which use both static and dynamic analysis in order to achieve their goal.

More realistically, we would like to translate some of the concepts common to abstract interpretation directly into our dynamic analysis framework. For example, the approach to completeness defined in abstract interpretation would be very interesting when mirrored in dynamic analysis.

Another direction for future improvements would be to enhance the framework to deal with randomized obfuscations such as poisons and antidotes.

5.3 Conclusions

The three scenarios presented in this thesis, and the wildly different constraints they add to the problems, make it clear that there cannot be one unified approach to solve program similarity in the context of malware analysis.

The open system scenario is ideal to apply domain knowledge in order to customize the program representations and the similarity analysis between them. This comes with many advantages, among which is the ability to accurately predict when an obfuscation will negatively impact the analysis at hand. In order to make good use of these inherent advantages, we built a program similarity tool for Android malware called R.E.H.A. that is able to group malware according to how many malicious behaviours they share. This led us to discover that malware samples, especially recent ones, are often mislabeled by commercial antivirus software that rely mostly on simple syntactic signatures.

Due to the lack of precision of our system, we investigated a more fine-grained code feature for malware similarity by extracting strands, or backward traces in the scope of basic blocks. After building a second tool for malware similarity called StranDroid, we verified that indeed the more precise nature of the feature led to better results in the classification of Android malware. The new system is not without its faults as static analysis in general is prone to precision loss due to the nature of the approximations used.

Due to the difficulty often encountered in disassembly binaries, we hypothesized a second scenario in which the malware similarity has to be designed to work with compiled programs

in mind. The natural solution to this problem is to resort to deep learning solutions. We explored the most common problems in this context and isolated two very common ones: undersized datasets and unbalanced datasets. For both of these problems, we designed a new technique for data augmentation specific to program classification problems. We then designed two deep learning models, a convolutional neural network and a long short-term memory in order to train them on a custom dataset generated using the aforementioned technique. After verifying that resulting networks were able to successfully classify the dataset, we used them to classify two common datasets composed of Windows malware samples.

The positive results of the classification of the malware datasets led us to investigate a second possible way to improve deep learning problems in this context: transfer learning. With this in mind, we experimented with learning the features from our custom built dataset and then used the network with a new head to successfully classify the malware samples.

The third scenario has been investigated in order to introduce dynamic analysis into the context of program similarity for malware classification. The lack of a proper formal foundation led us to establish a mathematical framework based on topological abstractions of the program semantics. Thanks to this framework, we showed how it is possible to express a dynamic analysis and, at the same time, show how an obfuscation could work against it. There is great potential in this approach which has only been touched upon.

We believe that all three scenarios investigated present some unique research challenges and we hypothesized many of them in this chapter and previous ones.

Bibliography

[1] The tigress c obfuscator. https://tigress.wtf/

[2] Aafer, Y., Du, W., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: International conference on security and privacy in communication systems, pp. 86–103. Springer (2013)

[3] Ammann, P., Offutt, J.: Introduction to software testing. Cambridge University Press (2016)

[4] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 583–600 (2016)

[5] Andronio, N., Zanero, S., Maggi, F.: HelDroid: Dissecting and detecting mobile ransomware. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID), *Lecture Notes in Computer Science*, vol. 9404, pp. 382–404. DOI 10.1007/978-3-319-26362-5_18

[6] Armin, J.: Mobile threats and the underground marketplace. APWG White Paper: Mobile (2013)

[7] Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS (2014)

[8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2014)

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

[10] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. In: Annual international cryptology conference, pp. 1–18. Springer (2001)

[11] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. In: Annual International Cryptology Conference, pp. 1–18. Springer (2001)

[12] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. Journal of the ACM (JACM) **59**(2), 1–48 (2012)

[13] Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: International Symposium on Formal Methods for Components and Objects, pp. 364–387. Springer (2005)

[14] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: The best of both worlds. Computing in Science & Engineering **13**(2), 31–39 (2011)

[15] Bengio, Y., LeCun, Y., Henderson, D.: Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden markov models. In: Advances in neural information processing systems, pp. 937–944 (1994)

[16] Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks **5**(2), 157–166 (1994)

[17] Bhodia, N., Prajapati, P., Di Troia, F., Stamp, M.: Transfer learning for image-based malware classification. arXiv preprint arXiv:1903.11551 (2019)

[18] Bierma, M., Gustafson, E., Erickson, J., Fritz, D., Choe, Y.R.: Andlantis: Large-scale android dynamic analysis. arXiv preprint arXiv:1410.7751 (2014)

[19] Blazytko, T., Contag, M., Aschermann, C., Holz, T.: Syntia: Synthesizing the semantics of obfuscated code. In: 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017, pp. 643–659. USENIX Association (2017)

[20] Boiman, O., Irani, M.: Similarity by composition. In: Advances in neural information processing systems, pp. 177–184 (2007)

[21] Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)

[22] Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: A binary analysis platform. In: International Conference on Computer Aided Verification, pp. 463–469. Springer (2011)

[23] Canavese, D., Regano, L., Basile, C., Viticchié, A.: Estimating software obfuscation potency with artificial neural networks. In: International workshop on security and trust management, pp. 193–202. Springer (2017)

[24] Canfora, G., Mercaldo, F., Visaggio, C.A.: An hmm and structural entropy based detector for android malware: An empirical study. Computers & Security **61**, 1–18 (2016)

[25] Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering **19**(4), 1040–1074 (2014)

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

[27] Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 175–186. ACM, New York, NY, USA (2014). DOI 10.1145/2568225.2568286. URL http://doi.acm.org/10.1145/2568225.2568286

[28] Chen, L.: Deep transfer learning for static malware classification. arXiv preprint arXiv:1812.07606 (2018)

[29] Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: Stormdroid: A streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 377–388. ACM (2016)

[30] Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1027–1040 (2019)

[31] Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional (2009)

[32] Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations (1997)

[33] Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (*POPL '98*), pp. 184–196. ACM Press (1998)

[34] Coogan, K., Lu, G., Debray, S.K.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011, pp. 275–284. ACM (2011)

[35] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. IEEE Transactions on Software Engineering **35**(5), 684–702 (2009)

[36] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. **277**(1-2), 47–103 (2002)

[37] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252 (1977)

[38] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the 4th ACM Symposium on Principles of Programming Languages (*POPL '77*), pp. 238–252. ACM Press (1977)

[39] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the 6th ACM Symposium on Principles of Programming Languages (*POPL '79*), pp. 269–282. ACM Press (1979)

[40] Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Conference Record of the Thirtyfirst Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 173–185. ACM Press, New York, NY (2004)

[41] Crussell, J., Gibler, C., Chen, H.: Attack of the clones: Detecting cloned applications on android markets. In: ESORICS, vol. 12, pp. 37–54. Springer (2012)

[42] Crussell, J., Gibler, C., Chen, H.: Andarwin: Scalable detection of semantically similar android applications. In: European Symposium on Research in Computer Security, pp. 182–199. Springer (2013)

[43] Cui, Z., Du, L., Wang, P., Cai, X., Zhang, W.: Malicious code detection based on cnns and multi-objective algorithm. Journal of Parallel and Distributed Computing **129**, 50–58 (2019)

[44] Dalla Preda, M., Giacobazzi, R.: Semantic-based code obfuscation by abstract interpretation. Journal of Computer Security **17**(6), 855–908 (2009)

[45] Dalla Preda, M., Giacobazzi, R., Marastoni, N.: Formal framework for reasoning about the precision of dynamic analysis. In: International Static Analysis Symposium, pp. 178–199. Springer (2020)

[46] Dalla Preda, M., Maggi, F.: Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. Journal of Computer Virology and Hacking Techniques **13**(3), 209–232 (2017)

[47] Dalla Preda, M., Mastroeni, I.: Characterizing a property-driven obfuscation strategy. Journal of Computer Security **26**(1), 31–69 (2018)

[48] David, Y., Partush, N., Yahav, E.: Statistical similarity of binaries. ACM SIGPLAN Notices **51**(6), 266–280 (2016)

[49] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition, pp. 248–255. Ieee (2009)

[50] Deshotels, L., Notani, V., Lakhotia, A.: Droidlegacy: Automated familial classification of android malware. In: Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014, pp. 1–12 (2014)

[51] Desnos, A.: Android: Static analysis using similarity distance. In: System Science (HICSS), 2012 45th Hawaii International Conference on, pp. 5394–5403. IEEE (2012)

[52] Drape, S., Thomborson, C., Majumdar, A.: Specifying imperative data obfuscations. In: ISC - Information Security, *Lecture Notes in Computer Science*, vol. 4779, pp. 299 – 314. Springer Verlag (2007)

[53] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI) (2010)

[54] Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE security & privacy **7**(1), 50–57 (2009)

[55] Faruki, P., Bharmal, A., Laxmi, V., Gaur, M.S., Conti, M., Rajarajan, M.: Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In: 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 414–421. IEEE (2014)

[56] Faruki, P., Laxmi, V., Bharmal, A., Gaur, M.S., Ganmoor, V.: Androsimilar: Robust signature for detecting variants of android malware. Journal of Information Security and Applications **22**, 66–80 (2015)

[57] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security, pp. 627–638 (2011)

[58] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z.: Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 679–696. IEEE (2018)

[59] Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., Waters, B.: Candidate indistinguishability obfuscation and functional encryption for all circuits. SIAM Journal on Computing **45**(3), 882–929 (2016)

[60] Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural detection of android malware using embedded call graphs. In: Proceedings of the 2013 ACM workshop on Artificial intelligence and security, pp. 45–54. ACM (2013)

[61] Gentry, C., Boneh, D.: A fully homomorphic encryption scheme, vol. 20. Stanford university Stanford (2009)

[62] Giacobazzi, R.: Hiding information in completeness holes - new perspectives in code obfuscation and watermarking. In: Proc. of The 6th IEEE International Conferences on Software Engineering and Formal Methods (SEFM'08), pp. 7–20. IEEE Press. (2008)

[63] Giacobazzi, R., Jones, N.D., Mastroeni, I.: Obfuscation by partial evaluation of distorted interpreters. In: O. Kiselyov, S. Thompson (eds.) Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'12), pp. 63 – 72. ACM Press (2012)

[64] Giacobazzi, R., Mastroeni, I., Preda, M.D.: Maximal incompleteness as obfuscation potency. Formal Asp. Comput. **29**(1), 3–31 (2017)

[65] Giacobazzi, R., Ranzato, F., Scozzari., F.: Making abstract interpretation complete. Journal of the ACM **47**(2), 361–416 (2000)

[66] Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. Journal of the ACM (JACM) **47**(2), 361–416 (2000)

[67] Gibert, D., Mateu, C., Planes, J.: A hierarchical convolutional neural network for malware classification. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2019)

[68] Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213–223 (2005)

[69] Guardsquare: Dexguard (2020). URL https://www.guardsquare.com/en/products/dexguard. [accessed 14-June-2020]

[70] Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., Song, D.: Juxtapp: A scalable system for detecting code reuse among android applications. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 62–81. Springer (2012)

[71] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778 (2016)

[72] Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)

[73] IDC: Smartphone users worldwide (2020). URL https://www.idc.com/promo/smartphone-market-share/os. [accessed 13-March-2020]

[74] Jain, M., Andreopoulos, W., Stamp, M.: Convolutional neural networks and extreme learning machines for malware classification. Journal of Computer Virology and Hacking Techniques **16**(3), 229–244 (2020)

[75] Jang, J.w., Yun, J., Mohaisen, A., Woo, J., Kim, H.K.: Detecting and classifying method based on similarity matching of android malware behavior with profile. SpringerPlus **5**(1), 1–23 (2016)

[76] Kang, J., Jang, S., Li, S., Jeong, Y.S., Sung, Y.: Long short-term memory-based malware classification method for information security. Computers & Electrical Engineering **77**, 366–375 (2019)

[77] Kapoor, A., Kushwaha, H., Gandotra, E.: Permission based android malicious application detection using machine learning. In: 2019 International Conference on Signal Processing and Communication (ICSC), pp. 103–108. IEEE (2019)

[78] Kebede, T.M., Djaneye-Boundjou, O., Narayanan, B.N., Ralescu, A., Kapp, D.: Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In: 2017 IEEE National Aerospace and Electronics Conference (NAECON), pp. 70–75. IEEE (2017)

[79] Keys, R.: Cubic convolution interpolation for digital image processing. IEEE transactions on acoustics, speech, and signal processing **29**(6), 1153–1160 (1981)

[80] Kiss, N., Lalande, J.F., Leslous, M., Tong, V.V.T.: Kharon dataset: Android malware under a microscope. In: The {LASER} Workshop: Learning from Authoritative Security Experiment Results ({LASER} 2016), pp. 1–12 (2016)

[81] Kukačka, J., Golkov, V., Cremers, D.: Regularization for deep learning: A taxonomy. arXiv preprint arXiv:1710.10686 (2017)

[82] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86. IEEE (2004)

[83] Lawrence, S., Giles, C.L., Tsoi, A.C., Back, A.D.: Face recognition: A convolutional neural-network approach. IEEE transactions on neural networks **8**(1), 98–113 (1997)

[84] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)

[85] LeCun, Y., Cortes, C., Burges, C.: Mnist handwritten digit database. AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist **2** (2010)

[86] Li, J., Sun, L., Yan, Q., Li, Z., Srisa-an, W., Ye, H.: Significant permission identification for machine-learning-based android malware detection. IEEE Transactions on Industrial Informatics **14**(7), 3216–3225 (2018)

[87] Li, L., Li, D., Bissyandé, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: A systematic study of malicious code grafting. IEEE Transactions on Information Forensics and Security **12**(6), 1269–1284 (2017)

[88] Li, Y., Jang, J., Hu, X., Ou, X.: Android malware clustering through malicious payload mining. In: International symposium on research in attacks, intrusions, and defenses, pp. 192–214. Springer (2017)

[89] Maggi, F., Valdi, A., Zanero, S.: Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In: Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM). ACM (2013)

[90] Mahmoudi, M., Nadi, S.: The android update problem: An empirical study. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 220–230 (2018)

[91] Maiorca, D., Ariu, D., Corona, I., Aresu, M., Giacinto, G.: Stealth attacks: An extended insight into the obfuscation effects on android malware. Computers & Security **51**, 16–31 (2015)

[92] Majumdar, A., Drape, S.J., Thomborson, C.D.: Slicing obfuscations: design, correctness, and evaluation. In: DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management, pp. 70–81. ACM (2007). DOI http://doi.acm.org/10.1145/1314276.1314290

[93] Malwarebytes: State of malware report (2020). URL https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf. [accessed 13-March-2020]

[94] Marastoni, N., Continella, A., Quarta, D., Zanero, S., Preda, M.D.: Groupdroid: Automatically grouping mobile malware by extracting code similarities. In: Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop, pp. 1–12 (2017)

[95] Marastoni, N., Giacobazzi, R., Dalla Preda, M.: A deep learning approach to program similarity. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, pp. 26–35 (2018)

[96] Marastoni, N., Giacobazzi, R., Dalla Preda, M.: Data augmentation and transfer learning to classify malware images in a deep learning context. Journal of Computer Virology and Hacking Techniques pp. 1–19 (2021)

[97] Marastoni, N.: Niccolò Marastoni's personal website. https://niccolomarastoni.github.io/articles.html (Mar. 2021)

[98] Mariconti, E., Onwuzurike, L., Andriotis, P., De Cristofaro, E., Ross, G., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models. arXiv preprint arXiv:1612.04433 (2016)

[99] McAfee: McAfee Labs Threats Report 2020. https://www.mcafee.com/enterprise/en-us/a reports/rp-quarterly-threats-nov-2020.pdf (Dec. 2020)

[100] Meng, G., Xue, Y., Xu, Z., Liu, Y., Zhang, J., Narayanan, A.: Semantic modelling of android malware for effective malware comprehension, detection, and classification. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 306–317 (2016)

[101] Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.: Malware images: visualization and automatic classification. In: Proceedings of the 8th international symposium on visualization for cyber security, p. 4. ACM (2011)

[102] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: International Conference on Compiler Construction, pp. 213–228. Springer (2002)

[103] Ochoa, M., Banescu, S., Disenfeld, C., Barthe, G., Ganesh, V.: Reasoning about probabilistic defense mechanisms against remote attacks. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017, pp. 499–513. IEEE (2017)

[104] OKane, P., Sezer, S., McLaughlin, K.: Obfuscation: The hidden malware. IEEE Security & Privacy **9**(5), 41–47 (2011)

[105] Oliva, A., Torralba, A.: Building the gist of a scene: The role of global image features in recognition. Progress in brain research **155**, 23–36 (2006)

[106] Ollivier, M., Bardin, S., Bonichon, R., Marion, J.Y.: How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In: Proceedings of the 35th Annual Computer Security Applications Conference, pp. 177–189 (2019)

[107] O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., et al.: Keras Tuner. https://github.com/keras-team/keras-tuner (2019)

[108] Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on knowledge and data engineering **22**(10), 1345–1359 (2009)

[109] Pasetto, M., Marastoni, N., Dalla Preda, M.: Revealing similarities in android malware by dissecting their methods. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), pp. 625–634. IEEE (2020)

[110] Pawlowski, A., Contag, M., Holz, T.: Probfuscation: an obfuscation approach using probabilistic control flows. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 165–185. Springer (2016)

[111] Perez, L., Wang, J.: The effectiveness of data augmentation in image classification using deep learning. arXiv preprint arXiv:1712.04621 (2017)

[112] Pratt, L.Y., Mostow, J., Kamm, C.A., Kamm, A.A.: Direct transfer of learned information among neural networks. In: Aaai, vol. 91, pp. 584–589 (1991)

[113] Programiz: C examples. `https://www.programiz.com/c-programming/examples` (Dec. 2020)

[114] Rakamarić, Z., Emmi, M.: Smack: Decoupling source language details from verifier implementations. In: International Conference on Computer Aided Verification, pp. 106–113. Springer (2014)

[115] Rawat, W., Wang, Z.: Deep convolutional neural networks for image classification: A comprehensive review. Neural computation **29**(9), 2352–2449 (2017)

[116] Reitermanova, Z.: Data splitting. In: WDS, vol. 10, pp. 31–36 (2010)

[117] Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., De Geus, P.: Malicious software classification using transfer learning of resnet-50 deep neural network. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1011–1014. IEEE (2017)

[118] Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft malware classification challenge. arXiv preprint arXiv:1802.10135 (2018)

[119] Rosenblum, N., Zhu, X., Miller, B.P.: Who wrote this code? identifying the authors of program binaries. In: European Symposium on Research in Computer Security, pp. 172–189. Springer (2011)

[120] S. O'Dea: Smartphone users worldwide (2020). URL https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide. [accessed 13-March-2020]

[121] Schrittwieser, S., Katzenbeisser, S.: Code obfuscation against static and dynamic reverse engineering. In: International workshop on information hiding, pp. 270–284. Springer (2011)

[122] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Computing Surveys (CSUR) **49**(1), 1–37 (2016)

[123] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.R.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. **49**(1), 4:1–4:37 (2016)

[124] Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE symposium on Security and privacy, pp. 317–331. IEEE (2010)

[125] Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: Avclass: A tool for massive malware labeling. In: International Symposium on Research in Attacks, Intrusions, and Defenses, pp. 230–253. Springer (2016)

[126] Serebryany, K.: Continuous fuzzing with libfuzzer and addresssanitizer. In: 2016 IEEE Cybersecurity Development (SecDev), pp. 157–157. IEEE (2016)

[127] Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Automatic reverse engineering of malware emulators. In: 30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA, pp. 94–109. IEEE Computer Society (2009)

[128] She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: Neuzz: Efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 803–817. IEEE (2019)

[129] Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. Journal of Big Data **6**(1), 60 (2019)

[130] Shu, X., Yao, D.D., Ryder, B.G.: A formal framework for program anomaly detection. In: International Symposium on Recent Advances in Intrusion Detection, pp. 270–292. Springer (2015)

[131] Sim, S.E., Easterbrook, S., Holt, R.C.: Using benchmarking to advance research: A challenge to software engineering. In: 25th International Conference on Software Engineering, 2003. Proceedings., pp. 74–83. IEEE (2003)

[132] Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)

[133] Swiecki, R.: Honggfuzz. Available online a t: http://code. google. com/p/honggfuzz (2016)

[134] Talha, K.A., Alper, D.I., Aydin, C.: Apk auditor: Permission-based android malware detection system. Digital Investigation **13**, 1–14 (2015)

[135] Venkatraman, S., Alazab, M., Vinayakumar, R.: A hybrid deep learning image-based analysis for effective malware detection. Journal of Information Security and Applications **47**, 377–389 (2019)

[136] Wang, C., Knight, J.: A security architecture for survivability mechanisms. University of Virginia (2001)

[137] Wang, H., Guo, Y., Ma, Z., Chen, X.: Wukong: A scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 71–82. ACM (2015)

[138] Wang, W., Zhao, M., Wang, J.: Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. Journal of Ambient Intelligence and Humanized Computing **10**(8), 3035–3043 (2019)

[139] Warren, H.S.: Hacker's delight. Pearson Education (2013)

[140] Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W.: Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 252–276. Springer (2017)

[141] Weiser, M.: Program slicing. IEEE Transactions on software engineering (4), 352–357 (1984)

[142] Wilde, N., Buckellew, M., Page, H., Rajlich, V., Pounds, L.: A comparison of methods for locating features in legacy software. Journal of Systems and Software **65**(2), 105–114 (2003)

[143] Winsniewski, R.: Android–apktool: A tool for reverse engineering android apk files (2012)

[144] Wueest, C.: Financial Threats Review 2017 (2017). URL https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-financial-threats-review-2017-en.pdf

[145] Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K.: Android malware detection based on system call sequences and lstm. Multimedia Tools and Applications **78**(4), 3979–3999 (2019)

[146] Xu, M., Ma, Y., Liu, X., Lin, F.X., Liu, Y.: Appholmes: Detecting and characterizing app collusion among third-party android markets. In: Proceedings of the 26th International Conference on World Wide Web, pp. 143–152 (2017)

[147] Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pp. 674–691. IEEE Computer Society (2015)

[148] Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., Sakuma, J.: Neural malware analysis with attention mechanism. Computers & Security **87**, 101592 (2019)

[149] You, I., Yim, K.: Malware obfuscation techniques: A brief survey. In: 2010 International conference on broadband, wireless computing, communication and applications, pp. 297–300. IEEE (2010)

[150] Yuan, Z., Lu, Y., Xue, Y.: Droiddetector: android malware characterization and detection using deep learning. Tsinghua Science and Technology **21**(1), 114–123 (2016)

[151] Zalewski, M.: Technical" whitepaper" for afl-fuzz. URl: http://lcamtuf. coredump. cx/afl/technical_details. txt (2014)

[152] Zhang, M., Duan, Y., Yin, H., Zhao, Z.: Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, pp. 1105–1116. ACM, New York, NY, USA (2014). DOI 10.1145/2660267.2660359. URL http://doi.acm.org/10.1145/2660267.2660359

[153] Zhang, Y., Sui, Y., Pan, S., Zheng, Z., Ning, B., Tsang, I., Zhou, W.: Familial clustering for weakly-labeled android malware using hybrid representation learning. IEEE Transactions on Information Forensics and Security (2019)

[154] Zhou, W., Zhou, Y., Grace, M., Jiang, X., Zou, S.: Fast, scalable detection of piggybacked mobile applications. In: Proceedings of the third ACM conference on Data and application security and privacy, pp. 185–196. ACM (2013)

[155] Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on Data and Application Security and Privacy, pp. 317–326. ACM (2012)

[156] Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: 2012 IEEE symposium on security and privacy, pp. 95–109. IEEE (2012)

[157] Zhu, H.J., You, Z.H., Zhu, Z.X., Shi, W.L., Chen, X., Cheng, L.: Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing **272**, 638–646 (2018)