Analysis and classification of malware based on symbolic execution and machine learning

Charles-Henry Bertrand Van Ouytsel

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

February 2024

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Axel **Legay** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Peter **Van Roy** (President)	UCLouvain/ICTEAM, Belgium
Pr. Charles **Pecheur** (Secretary)	UCLouvain/ICTEAM, Belgium
Dr. Thomas **Given-Wilson**	Diffblue, United Kingdom
Pr. Olivier **Bonaventure**	UCLouvain/ICTEAM, Belgium
Pr. Emilio **Coppa**	Sapienza University of Rome, Italy
Pr. Johannes **Kinder**	LMU Munich, Germany

Analysis and classification of malware based on symbolic execution and machine learning

by Charles-Henry Bertrand Van Ouytsel

© Charles-Henry Bertrand Van Ouytsel 2024
ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

This work was supported by the F.R.S-FNRS (FRIA grant).

I know that I know nothing.
SOCRATES

The greatest enemy of knowledge is not ignorance, it is the illusion of knowledge.
STEPHEN HAWKING

Preamble

As the world becomes increasingly dependent on technology, the number and variety of cyber threats continue to grow, implying significant challenges to the security of information systems. The ENISA Threat Landscape Report 2023 [ENI23] identifies malware as one of the main threats observed in cyberspace. Preventing the execution and propagation of these malicious codes is therefore essential. Over the past decades, malicious actors have developed new obfuscation and evasion techniques to modify existing malware into new variants and evade anti-virus technologies. Therefore, cybersecurity researchers must continually understand these new techniques, develop countermeasures, and maintain the effectiveness of their detection tools.

The field of machine learning offers new opportunities for researchers to automatically analyze the growing amount of suspicious codes observed in the wild. It allows existing approaches based on the syntactic and semantic properties of malicious codes to be extended in order to train classification models that can efficiently detect these threats. This capability helps to reduce the workload of malware analysts, allowing them to focus on the latest malware evasion tricks. However, machine learning models can suffer from the same limitations as traditional detection methods. For example, if the model learns from logs obtained during code execution, but the malicious code hides its activity, the model will not learn anything useful from the logs. Furthermore, the use of machine learning in cybersecurity tasks brings different challenges [Arp+22; Cav+23], such as building representative datasets to ensure robust models and studying vulnerabilities against adversarial examples.

In the cat-and-mouse game between cybercriminals and cybersecurity analysts, new techniques are constantly being explored to discover the malicious capabilities of unknown code. Symbolic execution, a program analysis technique generally used in software testing, allows for extensive exploration of malware codes. It has been successfully applied to malware analysis [Bal+17; Bru+08] and combined with machine learning to build models capable of detecting and classifying malicious codes [Sai+18; Seb+20]. Moreover, recent open source projects aim to incorporate symbolic execution into well-known malware analysis tools: SENinja [BCD22] extends BinaryNinja, AngryGhidra [Nal20] extends Ghidra, Modality [Kan20] extends radare2, ... However, symbolic execution is expensive and different challenges hinder the deployment of this solution for automated analysis: path explosion, proper

environment modeling, etc.

In this work, we propose different approaches to improve the combination of symbolic execution and machine learning for malware analysis. More specifically, the contributions of this thesis are the following:

- *The design of improved building blocks for malware analysis based on symbolic execution and machine learning*

  To overcome the limitations of static and dynamic malware analysis tools, we propose to extend previous works based on symbolic execution and machine learning. We examine the literature and discuss different improvements to enhance the capabilities of symbolic execution and machine learning for malware analysis. These improvements focus on binary exploration strategies, representation of malware behavior and limitations of existing classifiers.

- *The implementation and evaluation of SEMA -Symbolic Execution for Malware Analysis*

  Taking advantage of symbolic execution, SEMA explores the different execution paths of malware and extracts system calls in the form of a SCDG - System Call Dependency Graph. SEMA then proposes different machine learning methods to learn the signature of a malware family from several SCDGs and then detect its new variants. We present the practical implementation and the assessment of different improvements and strategies compared to the literature [Sai+18; Seb+20; Puo+21]: improved environment modeling, exploration methods, SCDG building strategies and a new malware classifier based on the Weisfeiler-Lheman kernel.

- *A detailed analysis of machine learning algorithms for packing detection*

  Packing is an obfuscation technique commonly used by malware to evade detection tools thereby making analysis more difficult. Knowing that an executable has been packed means that the analysis can be adapted to examine its true contents after unpacking. Various machine learning models have been studied for this task, based on their efficiency, robustness and costs. Particular attention has also been paid to the features used and their importance in the learning process.

- *An extension of the SEMA toolchain to support Federated Learning*

> To build effective machine learning models, it is essential to have a high diversity of data to represent the real world. SEMA has been extended to support federated learning, which allows multiple users to jointly train a model without directly sharing their dataset. This learning paradigm aims to increase collaboration between different actors to build efficient models.

This thesis is structured as follows. Chapter 1 provides a comprehensive overview of the three areas studied in this thesis, namely malware analysis, symbolic execution and machine learning. It also presents related works proposed in the literature. Subsequently, Chapter 2 discusses different building blocks proposed in the literature to combine symbolic execution and machine learning for malware analysis, and presents different improvements. Chapter 3 introduces SEMA -Symbolic Execution for Malware Analysis-, an implementation of the aforementioned building blocks, and presents several experiments to evaluate its performance. Chapter 4 tackles the problem of packing detection by machine learning and compares nine supervised classifiers, along with the relevance of 119 features. Chapter 5 offers an extension of SEMA through federated learning to build malware classifiers securely and collaboratively. Finally, Chapter 6 concludes with a discussion of this thesis and presents additional perspectives for future work.

Bibliographic notes

Conference Publications

1. C.-H. Bertrand Van Ouytsel and A. Legay. "Detection and classification of malware based on symbolic execution and machine learning methods". In: *Cybersec & AI* (2021).

2. C.-H. Bertrand Van Ouytsel, K. H. T. Dam, and A. Legay. "Symbolic Analysis Meets Federated Learning to Enhance Malware Identifier". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security ARES*. 2022, pp. 1–10.

3. C.-H. Bertrand Van Ouytsel and A. Legay. "Malware Analysis with Symbolic Execution and Graph Kernel". In: *Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings*. Springer. 2023, pp. 292–310.

4. C.-H. Bertrand Van Ouytsel, C. Crochet, K. H. T. Dam, and A. Legay. "Tool Paper-SEMA: Symbolic Execution Toolchain for Malware Analysis". In: *Risks and Security of Internet and Systems: 17th International Conference, CRiSIS 2022, Revised Selected Papers*. Springer. 2023, pp. 62–68.

5. A. D'Hondt, C.-H. Bertrand Van Ouytsel, and A. Legay. "Experimental Toolkit for Manipulating Executable Packing". In: *Risks and Security of Internet and Systems: 18th International Conference, CRiSIS 2023, To appear*. Springer. 2023.

6. S. Lucca, C. Crochet, C.-H. Bertrand Van Ouytsel, and A. Legay. "On Exploiting Symbolic Execution to Improve the Analysis of RAT Samples with angr". In: *Foundations and Practice of Security: 16th International Symposium, FPS 2023, Revised Selected Papers*. Springer. 2023.

Posters and Demos

1. A. D'Hondt, C.-H. Bertrand Van Ouytsel, and A. Legay. "Packing-Box: Playing with Executable Packing". In: *BlackHat Europe*. 2022.

2. A. D'Hondt, R. Jennes, S. Martinez Balbuena, C.-H. Bertrand Van Ouytsel, and A. Legay. "Packing-Box: Breaking Detectors and Visualizing Packing". In: *BlackHat Europe*. 2023.

Journal Publications

1. C.-H. Bertrand Van Ouytsel, O. Bronchain, G. Cassiers, and F.-X. Standaert. “How to fool a black box machine learning based side-channel security evaluation”. In: *Cryptography and Communications* 13.4 (2021), pp. 573–585.

2. C.-H. Bertrand Van Ouytsel, K. H. T. Dam, and A. Legay. “Analysis of machine learning approaches to packing detection”. In: *Computers & Security* (2023), p. 103536.

Miscellaneous Contributions

1. Presentation during the contest "My thesis in 180 seconds": `https://www.youtube.com/watch?v=AiB5iYaAM5o`.

Acknowledgments

Après quatre ans (et demi), il est temps de mettre fin à ce parcours ! Il s'en passe des choses durant une thèse ; des hauts (résultats prometteurs, papiers acceptés, voyages, rencontres...) mais aussi des bas (idées foireuses/déjà explorées dans la littérature, la Covid, bugs à foison ou encore ce maudit reviewer 2). En regardant le chemin parcouru, je suis très heureux d'avoir pu compter sur de nombreuses personnes qui m'ont accompagné durant ces années et j'aimerais leur adresser mes sincères remerciements.

Tout d'abord, merci à Axel pour m'avoir initialement proposé ce sujet de recherche. Merci pour tes conseils lors des écritures et des présentations, mais aussi de m'avoir fait confiance pour la rédaction du nouveau projet que je vais entamer ainsi que pour ceux que nous explorons avec les plus jeunes membres de l'équipe. Merci également pour ton engagement en faveur de la cybersécurité en Wallonie, qui nous permet de partager avec des personnes d'horizons très différents.

Je tiens ensuite à remercier les post-docs et professeurs qui m'ont soutenu dans ce cheminement et enrichi ma réflexion. Merci à Eduard Baranov pour sa relecture approfondie de ma thèse et ses nombreux commentaires. Merci à Benoit Duhoux pour ses encouragements, ses conseils et son soutien durant la rédaction. Merci à Thomas Given-Wilson pour son accompagnement et son aide au début de ma thèse ainsi que pour ses nombreuses remarques durant ma défense privée qui ont grandement contribué à l'amélioration de ma thèse. Merci à Olivier Bonaventure d'avoir fait parti de mon jury, pour son écoute, mais aussi et surtout, suite à une discussion après la séance de questions-réponses de l'un de ses cours, pour m'avoir présenté à Axel en décembre 2018. Merci à Charles Pecheur d'avoir accepté de faire partie de mon comité d'accompagnement, de mon jury et pour ses remarques bienveillantes. Merci à Johannes Kinder et Emilio Coppa d'avoir accepté de faire partie de mon jury ainsi que pour leurs questions et nos discussions durant ma défense. Enfin, merci à Peter Van Roy d'avoir accepté de présider mon jury.

Mes remerciements s'étendent à mes collègues en INGI pour la bonne ambiance qui y règne, surtout les temps de midi (un merci tout particulier à tous les whisters pour ces innombrables parties). En particulier, merci à Vanessa pour sa gentillesse, son écoute, sa disponibilité à toujours nous aider pour trouver des solutions. Beaucoup de personnes - c'est un euphémisme - ont rejoint l'équipe d'Axel durant ces quatre années ; Maxime, je te remercie pour

nos nombreuses discussions scientifiques/philosophiques et te souhaite bon courage pour ta fin de thèse ! Tom, j'aimerais te remercier particulièrement pour ton soutien et ta gentillesse durant tout ce parcours. Un grand merci à toute l'équipe qui a contribué, de près ou de loin au développement de SEMA et continue de le faire. Serena, Christophe, Dam, Dimitri, Marie, Samy, Bastien, Arnold, merci pour nos échanges et de m'avoir motivé à continuer ce projet. Merci également à tous les étudiants enthousiastes que j'ai pu encadrer dans différents cours ou pour des TFEs (Serena, Alexandre, Dimitri, Samy, Bastien, Arnold, ...). Les échanges que nous avons pu avoir et les défis auxquels nous avons pu être confrontés contribuent à alimenter ma passion pour la recherche scientifique.

Je remercie également tous mes amis pour avoir partagé mes joies et mes doutes, autour d'un verre ou d'une partie de jeu de société. Merci à Henri, Brieuc et Guillaume pour les pauses midi-math du jeudi. Merci à toute l'équipe du Kotangente 2019-2020 pour cette super année malgré le covid. Merci à Nicolas, Pauline, Marie-Laure, Pierre-Paul, Adrian, Thibault, Géraldine, Antoine, Bénédicte pour ses précieux moments partagés et pour tout ceux à venir. Merci à François, Arno et Martin pour leur amitié et leur soutien de toujours. Un immense merci à ma famille pour leur soutien inconditionnel et en particulier à ma maman et à mon tonton pour avoir toujours entretenu ma curiosité et mon étonnement envers le monde depuis tout petit. Enfin, j'aimerai remercier tout particulièrement Victoria pour avoir toujours été présente pour moi ces dernières années. Merci pour ton soutien de chaque jour, tous ces moments forts passés ensemble et tous ceux à venir. Merci pour tout.

Contents

Preamble **i**

Acknowledgments **vii**

Table of Contents **ix**

1 Background **1**
1.1 Malware analysis . . . 1
1.1.1 Binary analysis and malware . . . 1
1.1.2 Static analysis . . . 3
1.1.3 Dynamic analysis . . . 4
1.2 Symbolic Execution . . . 6
1.3 Machine Learning . . . 10
1.3.1 Generalities and algorithms . . . 11
1.3.2 Assessment methodology . . . 16
1.3.3 Adversarial machine learning . . . 18
1.4 Machine Learning for Malware Analysis . . . 19

2 Machine Learning and Symbolic Execution for Malware Analysis **23**
2.1 Introduction . . . 23
2.2 A toolchain based on Symbolic Execution and Machine Learning 25
2.3 Extraction of API calls with symbolic execution . . . 26
2.4 SCDG creation . . . 32
2.5 On Graph Comparison for Malware Analysis . . . 36

3 SEMA - Symbolic Execution for Malware Analysis: Implementation and Evaluation **43**
3.1 The Architecture of SEMA . . . 43
3.2 The angr framework . . . 45
3.3 SEMA-SCDG . . . 47
3.4 SEMA Classifier . . . 55
3.5 Experimental Results . . . 56
3.6 Conclusion and Future Work . . . 63

4 Packing detection: Feature Relevance and Classifiers Evaluation 65
4.1 Introduction . . . 65
4.2 Background . . . 68
4.2.1 Packing . . . 68
4.2.2 Interpretability, dimensionality reduction and feature selection . . . 71
4.3 Methodology . . . 73
4.3.1 Data Sets and Labeling Tools . . . 73
4.3.2 Experimental methodology . . . 75
4.3.3 Hyper-parameters Tuning . . . 77
4.4 Features Analysis . . . 77
4.4.1 First insights . . . 78
4.4.2 SHAP values . . . 78
4.4.3 Feature selection and PCA . . . 82
4.5 Classifier Effectiveness . . . 84
4.5.1 Evaluation of Classification Metrics . . . 84
4.5.2 Robustness . . . 86
4.5.3 Self-packed binaries . . . 87
4.6 Packing detection economics . . . 88
4.7 Conclusion and Future works . . . 89

5 Extending SEMA with Federated Learning 93
5.1 Introduction . . . 93
5.2 Deep learning auto-encoder and classifier . . . 95
5.3 Federated learning with homomorphic encryption . . . 97
5.4 Evaluation . . . 100
5.4.1 Federated Learning with homogeneous SCDGs . . . 101
5.4.2 Federated Learning with heterogeneous SCDGs . . . 103
5.5 Conclusion . . . 105

6 Conclusion 107

A Appendix Chapter 2 111
A.1 gSpan algorithm . . . 111

B Appendix Chapter 3 113
B.1 GetProcAddress Simprocedure . . . 113

C Appendix Chapter 4 115
C.1 Features extracted for packing detection . . . 115
C.2 Comparison of related works . . . 119

D Appendix Chapter 5 **127**
D.1 Confusion matrix 127

Background 1

This chapter aims to introduce various concepts related to the three important areas studied in this thesis: malware analysis, symbolic execution, and machine learning.

1.1 Malware analysis

We first provide background on binary analysis and various malware detection techniques.

1.1.1 Binary analysis and malware

Definition According to *Practical Malware Analysis - The Hands-On Guide to Dissecting Malicious Software* [SH12]:

> "Any software that does something that causes harm to a user, computer, or network can be considered malware [...]"

This definition covers a wide range of threats: Ransomware, Trojans, Spyware, Bot, Cryptominer. These are generally used by malicious actors during their campaign to gain control of the system and network, access private data, disrupt services, etc.

Although malicious actors could launch a malware campaign with a brand new malicious software, most of the malware observed in the wild are variants of already known malware [Coz+20; CTC18]. This means that the same code base is usually reused with different obfuscation techniques/improvements to evade detection. Some malware is also shared or sold in underground forums and adapted according to the attacker's goal. The term *malware family* is generally used to refer to variants of the same malware. Known malware families include : Mirai (2016), WannaCry (2017), NotPetya (2017), Emotet (2018), CovidLock (2020), etc.

The Portable Executable (PE) format PE file format is used by Microsoft Windows. Since PE files are the primary target of malware writers and the primary means of malicious software distribution on Windows [Vira], it is important to cover this file type.

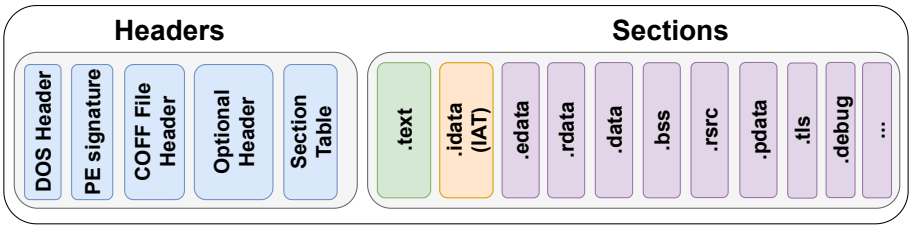


Figure 1.1: PE File Format Structure

PE can be divided into two categories: Header and Sections, as shown in Figure 1.1. The header part encapsulates information about the executable. As described by Microsoft [Mica], its components are the DOS header (a simple stub present for legacy reasons), the signature (*i.e.*, `PE\0\0` identifying the file as a PE file format), the COFF file header, and the optional header. The COFF header contains information about the machine type, the number of sections, a file creation timestamp, and the size of the optional header. The optional header contains important fields such as:

- `SizeOfCode`, the size of the code (text) section.
- `SizeOfInitializedData`, the size of the initialized data section.
- `SizeOfUninitializedData`, the size of the uninitialized data section (BSS).
- `AddressOfEntryPoint`, the address of the first x86 instruction to be executed.
- `ImageBase`, the preferred base address to use when loading the image into memory.

This optional header also contains other data such as alignment information, operating system information, `Checksum`, `DllCharacteristics`, etc. Then it exposes information related to different tables (*e.g.*, Export Table, Import Table, Resource Table,...) present in the binary. Finally, the **Section Table** contains an entry for each section of the executable with data related to their characteristics.

PE file sections of interest include:

- `.text` This section contains the instructions executed by the CPU and *generally* contains the entry point of the executable.
- `.idata` This section contains the information related to the imported functions required by the executable (if this section is not present, this

information is stored in .rdata). Imported functions are ordered in the Import Address Table (IAT) and can be referenced by name or ordinal (*i.e.*, the index of the function in the dll).

- `.edata` This section contains information related to exported functions (similar to .idata, this information could also be stored in .rdata).
- `.data`, `.rdata`, `.bss` These sections contain initialized, read-only initialized and uninitialized data respectively.
- `.rsrc` This tree-structured section stores resources used by the executable (*e.g.*, icons, strings, images, etc.).
- `.pdata` This section stores exception handling information.
- `.tls` This section is used for thread local storage of variables. It also could contain the definition of callback functions to be executed before execution at the entry point of the executable.
- `.debug` This section stores the debug information generated by the compiler.

Analyst's Objectives While malware analysis generally involves gaining knowledge of malicious behavior and evasion tactics, the ultimate goal of the malware analyst can vary. The most common goal is *malware detection*: deciding whether a given binary is malicious. Since analysts receive a large number of samples and manual analysis are time-consuming, another common goal is *malware similarity*: given a set of known malware binaries, we want to determine if the binary under analysis is similar to an existing family. This reduces the workload of analysts since previously gained knowledge about a malware family could be re-used to improve protection and attack recovery process (*e.g.*, which persistence mechanisms are used by the malware).
In general, malware analysts create *signatures* to detect and identify malware. A malware signature aims to represent the DNA of the malware by considering syntactic and/or semantic aspects of its family [Far+15; KV15; Bio+18]. There are a variety of approaches to analyze malicious files and to build these signatures. They are generally divided into *static* and *dynamic* approaches.

1.1.2 Static analysis

The first approach to analyze and detect malware is static analysis. This approach does not execute the binary under analysis. Instead, it focuses on syntactic properties of the binary such as its size, hashes, entropy, number of sections, imported functions in the IAT or the presence of certain strings/byte

sequences. These are used to infer the functionality of the malware. More advanced static analysis techniques could offer a better understanding of the malware behavior such as disassemblers/decompilers (*e.g.*, Ghidra, IDA, radare,...), Control Flow Graph (CFG) [CX11], etc. The elements gathered during static analysis allow to create syntactic signatures such as YARA signatures [Virb]. These signatures are known to be quick to apply, easy for an analyst to understand, and do not require the malware to be executed in a dedicated environment.

An example of a YARA signature designed to detect malware that displays "*Your files have been encrypted*", is shown in Code 1.1. This rule checks whether the analyzed file is an executable, its file size and the presence of the string.

```
rule SimpleRule{
  meta:
    description: "Toy Example"
  strings:
    $s1 = "Your files have been encrypted"
  condition:
    ( uint16(0) == 0x5a4d and filesize < 900KB and $s1)
}
```

Code 1.1: Example Yara rule

Limitations Unfortunately, static signatures are very sensitive to obfuscation techniques that modify the binary code to change its syntactic properties [MKK07]. The various elements used to build these signatures can be hidden or modified, leading to an easy bypass of this detection method, as illustrated in [Bio+18; Sai+18]. For example, the YARA signature of Code 1.1 could be bypassed by modifying the string $s1$, splitting that string into different parts (aggregated at runtime), encoding the text (*e.g.*, with xor/base64), or increasing the length of the binary. Packing, a special technique for obfuscating the contents of a binary, is discussed in detail in Chapter 4. More advanced static analysis based on decompiler/disassembler could still be used, but they generally require more time and deeper understanding on the part of the analysts to deal with the obfuscation strategies and recover (part of) the original binary.

1.1.3 Dynamic analysis

Another approach is dynamic analysis which requires executing the binary in a dedicated environment in order to record its behavior and its effects on the host system. The malware behavior could be observed at several levels [Or-+19]: function calls, volatile memory analysis, side-channel analysis,... Recorded information on the malware behavior could then be used to build a

behavioral signature [Mas+12; ZQW20] (*e.g.*, modified registry keys, network communication, API calls, ...). This approach is not affected by obfuscation as static analysis is, since obfuscated malware retains the same malicious behavior when executed. This type of analysis can be performed on emulated environments (*e.g.*, Anubis [MBN09]), virtual machines (*e.g.*, Amal [MAM15], Cuckoo [Jam+18], ...), bare metal [KVK11], etc. The information obtained from dynamic analysis tools could then be used with behavioral signatures such as Sigma rules [RP].

An example of a Sigma rule designed to detect malware that deletes a large number of files from a storage is shown in Code 1.2.

```
logsource:
  product: windows
  service: security
detection:
  selection:
    EventID: 4663 # Attempt to access an object
    ObjectType: 'File'
    AccessList: '%%1537'  # Right to delete the object.
    Keywords: '0x8020000000000000' # Success of the operation
  timeframe: 20s
  condition: selection | count() by SubjectLogonId > 10
```

Code 1.2: Example sigma rule

Limitations To evade these signatures, malware implements evasion techniques to determine whether it is running in a monitored environment. These techniques include checking CPU features of the host machine, installed libraries, user activity, etc. (see [Afi+19] for an overview). Malware can also exploit the fact that dynamic analysis only examines a single execution path. A carefully crafted program could trigger its malicious behavior under certain conditions thus avoiding the detection. For example, the toy example in Code 1.3 triggers the malicious behavior at 8 PM. In [Avl+21], the authors examine the variability of malware behavior across multiple executions and show that *"for at least 50% of the malware, 30% of the action observed in an execution will not appear in other machines"*.

```
int main(int argc, char** argv){
  time_t now;
  struct tm *now_tm;
  int hour;
  now = time(NULL);
  now_tm = localtime(&now);
  hour = now_tm->tm_hour;
  if (hour == 20){
    do_malicious_stuff();
  }
}
```

Code 1.3: Pseudo-code of malicious behavior with trigger condition

The limitations of static and dynamic analysis have been widely studied in the literature. In this thesis we focus on symbolic execution, a technique that addresses these limitations and is described in the next section.

1.2 Symbolic Execution

Introduced in [Kin76], symbolic execution is a program analysis technique originally used for software testing. This technique simulates program behavior in order to cover as many execution paths as possible. For this purpose, each input variable (*i.e.*, variable related to the user or the execution environment) is associated with a symbol. These symbols represent a set of possible values and are maintained within an execution state. During execution, program statements add conditions to these symbols. When a conditional statement is encountered, the new condition and its negation are considered. If both cases are satisfiable with respect to the current execution state, the two execution paths are considered and the initial execution state is forked. Otherwise, only the execution path with a satisfiable condition is continued. To illustrate these concepts, consider the simple program of Code 1.4, which checks whether the variable *a* is greater than 21 and the local variable *d* is equal to 100.

```
1  int check_code(int a, int b)
2  {
3    int c = 6, d = 0;
4    if (a >= 21){
5      d = b * c + 4;
6      if (d == 100){
7        return 1;
8      }
9    }
10   return 0;
11 }
```

Code 1.4: Example program

The symbolic execution tree of this simple program is shown in Figure 1.2. Paths in the tree represent the different execution paths.

Unlike normal execution, where the input variables take a concrete value, symbolic execution allows multiple execution paths to be explored simultaneously. In the context of malware analysis, it can be particularly helpful to avoid the evasion techniques and ensure the execution of the malicious behavior. However several challenges arise when dealing with symbolic execution, as highlighted in [Bal+18].

Environment A program interacts with the operating system through libraries and system calls. While these interactions cause side effects that can later affect the program's workflow (*e.g.*, file manipulation, registry key modification,...), it is generally impossible to account for all these interactions with

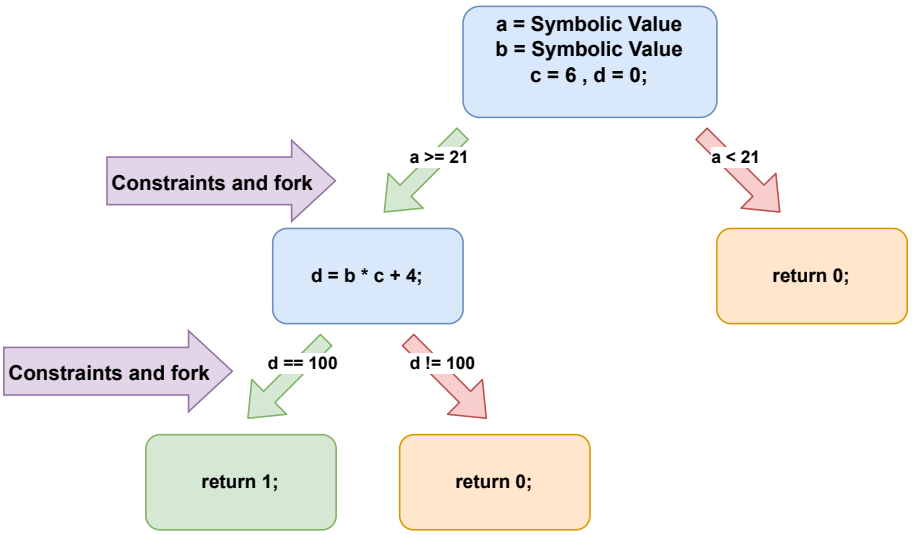


Figure 1.2: Example

symbolic execution. In fact, this would require symbolic execution of external libraries and a perfect representation of every possible execution environment, which is not feasible in practice.

To handle interactions with the environment, two approaches are generally considered: developing abstract models or using concrete execution to interact with the real environment. With abstract models, external procedures and operating system elements are expressed by ad hoc models to summarize their effect on a state. For example, KLEE [CDE+08] proposes a basic symbolic file system for each execution state, which is configured by the user. Automatic Exploit Generation (AEG) [Avg+14a] models the system environment as an input source with symbolic constraints including files, sockets, environment variables, and about 70 system calls. A similar approach is used by angr [Sho+16] for files/sockets and allows the user to create models (*SimProcedures*) for each system call/library function. However, this approach is generally labor-intensive and error-prone. Other symbolic engines interact with the real environment by executing external calls with concrete arguments (*e.g.*, DART [GKS05], CUTE [SMA05],...). S2E [CKC12] uses QEMU to emulate the entire software stack and only symbolically executes code of interest to the developer. Similar to S2E, Symbion [Gri+20] (based on angr) proposed to interleave symbolic and concrete execution. Their approach is highly modular and allows analysts to synchronize the observed real environment with the symbolic engine to improve exploration.

Path explosion While symbolic execution could theoretically allow exploration of all execution paths, in practice this is not feasible for real programs. Certain program constructs, such as loops or memory operations, can generate an exponential number of states. This problem requires the development of strategies and heuristics to reduce the number of states and maximize the coverage of the program's workflow.

Pruning provide a way to discard irrelevant execution paths, helping to mitigate this problem. In general, invoking the constraint solver allows discovering and pruning unsatisfiable states (*i.e.*, states in which no assignment to their symbolic variables could lead an execution to them). In [Avg+14a], AEG introduces precondition constraints (*e.g.*, buffer length, buffer with known prefix,...) to guide the exploration on an interesting path. The search space is reduced because states that do not satisfy these preconditions are pruned. On the other hand, postconditioned symbolic execution [Yi+15] reduces the search space by keeping track of previously explored path suffixes and considering them as postconditions. These postconditions are built from paths that reach the end of the program execution. When a branch condition is encountered, the corresponding previously explored postconditions are negated and added to the path constraints, allowing redundant paths to be pruned early. Following another approach, under-constrained symbolic execution [ED07] directly explores a piece of code (*e.g.*, a function) in isolation (*i.e.*, without considering the execution path leading to that piece of code). While this allows the exploration to focus directly on the relevant parts of the program, and reduces the number and length of execution paths, it does not take into account some constraints on the program execution. Therefore, it might consider paths that are normally unfeasible due to these missing constraints, or even code that should normally be unreachable.

To mitigate the path explosion, it is also useful to limit the exploration to the most "interesting" states according to some goal (*e.g.*, code coverage). With KLEE [CDE+08], the authors proposed two heuristics for state scheduling. On one hand, *Random Path Selection* maintains an execution tree of all execution paths followed by the current states (*i.e.*, the leaves of the tree are the current execution states). It randomly traverses the tree from the root to select the next state to explore. On the other hand, *Coverage-optimized search* selects states based on heuristics to reach uncovered code (*e.g.*, call stack, history, distance to uncovered instruction). In [Ma+11], *Directed Symbolic Execution* is proposed to find an execution path that leads to a specific target line. *Chopped Symbolic Execution* [Tra+18] is another technique that allows analysts to specify uninteresting part of a program that could be skipped during symbolic execution. This improves the efficiency of symbolic execution and those skipped parts could be analyzed later if needed. Machine learning techniques have also been used to develop exploration techniques. SyML [Rua+21] is a

prioritization technique that learns patterns of vulnerable execution paths (*i.e.*, paths that lead to a program crash). For this purpose, several features are extracted: number of branches visited, connectivity in the CFG, number of registers read/write,... Finally, LEARCH [He+21] is also a learning-based prioritization strategy that estimates a reward for each state before selecting them to maximize code coverage. It uses features such as stack size, state constraint, number of successors,...

Finally, state merging (*e.g.*, in Veritesting [Avg+14b]) allows to merge different paths into a single state. The new state is represented using the disjunction of constraints that would have been used for the individual states. However, a trade-off between the length of the constraints and the number of states should be considered. Disjunctions are difficult to handle efficiently by SMT solvers and this overhead could counteract the benefits of state merging [Kuz+12].

Symbolic memory addressing The introduction of symbolic variables could cause execution to read/write from/to symbolic memory address. This problem is called the *symbolic memory address problem* [SAB10] and could significantly impact the coverage and scalability of symbolic execution. To support these symbolic pointers and arrays, several models have been proposed and are summarized below (see [Bal+18] for more details):

- Fully concrete : To avoid the complexity associated with symbolic pointers, an unsound but popular solution is to concretize pointers to a single specific address ((*e.g.*, DART [GKS05] initialize these randomly). While this reduces the number of states and the complexity of the constraints, some paths associated with specific pointer values may be missed.

- Fully symbolic : This sound strategy considers every possible pointer value that satisfies the constraints of the symbolic variable. Two approaches coexist. With *state forking*, each read/write operation on a symbolic pointer address creates new states (*i.e.*, one for each of the possible values), and the constraints of the path are adapted in each state. On the contrary, *merge model* encodes conditions (*i.e.*, if-then-else formulas) representing the different possible values without creating new states. The overhead is thus shifted to the solver engine of the symbolic executor. This latter approach is used in angr [Sho+16].

- Partial : To provide a trade-off between soundness and scalability, Mayhem [Cha+12] introduced a partial *merge model*: it concretizes write addresses while keeping read addresses symbolic (if the range of possible values is less than a given threshold). This approach is also used by angr which optionally supports symbolic writes if the range of possible values is small enough (*i.e.*, less than 128).

Constraint solving During exploration, various constraints are added to the symbolic variables. As new constraints are added, it may become difficult to efficiently check their satisfiability, and SMT solving could quickly become a bottleneck in symbolic execution. Sebastio et al. [Seb+20] investigated how to efficiently use the z3 solver to handle constraints built during symbolic execution of a malware with angr. They identify which tactics of *z*3 are more suitable for this use case. In this thesis, these techniques are used and the handling of constraints is not explored further.

While all these challenges should be considered when designing a system based on symbolic execution, we choose to focus this thesis on the challenges related to environment interaction and path explosion. We use the angr symbolic engine, which uses a partial memory model, and reuse the strategies found by Sebastio et al. [Seb+20] regarding constraint solving.

Symbolic execution for malware analysis Various studies have proposed the use of symbolic execution to analyze malware. MineSweeper [Bru+08] proposes to automatically identify triggering conditions in malware from a set of triggering specifications using symbolic execution. Baldoni et al. [Bal+17] illustrate with a case study how to analyze a Remote Access Trojan (RAT) with angr. They were able to automatically derive the list of supported commands and the associated sequence of system calls used in the RAT while an analyst might spend hours trying to perform a similar analysis. They later extend their work [Bor+19] to show how to use symbolic execution to synthetize a command-and-control server for a RAT. Gritti et al. [Gri+20] propose *Symbion* to interleave symbolic execution (*i.e.*, angr) with concrete execution and illustrate their approach on a heavily packed evasive malware. Although their approach is efficient to limit the problems associated with symbolic execution, it requires the analyst to define instruction addresses in the binary to switch between symbolic and concrete contexts. Recently, Park et al. [Par+21] propose *DBHunter*, a system based on CFGs and API calls to identify behavior dispatchers associated with malicious actions of malware. They then demonstrate how it can be used by symbolic execution to improve exploration.

1.3 Machine Learning

This section provides pertinent background information on machine learning and its evaluation.

1.3.1 Generalities and algorithms

Machine Learning (ML) is a branch of artificial intelligence (AI) based on the assumption that systems can use data to learn and make decisions. ML algorithms allow to build models based on a set of data known as training data. These models can be used, for example, to create classifiers that take as input new data and output a label about this data (*e.g.*, *malicious* or *clean*). However, it is important to pay attention to the **over-fitting** issue. It happens when a model fits its training data exactly and is therefore unable to perform accurately on new data. Instead, the model captures noise, resulting in poor generalization performance.

There are three main categories of learning algorithms: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. All of the classifiers studied in this thesis rely on supervised learning, which consists of using an already labeled dataset to train the model. More precisely, we investigate six types of models to create our classifiers:

KNN The *K-Nearest Neighbors* (KNN) is a learning algorithm that treats each input data point as a point in the feature space with a particular label. When given a data point to classify, KNN finds the K-nearest points in the feature space with respect to a defined distance (*e.g.*, Euclidean distance). The data point is then assigned a class label based on the majority class of its K nearest neighbors. This process is illustrated in Figure 1.3, where the new data point is assigned the label A according to the majority class of its five neighbors. While KNN requires no training time and is easy to implement, the speed of this algorithm decreases significantly as the dataset grows.

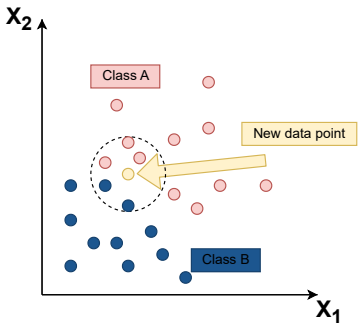


Figure 1.3: Binary classification involving two features (X_1 and X_2) and using a 5-Nearest Neighbors classifier

Probabilistic models *Naive Bayes classifiers* (NBC) are probabilistic models based on Bayes theorem:

$$\Pr(y_k|X_i) = \frac{\Pr(X_i|y_k)\Pr(y_k)}{\Pr(X_i|y_k)\Pr(y_k) + \Pr(X_i|\neg y_k)\Pr(\neg y_k)} \tag{1.1}$$

where y_k is a class in the set of classes $k \in \{1, .., K\}$, X_i is a feature vector for the set of data points $i \in \{1, .., n\}$, $\Pr(a)$ is the probability of a (where a is a class label), and $\Pr(a|b)$ is the probability of a given b (where a is a class label and b is a feature vector or vice versa). These classifiers use Equation 1.1 to estimate the probability of observing the label y_k for a sample according to the value of the feature vector X_i. These estimated probabilities are then combined to construct the classifier $\hat{y}$ given by the Equation 1.2:

$$\hat{y} = \underset{k\in\{1,..,K\}}{\arg\max}\, p(y_k) \prod_{i=1}^{n} p(X_i|y_k) \tag{1.2}$$

The naive adjective of the classifier comes from the assumption that the features of each data point are independent. Although this assumption is generally not valid, NBC is known to work well in practice [BN06]. In Chapter 4, we use a variant of NBC: the *Gaussian Naive Bayes Classifier* (GNBC). In order to estimate the probabilities, it makes the additional assumption that the continuous feature values are distributed according to a normal distribution.

Linear Models (Lin) There are classification models built as a linear combination of features, where the decision boundary $z(x)$ can be represented as a line or a plane separating the data into two classes. Consider Equation 1.3 with feature values x_i and weights w_i.

$$z(x) = w_0 + w_1x_1 + ... + w_px_p \tag{1.3}$$

Following Equation 1.4, this decision boundary can construct the classifier:

$$y = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

Since Lin is a very large family of algorithms, within this thesis we consider 2 types of Lin: logistic regression and linear support vector machines. *Logistic regression* (LR) is based on the logistic function used to modify the decision boundary and given in Equation 1.5:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.5}$$

This classifier and the logistic function are illustrated in Figure 1.4. The optimal weight vector $W =< w_1, ..., w_p >$ of LR can be obtained by maximizing the likelihood function L defined below:

$$L(W) = \prod_{i=1}^{n} \sigma(W^T X_i)^{y_i} \cdot [1 - \sigma(W^T X_i)]^{1-y_i}$$

where X_i is the i-th element from a dataset of size n and y_i is the label of X_i. A *Linear Support Vector Machine* (LSVM) aims to maximize the margin between two parallel hyperplanes that separate the data classes. Data points used to construct the two hyperplanes are called *support vectors* and are defined by Equation 1.6.

$$W^T X - b = 1 \quad \text{and} \quad W^T X - b = -1 \; . \tag{1.6}$$

where b is a bias parameter. Figure 1.5 shows the two hyperplanes (here two lines), the support vectors used to build these hyperplanes, and the margin width maximized by the algorithm.

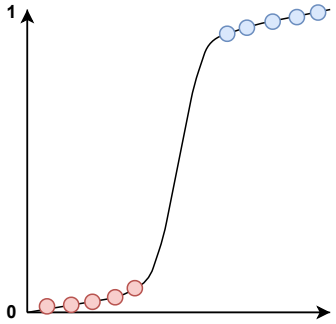


Figure 1.4: Logistic regression

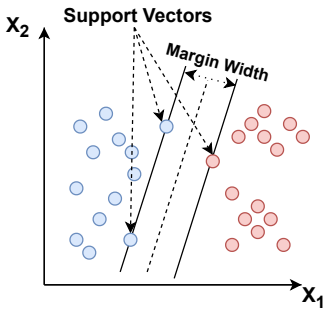


Figure 1.5: Linear support vector machine.

Tree-based Models *Decision trees* (DT) are classification models based on the sequential application of simple binary rules. These models start at the *root* of the tree and traverse it until reaching a *leaf*. A *leaf* is a node with no children, and each node has rules (*e.g.*, "*if* x_1 *of the sample is less than one, go to the right branch; if not, go to the left branch*" as shown in Figure 1.6).

Multiple DT models can be combined to construct more complex classifiers, solving the high variance error that single DT can suffer from. *Random Forests* (RF) combine slightly different DT with bagging. Bagging is a technique that combines multiple base classifiers (*e.g.*: decision trees) to improve classification performance. To create these base classifiers, different data sets are created by randomly taking subsets from the original data set. While a single DT can be

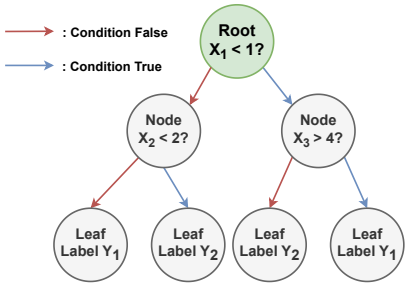


Figure 1.6: Example of a simple decision tree

effective, there is a risk of over-fitting. RF addresses this risk by aggregating the results of multiple DT; this reduces over-fitting while maintaining the predictive power of each tree. *Gradient Boosted Decision Trees* (GBDT) combine DT using *boosting*. Boosting is a technique that combines multiple base classifiers trained **sequentially**. Each base classifier is trained to improve the performance of the previous classifiers. Boosting gives more weight to misclassified data points to improve their classification when subsequent base classifiers are used. Typically, a GBDT starts with shallow trees, often called weak learners, that make good predictions only for some parts of the data. More shallow trees are gradually added to improve global performance by reducing the loss margin. Compared to RF, GBDT is usually more efficient, but harder to tune.

Neural networks (NN) are generally used in deep learning approaches and have been employed in hundreds of different classifiers for security tasks such as anomaly detection or malware classification. [Nas+18; Kal+18]. Chapter 4 uses the most popular variant of NN: *Multi-Layer Perceptrons* (MLP) algorithm. MLP can be seen as a generalization of Lin composed of multiple layers that apply nonlinear functions. Each layer has multiple neurons that takes inputs from the previous layer and output to the next. Each neuron applies a nonlinear function f to a weighted sum of input features to calculate its output y:

$$y = f(x_0 w_0 + w_1 x_1 + ... + w_p x_p + b) \tag{1.7}$$

where f is a nonlinear function (*e.g.*, logistic function) applied to a weighted sum of features x_i. In MLP, data is forwarded among the neurons of the different layers, as shown in Figure 1.7. The data is fed into the input layer and passes through one or more hidden layers, which represent intermediate processing steps, before reaching the output layer, which provides predictions. NN have demonstrated their efficiency on large datasets, creating incredibly complex models at the cost of training time, hyperparameters tuning and accurate preprocessing.

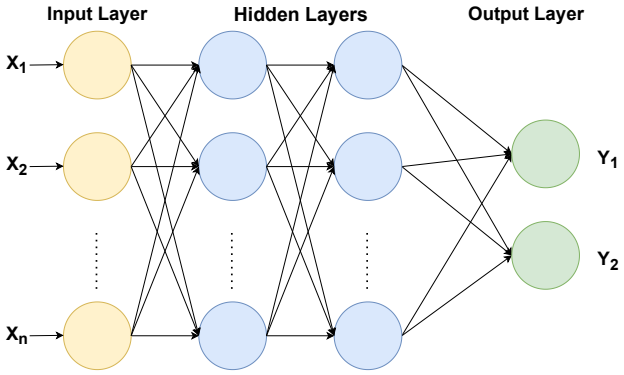


Figure 1.7: Multi-Layer Perceptron with two hidden layers.

Kernelized Support Vector Machines (KSVM) are an extension of Lin that uses hyperplanes to handle nonlinearly separable data. In KSVM, the original data space is mapped to a higher dimension space using a *kernel*. In machine learning, *kernel methods* are algorithms that allow us to compare different data points with a given similarity measure. Consider a set of data points $\mathcal{X} \in \mathbb{R}^p$, and let $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ be a function. The function k is a valid kernel on $\mathcal{X}$ if there exists a Hilbert space $\mathcal{H}_k$ (*i.e.*, a vector space providing an inner product that induces a distance function) and a feature map $\phi : \mathcal{X} \rightarrow \mathcal{H}_k$ such that $k(X_1, X_2) = \langle \phi(X_1), \phi(X_2) \rangle$ for $X_1, X_2 \in \mathcal{X}$, where $\langle \cdot, \cdot \rangle$ denotes the inner product of $\mathcal{H}_k$. There are many different kernels (*e.g.*, polynomial, gaussian, sigmoid, etc.). A well-known kernel is the Gaussian Radial Basis Function (RBF) kernel on $\mathbb{R}^p$, defined as:

$$k_{RBF}(X_1, X_2) = exp(-\frac{\|X_1 - X_2\|^2}{2\sigma^2}) \tag{1.8}$$

where σ is the bandwidth parameter and $\|X_1 - X_2\|$ represents the Euclidean distance between X_1 and X_2. Note that the RBF kernel provides an explicit definition of ϕ. In practice, this is not always necessary. In fact, the SVM uses the dataset only through inner products between data points. Thus, it is sufficient to have the kernel value $k(X_1, X_2)$ between each data point to build SVM-based classifier. This property is known as the *kernel trick* [Hof06]. A Gram matrix K, is defined with respect to a finite set of points $X_1, .., X_n \in \mathcal{X}$. Each element $K_{i,j} = k(X_i, X_j)$ for $i, j \in \{0, .., n\}$ represents the kernel value between a pair of data points. If the Gram Matrix K of the kernel k is positive semi-definite for every possible set of data points, then k is a valid kernel.

Due to the data transformation, the planes used to separate the data become higher dimensional hyperplanes, as shown in Figure 1.8 and Figure 1.9. Although KSVM works well on a variety of datasets, it is known to suffer from performance degradation when scaled to work with large numbers of samples.

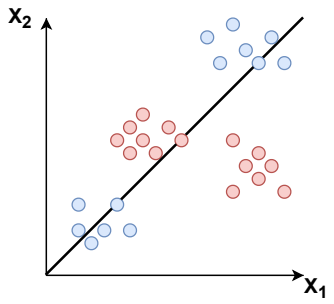


Figure 1.8: Decision boundary found by LIN.

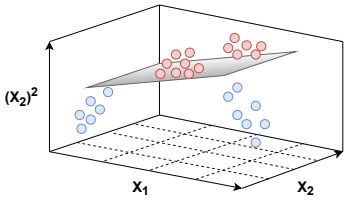


Figure 1.9: Decision boundary found by KSVM.

1.3.2 Assessment methodology

In a binary classification problem (*e.g.*, malicious vs benign), *false positives* (FP) are negative samples (*e.g.*, benign) classified as positive (*e.g.*, malicious) by a classifier. Conversely, *false negatives* (FN) are positive samples (*e.g.*, malicious samples) classified by a classifier with the negative label (*e.g.*, benign). Finally, true positives (TP) (as well as true negatives (TN)) are positive (or negative, respectively) samples that have been correctly classified.

Performance metrics To evaluate and compare the performance of different classifiers, it is essential to define clear metrics. We use the following:

- The **accuracy** metric is defined as the ratio of true positives and true negatives sample ($TP + TN$) over all samples: $\frac{TP+TN}{TP+TN+FP+FN}$
- The **precision** metric is defined as the ratio of positive samples correctly classified (TP) over all samples classified as positive ($TP + FP$): $\frac{TP}{TP+FP}$
- The **recall** metric is the ratio of positive samples correctly classified (TP) over all positive samples ($TP + FN$): $\frac{TP}{TP+FN}$
- The F-**score** metric is defined as $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}}$. It is a harmonic mean that represents both precision and recall symmetrically in one metric.

Receiver Operating Characteristics (ROC) Curve This curve represents the performance of a binary classifier at different classification thresholds, which is the value used to convert the intermediate output of a binary classifier into its final predicted label (*e.g.*, the output of a nonlinear function from the output layer of a MLP classifier is a continuous value between 0 and 1 that is converted into a label according to this threshold). The ROC curve is created by plotting the true positive rate (TPR) (also known as *recall*) versus the

false positive rate (FPR), which is the ratio of false positives to all negative samples. ROC curves corresponding to different classifiers are shown in Figure 1.10. A diagonal represents a completely random classifier since it classifies a sample equally as positive or negative (*i.e.*, leading to the same TPR and FPR). The more the curve is skewed above the diagonal, the more effectively the classifier discriminates between positive and negative. A perfect classifier would have a TPR of 1 with no false positives. The ROC curve

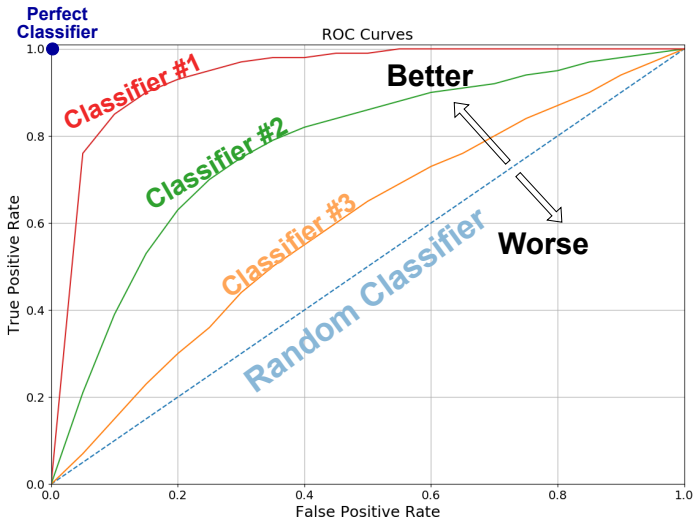


Figure 1.10: Illustration of different ROC curves

illustrates the trade-off between TPR and FPR and allows a more accurate comparison of the performance of different classifiers.

The Wilcoxon Rank Sum Test This statistical significance test compares the classification performance (*e.g.*, accuracy, F-score, etc.) of ML algorithms. The test assesses whether two independent sets of measures M (*i.e.*, the performance of two algorithms on different datasets) come from identical continuous distributions with equal medians. This tests whether one algorithm is significantly more efficient than another [HWC13].

The performance difference of the two classification algorithms A and B according to the measure M on the dataset D is given by:

$$\delta(M, D) = M(A, D) - M(B, D)$$

The differences are computed for each dataset D and assigned a rank corresponding to their place in ascending order. The ranks associated with positive

(and respectively negative) differences are summed. Finally, the smaller of the two values is used as the output value of the Wilcoxon test. From this value, we derive the p-value associated with the test and the (non-)superiority of one algorithm over another. The smaller the p-value, the stronger the indication that there is a significant difference between algorithms A and B. The result is determined by a threshold α, called the significance level, *e.g.*, $\alpha \leq 0.05$. If p-value $< \alpha$, then there is no difference between the algorithms.

1.3.3 Adversarial machine learning

The field of adversarial machine learning studies the robustness of ML algorithms against adversaries in various settings. This section aims to provide a brief overview of the field. Although attacks targeting different elements of the machine learning pipeline have been proposed in the literature [Pit+19], we focus our study on the creation of adversarial examples as shown in Figure 1.11. In this attack, the goal is to *slightly* modify a sample to change the classifier's decision about it.

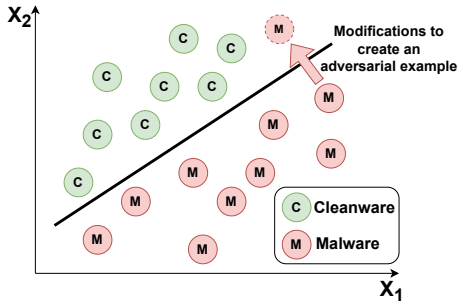


Figure 1.11: Illustration of adversarial examples

Threat model A threat model allows analysts to identify and structure the capabilities and objectives of an adversary in order to design an appropriate defense. According to [Car+19], different elements could be considered to model an adversary to a machine learning classifier:

- **Goal**: While the initial goal is to create an input to the algorithm that will be misclassified, an adversary's exact goal may vary. He may be more interested in misclassifying samples from a particular source class to a particular target class (also known as a targeted attack [CW17]). In other settings, any misclassification might be considered a success.
- **Knowledge**: The adversary may have access to various pieces of information about the target model. In a white box attack, the adversary has

full access to the inner workings of the models (*e.g.*, weights, feature importance,...), while in a black-box setting he only gets the output of the model for a given input. Moreover, different degrees of black box access could be considered: limited number of queries to the models, access to the predicted class probabilities,...

- **Capabilities**: Depending on the scenario, the attacker may have limited ability to modify inputs. For example, an attacker modifying malware should preserve its behavior and the binary should remain executable. While some attacks may also require minimizing the modifications applied to the input, these constraints are generally problem dependent and less relevant in the context of malware detection (*i.e.*, our goal is to create malware that evades the classifier no matter what modifications are applied). "*However, the well-defined nature of this threat model is helpful for performing principled work towards building strong defenses*" [Car+19].

Depending on the attacker model, different attacks could be considered. The literature [Pie+20] distinguishes two different spaces in which an attack on a machine learning model could be crafted. In **feature-space** attacks, the attacker is able to directly modify each component of the feature vector and send it to the target model. Conversely, in a **problem-space** attack, the attacker defines a finite set of transformations and applies them directly to the object (*e.g.*, the malware binary) to evade the model. Feature-space attacks are generally easier to apply because they have fewer constraints: it is not necessary to preserve the syntax of the original program, it is possible to modify features individually where they should be dependent, etc. For example, *Gradient-based* attacks (*e.g.*, [Big+13]) use the gradient loss of the model with respect to an input to apply a perturbation that maximizes this loss. On the other hand, while a problem-space attack is generally more difficult to apply, it is easier to preserve the functionality of the modified malware, making the attack more realistic. For example, *Reinforcement Learning* could be used to train an agent to take a sequence of actions (*e.g.*, modifying a PE executable) that maximizes a reward (*e.g.*, reducing the accuracy of a classifier [Son+22]). In this thesis, we consider attackers who attempt to misclassify samples of a specific class (*e.g.*: malware being misclassified as cleanware). The attackers have black box access to our models and the ability to modify the input in the problem space.

1.4 Machine Learning for Malware Analysis

Following the malware analysis approaches presented in Section 1.1, researchers developed machine learning models taking as input different stat-

ically and dynamically extracted features. These features lead to multiple classifiers and are described below:

- *Byte/Opcode N-gram* : This group of features focuses on the sequence of bytes/opcodes that occur in a PE file (*i.e.*, a *n*-gram is a contiguous sequence of *n* elements). Most research relies on at most 3-grams for the byte sequence (*e.g.*, [And+11; Lin+15; Ahm+16]). It is also common to compute metrics such as entropy to measure the disorder in the byte distribution and possibly detect obfuscation [Ahm+13; LH07]. Similarly, the n-gram opcodes can be used to represent the machine language instructions present in the binary. The n-gram frequencies could be used directly as features [Lin+15; Kho+15] or the occurrences could be aggregated by operation (*e.g.*, memory access, arithmetic operation, control flow modification,...) [Kho+15; ASL12]. In [Raf+17], authors feed raw bytes of an entire EXE file into a neural network to determine maliciousness. Convolutional Neural Network (CNN) classifiers have also been proposed by considering the malware as an image [Nat+11].

- *Strings* : The printable sequences of characters found in an executable file generally offer a great deal of information to an analyst. While some works limit their use to histograms related to string length [Ahm+16], other works build a feature vector by considering the presence of specific strings [Sch+00; Isl+13] including API function names, dll names, file names,... The API function names discovered in the IAT table are generally helpful in inferring the behavior of the program [Ahm+16]. Nevertheless, obfuscation and dynamic linking (*e.g.*, using `GetProcAddress`) could limit information gathering [KW18].

- *PE file characteristics* : Headers contain a wide range of information as illustrated in Section 1.1. It has been shown in [Yon12] that their distribution differs between benign and malicious software (*e.g.*, `NumberOfSection`, `AddressOfEntryPoint`, raw section size,...). Therefore, this information has been widely used as features in machine learning models [Sha+09; RH20; Asq16; YBK13].

- *Windows API calls* : While API function names found in the IAT table provide useful information about a sample behavior, the API calls observed at runtime (along with their arguments) provide a direct view of the binary's interaction with the OS [Xia+19]. The sequence of executed calls allows to build models of malicious behavior with Markov chain [AZ20], DNA sequence alignment algorithm [KKK15], frequent itemset mining [Qia+13] or common subgraph extraction [CJK07; Pal+13].

- *Network, registry and file system activity*: This group of features consists of behavioral characteristics extracted by observing the system: network communication [Bek+15] (*e.g.*, to a command and control server), registry modification [GSS15] to establish persistence, file modification [MAM15] (*e.g.*, modification of configuration files),...
- *Control Flow Graphs*: These graphs contain semantic and structural information about the binary. They have been combined with other graph structures (*e.g.*, function call graphs) to build hierarchical graph representations and train Graph Neural Network [Lin+22]. Additionally, size- and permutation-invariant graph representations, such as heat and wave representations [Sun+22], have been proposed.

While there is a significant overhead in extracting dynamic features, a recent study [Dam+23] shows that they do not always provide improvements in malware detection/classification compared to static features. According to the authors, this is likely due to the lack of some features after the sample has been executed in a sandbox. This is related to intrinsic characteristics of some malware families [Aon+23] (*e.g.*, feature set related to file system is empty if the malware does not create files) but also to the sandbox not triggering the malware behavior correctly (*e.g.*, evasion techniques, missing connection with command-and-control server,...). Nevertheless, these features allow to build behavioral signatures for specific malware families and bring other benefits such as better interpretability for human analysts [Aon+23] and better robustness against obfuscation.

Machine Learning and Symbolic Execution for Malware Analysis

2

This chapter discuss existing machine learning and symbolic execution approaches that have been developed in the context of malware detection. It then presents our malware analysis toolchain and introduces a simple toy example. The concepts behind the different building blocks of our toolchain are then explained in detail, followed by their application to the toy example.

2.1 Introduction

While machine learning have been widely applied for malware analysis, multiple classifiers proposed in the literature face difficulties in capturing the semantics of the analyzed binary [Smi+20] in their model. For example, in [Ram+12], the authors investigated which features allow to distinguish between malware and benign software from a Windows system. By extracting only seven features, they achieved a true positive rate of 98.6%. However, it appears that their models can only distinguish between Microsoft and non-Microsoft executables. In addition to the limitations of static and dynamic analysis exposed in Section 1 (*i.e.*, obfuscation and evasion techniques), this semantic gap hinders the robustness of classifiers against adversaries. Several papers show how easily some classifiers are vulnerable to adversarial examples [Dem+21]. For example, in [Son+22], the authors designed a reinforcement learning framework that sequentially applies simple actions (*e.g.*, adding/renaming a section, appending an overlay, breaking the checksum,...) to evade state-of-the-art classifiers based on statically extracted features. These classifiers generally suffer from the same drawbacks as the malware analysis techniques used to extract the features they use. Therefore, it is essential to explore novel methods for extracting malware features for machine learning.

The ability of symbolic execution to explore different paths has led to the development of a new set of machine learning based malware classification methods. In particular, in [Sai+18] the authors have proposed to combine symbolic execution with gSpan [YH02], a graph mining algorithm that detects the largest common subgraphs between two graphs. They first collect API calls

via symbolic execution and then connect them in a System Call Dependency Graph (SCDG), *i.e.*, a graph that abstracts the flow of information between these calls. In the training phase, gSpan computes the largest common subgraphs between malware in a given family. These then represent the signature for the family. In the classification phase, the approach extracts the SCDG from the binary and compares it to the signature of each family. This approach was extended in [Seb+20] with a focus on using efficient tactics for the SMT solver of angr.

Unfortunately, the above method has several limitations. The main one is related to the efficiency of symbolic execution which often faces the path explosion problem. In addition, its SCDGs connect only two calls that have arguments with the same value. This means that some dependencies between calls may be missed (*e.g.*, an argument's value is summed with a constant) and others may be invalid (*e.g.*, the same argument's value is used in a different context). Finally, relying on the largest common subgraphs may exclude important but isolated calls (*i.e.*, not in the connected components of the graph) that are specific to the malware.

The use of graphs poses a particular challenge in the application of traditional data mining and machine learning approaches that rely on vectors. To overcome these limitations, the authors in [Puo+21] propose the use of a graph kernel [NV21], which can be intuitively understood as a function that measures the similarity of pairs of SCDGs. In their work, the authors study unsupervised learning, but supervised learning classifiers such as SVM could also benefit from this kernel. The results in [Puo+21] show that their graph kernel outperforms gSpan in terms of accuracy. Unfortunately, the kernel still implicitly relies on detecting the largest commonalities between graphs. Isolated important API calls remain outside its scope. In this chapter, we take these consideration into account and propose the development of **heuristics for symbolic exploration** of malware, **improvements to the SCDG abstraction**, and the use of **new classifiers for malware detection and classification**. While this chapter presents different strategies to improve the combination of symbolic execution and machine learning for malware analysis, the implementation and evaluation of these solutions will be covered in the next chapter.

The chapter is organized as follows. Section 2.2 provides an overview of the different building blocks commonly used in a malware analysis toolchain based on symbolic execution and machine learning. It then introduces a simple toy example that is used at the end of each section to demonstrate the application of each building block. Section 2.3 discusses improvements to the symbolic execution engine in order to better explore the different malware execution paths. Section 2.4 presents our various strategies for creating SCDGs from the paths obtained by symbolic execution. Finally, Section 2.5 discusses the

limitations of existing machine learning classifiers based on SCDGs. It then presents a classifier based on the Weisfeiler-Lehman kernel, which overcomes these limitations.

2.2 A toolchain based on Symbolic Execution and Machine Learning

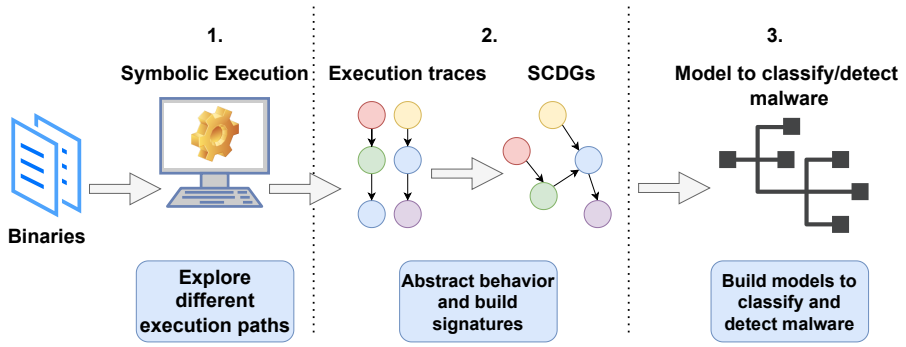


Figure 2.1: Illustration of the toolchain workflow

In the literature [Sai+18; Seb+20; Puo+21], symbolic execution and machine learning are applied to malware analysis following the toolchain workflow shown in Figure 2.1. It relies on the following key building blocks: First, a symbolic execution engine is used on a set of binaries from different malware families. During the symbolic execution of each binary, API calls and their arguments are extracted and recorded according to their respective execution paths to form *execution traces*. The execution traces of each binary are then used to create its corresponding SCDG. The SCDGs can then be used to train machine learning algorithms to either distinguish a malicious file from a benign one, or to directly identify a malware family. Finally, when a new binary needs to be analyzed, its SCDG is extracted and the previously trained classifier outputs its decision regarding its maliciousness/malware family. We take this toolchain workflow as a starting point and implement several improvements to each of its building blocks to improve symbolic exploration, SCDG creation, and classifiers.

Introduction of the toy example Let us consider a simple malware whose goal is to display the string `"I'm evil"`. Imagine that we use a honeypot to capture a version of the malware that contains this string as an argument to the `printf` function. Static analysis tools such as Yara [Virb] could detect such malware by matching the pattern `"I'm evil"` directly in the binary. Unfortunately, this detection can be easily defeated by creating a variant that

uses string obfuscation. This variant could be obtained by splitting the string as shown in Lines 13 and 14 of Code 2.1. A solution to this problem would be to use dynamic analysis which allows the malware analyst to monitor the execution of a malware (*e.g.*, via a sandbox). Looking at Code 2.1 again, the string `I'm evil` is not directly present in the code, but is constructed during execution. In fact, it is obtained as the result of the `strcat` function via the `func` function (Line 12) applied to the arguments `I'm` (Line 13) and `evil` (Line 14).
Unfortunately, as seen in Section 1.1, dynamic analysis could be limited: it only examines a single execution path associated with its own analysis environment. The malware could detect that it is in a monitored environment and decide to evade detection by hiding its behavior. Code 2.1 uses a time trigger, *i.e.*, `Sleep(500000)` (Line 2) and checks (Line 4) if the `sleep` action has actually finished (`uptimeBis-uptime`). It also checks for the presence of a debugger (function `IsDebuggerPresent`) in its environment. If either of these conditions is met, the malware adopts benign behavior, *i.e.*, it displays the message `hello world!` at Line 6. If neither condition is met, the malicious behavior is executed. Symbolic execution can be used to analyze this program and discover its two possible outcomes.

```
1   ULONGLONG uptime = GetTickCount();
2   Sleep(500000);
3   ULONGLONG uptimeBis = GetTickCount();
4   if ((uptimeBis - uptime)<500000 || IsDebuggerPresent())
5   {
6         MessageBox(NULL,"Hello world!","", MB_OK);
7   }
8   else
9   {
10        char message[20] = "";
11        HINSTANCE hlib = LoadLibrary("msvcrt.dll");
12        MYPROC func = (MYPROC) GetProcAddress(hlib,"strcat");
13        (func) (message, "I'm ");
14        (func) (message, "evil!!");
15        MessageBox(NULL, message, "", MB_OK);
16    }
```

Code 2.1: Malware code example

In the following sections, we present improvements to the different building blocks of this toolchain to explore symbolically programs, construct SCDGs and build classifiers. At the end of each building block section, we show its application to the toy example.

2.3 Extraction of API calls with symbolic execution

Since symbolic execution suffers from state space explosion, only a set of execution paths can be explored in a reasonable amount of time. The choice of

execution paths affects the set of encountered system calls and, consequently, SCDG. Therefore, it has a large impact on our machine learning models.

In this thesis, we investigate how path selection strategies influence the creation of SCDGs and therefore the performance of machine learning classifiers. To illustrate the considered strategies, we will consider the execution tree shown in Figure 2.2. It consists of possible execution states associated with different parts of the binary labeled from A to M.

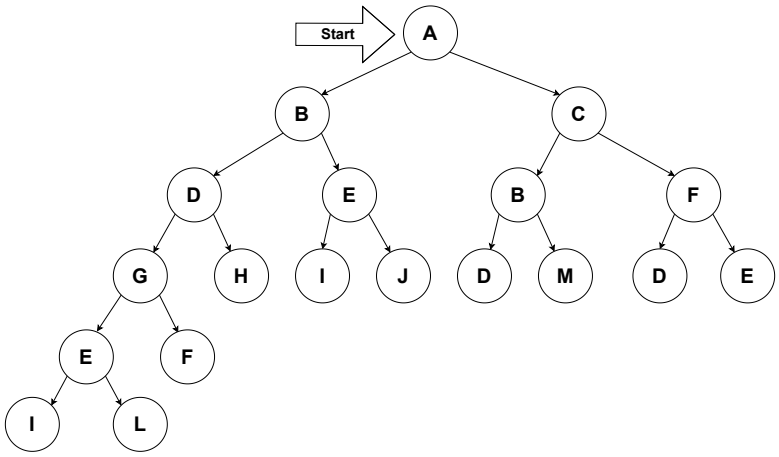


Figure 2.2: Example execution tree of a program. Each letter represent a different execution state.

In [Sai+18; Seb+20], the authors use a Breadth-First Search (BFS) approach, *i.e.*, at each execution step, all current paths are explored simultaneously. Although they allow limiting the number of concurrently explored states at each execution step, this approach can lead to a significant increase in the number of states and therefore high memory consumption. As shown in Figure 2.3, the number of states increases at each execution step, and redundant states (*i.e.*, states with the same letter) are explored.

To limit the path explosion problem, we propose to explore a subset of paths at a time. We prioritize states from which new assembly instruction addresses can be explored (*i.e.*, instruction addresses not previously visited). Our Coverage Breadth-First Search Strategy (**CBFS-Strategy**) is presented in Algorithm 1. The algorithm starts by iterating through the set of available states S and adding the states that lead to an unexplored part of the code to the list R of states to explore next (line 7). If there are L states that fulfill this condition, the algorithm directly returns R (line 9). Otherwise, it completes the set R with randomly selected states (Line 10) and iterates again through the set of available states S. If it finds a state s that currently has a shorter execution path than any other state s_1 in R (and s_1 does not lead to a new

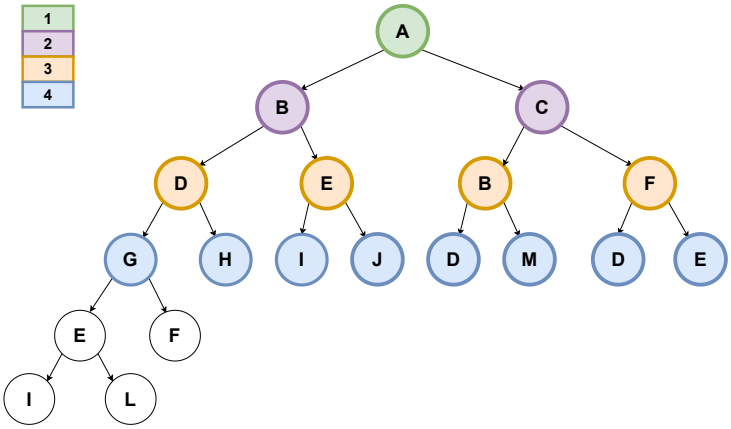


Figure 2.3: Explored states of the execution tree after 4 steps with BFS exploration

instruction), it removes s_1 from R (Line 13) and adds s to R (Line 14). At the end, it returns R to allow a new execution step to be performed on the states of R.

Algorithm 1 CBFS exploration

```
1: Inputs: A set of states: S
2:         Limit of states: L
3: Output: A set of L states requiring further exploration: R
4: R ← []
5: for state s ∈ S do
6:     if new(s.next_ip) then
7:         Add s to R
8:     if len(R) == L then
9:         Return R
10: Add randomly L−len(R) states from S to R
11: for state s ∈ S do
12:     if {∃s1 ∈ R|s.depth < s1.depth & !new(s1.next_ip)} then
13:         Remove s1 from R
14:         Add s to R
15: Return R
```

The resulting exploration of the execution tree using this strategy is illustrated in Figure 2.4. This strategy avoids a number of redundant states and therefore improves coverage compared to the **BFS-Strategy** (*i.e.*, for the same number of states explored, CBFS explores a more diverse part of the code than BFS).

In addition to **CBFS-Strategy**, we also implement a Coverage Depth-First Search Strategy (**CDFS-Stategy**), which is described in Algorithm 2 (the main difference with CBFS-Strategy is the condition to select $s1$ states at Line 12).

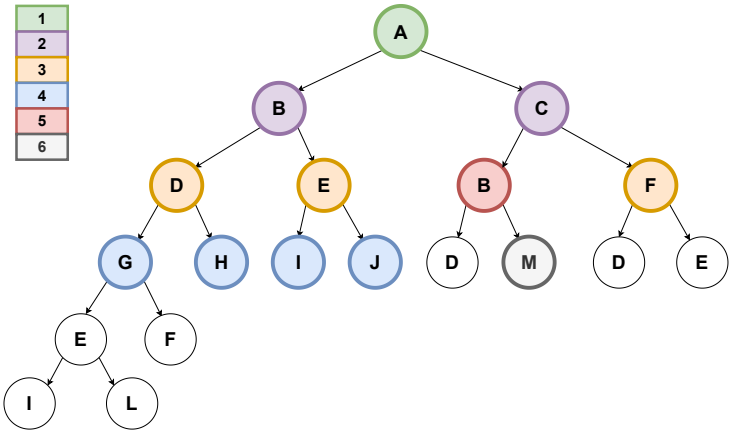


Figure 2.4: Explored states of the execution tree after 6 steps with CBFS exploration

Note that symbolic execution with this coverage heuristic is not new (*i.e.*, KLEE [CDE+08] uses a similar heuristic), but has not yet been tested in the context of a malware analysis toolchain.

Algorithm 2 CDFS exploration

```
Inputs: A set of states: S
        Limit of states: L
Output: A set of L states requiring further exploration: R
R ← []
for state s ∈ S do
    if new(s.next_ip) then
        Add s to R
    if len(R) == L then
        Return R
Add randomly L−len(R) states from S to R
for state s ∈ S do
    if {∃s1 ∈ R|s.depth > s1.depth & !new(s1.next_ip)} then
        Remove s1 from R
        Add s to R
Return R
```

The exploration of the execution tree using this strategy is shown in Figure 2.5. Note that the efficiency between **CDFS** and **CBFS** may vary depending on the execution tree. In our example, the **L** state is explored quickly while the **M** state is not.

Another major bottleneck associated with path explosion is the handling of loops within the binary. Symbolic loop conditions can produce infinitely repeating behavior. In these cases, deciding whether to stay in or exit the loop is a difficult decision that has been the subject of several studies, including

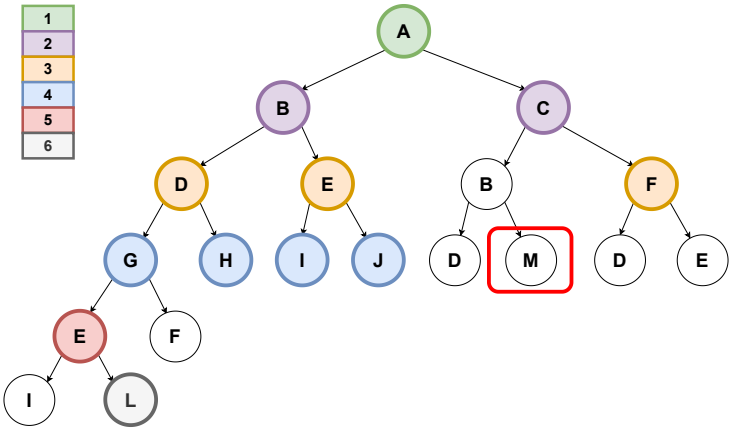


Figure 2.5: Explored states of the execution tree after 6 steps with CDFS exploration

Loop-Extended Symbolic Execution [Sax+09], Read-Write set [BCE08], and bit-precise symbolic mapping [XMW17]. Since these approaches may be too time consuming, we adopt two intermediate heuristics from [Seb+20]. The first one is applied to loops that have a symbolic value in their condition. Such loops can generate two states at each iteration: one that leaves the loop for symbolic values that no longer satisfy the condition, and one that remains in the loop for other values. This last state is then used to iterate the loop again. We decided to limit such loops to four iterations by default, as suggested by Sebastio et al. [Seb+20]. This method of breaking loops is shown in Figure 2.6. In cases where loops do not rely on symbolic values, they may still result in an infinite number of behaviors (*e.g.*, due to an incomplete environment model), the second loop heuristic involves restricting these loop executions to a finite, predetermined number of iterations, followed by switching to another state for further exploration.

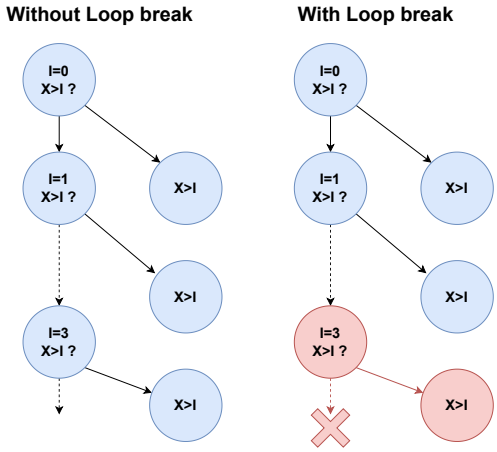


Figure 2.6: Illustration of the loop-breaking heuristic for a loop implying a symbolic value. While the symbolic execution stays in the loop without Loop break, the heuristic stops the execution of the loop execution of the loop after four iterations.

Application to the toy example Looking at the Code 2.2, symbolic execution allows to consider the values returned by `GetTickCount` and `IsDebuggerPresent` as symbolic and explore the malicious ("I'm evil!!") and benign ("Hello World!") behavior.

```
    ULONGLONG uptime = GetTickCount();
    Sleep(500000);
    ULONGLONG uptimeBis = GetTickCount();
    if ((uptimeBis - uptime)<500000 || IsDebuggerPresent())
    {
    MessageBox(NULL,"Hello world!","", MB_OK);
    }
    else
    {
     char message[20] = "";
     HINSTANCE hlib = LoadLibrary("msvcrt.dll");
     MYPROC func = (MYPROC) GetProcAddress(hlib,"strcat");
     (func) (message, "I'm ");
     (func) (message, "evil!!");
     MessageBox(NULL, message, "", MB_OK);
 }
```

Code 2.2: Malware code example

By recording the API calls observed in these two execution paths, we obtained the two execution traces A and B, shown in Figure 2.7. *TickCountValue* corresponds to the return value of `GetTickCount` and *IsDebugValue* corresponds to the return value of `IsDebuggerPresent`. Note that all buffer addresses have been resolved to their corresponding strings (arguments of `LoadLibrary`, `GetProcAddress`, `strcat` and `MessageBox`).

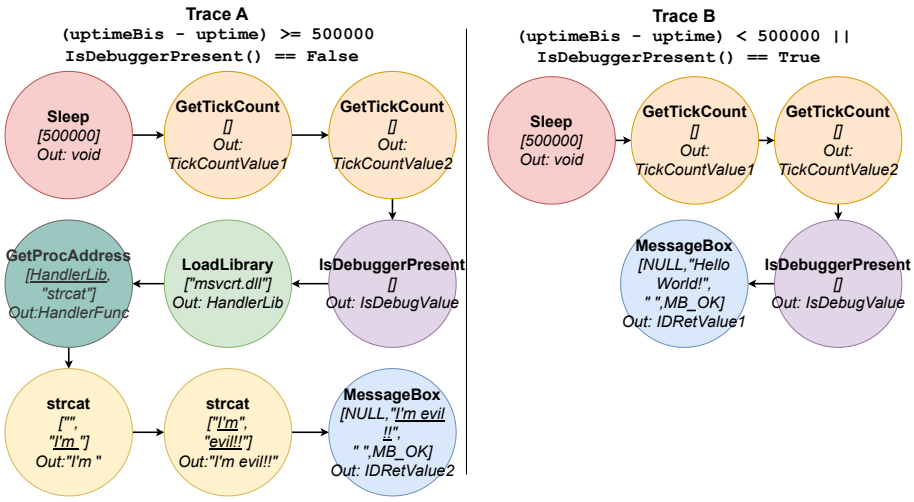


Figure 2.7: Traces extracted from the toy example

2.4 SCDG creation

A graph G is defined as a pair (V, E), where V is a set of vertices and $E = \{\{u, v\} \subseteq V\}$ is a set of edges. The set of edges and the set of vertices of G are denoted with $E(G)$ and $V(G)$, respectively. We consider labeled graphs, where a label function $l : V(G) \longrightarrow \Sigma$ assigns a label from Σ to each vertex of G. We use $l(v)$ to denote the label of vertex v. A graph G' = (V',E') is a *subgraph* of G=(V,E) if $V' \subseteq V$ and $E' \subseteq E$. During symbolic execution, all API calls and their arguments are recorded and collected into execution traces (*i.e.*, one execution trace for each path). These traces are then used to construct a System Call Dependency Graph of the binary[1]. Following [Sai+18; Seb+20], SCDGs are graphs in which each vertex is labeled with the name of an API call, and edges represent (an abstraction of) the flow of information between these calls. In practice, the construction of each SCDG involves merging and building connections between calls from one or more execution paths. In this section, we define several **SCDG-Strategies** to construct SCDGs.

First, let us consider strategies related to a single execution path. The simplest one is to take edges connecting two calls if they both share an argument with an identical value. For example, such edge type is created from two calls that use the same file handler. Using only this type of connection is called

[1]While system calls are well defined for ELF binaries, the concept is more complex and less accessible in Windows, where the user interacts with the kernel through a wide variety of API calls. In this thesis, we choose to keep the expression *System Call Dependency Graph* (SCDG), but a recent work [Puo+21] proposes the expression *External Call Dependency Graph* (ECDG) to take into account this discrepancy.

the one-edge strategy, or **SCDG-Strategy 1**. We can also consider two other edges type in addition to this first one. The second edge type is established between two calls that share an argument with the same symbolic value. For example, consider a symbolic file size returned by a call, and the program adds a specific value to this size before passing it as an argument to another call. Finally, we consider that two calls can be linked if an argument of the first call is the calling address of the second call. This situation occurs when a function is dynamically loaded with *getProcAdress*. Using the three types of edges is called the multipe edge types strategy, or **SCDG-Strategy 2**. Additionally, we assign a label to each edge with the indexes of the argument in both calls (the return value of a call is given index 0). According to the experiments of Chapter 3, **SCDG-Strategy 1** results in the loss of significant dependencies between calls and leads to more disconnected vertices in SCDGs. Symbolic values can be changed before passing to another call, and *GetProcAddress* is often used to mask the true contents of the import table in PE files.

An example of an SCDG is shown in Figure 2.8 with **SCDG-Strategy 1** and **SCDG-Strategy 2**. Below we detail the SCDG creation with **SCDG-Strategy 2**:

- The program calls *GetProcAddress*, which returns the address of the *CreateFile* call. A vertex is then created for *GetProcAddress*.
- The *CreateFile* call returns a handle to the file with the given filename. The vertex *CreateFile* and an edge between *GetProcAddress* and *CreateFile* are constructed because the address of CreateFile is obtained by the call to *GetProcAddress*.
- A new vertex (*SetFilePointer*) is created after calling *SetFilePointer* on the previous file handle. An edge is added between the returned argument of *CreateFile* (index 0) and the first argument of *SetFilePointer* (index 1).
- The vertices *ReadFile* and *WriteFile* are created and connected using the same principles.

In SCDGs, a vertex is created for each API call that occurs during an execution trace. This results in the creation of multiple vertices with identical labels when a single API is called multiple times. API calls may occur in different contexts (*i.e.*, different program instruction addresses). However, similar calls that share the same API name and instruction address may occur in the same execution trace, such as within a loop. In such cases, it may be appropriate to merge these calls into a single vertex, resulting in a more concise SCDG representation [Seb+20]. This merging strategy is called **SCDG-Strategy 3** and could be combined with either **SCDG-Strategy 1** or **SCDG-Strategy 2**. It has several advantages. First, it reduces the size of the SCDG,

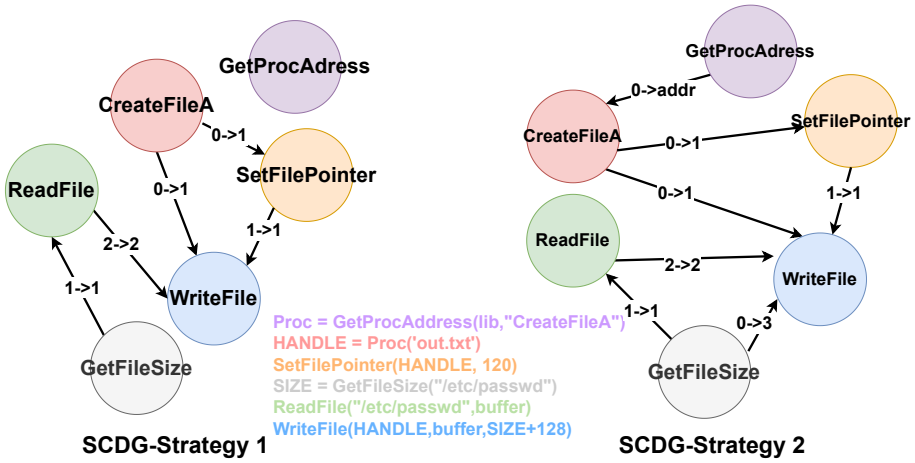


Figure 2.8: Example of SCDGs built with SCDG-Strategy 1 and SCDG-Strategy 2

which should improve classification and detection performance. In addition, it may reduce the effect of some calls during the learning phase. For example, the *wabot* malware executes up to a hundred instances of the *WriteFile* call. If not merged, some calls could make up a significant portion of the execution trace (and therefore the signature), potentially interfering with the learning process. However, there are malware families where **SCDG-Strategy 3** combines calls with distinct purposes, resulting in a possible loss of certain malware behaviors.

The above strategies apply only to a single execution trace. In situations where multiple traces are extracted from symbolic execution, additional strategies should be considered. One can choose to create an SCDG that is the disjoint union of the corresponding SCDGs of individual traces. This approach is referred to as **SCDG-Strategy 5**. In contrast, **SCDG-Strategy 4** involves combining calls from different execution paths. These strategies are illustrated in Figure 2.9. In this example, two traces call `GetProcAddress` and `CreateFileA` in the same context (*i.e.*, same arguments) before showing different behaviors. **SCDG-Strategy 5** is easier to apply, but could result in the union of a large number of disconnected graphs because each trace is considered separately. By definition, **SCDG-Strategy 4** and **SCDG-Strategy 5** are mutually exclusive, while the other strategies are not. Our experimental results indicate that **SCDG-Strategy 4** can improve the efficiency of the graph mining algorithm for families with a large number of paths since it allows a more compact representation.

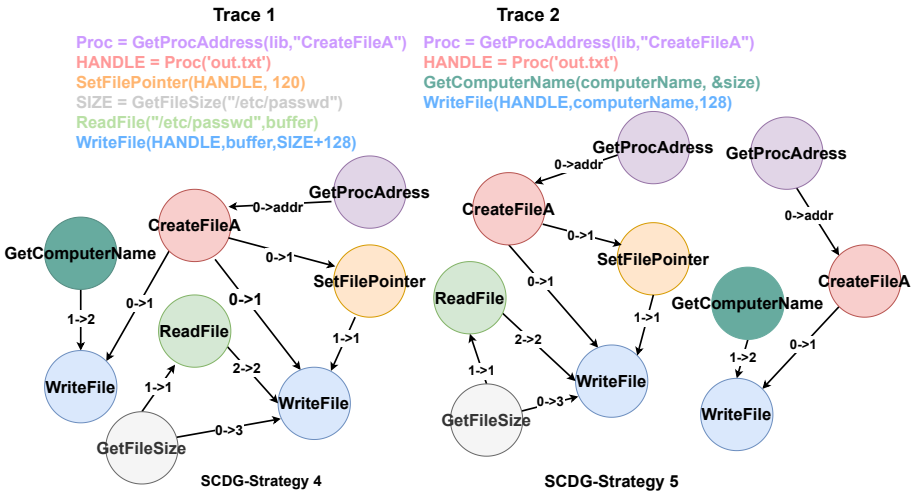


Figure 2.9: Example of SCDGs built with SCDG-Strategy 4 and SCDG-Strategy 5

Application to the toy example Looking at the traces obtained from the toy example and shown in Figure 2.10, we can now construct the SCDG.

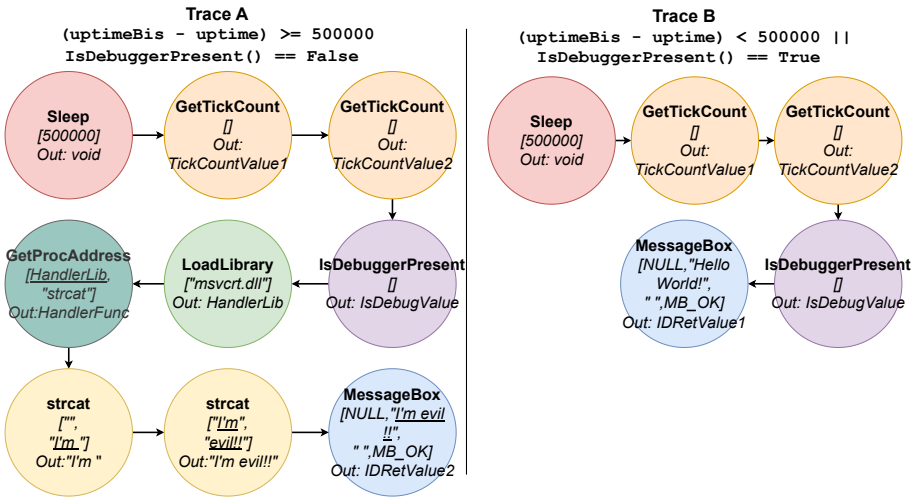


Figure 2.10: Traces extracted from the toy example

As an example, we use **SCDG strategies 2 and 4** to build the SCDG represented in Figure 2.11. The argument values used to construct the edges are colored in red.

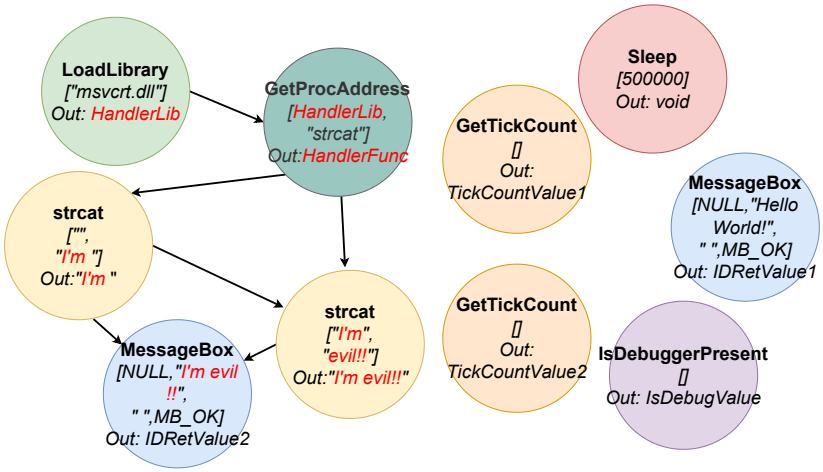


Figure 2.11: An example of SCDG obtained for the toy example

2.5 On Graph Comparison for Malware Analysis

Each SCDG provides an abstraction of a single binary. Our next step is to identify the similarities in SCDGs from samples of a family in order to build a classifier. In this section, we discuss the capabilities and limitations of three different classifiers, which are then evaluated in Chapter 3.

We are interested in using graph comparison to extract and compare malware signatures represented by our SCDGs. In particular, graph isomorphism is considered a powerful approach that allows us to detect structural similarities between graphs that may not be identical. Two labeled graphs G and H are said to be isomorphic ($G \simeq H$) if there exists a bijection $\phi : V(G) \longrightarrow V(H)$ such that $(u, v) \in E(G)$ if and only if (1) $(\phi(u), \phi(v)) \in E(H)$ (for all $u, v \in V(G)$), and (2) $l(v) = l(v')$ for all $(v, v') \in \phi$. There is a large number of graph similarity measures based on isomorphism, *e.g.*, subgraph isomorphism, or similar concepts such as largest common subgraph. Unfortunately, reducing the comparison of two graphs to checking their isomorphism is known to be restrictive, since it requires that both graphs have exactly the same structure. This situation is rarely encountered when comparing (classes of) malware. The situation is illustrated in Figure 2.12, where two malware from the same family are considered different because the vertex $SetFilePointer$ cannot be covered by an isomorphic relationship.

To address this problem, the authors in [Sai+18] proposed an approach based on gSpan. This is a popular algorithm for frequent graph-based pattern mining. The gSpan algorithm is presented in Appendix A.1. Given a dataset of graphs $\mathbb{G}$ and a desired support min_supp, gSpan tries to extract all subgraphs that exist in at least min_supp graphs of $\mathbb{G}$. If $\mathbb{G}$ represents a set of malware

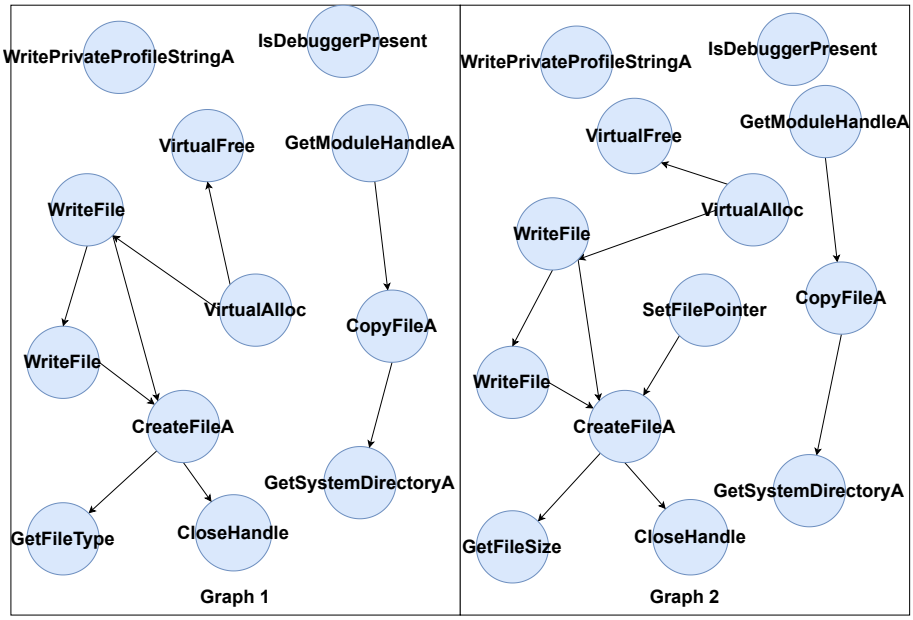


Figure 2.12: Example of non-isomorphic graphs with high similarities

from the same family, the output set of common subgraphs $\mathbb{S}$ represents their signatures. These signatures can then be interpreted by analysts who can verify the consistency of the malicious behavior detected.

The classifier based on gSpan first extracts signatures from each malware family. We obtain the signature G_i of each family by computing the largest common subgraphs between different SCDGs of malware belonging to the same family. In the classification phase, we compare the SCDG of the new binary with that of each signature to obtain their largest common subgraph G'_i. The new binary belongs to the malware family with the higher similarity. The similarity score S between the graphs G'_i and G_i is computed as follows (this similarity score was first used in [Sai+18]).

$$S(G'_i, G_i) = \frac{num_edges(G'_i)}{num_edges(G_i)}$$

We then classify the new sample by finding the family i with the highest similarity score. Unfortunately, relying on computing the largest subgraphs may ignore small but important connected components that do not belong to the largest subgraphs. The situation is illustrated in Graph 3 of Figure 2.13, which shows the application of gSpan on Graph 1 and 2 from Figure 2.12. We observe important calls such as *IsDebuggerPresent* may be ignored. An inefficient solution might be to increase the number of subgraphs. Unfortunately, for graph larger than our example, this approach will mostly favor

a variant of the largest connected component, as we will see in Chapter 3. Variants of gSpan [INR] propose to remove graph duplicates during the mining process to avoid the largest connected component variants, but this approach is computationally intensive.

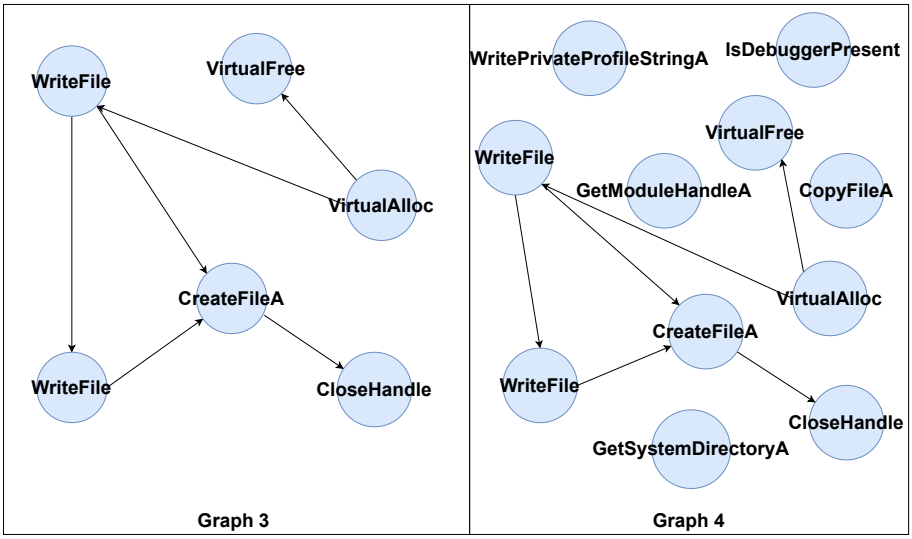


Figure 2.13: Graph 3 represents the subgraph extracted with gSpan from graphs of Fig. 2.12. Graph 4 corresponds to the information used by the simple kernel from [Puo+21]

To overcome this problem, we resort to the concept of graph kernels. Similar to the kernels introduced in Chapter 1, graph kernels are functions designed to measure the similarity of pairs of graphs by mapping them to another feature space. They allow algorithms such as KSVM to build a model for classifying graphs. It is common for kernels to compare data points using differences between data vectors. However, the structures of graphs are invariant to permutations of their representations (*i.e.*, the order of edges/vertices does not affect the structure and distance between graphs). This motivates the need to compare graphs in a way that is permutation invariant. Moreover, to avoid strict comparison (which would be equivalent to isomorphism), it is common to use smoother comparison metrics, such as *convolutionnal kernels*, for better generalization capabilities. Convolutionnal kernels divide structures (*i.e.*, graphs in our case) into substructures (*e.g.*, edges, subgraphs, paths, etc) and then evaluate a kernel between each pair of such substructures.

In [Puo+21], the authors propose a similarity metric for malware behavior graphs based on common vertices and edges. Specifically, they define a similarity σ between two graphs G and H as:

$$\sigma(G,H) = \alpha\sigma_{vertices}(G,H) + (1-\alpha)\sigma_{edges}(G,H) \tag{2.1}$$

where α is the vertex-edge factor, which allows to adjust the weight of vertices and edges in the similarity function (set to 0.25 at the end of their work). The vertex similarity is defined as:

$$\sigma_{vertices}(G,H) = \frac{|V(G) \cap V(H)|}{min(|V(G)|, |V(H)|)} \tag{2.2}$$

and the edge similarity as:

$$\sigma_{edges}(G,H) = \frac{|CC_{max}(G \cap H)|}{min(|CC_{max}(G)|, |CC_{max}(H)|)} \tag{2.3}$$

where $V(G)$ is the set of vertex of G, $G \cap H$ is the largest common subgraph between G and H, and $CC_{max}(G)$ is the largest connected component of G.

While this approach adds information about all vertex labels compared to gSpan, it suffers from similar drawbacks as gSpan. In fact, it focuses on the largest connected component and neglects edges in other connected components. This problem is illustrated in Graph 4 of Figure 2.13, which shows the information used by the kernel when applied to graphs from Figure 2.12. One can see that the kernel identifies the similarities between vertices of Graph 1 and Graph 2. However, it ignores important edge dependencies such as *GetModuleHandle*, *CopyFileA*, and *GetSystemDirectoryA*.

To address this problem, we explore other graph kernels that are able to compare the local structures of graphs. In this approach, two vertices of different graphs are considered similar if they share the same labels. The two vertices are considered more similar if they also share similar neighborhoods (*i.e.*, vertices with the same labels). Using this approach, Shervashidze et al. [She+11] introduced graph kernels based on the 1-dimensional Weisfeiler-Lehman (WL) isomorphism test. Let G and H be graphs and $l : V(G) \cup V(H) \rightarrow \Sigma$ be a function that assigns labels to their vertices. By several iterations $i = 0, 1, ...$, the 1-WL algorithm computes a new label function $l_i : V(G) \cup V(H) \rightarrow \Sigma$, with each iteration allowing comparison of G and H. Let $N(v)$ be the neighborhood of a vertex $v \in G$ in $V(G)$, *i.e.*, $N(v) = \{u \in V(G) | (v,u) \in E(G)\}$. In the first iteration, $l_0 = l$, and in subsequent iterations,

$$l_i(v) = relabel(l_{i-1}(v), sort(l_{i-1}(u) | u \in N(v))) \tag{2.4}$$

where $v \in V(G) \cup V(H)$, sort(S) returns a sorted tuple of S and the function $relabel(p)$ maps the pair p to a unique value in Σ that has not been used in previous iterations. If the cardinality of l_i is equal to the cardinality of l_{i-1}, the algorithm stops. The idea of the WL subtree kernel is to compute the

previous function for $h \geq 0$ and after each iteration i to compute a feature vector $\phi_i(G) \in \mathcal{R}^{|\Sigma_i|}$ for each graph G, where $\Sigma_i \subseteq \Sigma$ denotes the image of l_i. Each component $\phi_i(G)_{\sigma_j^i}$ counts the number of occurrences of vertices labeled with $\sigma_j^i \in \Sigma_i$. The total feature vector $\phi^{WL}(G)$ is defined as the concatenation of the feature vectors of all h iterations, *i.e.*,

$$\phi^{WL}(G) = (\phi^0(G)_{\sigma_1^0}, ..., \phi^0(G)_{\sigma_{|\Sigma_0|}^0}, \phi^h(G)_{\sigma_1^h}, ..., \phi^h(G)_{\sigma_{|\Sigma_h|}^h}) \tag{2.5}$$

Finally, to compute the similarity between two different feature vectors, we apply the following formula:

$$k_{WL}(G, G') = \sum_{\phi \in \phi^{WL}(G)} \sum_{\phi' \in \phi^{WL}(G')} \delta(\phi, \phi') \tag{2.6}$$

where δ is the Dirac kernel, *i.e.*it is 1 if its arguments are equal and 0 otherwise. The more labels the two graphs have in common, the higher this kernel value will be. Compared to gSpan and the kernel of [Puo+21], this kernel targets similarities related to all vertices and edges of the largest subgraph, but also local similarities. This is illustrated in Figure 2.14, where dependencies between *GetModuleHandle*, *CopyFileA*, and *GetSystemDirectoryA* are kept in the learning process.

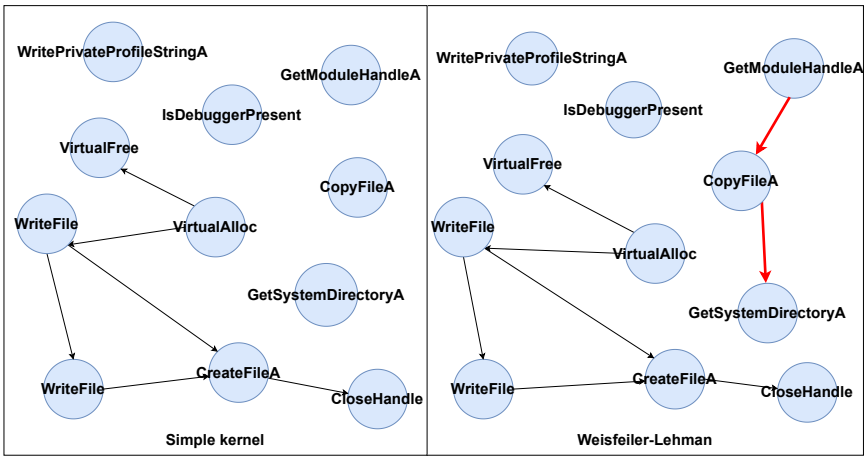


Figure 2.14: The figure shows that, contrary to the simple kernel from [Puo+21], the Weisfeiler-Lehman kernel captures all common edges between the graphs of Figure 2.12

Application to the toy example To illustrate the application of these classifiers, we consider a variant of our toy example. The initial toy example code is shown in Code 2.3 while its variant is shown in Code 2.4.

```
ULONGLONG uptime = GetTickCount();
Sleep(500000);
ULONGLONG uptimeBis = GetTickCount();
if ((uptimeBis - uptime)<500000 || IsDebuggerPresent()){
MessageBox(NULL,"Hello world!","", MB_OK);}
else{
 char message[20] = "";
 HINSTANCE hlib = LoadLibrary("msvcrt.dll");
 MYPROC func = (MYPROC) GetProcAddress(hlib,"strcat");
 (func) (message, "I'm ");
 (func) (message, "evil!!");
 MessageBox(NULL, message, "", MB_OK);}
```

Code 2.3: Malware code example

```
ULONGLONG uptime = GetTickCount();
Sleep(500000);
ULONGLONG uptimeBis = GetTickCount();
if ((uptimeBis - uptime)<500000 || IsDebuggerPresent()) {
MessageBox(NULL,"Hello world!","", MB_OK);}
else {
 char* fl[2] = {"cat","str"};
 char buf[10],message[20];
 strcpy(buf, fl[1]);
 strcpy(buf+3, fl[0]);
 HINSTANCE hlib = LoadLibrary("msvcrt.dll");
 MYPROC func = (MYPROC) GetProcAddress(hlib,buf);
 (func) (message, "I'm ");
 (func) (message, "evil!!! ");
 MessageBox(NULL, message, "", MB_OK);}
```

Code 2.4: Variant of the Code 2.3

If we consider **SCDG strategies 2 and 4**, we obtain the SCDGs shown in Figure 2.15. The signature obtained from these two SCDGs with gSpan is illustrated in Figure 2.16. As described in this section, the kernel from [Puo+21] will not only use this biggest common subgraph but will also consider all vertices labels to compare SCDGs. Moreover, the Weisfeiler-Lehman kernel will also consider the other common subgraphs.

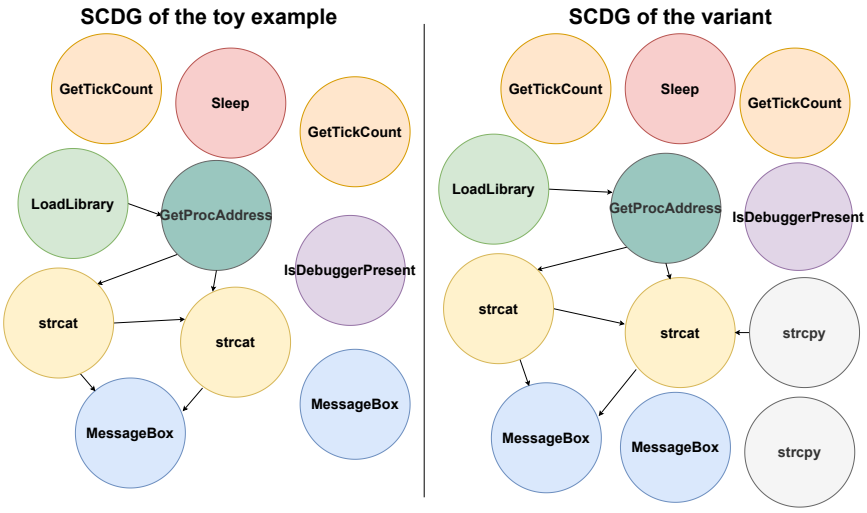


Figure 2.15: SCDGs of the toy example and its variant

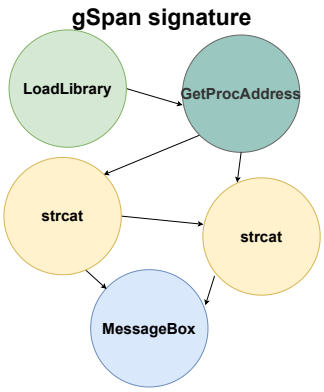


Figure 2.16: gSpan signature obtained from the toy example and its variant

SEMA - Symbolic Execution for Malware Analysis: Implementation and Evaluation

3

This chapter introduces SEMA, which stands for Symbolic Execution for Malware Analysis, and its various building blocks. In particular, it describes the implementation of API extraction, SCDG creation and classifiers. We then present the performed experiments and the analysis of our results.

3.1 The Architecture of SEMA

SEMA- Symbolic Execution for Malware Analysis - is an open source tool [Ber+] capable of analyzing, detecting and classifying malware. It is implemented in Python (version 3.8) and is designed to be modular, facilitating extensions and adaptability. The architecture of SEMA consists of two main components, illustrated in Figure 3.1: `SEMA-SCDGs` and `SEMA-Classifier`.

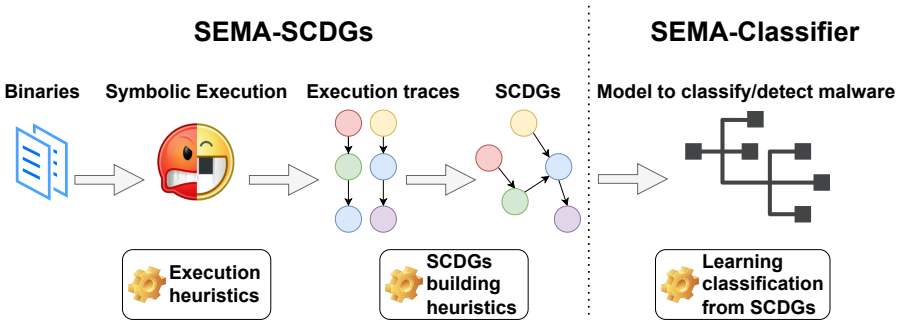


Figure 3.1: Overview of SEMA workflow

`SEMA-SCDGs` is implemented as an extension of the angr symbolic execution framework [Sho+16]. Its objective is to efficiently explore malware binaries, to construct SCDGs using the obtained execution traces and to output the SCDGs as JSON files containing calls and arguments. It can handle

ELF and PE files[1]. It provides exploration strategies and SCDG-strategies as described in Chapter 2. To avoid symbolic exploration of external libraries such as `user32.dll`, we generate default dummy API implementations using Microsoft documentation signatures. In addition, we implemented dedicated versions of over 80 common API calls using abstractions proposed by angr.

`SEMA-Classifier` implements the malware detection/classification component. Given a set of multiple JSON files containing SCDGs as input, it can train classifiers and store them for future use. The models are based on either graph mining with *gSpan* [YH02] or graph kernels and support vector machines. When encountering unknown binaries, this module loads previously trained models that can be used for classification. Similar to `SEMA-SCDGs`, `SEMA-Classifier` is modular and allows easy addition of new classifiers.

Previous works [Sai+18; Seb+20; Puo+21] proposed a similar toolchain based on symbolic execution, SCDGs, and machine learning to detect and classify malware. Said et al. [Sai+18] proposed to detect the Mirai botnet from cleanware by extracting SCDGs using symbolic execution. They focus on ELF binaries (Mirai samples and cleanware), do not tune angr for binary exploration, and construct their SCDGs as the disjoint union of SCDGs corresponding to individual execution traces. They then use gSpan to extract the largest common subgraphs between SCDGs and construct signatures. Different parameters related to gSpan have been investigated such as the minimum support, the number of subgraphs, the similarity threshold, or the consideration of edge directions. These experiments show an interesting trade-off between edge directions and subgraph commonalities. Sebastio et al. [Seb+20] investigate how to tune the SMT solver to improve malware exploration and classification. They explore PE binaries with angr and a one-hour timeout to distinguish cleanware from malware and identify their malware family. Symbolic exploration is improved by considering generic *simprocedure* for most API calls. They also limit the maximum number of concurrent states to explore, implement a timeout for each execution step, and implement the loop-breaking heuristic already introduced in Section 2.3. From the obtained execution traces, they compare strategies for merging traces and calls as presented in Section 2.4. They evaluate gSpan with different parameters to create signatures and classify new binaries. Finally, Puodzius et al. [Puo+21] proposed to apply unsupervised learning to SCDGs to discover clusters of malware families. Their symbolic exploration and SCDGs use the same parameters as [Seb+20]. To train their classifiers, they build a similarity metric based on common vertices and edges (using gSpan), which is presented in Section 2.5.

SEMA extends these works by improving the symbolic execution cover-

[1]With the exception of .NET files. Symbolic exploration of malware that manipulates its control flow (*e.g.*, with process injection) is not handled either.

age, proposing a new classifier, and investigating the link between classifiers and the SCDG representation. In particular, we propose to prioritize the unseen part of the code during exploration and to improve the modeling of the OS environment (*e.g.*, dynamic import of functions with `GetProcAddress`). These improvements allow us to build SCDGs within a ten minute timeout and improve the range of binaries that symbolic execution can analyze. We build SCDGs with new edge types and propose a new classifier based on the Weisfeiler-Lheman kernel. We show that our classifier makes better use of the SCDG information and outperforms previous classifiers in terms of training time and accuracy.

As already suggested by Said et al. [Sai+18], we do not pretend to replace syntactic analysis tools, but to provide a new tool that could be combined with others. This combination would allow to cover the shortcomings of each tool, as exposed in Section 1.1. Since syntactic approaches generally have negligible cost compared to behavioral approaches like SEMA, it might be interesting to apply a syntactic approach first. If the analyzed binary matches a syntactic signature, or a syntactic approach has high confidence in its classification decision, we do not need to construct a SCDG of the analyzed binary. Otherwise, if the syntactic approach does not produce a decision about the analyzed sample, we could apply a behavioral approach such as SEMA. After analyzing the sample, the syntactic signatures could be updated to take the sample into account. This ensures that the cost of behavioral analysis is paid only once, if the sample needs to be analyzed again. While this combination is not discussed further in this thesis, it remains a promising future research direction.

This chapter is organized as follows. Section 3.2 briefly introduces angr, the symbolic execution framework used in SEMA. Section 3.3 then presents `SEMA-SCDGs`, which explores the input binary and produces the corresponding SCDG. Section 3.4 presents `SEMA-Classifier` which implements three different classifiers for malware detection/classification. Section 3.5 presents experiments performed to evaluate the exploration strategies, SCDG strategies and classifiers. Finally, Section 3.6 concludes and presents possible future work.

3.2 The angr framework

For its symbolic execution engine, SEMA uses angr [Sho+16], an open-source Python framework. This choice was motivated by the ability to easily prototype new ideas in Python, the great modularity offered by angr to define new strategies for symbolic execution, and the important community around this framework. The architecture of angr is presented in Figure 3.2. When a binary is analyzed, it is first loaded into memory using CLE (*CLE Loads Everything*),

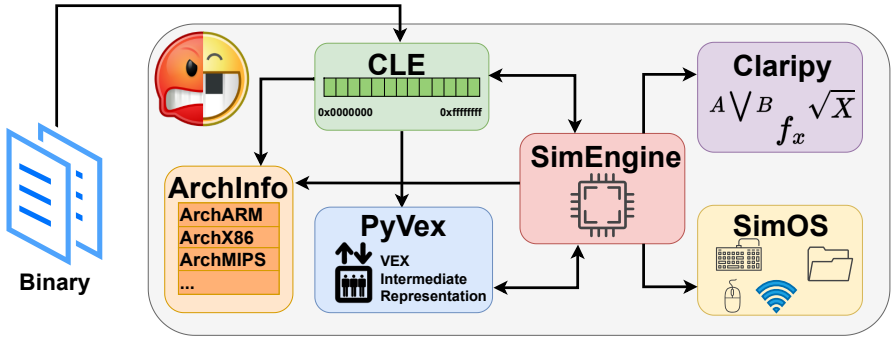


Figure 3.2: Illustration of angr internals inspired from [UG23]

the angr loader. By guessing the architecture with `Archinfo` (an architecture database using headers and heuristics), `CLE` creates a representation of the program as if it were loaded into memory. Then `PyVEX` allows the machine code to be lifted into an intermediate representation: `VEX`. This representation is used by the `SimEngine` to simulate the execution of the initial binary. Its goal is to instantiate an initial state of the binary (*i.e.*, `SimState`) containing information about registers, memory,... Then, for a given state, its `Simulation Manager` executes the corresponding block of `VEX` instructions get its set of successors states. It also adds constraints to `SimStates` related to the execution path. `Claripy` handles all constraints added during execution using an SMT solver (Microsoft's Z3). Finally, `SimOS` is responsible for all abstractions related to the operating system (*e.g.*, files, network, etc.) and provides summaries of common library functions (*i.e.*, `SimProcedures`).

As we are providing exploration strategies, we are particularly interested in two abstractions provided by the `SimEngine`: `SimState` and `Simulation Manager`.

SimState: It represents a snapshot of the execution state of the program at a given time. It provides access to registers, instructions, memory, and current constraints on the execution paths. To improve the exploration and to allow the creation of the SCDG, some variables have been added to the SimState for `SEMA` including `id` (unique identifier for the state), `JumpTable` and `JumpExceded` (variables that allow to track the control flow followed by the state and to identify if the state is stuck in a loop), ... A plugin dedicated to environment variables has also been created, allowing to set/modify common variables and keep track of environment variable requests.

Simulation Manager: It is the interface to control symbolic execution over groups of states and to explore the state space of the program. States are

grouped into stashes, which allow them to be ordered according to different characteristics:

- *active*: Stash where states currently used for exploration are stored. To reduce memory consumption, an option to limit the number of states in this stash has been added: *simul_state*, which decides how many states can be in this stash at the same time. If this limit is exceeded, some states will be moved to the *Pause* stash.
- *pause* (SEMA): The stash to which states will be moved to wait until some space becomes available in the *Active* stash.
- *newAddr* (SEMA): The stash where states leading to new instruction addresses (not yet explored) of the binary are kept. This stash extends the pause stash and is used by our exploration strategies to prioritize states when space becomes available in the *Active* stash.
- *excessLoop* (SEMA): The stash where states that exceed the threshold associated with loops are moved. When new states are needed, and there is no state available in the active and pause stashes, states in this stash are used to resume exploration and increase the loop threshold.
- *deadended*: The stash to which states are moved when they have no successors and their execution is therefore finished (*e.g.*, `exit()` function call).

3.3 SEMA-SCDG

SEMA uses the angr framework to symbolically explore a binary and extract its API calls. Following the elements discussed in Section 2.3, there are several options available to tune the exploration of the analyzed binary:

- BFS | CDFS | CBFS: Exploration strategy used to select the next states to explore. Default: CDFS
- NB_ACTIVE: Maximum number of states contained in the *active* stash. It represents the maximum number of concurrently executed states during an exploration step. Default: 5
- LOOP_THRESHOLD: Maximum number of iterations allowed for a state within a loop with symbolic condition before it is moved to the *excessLoop* stash. Default: 4
- TIMEOUT: Timeout before ending symbolic execution of the binary. Default: 600 seconds

- Z3_CONFIG: Use the optimized version of z3 by Sebastio et al. [Seb+20]. Default: True
- DISJOINT_TRACES: Consider disjoint union of traces for SCDG. Default: False
- MERGE_CALLS: Merge calls with similar arguments in SCDG. Default: True.
- THREE_EDGES: Consider the three edge types when building the SCDG. Default: True.
- OUT_FORMAT: The format to output the SCDG, can be either JSON or GS file (input for gSpan). Default: JSON

The Code 3.1 shows an exploration step of SEMA. First, an execution step (*i.e.*, the execution of a block of instructions) is applied to the states from the *active* stash (Line 6). The states are then examined to identify states currently in a loop (Line 13) or leading to a new instruction address (Line 16). States are then selected using the current exploration strategy (Line 19) as defined in Section 2.3. Finally, the process is repeated until no states remain or the timeout specified to SEMA has passed (Line 26).

```
1  Input:  A strategy Strat
2          A simulation manager Simgr
3  Output: An updated simulation manager Simgr
4
5  # Apply execution step to states from the '
       active' stash
6  Simgr.step(Simgr.active)
7
8  #Identify new states produced by a fork and
       give them an ID
9  Compare_state_snapshot()
10
11 """Move states exceeding their loop counter
12  to the 'excessLoop' stash"""
13 Remove_exceeded_jump(Simgr.active,Simgr.
       excessLoop)
14
15 #Identify states leading to unexplored
       instructions addresses
16 Check_new_address(Simgr.active,Simgr.newAddr)
17
```

```
18 #Select states according to exploration
      strategy
19 Select_states(Strat,Simgr.active,Simgr.pause,
      Simgr.newAddr)
20
21 # If necessary, continue to explore states from
       'excessLoop'
22 if empty(Simgr.active,Simgr.pause,Simgr.
      newAddres):
23         Select_states(Strat,Simgr.excessLoop)
24
25 #If no more states are available or the time is
       out, exit
26 if empty(Simgr.active) or time_out:
27         exit()
```

Code 3.1: Pseudo code of the exploration of a binary

To capture API calls, SEMA uses the breakpoint interface provided by the angr API. Whenever an API call is encountered during an execution step, a callback allows us to record the API name and examine the current execution state. Using the Windows documentation to get API signatures, we extract the API arguments and try to resolve (*i.e.*, decode the byte sequence) those that correspond to strings (*i.e.*, the arguments of type LPWSTR, PWSTR, LPSTR,...).

Environment Modeling Proper environment modeling is a major challenge in developing efficient symbolic execution techniques. In fact, when we use symbolic execution we avoid exploring/executing the API call code (*i.e.*, since performing such an operation would drastically increase the computation time [Lin17]). In angr, when a call to an external library occurs, the call is hooked into a simulated procedure called *Simprocedures*, which summarizes the effect of the function on the state. A simple but crude implementation of such a procedure is to assume that the external function returns a symbolic value without any constraints. However, this requires knowing the **calling convention** and the number of arguments of the API function, which can be found in the Microsoft documentation [Micb]. Most Windows API functions use the `__stdcall` calling convention, where the callee cleans the stack. This means that the API call arguments pushed onto the stack are normally popped at the end of the API function code, and this should be done in the same way by the corresponding *Simprocedure*. On the contrary, the `__cdecl` calling convention (*i.e.*, the default convention for C and C++ programs) lets the caller clean up the stack and allow the implementation of variadic functions. These two calling conventions are illustrated in Code 3.2 and 3.3.

```
push 0x00
push 0x402000
push 0x402010
push 0
call MessageBoxA
push 0
call ExitProcess
```

Code 3.2: `__stdcall` calling convention

```
push edi
push edx
push ecx
push eax;
push [ebp+var_8]
call wsprintfA
add esp, 14h
```

Code 3.3: `__cdecl` calling convention

The function signature from Microsoft's API documentation (including argument number/type, return type, and calling convention) has been compiled into multiple json files (*i.e.*, one for each dll) as shown in Code 3.4.

```
"GetTempFileNameA": {
        "cc": "__stdcall",
        "returns": "UINT",
        "name": "GetTempFileNameA",
        "arguments": [
            {
                "type": " LPCSTR",
                "name": "lpPathName"
            },
            {
                "type": " LPCSTR",
                "name": "lpPrefixString"
            },
            {
                "type": " UINT",
                "name": "uUnique"
            },
            {
                "type": " LPSTR",
                "name": "lpTempFileName"
            }]}
```

Code 3.4: Example of API signature in json

From these signatures, we create default implementations for each *Simprocedure*, as presented in Code 3.5. These *Simprocedures* return symbolic values with no constraints. In practice, such a solution allows us to observe multiple API calls related to malicious behavior in 26 percent of the analyzed families. Indeed, this solution may generate output that is not defined in the API specification. In addition, it ignores many potential effects of the call, including modification of input parameters (*e.g.*, modification of a buffer

passed as an argument to memcpy) or interaction with the operating system (*e.g.*, modification of a registry key, creation of a file,...). This can lead to inconsistent executions if these elements affect the rest of the execution (*e.g.*, missing or inaccurate path constraints).

```
1     def generic_sim_procedure(self, args, ret):
2         s = "lambda self, " + ", ".join(args)
3         if ret != "void":
4             s += ': self.state.solver.BVS(
5             "retval_{}".format(self.
                display_name),
6             self.arch.bits)'
7         else:
8             s += ": None"
9         return eval(s)
```

Code 3.5: Implementation of a generic *Simprocedure*

We propose implementations of *Simprocedures* that follow different principles to fix these issues. First, we restrict the ranges of outputs to those given in the API specification. For example, if the output is an integer variable that can only take four values, *Simprocedures* would generate those values instead of the full range of integers. To improve exploration, we emulate the changes made by some API calls with concrete values. For example, a call may change some of its inputs (*e.g.*, a buffer passed as an argument to contain a path) or even some environment variables. Note that writing models such as *Simprocedures* is generally a time-consuming task [Bal+18] that needs to be done for each call in our dataset that causes problems. We have created a *Simprocedures* library that will be continuously enriched with new implementations according to our experiments. Currently, over 80 common API calls have been implemented using the abstractions proposed by angr. Finally, angr already proposes different implementations of *Simprocedures* but these are mainly related to calls used on Linux systems (*e.g.*, the POSIX library).

As an example, the implementation of the `GetTempFileNameA` *simprocedure* is shown in Code 3.6. According to the Microsoft documentation [Micb], `GetTempFileNameA` *"creates a name for a temporary file. If a unique filename is generated, an empty file is created and the handle to it is released [...], [the temporary name] is of the form* `<path>\<pre><uuuu>.TMP`*"*. The arguments are resolved in Line 9,10 and 11. Note that strings are resolved with a separate method `decodeString` to easily extend this procedure to wide strings (*i.e.*, `GetTempFileNameW`). Following the specification, a `uid` is created if the passed argument is 0 (Line 12 and 13), the file is created (Line 16) and the buffer argument `lpTempFileName` is filled with the temporary name (Line 22).

```
class GetTempFileNameA(angr.SimProcedure):
    def decodeString(self, ptr):
        fileName = self.state.mem[ptr].string.
            concrete
        return fileName

    def run(self, lpPathName, lpPrefixString,
     uUnique, lpTempFileName):

        dirname = self.decodeString(lpPathName).
            decode("utf-8")
        name = self.decodeString(lpPrefixString)
            [:3]
        uid = self.state.solver.eval(uUnique)
        if uid == 0:
            uid = int(timer.time())
        hexnum = "{0:0{1}x}".format(uid, 2)

        fd = self.state.posix.open(
            dirname + name + hexnum +
             ".TMP\0", self.state.solver.BVV(2,
                self.arch.bits)
          )
        newName = dirname + name + hexnum + ".TMP
            \0"
        newName = self.state.solver.BVV(newName)
        self.state.memory.store(lpTempFileName,
            newName)
        return int(hexnum, 16)
```

Code 3.6: Implementation of `GetTempFileNameA` *Simprocedure*

In Appendix B.1, we present the Simprocedure related to `GetProcAddress`. According to the Microsoft documentation [Micb], `GetProcAddress` *"retrieves the address of an exported function (also known as a procedure) or variable from the specified dynamic link library (DLL)"*. This function is often used by malware to import a function at runtime without declaring it in the IAT table. It is therefore important to ensure that functions imported in this way are correctly redirected to the appropriate *Simprocedure*.

While a significant number of *Simprocedures* have been implemented (*i.e.*, more than 80), there are still a large number of Windows APIs that do not have a dedicated *Simprocedure*. Similarly, specific OS features such as exception

handling, the file system, shell commands, or obscure x86 instructions are difficult to model extensively. These limitations are common to symbolic execution approaches that model the environment [Bal+18] and affect their scalability, as the path explosion problem does. Therefore, an analyst *could* need to extend SEMA to analyze new samples different from those analyzed in this chapter. However, this implementation cost is usually paid once and helps to increase the capabilities of SEMA. The binaries that use multiple threads/processes are also difficult to analyze automatically and generally require human intervention [Luc+23; Bal+17]. Finally, some obfuscation strategies such as packing can be bypassed by SEMA at the cost of high overhead (z3 solver time, self-modifying code,...).

From execution traces to SCDGs When binary exploration is complete (*i.e.*, either execution times out or there are no more paths available), the collected execution traces are used to construct the SCDG. The process of building an SCDG from a set of execution traces `T` and the SCDG strategies described in Chapter 2 (namely `merge_trace`,`merge_call` and `three_edges`) is outlined in Code 3.7. First, we initialize dictionaries containing the (`vertices`) of the SCDG, the `edges` of the SCDG, the calls observed in the currently analyzed execution trace (`calls_current_traces`) and the arguments of all calls already analyzed (`all_args`). We iterate through all calls (Line 14) of all traces (Line 12). The SCDG creation is adapted according to the `merge_trace` parameter (Line 16). For each call, we check whether the corresponding vertex should be created with `is_new_vert`. In the case of `merge_trace`, we check the presence of a call with same address and name in `vertices` (Line 17). In the other case (*i.e.*, the disjoint union of traces), we check the presence of a call with same address and name in `calls_current_traces` (Line 22). If the call is in `vertices`/`calls_current_traces` (Line 32), the function also checks if `merge_call` is enabled or if there is already a call with the same arguments in `vertices`/`calls_current_traces` (Line 29). In this case, the function returns `False` (Line 34), otherwise it returns `True` (Line 37).

When a vertex is created, we iterate through its arguments to create the corresponding edges with `create_edges()` (Line 17 and Line 24) and add the arguments to `all_args` (Line 20 and Line 25). This function simply iterates through all elements of `all_args`. If `arg` has the same value as an element of `all_args` and that element is related to a call of `calls_current_traces`, we create an edge. Finally, if `merge_trace` is disabled, we clean up `all_args` because each trace is considered independently (Line 27).

```
1 Inputs: A set of execution traces T (Calls + arguments)
2         merge_trace (True or False)
```

```
3           merge_call (True or False)
4           three-edges (True or False)
5   Output: SCDG of the executable (vertices and edges)
6
7   vertices={} # vertices of the SCDG
8   edges={} # edges of the SCDG
9   calls_current_trace={} #calls observed in the current
        trace
10  all_args={} #args observed and the associated vertices
11
12  for trace in T:
13  #Parse the trace and add all its elements to the SCDG
14    for call in trace:
15        calls_current_trace.add(call['addr']+'.'+call['name
              '])
16        if merge_trace:
17            if is_new_vert(call, vertices):
18                for arg in call[args,addr,ret_value]:
19                    create_edges(arg, all_args, edges, three-
                          edges, calls_current_trace)
20                    all_args.add(arg,call)
21        else:
22            if is_new_vert(call, calls_current_trace):
23                for arg in call[args,addr,ret_value]:
24                    create_edges(arg, all_args, edges, three-
                          edges, calls_current_trace)
25                    all_args.add(arg,call)
26      if not merge_trace:
27        all_args.clean()
28    calls_current_trace.clean()
29  #Check if the call create a new vertice and updates
        vertices and all_args
30  def is_new_vert(call,vertices_list):
31    #Check if a call with same address and name is in
          vertice_list
32    if call['addr']+'.'+call['name'] in vertices_list:
33        if merge_call or (exact_match(call,vertices_list)):
34            return False
35    else:
36        vertices_list.add(call['addr']+'.'+call['name'])
37    return True
```

Code 3.7: Pseudo-code to build SCDGs

As suggested in [CJK07], since malware often read from their own executable (*e.g.*, to replicate), while this behavior is less common in benign files, dependencies could be added to the SCDG to model this behavior. To capture this behavior in our SCDG, we decide to add a `Main` call at the beginning of

each trace, taking as argument the name of the executable file.

3.4 SEMA Classifier

SEMA Classifier is an interface that allows to manage the common operations of the machine learning pipeline when implementing new classifiers: initializing the dataset, training, testing, and storing classification results. The interface is similar to standard library such as sklearn and contains the following operations:

- `init_classifier(params)` initializes the classifier with the given parameters `params`.
- `train(dataset)` trains the classifiers with the input dataset.
- `classify(dataset)` outputs a malware family for each sample of the input dataset.
- `detect(dataset)` outputs a decision (malicious/clean) for each sample of the input dataset.
- `save_model(out_path)` save the model for future uses.
- `load_model(in_path)` loads the model from the input path.

Following Section 2.5, we implement three classifiers using this interface: gSpan classifier, SVM with the kernel from [Puo+21] and SVM with the Weisfeiler-Lehman kernel.

gSpan Classifier This classifier is based on quickSpan [INR], an optimized implementation of gSpan that uses partial pruning, parallelization, avoids to store/build unnecessary intermediate structures,... During the training phase, the largest common subgraph is extracted from the training set to build each family signature. Then, during the classification phase, the largest common subgraphs between each signature and the sample to be classified are computed. Finally, the sample is assigned the family label corresponding to the signature with the highest similarity (and above a certain threshold if we consider cleanware) as defined in Section 2.5. If the sample is equally similar to multiple families, the largest signature is preferred. Different parameters can be specified for this classifier:

- `DIRECTED_GRAPHS`: Consider edge direction during graph mining. Default: False.

- SUPPORT: Percentage of the training set that must contain the mined subgraphs. Default: 0.75.
- BIGGEST_SUBGRAPHS: Number of subgraphs to mine. Default: 5.
- THRESHOLD: Similarity threshold to consider for malware detection. Default: 0.5.

SVM with kernel from [Puo+21] With this kernel, the training phase consists of computing the similarity between each pair of malware from the training set. Similar to [Puo+21], we use gSpan to compute the edge similarity. The vertex-edge factor (see Section 2.5) is a parameter that is set to 0.25 by default. As discussed in Chapter 1, we generate a Gram matrix from the training set to train this classifier (note that we take advantage of the fact that a Gram matrix is symmetric and compute only half of it). In the classification phase, we compute the Gram matrix between each SCDG to be classified and the SCDGs from the training set. For this classifier we use the scikit-learn library version 1.0.2.

SVM with the Weisfeiler-Lehman (WL) Kernel In the case of the WL kernel [She+11], the training phase first consists in computing, for each sample of the training set, the feature vector corresponding to the application of the 1-dimensional WL test with the specified number of iterations (*i.e.*, 5 is the default value used). These are then used to construct the Gram matrix of the dataset. As before, this matrix represents an implicit version of the kernel and is used to build an SVM classifier and classify new samples. For this classifier, we use the scikit-learn library version 1.0.2 and the GraKeL library [Sig+20] version 0.1a8.

3.5 Experimental Results

We conduct experiments to compare exploration techniques, SCDG strategies, and classifier performance which are presented below.

Our evaluation set consists of 1792 malware samples, divided into 15 families, plus 150 cleanware samples. The exact composition of the dataset is given in Table 3.1. In terms of origin, 64 percent of the samples were obtained through direct collaboration with Cisco. The remaining 36 percent were extracted from MalwareBazaar [Abu23]. The samples were tagged using AVClass [Seb+16], a Python tool that takes advantage of the VirusTotal report to tag malware samples. This tool is fed with VirusTotal reports and outputs the most likely family of each sample. To evaluate detection performance, we used 150 open source programs found online [Séb19]. We only consider

Family	#samples		Family	# samples
bancteain	91		remcosRAT	446
delf	78		sfone	32
fickerstealer	85		sillyp2p	269
gandcrab	92		simbot	126
ircbot	36		sodinokibi	50
lamer	170		sytro	77
nitol	71		wabot	134
redlinestealer	35		cleanware	150

Table 3.1: Composition of the dataset

malware families that *detonate*: *i.e.*, we observed at least 50 API calls [Küc+21] during their symbolic execution.

In the rest of this section, all experiments were performed on a desktop PC with an Intel Core i7-8665U CPU (1.90GHz x 8) and 16GB RAM running Ubuntu 18.04.5. Our experimental results are based on our ability to efficiently extract SCDGs. In all experiments, we used a timeout of **ten minutes** for each SCDG. Note that 20 percent of the SCDGs were computed on time, while 80 percent were never fully computed. For the case of BFS-Strategy, we used the same parameters as in [Seb+20] (loop threshold of 4, unlimited number of states to explore, z3 optimization enabled). However, for CDFS-Strategy and CBFS-Strategy, we imposed a limit of 10 states that could be explored simultaneously. All metrics for multi-class classification are weighted averages.

The SCDGs are constructed using the various exploration strategies defined in Section 2.3 and the SCDG strategies defined in Section 2.4. As a reminder, **SCDG-strategy 1-2-3-4-5** corresponds to the one edge strategy, the multiple edges strategy, the merge call strategy, the merge trace strategy, and the disjont union strategy, respectively. The other SEMA parameters are set to their default values, which are presented in Section 3.3. After evaluating our classifier with these different strategies, we compare it to an SVM classifier trained on Ember's static features [AR18] for malware detection. These features consist of *byte/opcode N-grams*, *strings*, and *PE file characteristics*. We then illustrate how easily it could be fooled with adversarial examples generated with frameworks such as MAB-malware [Son+22], while our approach exhibits better accuracy with robustness against these attacks.

gSpan We first apply gSpan to SCDGs obtained using combinations of different strategies and the default parameters presented in Section 3.4. Signatures are obtained by randomly sampling 60% of the SCDGs of each family; these SCDGs constitute the training set. Other SCDGs are then classified to evaluate the quality of these signatures; these SCDGs constitute the test set. The

same process is repeated three times for each classifier and the performance is averaged.

SCDG-strategy					BFS-Strategy			CBFS-Strategy			CDFS-Strategy		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.684	0.651	0.667	0.736	0.655	0.693	0.693	0.639	0.664
x		x		x	0.614	0.594	0.614	0.686	0.615	0.648	0.645	0.586	0.614
x			x		0.679	0.587	0.629	0.623	0.465	0.532	0.68	0.597	0.636
x				x	0.602	0.556	0.578	0.587	0.448	0.508	0.632	0.578	0.603
	x	x	x		0.685	0.566	0.619	0.721	0.604	0.657	0.652	0.609	0.629
	x	x		x	0.623	0.552	0.585	0.674	0.568	0.616	0.61	0.587	0.598
	x		x		0.683	0.589	0.632	0.698	0.555	0.619	0.669	0.597	0.631
	x			x	0.609	0.534	0.569	0.651	0.534	0.586	0.629	0.54	0.581

Table 3.2: Results of gSpan classifier with different exploration and SCDG strategies

In Table 3.2, we observe that **CBFS-Strategy** generally outperforms **CDFS-Strategy** and **BFS-Strategy**. While inspecting the results, we noticed that **BFS-Strategy** ran out of memory on 7 percent of the binaries, reducing its performance compared to **CBFS-Strategy**. While **SCDG-strategy 4** showed improvements over **SCDG-strategy 5**, it should be noted that **SCDG-strategy 5** has significant overhead in SCDG creation (up to 100 times slower) and signature size (5 times larger on average). In general, the best performance was achieved by combining **SCDG-strategy {1,3,4}**. Looking at the best classifier in Figure 3.4, we see a lot of confusion between different classes. This can be explained by plotting the similarities between signatures built with gSpan, as shown in Figure 3.3.

There, different signatures share important similarities, leading to confusion between different malware families, as shown in Figure 3.4. This problem is directly related to the problem exposed in Section 2.5, namely that gSpan focuses on the largest subgraph while neglecting other components.

Kernel from [Puo+21] We now turn to the application of the kernel from [Puo+21]. Table 3.3 shows that the overall performance has increased compared to gSpan. In addition, **CBFS-Strategy** and **CDFS-Strategy** outperformed both **BFS-Strategy** and **SCDG-strategy {1,3,4}** and appeared to be more efficient. However, Figure 3.5 shows that several families were still indistinguishable.

Weisfeiler-Lehman Kernel (WL) Finally, we examined the SVM classifier using the WL kernel. The results in Table 3.4 clearly outperformed the others, achieving a F_1 score of 0.929 with **CDFS-Strategy** and **SCDG-strategy {2,3,4}**. Families were better discriminated, as shown in Figure 3.6. These results confirm our assumption in Section 2.5: exploiting the local structure of an SCDG increases the efficiency of machine learning in malware classification.

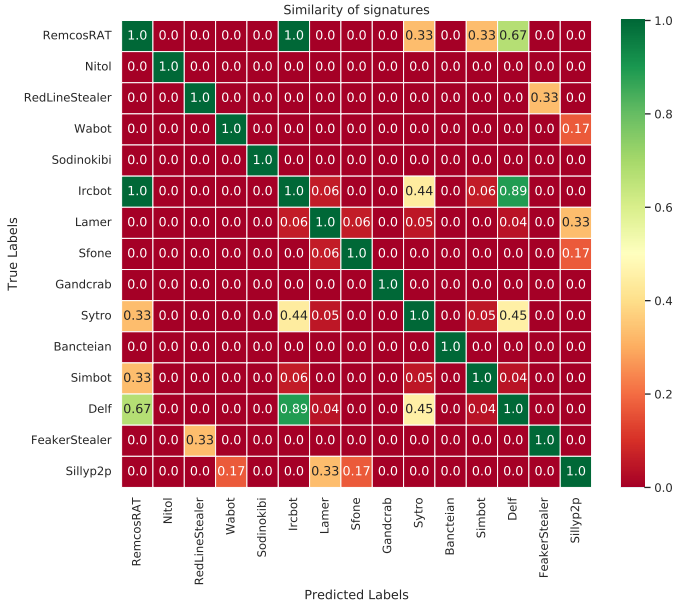


Figure 3.3: Similarity matrix between signatures obtained with gSpan

SCDG-strategy					BFS-Strategy			CBFS-Strategy			CDFS-Strategy		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.769	0.723	0.745	0.851	0.826	0.838	0.847	0.813	0.829
x		x		x	0.738	0.645	0.688	0.813	0.752	0.781	0.802	0.738	0.768
x			x		0.747	0.632	0.684	0.798	0.747	0.771	0.835	0.772	0.802
x				x	0.714	0.654	0.682	0.781	0.733	0.756	0.763	0.718	0.739
	x	x	x		0.754	0.691	0.721	0.845	0.812	0.828	0.833	0.787	0.8093
	x	x		x	0.728	0.673	0.699	0.785	0.736	0.759	0.77	0.714	0.741
	x		x		0.742	0.71	0.725	0.827	0.778	0.801	0.82	0.754	0.786
	x			x	0.711	0.649	0.678	0.771	0.703	0.735	0.74	0.698	0.718

Table 3.3: Results of SVM classifier and kernel of [Puo+21] with different exploration and SCDG strategies

SCDG-strategy					BFS-Strategy			CBFS-Strategy			CDFS-Strategy		
1	2	3	4	5	Precision	Recall	F_1-score	Precision	Recall	F_1-score	Precision	Recall	F_1-score
x		x	x		0.86	0.855	0.857	0.897	0.879	0.867	0.929	0.925	0.924
x		x		x	0.812	0.795	0.803	0.867	0.862	0.864	0.885	0.877	0.881
x			x		0.895	0.891	0.891	0.895	0.891	0.886	0.939	0.933	**0.929**
x				x	0.834	0.828	0.831	0.862	0.858	0.859	0.891	0.887	0.888
	x	x	x		0.852	0.847	0.846	0.865	0.864	0.852	0.936	0.931	**0.929**
	x	x		x	0.832	0.824	0.827	0.85	0.842	0.846	0.915	0.91	0.912
	x		x		0.895	0.891	0.892	0.894	0.881	0.874	0.937	0.933	**0.929**
	x			x	0.847	0.836	0.841	0.86	0.851	0.855	0.918	0.911	0.914

Table 3.4: Results of SVM and WL kernel with different exploration and SCDG strategies

Malware Detection and Adversarial Examples We then evaluate the performance of our classifier for binary classification: malware or cleanware

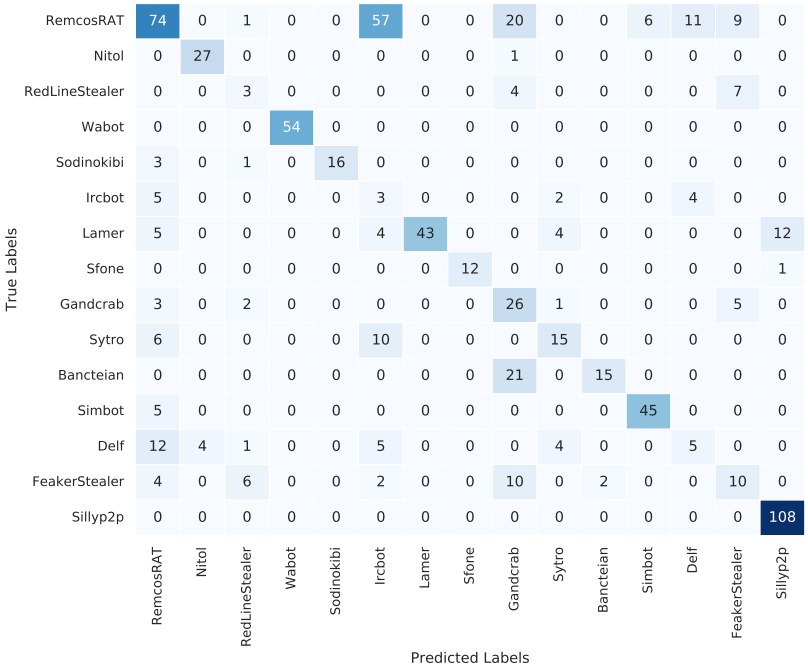


Figure 3.4: Confusion matrix obtained for gSpan with CBFS-strategy and SCDG-strategy {1,3,4}.

and compare it to a classifier based on static features. For each classifier, we select the previously best combination of strategies. gSpan is used with **CBFS**, **SCDG-strategy {1,3,4}** and a similarity threshold of 0.5 (to decide whether the sample is similar to a signature of malware family or not). The kernel from [Puo+21] is used with **CBFS-strategy** and **SCDG-strategy {1,3,4}**. WL kernel is used with **CDFS** and **SCDG-strategy {2,3,4}**. As expected, our classifiers are robust against adversarial examples based on static features. This is illustrated in Table 3.5, where we show that the MAB malware adversarial framework [Son+22] can easily corrupt Ember features [AR18] used in most static classifiers. As already highlighted in the literature, although these static classifiers provide good performance, they are highly vulnerable and there is a need to develop new approaches. Unfortunately, there is also a lack of framework to generate adversarial attacks on classifiers based on binary execution which could be more interesting to evaluate our models [Lin+23; Sha+22].

Training time. In general, the WL kernel outperforms gSpan by a factor of 15 and the kernel in [Puo+21] by a factor of 10,000 regarding the training time. We suspect that the overhead is due to the extensive use of pairwise

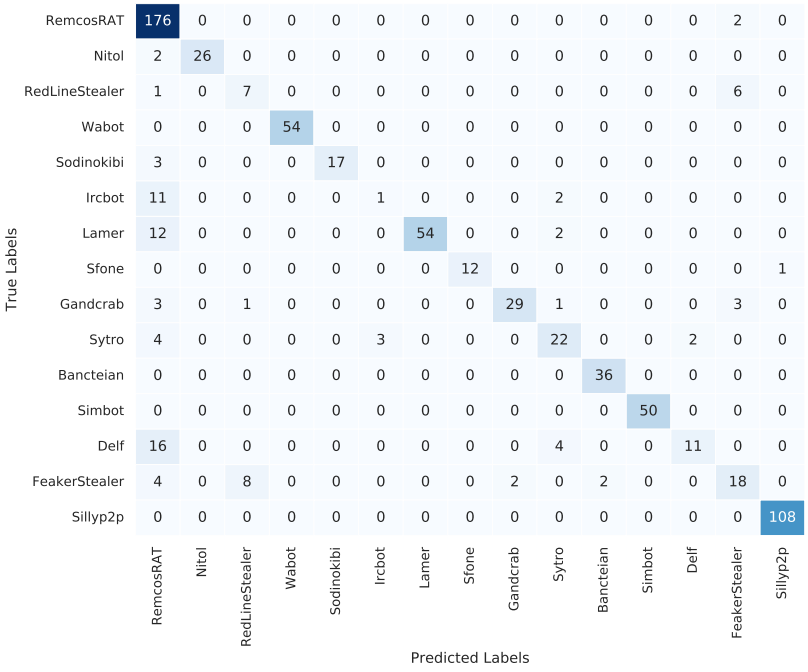


Figure 3.5: Confusion matrix obtained for SVM classifier and kernel from [Puo+21] with CBFS-strategy and SCDG-strategy {1,3,4}

Model	Initial dataset			Mutated dataset		
	Precision	Recall	F_1-score	Precision	Recall	F_1-score
gSpan	0.911	0.914	0.911	0.911	0.914	0.911
SVM kernel from [Puo+21]	0.965	0.95	0.957	0.965	0.95	0.957
SVM Weisfeiler-Lehman	0.989	0.975	0.981	0.989	0.975	0.981
SVM Ember features	0.964	0.965	0.964	0.763	0.564	0.502

Table 3.5: Comparison in the context of adversarial examples (mutated dataset)

graph mining in the similarity metric presented in Section 2.5. Compared to the kernel in [Puo+21], gSpan reduces this number of computations by first creating a signature for each family before comparing that signature to the binary to classify.

Our cleanware dataset consists of open source software, which might have specificities compared to other cleanware datasets. Moreover, this dataset is small and the metrics may not reflect the reality as explained by Arp et al. [Arp+22]. Therefore, we carefully checked the confusion matrix for the detection task and it confirmed that cleanware were correctly distinguished from malware (*i.e.*, only a few cleanware were misclassified). Finally, we do not take into account the release date of the samples when dividing our

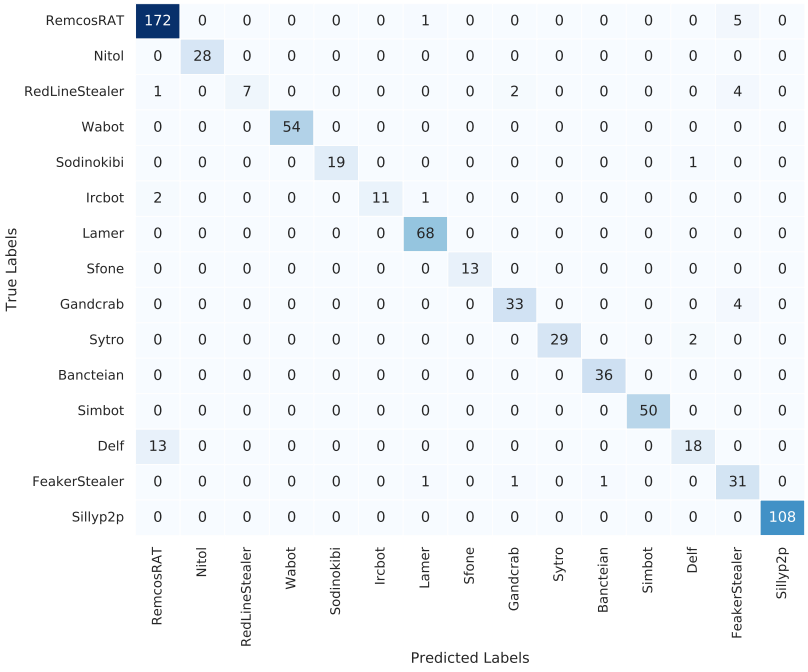


Figure 3.6: Confusion matrix obtained for WL kernel with CDFS-strategy and SCDG-strategy {2,3,4}.

dataset into training and test sets. Therefore, the observed performance may be slightly biased compared to a real-world deployment.

Several conclusions can be drawn from these experiments. First, **SCDG-strategy 2** gives overall better results than **SCDG-strategy 1** with WL kernel. This is not the case for the other classifiers where this information seems to lead to over-fitting and **SCDG-strategy 1** should be preferred. Furthermore, the effect of **SCDG-strategy 3** is mixed. While it improves classification for classifiers based on the largest common subgraph, its effect varies when combined with other strategies. Finally, while **SCDG-strategy 5** results in significant overhead, it does not improve the performance of any classifier. On the other hand, **SCDG-strategy 4** leads to more compact signatures, better computation times, and good classification performance. Regarding exploration strategies, **BFS-strategy** is generally outperformed by **CBFS-strategy**, while **CDFS-strategy** outperforms other strategies when used with the WL graph kernel. There is a dependency between the construction of the malware behavioral abstraction (*i.e.*, exploration and SCDG strategies) and the efficiency of the classifier. Therefore, the specifics of the classifier should be considered when constructing the malware representation. While gSpan showed poor performance for some malware families, the algorithm

build signatures that are more interpretable compared to other classifiers. This quality should not be neglected since it allows analysts to directly get knowledge about the behavior of the analyzed malware family and possible bias in their dataset [Arp+22] when signatures do not correspond to a malicious behavior.

3.6 Conclusion and Future Work

This chapter introduces SEMA, an open source toolchain for malware analysis using symbolic execution and machine learning. It evaluates different heuristics for exploring a binary, building an abstraction of its behavior, and training classifiers. The proposed exploration strategies and the Weisfeiler-Lehman kernel outperform similar approaches based on symbolic execution from the literature [Sai+18; Seb+20; Puo+21]. Our experiments highlight the need to adapt the construction of the malware representation (*i.e.*, the SCDG) to the classifier used. SEMA is easily adaptable to new strategies for symbolic analysis or classification. While *Simprocedures* will be continuously added to improve environment modeling, our future work ideas include:

New exploration heuristics As explained in Chapter 1, recent research has proposed to guide the exploration of the binary thanks to machine learning [Rua+21; He+21]. These approaches are based on several features related to instructions visited, properties of the current path, number of system calls visited, properties of the CFG,... We could design new exploration strategies for malware analysis based on similar techniques and on heuristics proposed in the literature to identify triggering conditions [Bru+08; Par+21].

Interleaving Symbolic Execution The lack of a proper environment model is generally a major problem that hinders the use of symbolic execution on real binaries. As suggested by Gritti et al. [Gri+20], interleaving symbolic execution with concrete execution could overcome this challenge by avoiding modeling part of the environment for the symbolic execution engine. They propose to concretely execute parts of the code that are not relevant to the analysis, generating an execution state that could then be used when switching to symbolic execution. Adapting their approach to SEMA could greatly improve the efficiency of our analysis and allow us to manage techniques that are currently outside of the scope of SEMA (*e.g.*, process injection).

Comparison with other approaches Our experiments compared SEMA with a machine learning classifier based on Ember features. As discussed in Section 1.4, there is a wide range of other features that could be used to

build machine learning classifiers. Different works [NP17; AZE21; Dam+23] proposed to use API calls extracted by dynamic analysis to build malware detectors. We could construct SCDGs based on these extracted API calls to feed our classifiers. This could allow us to study the different performance trade-offs between classifiers based on static, dynamic and symbolic analysis.

Unsupervised Learning Our approach currently requires us to know *apriori* the malware families we want to detect. As suggested in the literature [DGL21; Puo+21], we could adapt SEMA to identify new malware families thanks to unsupervised learning algorithms such as OPTICS [Ank+99] or DBSCAN [Est+96]. Similar to supervised learning, the WL kernel could improve the performance of existing unsupervised approaches and allow us to avoid relying on labeling tools to build our models.

Robustness While it is well known that it is easy to design adversarial examples against models based on static features [Lin+23; Dem+21], models based on execution traces are generally more robust. Nevertheless, recent research [Sha+22; Arp+22] highlights the need to evaluate attacks on these models. We could imagine an attacker trying to bypass our classifiers by modifying the SCDG structure: *e.g.*, inserting/replacing executed API calls to modify the SCDG and confuse the classifier. In this context, it could be helpful to consider gSpan, since this classifier could be robust against different attacks than the WL kernel. Moreover, its interpretability could help to understand how the adversarial attack affects its signature, while the same is more complex to study for the other classifiers. Another attack surface is the symbolic execution engine: *e.g.*, the path explosion could be exploited by a malware to hide its malicious behavior (*e.g.*: thanks to junk code or opaque predicates [Min+15]). SEMA should be improved and evaluated against these attacks in future work.

Packing detection: Feature Relevance and Classifiers Evaluation

4

In this chapter, we are investigating the problem of packing detection, which is a common obfuscation technique used by malware. First we present existing approaches to packing detection and the life cycle of a packed program. Then, we propose to study different machine learning classifiers for packing detection. In particular, we focus on the feature relevance, effectiveness, and robustness of the classifiers, and compare the resources they require.

4.1 Introduction

Packing is a widely used technique that can evade or interfere with malware analysis. It works through various methods such as compressing or encrypting a program's sections, encoding a program in a virtual machine, or fragmenting a program's behavior. These packing methods complicate malware analysis because they obfuscate static features (*i.e.*, features extracted without running the binary) and make the behavior of the sample observed by dynamic analysis more complex and difficult to analyze. While benign programs typically use packing for compression or to protect intellectual property, VirusTotal [Vira] states that most new malware samples are merely re-packed versions of previously seen malware.

An effective malware detection or classification engine must be able to detect whether a program under analysis is packed. A packed program can be unpacked or treated with greater suspicion to improve analysis and response. Combined with a fast and efficient static malware detector, such as the one proposed by Baldangombo et al. [BJH13], a complete malware detection engine can provide high-end performance malware detection and classification. Regarding SEMA, we observed that some packers could drastically affect the efficiency of symbolic execution (*e.g.*, UPX could slow down the analysis of a simple "hello world" binary by 15 minutes). We suspect that this overhead is related to self-modifying code, which drastically affects the efficiency of the lifting process in angr. Therefore, we need to properly detect packed binaries in order to handle them differently and improve our malware analysis

workflow.

Syntactic signatures are a popular approach for packing detection because they are easy to compute and can achieve high accuracy with known packers. For example, to build signatures, YARA [Virb] typically considers simple static features such as byte sequences, strings, or file features such as file size or hash. Although such signatures are effective against known packers, they can be easily bypassed by new packers [Bio+18]. Using entropy is also effective in packing detectors, since packed files *often* have higher entropy than unpacked files. Lyda et al. [LH07] were the first to address the problem of packing detection using entropy in an academic publication. Although studies like [HLL09] improved this detection method, [Uga+12] shows that inserting a selected set of bytes into an executable maintains a low entropy. Finally, several works have proposed heuristic-based techniques to build their detectors. For example, Choi et al. [Cho+08] proposed PHAD, a packing detector based on eight features, including the number of executable and writable sections, the number of section names that are not printable, the sum of the size of each section compared to the file size, or the executable permission of the entry point section. Treadwell et al. [TZ09] highlight the importance of the IAT table, the TLS section, and specific section names in their detector. This wide range of features, statically extracted at low computational cost, is widely used as the basis for popular antiviruses such as ClamAV [Cla] or tools such as PEiD [PEI].

To improve packing detection, different works have used machine learning techniques involving supervised learning on a variety of statically extracted features. Perdisci et al. [PLL08a] trained different models (*e.g.*, multilayer perceptron, naive Bayes model, etc.) on nine features related to section properties, IAT table, and entropy. Ugarte-Pedrero et al. [Uga+14] use the same features (adding n-grams) and trained different models and rule-based classifiers, focusing on anomaly detection. After reviewing the literature, Biondi et al. [Bio+19] chose to extract 119 features divided into six categories: Byte Entropy, Entry Bytes, Import Functions, Metadata, Resource, and Sections. They trained naive Bayes models and several tree models with different ground truths. They then investigated which models offered the best trade-off between accuracy and training cost. Finally, Mantovani et al. [Man+20] as well as Aghakhani et al. [Agh+20] highlight the limitations of current machine learning approaches. The former illustrated the difficulties of ML models in detecting low entropy packing schemes; the latter showed the overlap between packing detection and malware detection based on static features in current ML models. A detailed presentation of existing ML approaches to packing detection is presented in Appendix C.2. Recent research in unsupervised learning [Nou+21] studied how to build clusters related to packers to identify variants of known packers or new packers. This is beyond the scope of this chapter which focus

on supervised learning algorithms. Finally, as explained in [SPB22], recent studies show that machine learning models tend to learn spurious correlations from data, making some features more important than they should be. In the context of vulnerability detection, the authors suggest to use explainable AI methods to better understand the importance of the features used.

Unfortunately, there is a lack of knowledge about the impact of all the proposed features on the decision processes of machine learning models. There is no conclusive evidence as to which of the existing models is more efficient and robust. The costs associated with different models and their features are uncertain. While a recent survey [UAB19] introduces economic analyses related to malware detection, aiming to present the existing trade-offs between analysis accuracy, time and cost when deploying a malware analysis infrastructure, researchers have proposed limited analyses related to retraining costs for packing detection [Bio+19].

We chose to evaluate the feature set proposed in [Bio+19] because it gathered a wide range of features known in the literature. We extended this study by examining the importance and impact of these features using nine different classifiers with multiple strategies: variance, mutual information, Shapley values, and feature selection. We compared and contrasted the nine supervised machine learning classifiers in terms of effectiveness and resilience against adversarial attacks, and we examined the cost of deployment. Based on our work, we propose the following contributions:

- **Feature Analysis.** We use several different methods to explore the importance of 119 features in packing detection: examining variance and mutual information of the features and Shapley values. We then reduced the feature set using feature selection methods, principal component analysis, or Shapley value selection methods.

- **Effectiveness of Classifiers.** We studied the effectiveness of nine supervised machine learning classifiers applied to a large dataset of observed *in the wild* samples and several binaries we packed ourselves. We used Wilcoxon tests to ensure the significance of the comparison between the algorithms.

- **Adversarial Examples.** We generated adversarial examples against our classifiers by making simple modifications to packed binaries. This allowed us to study the robustness of the classifiers and better understand their limitations.

- **Economical analysis.** We then trained and evaluated our classifiers on different samples collected chronologically. An economical analysis (following the approach proposed in [UAB19] and the work of [Bio+19])

highlighted the trade-offs per classifier in terms of accuracy, efficiency, training time, and memory.

The chapter is organized as follows: Section 4.2 provides specific background on packing, feature importance/selection and Shapley values. Section 4.3 presents the composition of the datasets and the labels used. We then describe our methodology for examining the 119 features and the nine algorithms. We also explain the configuration we used for our ML algorithms (*i.e.*, preprocessing and hyperparameter tuning). Section 4.4 examines the relevance of different features in the decision process of the classifiers, including the calculation of Shapley values, feature selection methods, and principal component analysis. Section 4.5 evaluates the ML algorithms for effectiveness, relevant limitations, and robustness against simple adversarial attacks. Section 4.6 explores economic trade-offs for the different classifiers and tests their resilience using a chronological dataset. Finally, Section 4.7 discusses the validity of our results, inherent limitations, further work and concludes this chapter.

4.2 Background

This section briefly introduces packing, Shapley values, and dimensionality reduction methods.

4.2.1 Packing

A packing program, or packer, is a program that compresses/encrypts an executable, creating a new executable by bundling the original compressed executable with its decompression/decryption code. When executed, the packed executable first executes a decompression/decryption routine and then passes control to the rebuilt executable. Typical packing techniques are implemented with a small stub of the program that performs the initial decompression/decryption or runs a virtual machine to allow the program to be executed.

Figure 4.1 illustrates the life cycle of a typical packed program. First, the original program is transformed by obfuscating its contents (*e.g.*., compression, encryption, virtualization, deletion of imports, junk code, etc.) Then an unpacking stub is added that contains the routine to recover the original executable. The stub becomes the new entry point of the executable. When the program is executed, the unpacking routine allows the original executable to be reconstructed in process memory. The IAT table is rebuilt to allow the executable to call functions from DLLs, and control flow is finally passed to the original entry point of the executable.

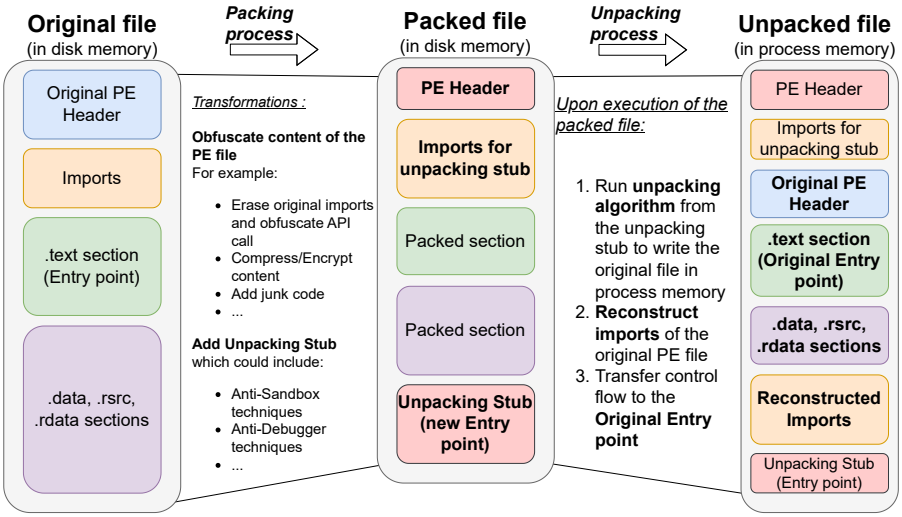


Figure 4.1: Life cycle of a packed program

Taxonomy A recent survey [Mur+22] describes four main types of packers:

- Compressors: Compressors compress sections of the original file to reduce the size of the original executable. In general, compressors do not insert anti-reversing tricks into packed files.
- Crypters: Crypters encrypt/obfuscate sections of the executable file and decrypt them when the unpacking stub is executed. This makes the file difficult to analyze statically.
- Protectors: Protectors combine the capabilities of both compressors and crypters.
- Bundlers: Bundlers combine multiple executable and data files into a single bundled executable that can access these multiple files without writing them to disk.

While packers such as compressors usefully reduce the size of an executable, adversaries can exploit these properties to conceal their code. Because packing an executable changes the file's signature and obfuscates its contents, packing can help evade static analysis. This forces analysts to unpack the executable if they want to use static analysis tools. Anti-unpacking techniques incorporated by the packer can also hinder dynamic analysis tools (anti-debugging [GTG07], anti-emulation [Fer08], etc.) and reduce the effectiveness of malware detectors. In [Uga+15], the authors discuss the complexity of different packers, ranging from I to VI. While the simplest packers (*i.e.*, Type

I, such as UPX) execute a single unpacking routine before passing control flow to the unpacked executable, other packers may contain multiple unpacking layers organized in a complex topology (*i.e.*, Type III, such as UPolyX). Finally, the most complex packers can unpack a single fragment of the original executable at a time (*i.e.*, Type VI, such as Armadillo).

While this chapter focuses on a single packer used on an executable, multiple packers are often used on the same executable. One goal of this "multi-stage packing" is to evade signature-based detection. Malware is more difficult to detect when a known packer is used to achieve a high compression ratio in conjunction with a custom packer that is unknown to the antivirus software. To illustrate the diversity of packers, we will now describe three packers with different capabilities that are present in our datasets.

UPX The Ultimate Packer for eXecutables (UPX) is an open source packer that supports Windows programs/DLLs, DOS programs, and Linux executables/kernels. This packer works by adding three sections to an executable file: **UPX0**, an empty section that acts as an area where the decompressed code will go; **UPX1**, the entry point section where the decompression code will go; and **UPX2** or .rsrc, which contains resources such as IAT. When an UPX-packed executable is executed, the content of the UPX1 (the entry point) section is executed first to decompress the original program into the UPX0 section. The execution flow then jumps to the UPX0 section and executes the original program. This process uses predictable section names, so looking for sections named UPX0 or UPX1 will tell you if an executable was packed with the UPX packer.

Yoda's Protector Yoda's Protector is an open source packer that supports Windows programs/DLLs and is based on *Yoda's Crypter*. This packer has the ability to compress or encrypt each section of an executable while preserving the section's structure. It adds its unpacking stub to a new section **.yP**. It supports several protection features: anti-reversing tricks (*e.g.*, CRC check to make sure the packed executable is not modified by an analyst), removal of debug information, API obfuscation by IAT redirection (only those exported by *kernel32.dll*, *user32.dll* and *advapi32.dll*), and similar.

BobSoft Mini Delphi Written in Delphi, BobSoft Mini Delphi splits the original executable into many parts which are encrypted and stored in different resource sections. It also adds a wide range of checks to the packed executable to determine whether it is running in a monitored environment (*e.g.*, in a virtual machine). These checks include cursor movements, running processes, processor information, etc.

4.2.2 Interpretability, dimensionality reduction and feature selection

Interpretable Machine Learning To better understand and explain a model's decision, interpretable machine learning has become increasingly popular [Mol20]. Techniques in this area help to identify the impact of each feature on a model's decision process. One of the techniques, the **Shapley values**, has an interesting approach that explains the independent contribution of each feature to the outcome. In the **Shapley values** approach, each feature is assumed to be a "player" in a game where the prediction is the payout. The aim is to attribute each part of this payout (*i.e.*, the prediction) to each feature. Let S be a subset (*i.e.*, coalition) of the features used in the model, $\hat{f}(.)$ be the model to interpret, and X be a vector $< x_1, ..., x_p >$ of size p containing feature values of the instance to be explained. $val_X(S)$ is defined as the prediction for the marginalised feature values in the coalition S over features that are not in the set S :

$$val_X(S) = \int \hat{f}(x_1, ...x_p) d\mathbb{P}_{x \notin S} - E_X(\hat{f}(X)) \tag{4.1}$$

where $\mathbb{P}_{x \notin S}$ is the space of features not contained in S and $E_X(\hat{f}(X))$ is the mean outcome of the model $\hat{f}(.)$.

The Shapley value associated with feature j is its contribution to the prediction, weighted and summed over all possible combinations of features:

$$\phi_j(val_X) = \sum_{S \subseteq \{1,...,p\} \setminus j} \frac{|S|!(p - |S| - 1)!}{p!} (val_X(S \cup \{j\} - val_X(S)) \tag{4.2}$$

where p is the number of features.

Since evaluating Shapley values requires significant computation time, most real applications make use of approximations like **SHAP values**. SHAP stands for SHapley Additive exPlanations and represents the Shapley value as a linear model. More details can be found in [Mol20] for the interested reader.

Variance filtering The variance $Var(.)$ is a statistical measure describing the variability of a feature X and is defined as :

$$Var(X) = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - E(X))^2 \tag{4.3}$$

where n designates the number of data points, X_i is the feature value of data point i, and $E(X)$ is the mean value of the variable X. Examining the variance of each feature allows us to see if there are features that do not vary in the dataset (*i.e.*, the variance around 0). Filtering these features is a simple technique to discard features that add little information to the model.

Mutual Information This statistical measure $I(.;.)$ quantifies the amount of information shared by two variables. It allows us to measure the dependence between these variables using the concept of entropy (*i.e.*, the measure of the uncertainty of a variable). The mutual information between a feature X and the label of the data point Y is defined as:

$$I(X;Y) = H(X) - H(X|Y) \tag{4.4}$$

where $H(X)$ is the entropy of X, and $H(X|Y)$ is the conditional entropy of X given Y (i.e., the uncertainty of X when Y is known). Mutual information allows the discovery of non-linear relationships between variables and is often used to select features according to their relevance.

Principal Component Analysis (PCA) This algorithm reduces high-dimensional data by projecting it linearly while preserving as much data variance as possible. Given a normalised dataset $D \in \mathbb{R}^{n \times d}$, where n designates the number of data points and d describes the dimension of the data, it first computes the covariance matrix $cov(.)$ of D as follows:

$$cov(D) = \frac{1}{n-1} D^T D \tag{4.5}$$

where D^T is the transpose of the matrix D. It then computes the eigenvectors and corresponding eigenvalues of $cov(X)$. The k eigenvectors with the largest eigenvalues are selected as *principal components*. Finally, the dataset is projected onto these selected principal components.

Features selection This approach reduces the dimensionality of the machine learning input by removing irrelevant/redundant features and retaining only relevant features [Bom+20]. Reasons for its use include reducing feature extraction time, improving training time, reducing memory consumption, simplifying models, or avoiding the curse of dimensionality [BN06]. There are three categories of feature selection methods: filter, wrapper, and embedded. Filter methods do not take into account the characteristics of the machine learning models (*e.g.*, variance analysis); wrapper methods use model predictions to select features (*e.g.*, Recursive Feature Elimination), and embedded methods include feature selection into the training process (e.g., Lasso for linear models [Mol20]). On behalf of variance analysis and mutual information, we will also use two feature wrapper selection methods: Recursive Feature Elimination (RFE) recursively eliminates features of lesser importance (*i.e.*, features that have less weight in a model or features that are less present in the tree-based model). This means that we end up focusing on smaller but more relevant sets of features. The SHAP selection method [ME20] calculates

SHAP values to rank features by importance, and then trains different models by progressively considering the best features for training.

It should be noted that this feature selection allows to directly reduce the dimensionality of our dataset by removing redundant/useless features, whereas PCA allows to reduce dimensionality by creating new features.

4.3 Methodology

This section details the three phases of our experimental setup: (1) dataset construction, (2) methodology, and (3) tuning of ML algorithms.

4.3.1 Data Sets and Labeling Tools

Our datasets come from a malware feed provided by Cisco, which provides 1,000 new malware samples per day as they are found online. For this work, malware samples were collected between 2019-09-17 and 2020-07-29. For each sample included in our dataset, we extracted the 119 static features identified in [Bio+19]. Occasional errors in the extraction software require the exclusion of an insignificant number of samples; we can safely ignore these.

Samples are labeled as packed or not packed to provide ground truth for training ML classifiers. We took a similar approach to VirusTotal; we used multiple detector engines for classification and then decided based on these results. The following four classification engines were used:

- **PEframe** [Ama] is an open source tool written in Python and capable to perform static analysis of portable executables (PE), object code, DLLs, etc. We used the tool for packer detection, but it is possible to use it to obtain other information such as macros, anti-analysis techniques, etc.
- **Manalyze** [Kwi] is a static analyzer often used by large companies. Its packer detection plugin works through signatures and custom heuristics. Signatures are generally related to the names of PE sections. For example, UPX renames some sections to UPX0, UPX1, etc. after packing. Heuristics are mostly related to entropy-based features or anomalies in the PE structure, such as unusual section names or sections that are both writable and executable, etc.
- **PEiD** [PEI] is a free and popular signature-based detector. Relevant signatures contain low-level byte patterns that can match either at the entry point or anywhere within a PE file.
- **DIE** Detect-it-Easy [hor] is an open source application that allows users to write signatures in a JavaScript-like language. A large database of signatures is available in the github repository.

When classifying a sample, a key metric is the number of detectors that label the sample as packed, and the threshold of detectors to consider the sample as packed. In [Bio+19], the authors build different ground truths based on existing packing detectors. Their first ground truth uses the threshold of one detector, and their second ground truth uses the threshold of three detectors. They find that a ground truth based on a large consensus (three detectors) reduces the number of packing families and improves model performance in a laboratory setting. However, this method does not generalize well when applied to new packers. The ground truth constructed using a threshold of one detector proves to be more flexible and allows for a better packer diversity in the dataset. Therefore, our dataset is labeled with a ground truth constructed using a one-detector threshold. While our experiments typically require identification of whether a sample is packed or not, we also need to know which packer is used to investigate false negatives. If there is a conflict between the labels proposed by our detector, we manually investigate the false negative to identify its label.

For our experimental results, we first use the Big dataset with 168,410 samples. This dataset consists of samples collected from 2019-09-17 to 2020-04-30 and has 94,621 packed samples and 73,789 not packed samples according to our labeling. The distribution of packer detections per each detector is shown in Table 4.1 and the malware families detected by AVClass [Seb+16] are shown in Table 4.3. We observe that UPX is the most common packer detected by our detectors. It is important to note that there are several versions of UPX (different released versions but also modified versions since it is open source). UPX is also often used in combination with other packers [Coz+18; Uga+15] to take advantage of its compression capabilities. Therefore, the high occurrence of UPX in the table corresponds to a complex reality.

We use a second dataset Chrono with 45,795 samples for an economical analysis of our classifiers. We compare long-term accuracy, memory, and time required for training/testing. This dataset consists of samples collected from 2020-05-01 to 2020-07-29 and, according to the labeling, has 22,071 packed samples and 23,724 not packed samples. The dataset is divided into four subsets, each corresponding to a separate time period: Chrono 1 designates malware obtained from 2020-05-01 to 2020-05-14. Chrono 2 designates malware obtained from 2020-05-15 to 2020-05-28. Chrono 3 designates malware obtained from 2020-07-01 to 2020-07-13. Chrono 4 designates malware obtained from 2020-07-14 to 2020-07-29. The distribution of packer detections can be seen in Table 4.2, and Table 4.3 shows the malware families detected by AVClass. As with the Big dataset, half of the samples are not detected as belonging to any particular family.

Packers	PEiD	DIE	Manalyze	PEframe	Packers	PEiD	DIE	Manalyze	PEframe
ah	2928	0	0	2905	nullsoft	1863	0	0	1859
armadillo	65	0	11	1036	obsidium	1	0	1	20
ascrypt	0	0	17	0	packman	224	224	0	0
aspack	3560	5717	3990	3508	pe diminisher	63	0	0	0
asprotect	75	0	9	564	pecompact	1827	942	870	975
bobsoftminidelphi	1270	0	0	0	peninja	0	0	380	0
crypto-lock	72	0	0	0	pestuboep	0	0	3789	686
d1s1g	0	0	195	0	petite	1	0	1	1
dxpack	0	6202	0	0	pklite	8	17	7	9
enigma	66	0	39	0	polyene	0	0	76	0
fsg	166	68	63	246	protect shareware	0	0	55	0
inno setup module	11	0	0	0	safeguard	248	0	0	234
kbys	19	20	54	0	ste@lth pe	104	0	0	94
kkrunchy	271	0	173	259	telock	807	0	645	903
mew	1	3	0	3	themida	1	0	1	2
molebox	3	0	13	0	upack	125	152	27	124
mpress	0	2388	0	1179	upolyx	0	0	760	0
neolite	139	138	0	5	upx	15134	70665	23685	14959
nspack	21	58	15	26	yoda's protector	0	0	0	1472

Table 4.1: Detection made by the different packing detectors on data set Big.

Packers	PEiD	DIE	Manalyze	PEframe	Packers	PEiD	DIE	Manalyze	PEframe
ah	1477	0	0	1443	nullsoft	211	0	0	210
armadillo	3	0	3	125	obsidium	0	0	0	31
ascrypt	0	0	3	0	packman	23	23	0	0
aspack	946	1100	1081	945	pe diminisher	9	0	0	0
asprotect	5	0	0	59	pecompact	166	42	29	43
bobsoftminidelphi	405	0	0	0	peninja	0	0	37	0
crypto-lock	16	0	0	0	pestuboep	0	0	175	118
d1s1g	0	0	16	0	petite	0	0	1	0
dxpack	0	412	0	0	pklite	1	2	0	0
enigma	3	0	1	0	polyene	0	0	22	0
fsg	77	1	2	36	protect shareware	0	0	17	0
inno setup module	5	0	0	0	safeguard	34	0	0	39
kbys	0	0	0	0	ste@lth pe	81	0	0	166
kkrunchy	24	0	34	24	telock	605	0	376	604
mew	0	0	0	0	themida	0	0	0	0
molebox	1	0	8	0	upack	42	44	7	42
mpress	0	363	0	129	upolyx	0	0	192	0
neolite	0	0	0	3	upx	3675	13762	3093	3767
nspack	1	2	1	2	yoda's protector	0	0	0	654

Table 4.2: Detection made by the different packing detectors on data set Chrono.

4.3.2 Experimental methodology

All ML algorithm implementations used are from the scikit-learn library [Ped+11] version 0.23.2. Namely, we used KNN, GNBC, LR, LSVM, DT, RF, GBDT, MLP and KSVM. All experiments are performed on a desktop PC with an Intel Core i7-8665U CPU (1.90GHz x 8) and 16GB RAM running Ubuntu 18.04.5. Our methodology is as follows.

First, we investigate which features are more significant for each ML algorithm in Section 4.4. To this purpose, we use different approaches, in

Big	#	Chrono 1	#	Chrono 2	#	Chrono 3	#	Chrono 4	#
dinwod	5833	virlock	1498	virlock	1633	mintluks	1164	mintluks	700
razy	4632	zusy	657	zusy	872	virlock	491	virlock	465
mansabo	4316	upatre	497	berbew	593	berbew	340	mansabo	411
sytro	3079	berbew	393	delf	396	mansabo	285	berbew	286
berbew	3023	delf	385	wapomi	359	delf	265	wabot	271
upatre	3001	genkryptik	379	sillywnse	270	wapomi	255	zusy	252
eati	2644	wapomi	313	genkryptik	255	zusy	220	sfone	248
banload	1987	downloadguide	290	blackmon	193	upatre	194	drolnux	225
ipamor	1976	dinwod	171	upatre	181	stihat	185	upantix	212
wacatac	1231	sytro	133	dinwod	153	sfone	184	upatre	168

Table 4.3: Top malware families recognized in Big data set and Chrono data set according to AVClass.

particular we look at variance, mutual information and Shapley values. We then reduce the dimensionality of the data using K-best feature selection, SHAP selection method and principal components analysis. These approaches were applied to all nine ML algorithms using the Big dataset. The Big dataset is split using 70% of the data for training and 30% for experimental evaluation of the model after hyperparameter tuning and dimensionality reduction. The split is stratified to ensure an equal proportion of packed and unpacked samples between the training and test sets.

We then evaluate the effectiveness of all ML algorithms with different metrics on the dataset Big in Section 4.5. We use the hyperparameters and dimensionality reduction obtained at the beginning and the same dataset distribution (*i.e.*, the same 70% of data for training and 30% for testing). The significance of these results is assessed using Wilcoxon tests. We use the one-sided Wilcoxon rank sum test available in the scientific computing library SciPy[Vir+20] and take the significance level $\alpha = 0.05$ which is a common value used in the literature [Wil45]. We then examine false positives and false negatives to better understand the efficiency of the model, as well as the accuracy on binaries that we pack ourselves. Next, we evaluate the robustness of the nine ML algorithms against adversarial examples.

Lastly, we train the tuned ML algorithms on the Big dataset and assess their economical and temporal performance in Section 4.6. We test the classifiers on the Chrono dataset, which we previously divided into small temporal periods (four periods in total, each comprising two weeks). This approach evaluates the effectiveness of the ML algorithms over time and relates this information to training costs. The experiments we conduct use datasets from different time periods, so we use the entire early dataset for training.

4.3.3 Hyper-parameters Tuning

Each of the nine ML algorithms has unique hyperparameters that can be tuned to improve performance on different datasets and classification problems. We test different values to find the best combination of hyperparameters. We define the best combination by accuracy, with equal accuracy results determined by training time. We test each combination using 70% of the Big dataset for training and apply 10-fold cross-validation with this training set.

The tested hyperparameters for each ML algorithm are listed below with their final values.

- KNN: Number of neighbors $[1; 30]$. Selected value : 16.
- DT: Criteria to measure the quality of the split (entropy or gini impurity), the minimum number of samples required for a leaf $[2; 12]$, and the maximum depth of the tree $[3; 12]$. Selected values: gini as a criterion to measure the quality of the split, 10 as the minimum number of samples required for a leaf, and 8 as the maximum depth of the tree.
- RF and GBDT: Number of estimators $[2; 50]$ (RF) and (GBDT), we use the best parameters of DT for a single tree. Selected value: 20 trees as estimators for both algorithms.
- MLP: Architecture of the network (*i.e.*, $[1; 3]$ hidden layers and 25, 50, 100 neurons), activation functions (identity, hyperbolic tangent, logistic, rectified), and solver (adam, sgd, or lbfgs). Selected values: 50 neurons in the first layer and 100 in the second layer for architecture; stochastic descent and logistic for solver and activation function.
- KSVM: Kernel used (linear, polynomial, rbf, or sigmoid). Selected kernel: radial basis functions (rbf).

Note that GNBC, LR, and LSVM have no hyperparameters affecting accuracy in the context of our problem.

4.4 Features Analysis

This section examines the relevance and significance of the 119 different features gathered in [Bio+19]. Initially, Biondi et al. grouped these features into six categories (Metadata, Section, Entropy, Entry bytes, Import Functions and Resource features) and studied the impact of each group. In this section, we propose to analyze the features individually to better understand their influence on the classifiers as proposed in the literature [SPB22]. A complete list of the features and their categories can be found in Appendix C.1.

We measure the relevance and significance of the features using different approaches: variance analysis, mutual information analysis, and SHAP values. Then, we applied feature selection and dimensionality reduction methods to reduce the set of features.

4.4.1 First insights

Variance Analysis We first performed a simple variance analysis on the 119 features to explore the relevance of each feature. The goal of this analysis was to find features with low variance (*i.e.*, features that generally keep the same value). We observed nine features with very low variance (less than 0.1). These included **DLL2**, **DLL4**, **DLL5**, **DLL6**, **DLL7**, **CODE_NOT_X**, **EP_NOT_X**, **ADDR_NOT_ALIGN** and **DBG_DIR**. We might expect these features to have little impact on the model's decision and to be discarded by feature selection methods.

Mutual Information Analysis To further explore the feature space, we computed the mutual information between each feature and the target label. We observed the ten features with the largest mutual information include **ENT_HEADER**, **ENT_EP_SEC**, **RATIO_SEC_EP**, and **RATIO_MAL_API** as well as features related to specific **BYTE**. The ten worst features found were the same as those identified by variance analysis, as well as **STACKC_SIZE**, **ALIGN** and **X_NOT_CODE**.

4.4.2 SHAP values

As discussed in Section 1.3.1, SHAP values can help us better interpret model decisions after the training phase. To examine the different classifiers, we randomly select 1000 samples from each class and plot their SHAP values in Figure 4.2-4.10, which shows the 20 most influential features. Each point corresponds to a sample, and the color indicates whether the feature has a high value (red) or a low value (blue). Points to the right indicate that the value of the feature tends to classify the sample as packed while points to the left indicate the opposite.

From these figures, we can identify the more influential features:

- **STAND_SEC** feature corresponds to *the number of standard sections in the binary.* As explained in Section 4.2.1, some packers rename existing sections or create new ones. Therefore, in all classifiers, the presence of a small number of standard sections pushes the model to classify the binary as packed.
- **EP_NOT_CODE** corresponds to *entry point not in code section* while **EP_NOT_STD** corresponds to *entry point not in standard section.* This

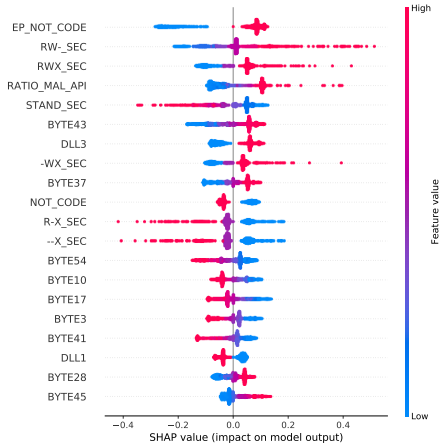


Figure 4.2: SHAP values for the LR classifier. Only the 20 most important features are represented.

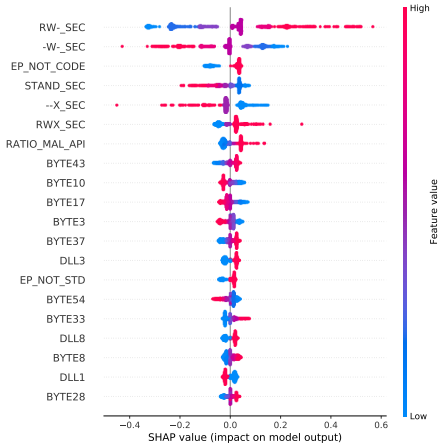


Figure 4.3: SHAP values for the LSVM classifier. Only the 20 most important features are represented.

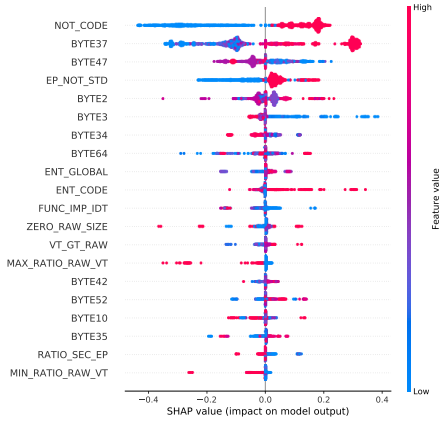


Figure 4.4: SHAP values for the DT classifier. Only the 20 most important features are represented.

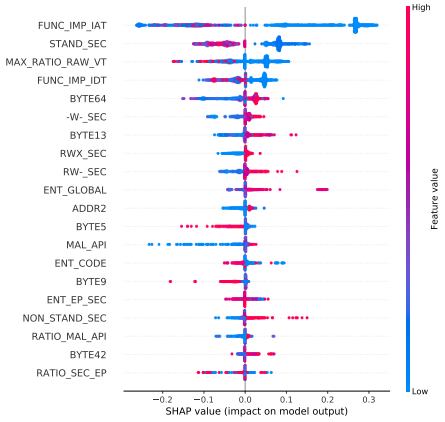


Figure 4.5: SHAP values for the GBDT classifier. Only the 20 most important features are represented.

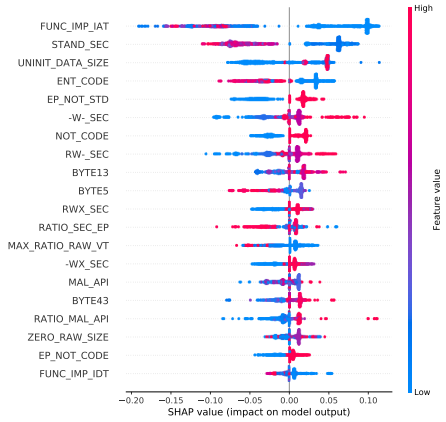


Figure 4.6: SHAP values for the RF classifier. Only the 20 most important features are represented.

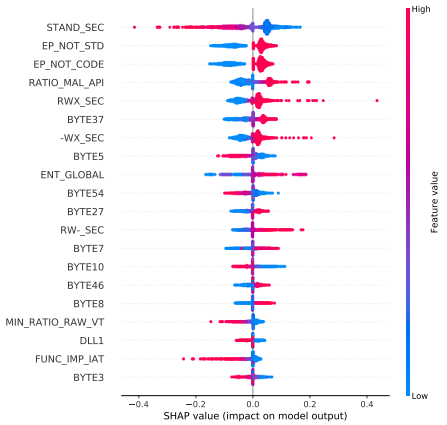


Figure 4.7: SHAP values for the MLP classifier. Only the 20 most important features are represented.

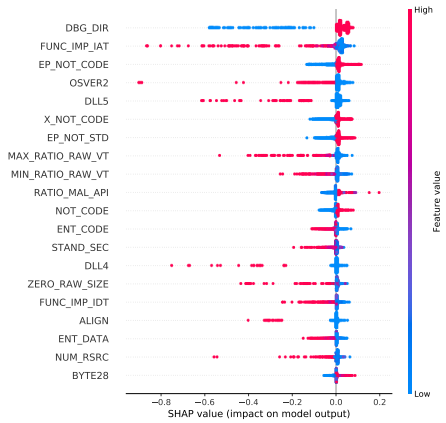


Figure 4.8: SHAP values for the GNBC classifier. Only the 20 most important features are represented.

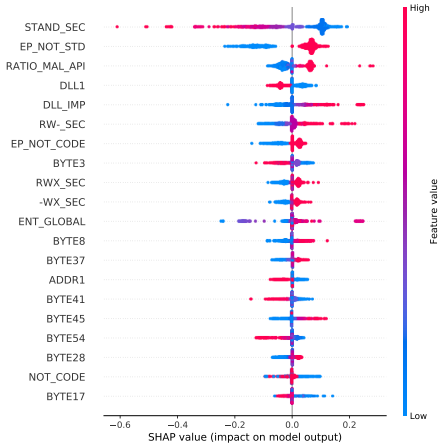


Figure 4.9: SHAP values for the KSVM classifier. Only the 20 most important features are represented.

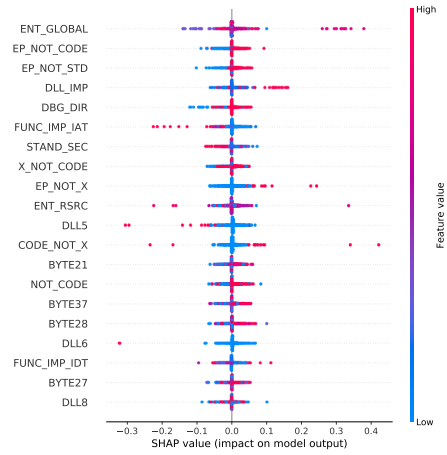


Figure 4.10: SHAP values for the KNN classifier. Only the 20 most important features are represented.

can happen if the entry point corresponding to the unpacking stub is in a dedicated section.

- **RATIO_MAL_API** is related to *the number of API calls imported* by the binary and *generally found in malicious software* according to [ZFA15]. More precisely, it represents the ratio of the number of these "malicious" API calls imported to the total number of API calls imported. These API calls are not malicious in themselves, but allow for easy modification or obfuscation of the control flow of a running program. This is achieved by allocating memory to store unpacked files, getting the address of certain API function calls after unpacking, etc. As expected, a higher ratio means a higher chance of a packed binary.

- **NOT_CODE** corresponds to the *absence of code sections in the PE file*. This is closely related to non-standard sections. Some packers change the structure of the PE file by creating new sections that replace/embed classic code sections.

- **RWX_SEC** and **-WX_SEC** features correspond to *number of Readable, Writable, and Executable sections* and *number of Writable and Executable sections*, respectively. Normally, a section that is both writable and executable denotes self-modifying code (*e.g.*, code that will decrypt its contents at runtime.) Although some packers may try to modify section properties at runtime, we observe a significant amount in our dataset that create them directly with the required properties.

- **FUNC_IMP_IAT** and **FUNC_IMP_IDT** are related to *the number of*

functions imported and found in IAT and IDT, respectively. This observation is directly related to the importance of **RATIO_MAL_API**. Packed files generally hide their behavior by loading via dynamic functions after unpacking. Thus, a limited number of functions are often imported directly, which explains the low value for these features in packed samples (see Figures 4.5 4.6 4.8 4.10).

- **ENT_GLOBAL** corresponds to the *byte entropy of the entire PE file* and **ENT_CODE** corresponds to the *byte entropy of the code section.* Since packing implies compression and/or encryption, the entropies of sections are often high. This explains why this feature is used in the literature [LH07; Cho+08; Jeo+10; RV15; Uga+14].

- **BYTEx** features represent raw values of the xth byte after the entry point. These values seem to affect many models. This is likely due to the dominance of some packers in the dataset. As a traditional packing detector, ML models mimic a partially signature-based approach.

We can summarize our findings by recalling the process of packing and unpacking. A packed file typically begins by executing the stub responsible for unpacking the file. This explains the role of **STAN_SEC**, **EP_NOT_CODE**, **EP_NOT_STD** and **NOT_CODE** which probably reflect the layout of the binary. A limited number of powerful imported functions hide the functionality of the binary. These limited number of functions (e.g.,*GetProcAddress* which dynamically loads other functions) are labeled as "malicious" by [ZFA15], which explains the features **FUNC_IMP_IAT**, **FUNC_IMP_IDT** and **RATIO_MAL_API**. Finally, packers are usually compressors and/or encryptors. So the code has to modify itself to access the original binary. This explains the importance of the **ENT_GLOBAL**, **ENT_CODE** and **ENT_GLOBAL** features.

4.4.3 Feature selection and PCA

Some features turn out to be more relevant than others. Thus, we can safely improve training time, memory usage, and feature extraction time by reducing the number of features. In this section, we will use two different wrapper methods: **Recursive Feature Elimination (RFE)** and SHAP selection [ME20]. We also compare these methods with the use of **Principal Component Analysis (PCA)** to reduce data dimensionality. In the end, we chose the method that gave the best feature reduction with a limited loss of recall (i.e., 0.5%).

We applied RFE to the following models: DT,RF,GBDT,LR, and LSVM (RFE is not applied to the other models because it is not directly applicable). The SHAP selection method was applied to all classifiers, considering features

Classifier	Old Accuracy	Method	New # features/components	Accuracy
KNN	0.972	PCA	13	0.971
GNBC	0.745	SHAP	19	0.811
LR	0.936	RFE	38	0.931
LSVM	0.935	RFE	55	0.935
DT	0.944	SHAP	42	0.936
RF	0.952	RFE	37	0.953
GBDT	0.973	RFE	54	0.973
MLP	0.973	PCA	32	0.968
KSVM	0.976	SHAP	26	0.972

Table 4.4: Top results from applying RFE/SHAP selection/PCA.

from one to 119. Finally, we applied PCA to all the classifiers. We followed a methodology similar to RFE: we tried a different number of components and kept the smaller number that resulted in a limited loss of recall. This allowed us to ignore several features with low variance. We kept the best of the three different methods. Our results are shown in Table 4.4.

We believe that many features have limited impact, given the minimal impact of dimensionality reduction on the performance of the ML algorithm (see Section 4.4.1.) Our results confirmed that we can discard low-variance features (identified earlier) without compromising the high performance of principal component analysis. PCA aims to preserve as much variance as possible by creating new features by linearly combining others. Limiting the number of principal components (i.e., the amount of variance conserved) shows that low-variance features have a negligible impact after dimensionality reduction.

In this section, we have observed that some features widely discussed in the literature have a large impact on the decision processes of machine learning models, while other features are less influential. Features such as **number of standard sections**, properties of **the entry point section**, number and type of **imported API functions**, **entropy** and specific **section permissions** seem to control aspects of the ML decision processes. The importance of **BYTE** features is likely due to the dominance of specific packers such as *UPX* (which reproduce a partially signature-based approach). We found that identifying these key features for packing detection using ML models could reduce feature extraction costs, memory consumption and training costs, as well as provide a better understanding of the packing detection problem and the limitations of static approaches.

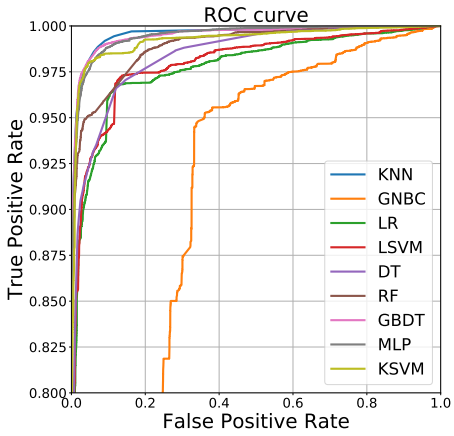


Figure 4.11: ROC curves for all 9 ML using Big data set.

Classifier	Precision	Recall	*F*-score
KNN	0.9762	**0.9733**	0.9748
GNBC	0.7675	0.9531	0.8503
LR	0.9554	0.9208	0.9378
LSVM	0.9521	0.9313	0.9416
DT	0.9647	0.9206	0.9421
RF	0.9812	0.9358	0.958
GBDT	0.9822	**0.9707**	0.9764
MLP	0.9777	**0.9668**	0.9722
KSVM	0.9804	**0.9694**	0.9749

Table 4.5: Detailed metrics for all 9 ML trained classifiers on Big data set.

4.5 Classifier Effectiveness

This section focuses on the effectiveness of the classifiers by looking at different classification metrics observed on a large dataset. We examine false negatives to identify the packers that evade our classifier and investigate the characteristics of these packers. Finally, we evaluate the robustness of our classifiers by determining how easy it is to create adversarial examples that evade detection.

4.5.1 Evaluation of Classification Metrics

First, we examine the ROC curves associated with the different classifiers, as shown in Figure 4.11. From these curves, we can observe four classifiers that perform well: KNN, GBDT, MLP and KSVM. GNBC performs poorly.

Table 4.5 presents a more detailed exploration of the metrics used to evaluate the different classifiers. The corresponding Wilcoxon test, evaluated with a 10-fold Table 4.6, allows to confirm our results.

GNBC gives the worst *F*-score statistically, but is the fastest model to train. The linear models LR and LSVM offer average performance and provide good baselines to observe advantages of the other models. Looking at the overhead required to train LSVM, LR serves as a better baseline. The tree-based models DT, RF, and GBDT perform well with less training time, suggesting why these models have been studied extensively in the literature. RF and GBDT both show high precision in their decisions; GBDT has the best recall compared to the other classifiers. Finally, MLP and KSVM perform well, similar to the top classifiers.

	KNN	GNBC	LR	LSVM	DT	RF	GBDT	MLP	KSVM
KNN	1.0	0.0625	0.0625	0.0625	0.0625	0.125	0.4375	0.125	0.3125
GNBC	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
LR	1.0	0.625	1.0	0.9375	0.9375	1.0	1.0	1.0	1.0
LSVM	1.0	0.0625	0.125	1.0	0.9375	1.0	1.0	1.0	1.0
DT	1.0	0.0625	0.125	0.125	1.0	0.8125	1.0	1.0	1.0
RF	0.9375	0.0625	0.0625	0.0625	0.3125	1.0	0.9456	0.9375	0.9375
GBDT	0.6875	0.0625	0.0625	0.0625	0.0625	0.0544	1.0	0.4375	0.4375
MLP	0.9375	0.0625	0.0625	0.0625	0.0625	0.125	0.6875	1.0	1.0
KSVM	0.8125	0.0625	0.0625	0.0625	0.0625	0.125	0.6875	0.0625	1.0

Table 4.6: p-values of the one-sided Wilcoxon rank sum hypothesis test to assess the result of Table 4.5. Values close to 0 indicate whether the median accuracy of the algorithm in row i is greater than the median accuracy of the algorithm in column j.

According to these metrics and the Wilcoxon test, GBDT, KNN and KSVM offer the best performance statistically. However, KNN takes more time to classify a sample because it has to compute the distance between the sample and the training data. So GBDT and KSVM seem to be better solutions. We can see that most of the classifiers have a higher precision (fewer false positives) than their recall (related to the number of false negatives).

To improve our understanding of false negatives, we decided to extract which samples were poorly classified and the characteristics of the associated packer. Figure 4.12 shows the top classes of poorly classified packed files, and where this happens in the training set, for GBDT. The same packers appear in the results of the other classifiers (with the addition of *nullsoft* and *yoda's protector*).

If we examine the features that lead these classifiers to make a wrong decision, we find that the misclassified packed binaries do not share the same characteristics as the other packed binaries. Most misclassified packed binaries have a **FUNC_IMP_IAT** and **FUNC_IMP_IDT** similar to non-packed binaries. Most misclassified packed binaries also have a high number of **STAND_SEC**, or standard sections, and a high **MAX_RATIO_RAW_VT**, or the maximum ratio of raw data to virtual size among all sections. As seen in Section 4.4, these features all have a high impact on the decision processes of most classifiers. (The **MAX_RATIO_RAW_VT** feature is mainly used in tree-based models). Our classifier seems to have a bias towards these packers due to their low occurrence in the dataset compared to other known packers like UPX or ASPack. This recalls previous research [Agh+20; Man+20] on the limitations of static approaches and paves the way for future research around new features or hybrid approaches.

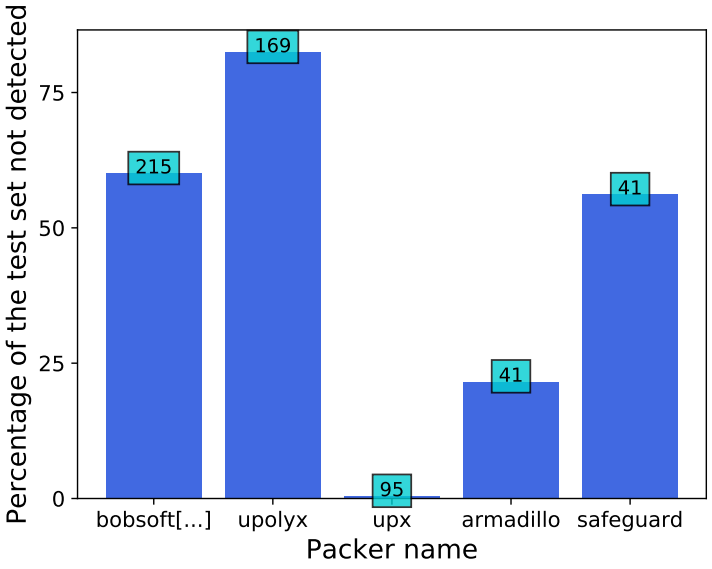


Figure 4.12: Top packers evading GBDT classifier. For each packer, the Y-axis represents the percentages of evading samples found in the test set while the number indicates how many binaries it represents.

4.5.2 Robustness

To evaluate the robustness of our classifiers, we created adversarial examples by applying simple modifications to packed samples. To do this, we randomly extracted 3000 correctly classified packed binaries from the Big dataset. We then ran MAB-malware [Son+22], an adversarial example generation framework, on the selected binaries. MAB-malware modifies various elements of a binary, such as renaming, adding, or appending sections. MAB-malware can also add an overlay, remove or debug information or a certificate, break the checksum, etc. Figure 4.13 shows how some classifiers that were very effective at detecting packing can be easily bypassed by the MAB malware modified binaries. For example, DT and KSVM were so easily defeated by the MAB malware that the results are not even shown in Figure 4.13; the evasion rate for these classifiers was too high after a single modification.

After sufficient mutation with MAB malware, LSVM, LR and RF are also evaded. MLP, GNBC, KNN and GBDT seem to be more resistant to these mutations. In particular, GBDT remains largely unaffected.

The most effective MAB malware mutations are **addition of a section** (SA: Section Add) or **rename of a section** (SR: Section Rename). Figure 4.14a and Figure 4.14b illustrate the sequence of modifications used to escape LSVM and GBDT. The classifiers turn out to be overly dependent on section-related

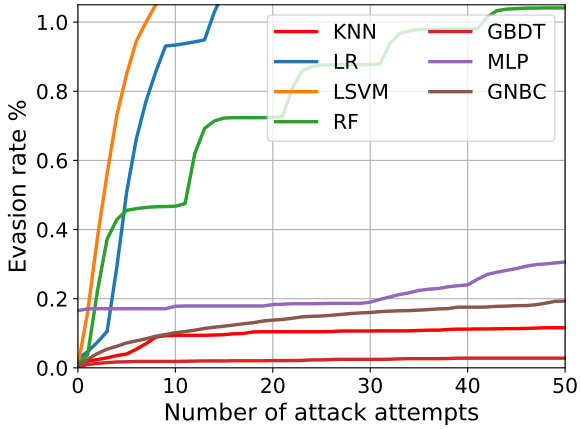
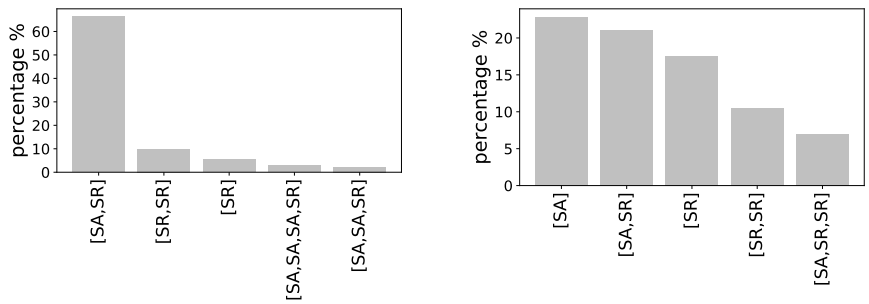


Figure 4.13: Evasion rate observed by MAB-malware on our classifiers. DT and KSVM are not shown since they are totally evaded after the first mutation.

features. They mimic some simple heuristics implemented in detectors on section names (*e.g.*, Manalyze [Kwi]) and fail to generalize packing detection properly.



(a) LSVM **(b) GBDT**

Figure 4.14: Modification(s) applied for samples which successfully evade (a) LSVM classifier (b) GBDT classifier

4.5.3 Self-packed binaries

Finally, we evaluate our different classifiers on self-packed binaries using the Packing-Box [DBL22b]. This is a framework that allows users to create and manipulate datasets of binaries packed with selected **known** packers. This experiment allowed us to evaluate the performance of the classifiers against a known ground truth (as opposed to our previous ground truth, which may contain errors due to the labeling process). Table 4.7 shows that all classifiers

except KNN and KSVM detect perfectly UPX packers. Similarly, KSVM does poorly with MPRESS, while KNN works better. The other packers are not well represented in our dataset in terms of the packer distribution of Section 4.3. This may explain why packers like Amber are barely detected by most of our classifiers. Overall, the most interesting results are shown by RF, GBDT and MLP. Out of the six packers, RF and GBDT correctly detect five, while MLP correctly detects four. However, MLP is the only classifier that correctly detects Amber. Despite its speed and simplicity, GNBC detects four packers.

Classifier	UPX	Amber	Eronana	mew	MPRESS	rlpack
KNN	0.69	0.0	0.85	0.95	0.99	0.99
GNBC	1.0	0.0	0.01	0.99	1.0	0.99
LR	1.0	0.0	1.0	0.99	1.0	0.0
LSVM	1.0	0.0	0.99	0.99	1.0	0.0
DT	1.0	0.0	0.01	0.99	1.0	0.99
RF	1.0	0.0	0.99	0.99	1.0	0.99
GBDT	1.0	0.0	1.0	0.98	0.81	0.99
MLP	1.0	1.0	0.64	0.22	1.0	0.99
KSVM	0.55	0.0	0.99	0.99	0.34	0.99

Table 4.7: Accuracy of the different classifiers on binaries packed with the packing box.

4.6 Packing detection economics

This section focuses on the deployment costs of classifiers by proposing an economical analysis that considers long-term accuracy, memory, and training/inference time. To this end, we examine the effectiveness of all nine ML algorithms on chronological data. We then briefly consider the economics behind the different models (i.e., the resources required and the resulting performance), as previously proposed by [UAB19] in the context of malware detection.

In our analysis, we first use the BIG dataset to train each classifier, training on samples collected from 2019-09-17 to 2020-04-30. We then test the resulting classifiers against the CHRONO dataset, which consists of malware samples collected between 2020-05-01 and 2020-07-29. As described in 4.3.1, the CHRONO dataset contains four subsets, each corresponding to a separate time period, labeled T1 through T4. **Our malware is harvested from a regular malware feed from Cisco, which consists of observed malware**[1] **'in the wild' on**

[1]It is worth noting that the observation time is different from the creation time, i.e. there are situations where it takes a considerable amount of time for a specialist to observe an

the given date. As a result, we observe changes in malware distribution during the different time periods, as shown in Section 4.3.1. Separating the datasets by time period allows us to show evolving trends in packers and the impact of this distribution shift.

The accuracy of ML classifiers over time is shown in Table 4.8. As expected, the effectiveness of most ML-trained classifiers decreases with time. This corresponds to a distribution shift as suggested in [Bio+19], which can be explained by different trends of the used packers depending on the period of interest. According to the table, KNN, RF, GBDT, MLP, and KSVM are less affected by the distribution shift. On the other hand, the accuracy of GNBC varies significantly between the different time periods, reaching its best performance at T4, but the worst performance of all classifiers at T2. This surprising result could be related to the inner workings of GNBC: the probabilities evaluated to build the model (see Section 1.3.1) make it highly dependent on the dataset distribution used for training, and T4 should have a distribution that fits these probabilities. To make an informed decision about which model to choose, we can put our observations into perspective by considering the training time, inference time (average time for 10,000 samples), and memory requirements of the different models listed in Table 4.8. This allows us to choose the model that best fits our available resources and specific requirements. Such considerations are especially important in real-world applications where these classifiers are integrated into larger malware analysis toolchains. For example, KNN uses the entire dataset, resulting in higher inference time and memory usage. MLP and KSVM have longer training times but shorter inference time and require less memory. In contrast, GBDT and RF offer a good trade-off between efficiency, training/inference time, and memory usage while using a small number of features. Table 4.8 also illustrates how reducing the number of features improves training/inference time and reduces memory for most models while maintaining their accuracies.

4.7 Conclusion and Future works

The feature significance analysis shows that across nine different ML algorithms, there are a small number of static features that have a high influence on the decision process. These are: number of standard sections, entry point not in code/standard section, malicious API calls, missing code section, number of readable-writable-executable and writable-executable sections, number of functions imported and found in IAT/IDT, the global/code entropy, and specific byte values.

active malware. Since it is impossible to know the creation time, we base our analysis on the observation times given in datasets recognized by the community and specialists.

Classifier	Baseline	T1	T2	T3	T4	Training time[s]	Inference* time [s]	Memory [kb]	# features
KNN	0.9697	0.9709	0.9707	0.9354	0.9378	0.5	50	113 200	119
	0.9722	0.9709	0.9672	0.9245	0.9297	0.5	1	15 100	13 (PCA)
GNBC	0.7457	0.6221	0.5955	0.7714	0.8549	0.2	0.01	4	119
	0.8114	0.7849	0.7386	0.8406	0.8921	0.1	0.01	1	19
LR	0.9364	0.924	0.9343	0.9227	0.9264	22	0.01	2	119
	0.9314	0.8979	0.8954	0.9103	0.926	9.5	0.0005	1	38
LSVM	0.9355	0.9164	0.9297	0.9131	0.9258	2082	0.01	7	119
	0.9351	0.9166	0.9333	0.9114	0.9239	1281	0.007	4	55
DT	0.9448	0.9081	0.9096	0.901	0.9223	1	0.002	9	119
	0.9365	0.8731	0.8901	0.9091	0.9116	0.25	0.001	9	42
RF	0.9521	0.937	0.9424	0.949	0.9479	2	0.02	154	119
	0.9539	0.942	0.9516	0.9472	0.9476	2	0.02	159	37
GBDT	0.9734	0.9721	0.9737	0.9383	0.9371	22	0.01	164	119
	0.9736	0.9733	0.9741	0.9368	0.9358	11	0.01	164	54
MLP	0.9736	0.9741	0.9735	0.9381	0.9369	476	0.045	190	119
	0.9665	0.9612	0.9614	0.93	0.9374	466	0.045	121	32 (PCA)
KSVM	0.9766	0.9792	0.9749	0.9306	0.9355	2781	21.3	8900	119
	0.972	0.9736	0.9721	0.937	0.9388	1212	5.9	3100	26

Table 4.8: Evolution of the accuracy of nine ML-trained classifiers trained on Big dataset and tested on Chrono dataset split by time period. The metrics of each classifier are shown with and without feature reduction methods. *Inference time is the average time to evaluate 10,000 samples.

We compared the effectiveness of nine different ML algorithms with classical metrics, including adversarial examples and economics. Overall, KNN, GBDT, and MLP proved to be the most effective and robust classifiers. These three algorithms performed well in terms of accuracy, precision, recall, and F-score while remaining resilient to adversarial examples. KNN and GBDT performed particularly well. The other classifiers were more vulnerable to simple changes in section names and numbers. Observing which known packers were classified poorly allows us to discover the limitations of our static features and highlight the attention needed to build ground truth.

Finally, we used chronological samples to evaluate the long-term effectiveness of the nine ML algorithms. The results showed that, as expected, most classifiers lost accuracy over time. However, GBDT combined a low training cost with a modest memory requirement, making GBDT the most effective algorithm.

Below, we briefly discuss future research directions.

Ground Truth A challenge to the validity of machine learning work applied to malware is the selection of an accurate **ground truth** [Arp+22]. Our ground truth construction relies on several approaches (*i.e.*, PeID, Manalyze, PEframe, and DIE) that have limitations. Our use of the consensus approach should mitigate some of them (see also [Bio+19]). However, the approach is vulnerable to rare or unknown packers. The robustness exploration (see Section 4.5) highlights this problem by showing packers that are misclassified,

the reason for the misclassification, and how to create adversarial examples. We show how, as with our ground truth generation tools, section-related features are important in some classifiers. This parallel may explain some possible limitations of our ground truth.

Dynamic Features The problem of packing detection as part of a larger malware detection and defense system motivated our decision to focus on static features. However, as shown in the literature [Man+20] and in our experiments with adversarial examples, some static features can be bypassed. Thus, several related works have considered dynamic features [Bat+13; Che+18; KPY07]. These dynamic features could likely represent the behavior of the analyzed binaries and better generalize the packing detection. However, extracting these features can be costly, and the trade-offs need to be examined.

Explainable AI Examining correlated features and their effect on SHAP values presents another challenge. While SHAP values are good indicators of classifier decision making, they do not take into account possible dependencies between features. Regarding the SHAP calculation [Mol20], replacing feature values with random instance values is simpler than sampling from the marginal distribution. However, it ignores dependencies between features and gives significant weight to unlikely data points. Furthermore, our feature analysis is based on a real-world dataset that is unbalanced towards UPX packers (from different versions). It would be interesting to apply similar feature analysis and evaluation to a balanced training set and observe the difference in our model behavior.

Packing Classification The problem of packing detection has also been extended to packing classification [Sun+10; Dam+22]. While these works generally focus on already known packers, they do not investigate which features influence the decision process, nor which approach is more resilient in an adversarial context. Moreover, these classifiers could also be generalized to detect properties of packers using the terminology proposed in [Mur+22] (*i.e.*, compressor, cryptors, virtualizer, etc.) or packer Type [Uga+15].

Adversarial Examples We used MAB-malware [Son+22] to generate adversarial examples. To our knowledge, there is no dedicated tool that targets packing detectors. Another direction of research could be to design specific attacks on packing detectors by targeting their more influential features. The potential benefits of dynamic features to counter these attacks are worth investigating.

Uncertainty Quantification Finally, real-world applications would benefit from estimating the uncertainty of our classifiers' decisions. Knowing the confidence level of our classifier could be used to design a modular approach in a malware analysis toolchain. A high confidence would mean sending the binary to an unpacker, and a moderate confidence could initiate an evaluation of more (dynamic) features to improve the confidence level. Recent research [Bar+22] has looked at estimating uncertainty (*e.g.*, using conformal prediction) and shown benefits against adversarial examples or concept drift in the malware ecosystem.

Extending SEMA with Federated Learning

5

This chapter discusses the application of federated learning in the context of malware classification. We propose to deploy SEMA on multiple clients and use federated learning to collaboratively build a model while preserving the privacy of client data. After introducing our deep learning and federated learning architecture, we perform several experiments to evaluate the obtained models by varying the symbolic exploration strategies on the client and the aggregation scenario.

5.1 Introduction

While machine learning is a promising way to make malware identification more efficient, robust and scalable [Kas21; SS21], the use of a representative dataset is essential. As discussed by Arp et al. [Arp+22], the design of a learning-based system begins with the acquisition of a representative dataset that aims to capture the true data distribution of the underlying security problem. While it is difficult to avoid sampling biases when building a dataset (*e.g.*, due to limitations on the number of data sources, the number of files observed, the time frame considered, ...), attempting to understand and limit these biases is essential to build a robust system.

In a centralized ML configuration, a large amount of data is collected in one place to train an ML model. This data comes from different hosts, networks, cloud-based components or even different (anti-virus) companies. However, collecting all the data on a central server for training is not really practical. Today, data is a valuable resource for companies that do not want to share it for competitive reasons. Users' privacy should also be protected by not revealing their preferences (*e.g.*, what software they install on their computers).

To address this challenge, McMahan et al. [McM+17] introduce a new learning paradigm: federated learning (FL). In federated learning, training data does not leave the device from which it originated. Instead, each device (called *client*) uses its data to train a local model, and then all the local models are aggregated by a central server to build a global shared model. The model is then sent back to the clients, which refine the training, and a new iteration (or *round*) of the federated learning process begins. This learning technique has been

successfully applied to PE malware detection using static features [VHH22; LH20], anomaly detection on IoT devices [Ngu+19; Rey+22; GR22; Zha+19] or Android malware detection [Hsu+20; GMD21].

While FL avoids direct data sharing, observing model updates could allow malicious actors to perform *inference attacks*, violating the privacy of participants. These attacks could leak data membership [Wan+19; ZLH19; Mel+19] (*i.e.*, discover if a given data point was used in training), reveal properties of the training data [Mel+19], or even reconstruct the training data [Wan+19]. Various countermeasures have been proposed to protect against these attacks. On the one hand, differential privacy noise [Aba+16] could be added to the local updates [Tru+20] or by the aggregation server [Wei+20] to reduce the recovered information and imply a privacy/utility trade-off (*i.e.*, the more noise is added, the more privacy is preserved and the performance degrades). On the other hand, secure aggregation mechanisms based on cryptographic primitives [Man+23] (*e.g.*: secure multi-party computation [Geh+23] or fully homomorphic encryption [Zha+20]) provide a privacy guarantee with respect to aggregation at the cost of communication/computational overhead.

In this chapter, we propose an FL architecture that allows multiple participants to jointly learn an accurate deep learning model for malware classification. These models are built without sharing local datasets while protecting the privacy of the participants. In our system, each client has its own malware dataset from which it extracts SCDGs using SEMA. Each client trains a deep learning model to encode and classify the SCDGs. The model consists of a graph auto-encoder that vectorizes the input SCDGs and a classifier that identifies the malware family according to the output of the auto-encoder.

To collaboratively learn a classifier, each client sends its model updates encrypted with homomorphic encryption to an aggregation server, which combines the local model parameters into a global model. A key client is selected to hold the secret key for encrypting the model parameters. To protect against inference attacks, we replace the key client in each training round. Even if the aggregation server or clients take a snapshot of the training model, their ability to infer data information from other clients is limited. To evaluate our deep learning model, we use two datasets, each corresponding to a different experimental setup. In the first setup, each client has access to a limited part of the dataset and applies the same exploration strategy, namely the CDFS exploration strategy presented in Chapter 2, to generate its SCDGs. In the second setup, each client has access to the full dataset and defines its own exploration strategy to generate SCDGs. Each exploration strategy results in a different representation of the malware. Therefore, this setup investigates whether the clients' models can benefit from unknown exploration strategies. In both setups, in addition to full aggregation of the deep learning model, we experiment with partial aggregation, where clients

jointly build an auto-encoder to improve malware encoding, but do not share their classifier.

The chapter is organized as follows. Section 5.2 presents a deep neural network model to encode and classify the system call dependency graphs. Our federated learning architecture for training the deep learning model is presented in Section 5.3. Section 5.4 presents the evaluation of the architecture in different scenarios. Finally, we conclude and present possible future work in Section 5.5.

5.2 Deep learning auto-encoder and classifier

While several works in the literature use federated learning to build malware classification models based on static features [VHH22; LH20; Hsu+20; GMD21], we choose to use the SCDG generated by SEMA to build more robust models. We use a deep learning model to encode the SCDG and classify malware. The model consists of a graph auto-encoder connected to a classifier module, as shown in Figure 5.1. The graph auto-encoder is divided into two parts: an encoder and a decoder. First, the encoder $E(G)$ transforms the graphs $\mathbf{G}$ obtained by SEMA into feature vectors X of fixed length. This vector representation minimizes the reconstruction error with the graph $\mathbf{G}'$ obtained by the decoder $D(X)$. Such an auto-encoder architecture is known to provide a feature representation with a better generalization capability than learning a representation without a decoder to directly optimize the classification objective [Vin+10]. Finally, the classifier module $F(X)$ takes the feature vector as input and produces a predicted class L associated with that feature vector.

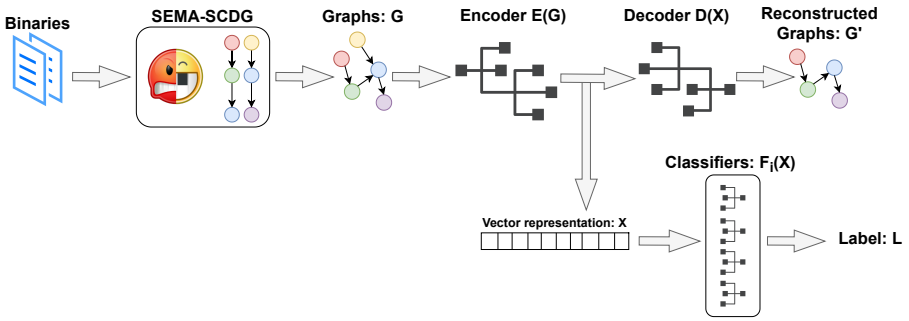


Figure 5.1: Workflow of our deep learning classifier

Our graph auto-encoder consists of Long Short Term Memory (LSTM) layers for embedding SCDGs into feature vectors. An LSTM layer is a neural network layer capable of modeling dependencies in sequential data of arbitrary size. The graph auto-encoder trains its LSTM layers using two modules: the

encoder and the decoder. Let **G** be an SCDG. To encode **G**, we take as input the sequence of edges (*i.e.*, the two labels of connected vertices) produced by SEMA. The labels of each node (*i.e.*, names of the API calls) are encoded into a numeric value using FastText [Boj+17] as proposed in [DGL21]. The encoder module $E(G)$ transforms the graph structure, *i.e.*, the sequence of edges from **G**, into a feature vector of fixed length, *i.e.*, $X = E(G)$. The decoder module $D(X)$ is trained to reconstruct the original graph G' from its feature vector X given by the encoder module, *i.e.*, $G' = D(X)$. This graph auto-encoder is trained to minimize the reconstruction error Loss_1:

$$\text{Loss}_1(G, G') = \frac{1}{2}\sum_{i=1}^{n} ||s_i - \hat{s}_i||^2$$

where $||s_i - \hat{s}_i||$ is the difference between the sequence element s_i of the original graph G and its reconstructed component $\hat{s}_i$ in the graph G'.

The classifier module is a combination of n binary classifiers, $\{\mathbf{F}_i(X)\}_{i=1}^{n}$, $n > 0$, each corresponding to a different malware family. These classifiers use the output X of the encoder $E(G)$. Let X be an output of $E(G)$. The module is defined as:

$$\hat{y}_i = \text{softmax}([\mathbf{F}_i(\mathbf{X})]_{i=1}^{n})$$

$$\mathbf{L}(X) = \arg\max_{i=1}^{n}(\hat{y}_i)$$

where $\mathbf{L}(X)$ is the predicted malware family for the input SCDG converted to the feature representation X and $\mathbf{F}_i$ is a linear model as defined in Section 1.3.1.

This module is trained to optimize the binary cross entropy loss:

$$\text{Loss}_2 = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

where $\hat{y}_i$ is the output of the softmax function and y_i is the true malware family associated with X: $y_i = 1$ if the SCDG belongs to family i and 0 otherwise.

To train our model, we use the Adam optimizer algorithm [KB14], which updates the model parameters at each training step on the training data to optimize the total loss function. The total loss function LossTot is the sum of the losses from the graph auto-encoder and the classifier module.

$$\text{LossTot} = \text{Loss}_1 + \text{Loss}_2$$

5.3 Federated learning with homomorphic encryption

In this section, we present our approach to train our model on different clients using federated learning. First, we recall the federated learning (FL) approach [Kon+16]. FL is a collaborative learning approach between $\mathbf{N}$ clients using an aggregation server. In the training process, clients use their private dataset to train their local model, and then all local models are sent to the aggregation server to be merged into a global model. The parameters of the global model are then sent to the clients to update their local model. After a sufficient number of local training and update exchanges between the aggregation server and the associated clients, the client models can converge to an optimal learning model. While FL avoids direct sharing of client datasets, it does not guarantee privacy due to data inference attacks. Differential privacy noise is a popular countermeasure that faces the utility/privacy trade-off [Boe+23]: adding more noise improves privacy but decreases the accuracy of the model. Therefore, the use of cryptographic primitives such as homomorphic encryption [Pho+18; Aon+16] or multi-party computation (MPC) [Bur+10] is an interesting approach to preserve privacy. In this work, we choose to use the homomorphic encryption scheme.

Homomorphic Encryption Let us recall the homomorphic encryption scheme [Reg09; Cho+21; KPP20]. Let $(\mathbf{sk}, \mathbf{pk})$ be a pair of secret and public keys respectively and $\mathbf{z}$ be a plaintext. The public key $\mathbf{pk}$ is used to encrypt $\mathbf{z}$ into a ciphertext $\bar{z}_{\mathbf{pk}}$. Let HEnc be a function that encrypts a plaintext into a ciphertext with a public key, *e.g.*, $\bar{z}_{\mathbf{pk}} = \text{HEnc}(\mathbf{z}, \mathbf{pk})$. Let HDecrypt be a function that decrypts a ciphertext using a secret key into a plaintext, *i.e.*, $\mathbf{z} = \text{HDecrypt}(\bar{z}_{\mathbf{pk}}, \mathbf{sk})$.

The important property of homomorphic encryption is that users can process and perform calculations on encrypted data without having access to the original data. After the operations have been performed, the user decrypts the processed data with the secret key and obtains the expected result [CRT18; Nik+13]. Given the plaintexts $\mathbf{x}$ and $\mathbf{z}$, the multiplication of the ciphertext of $\mathbf{z}$, *i.e.*$\bar{z}_{\mathbf{pk}}$, and the plaintext $\mathbf{x}$, is a ciphertext of $\mathbf{z} \cdot \mathbf{x}$, *i.e.*, $\text{HEnc}(\mathbf{z} \cdot \mathbf{x}, \mathbf{pk}) = \bar{z}_{\mathbf{pk}} \cdot \mathbf{x}$. The addition of two ciphertexts $\bar{x}_{\mathbf{pk}}$ and $\bar{z}_{\mathbf{pk}}$ is also a ciphertext of $\mathbf{z} + \mathbf{x}$, *i.e.*, $\text{HEnc}(\mathbf{z} + \mathbf{x}, \mathbf{pk}) = \bar{z}_{\mathbf{pk}} + \bar{x}_{\mathbf{pk}}$.

Federated Learning Protocol We assume that all participants are honest-but-curious: they follow the protocol but try to infer as much information as possible from the values (shares) they receive. Each client encrypts its local updates to the model weights, and the global update is computed on the encrypted data using the addition operator of homomorphic encryption. The Algorithm 3 describes the m training rounds of $\mathbf{N}$ clients.

Algorithm 3 Workflow of the federated learning approach

1: **for each** *round* $\in [1 \dots m]$ **do**
2: Randomly choose a *Key-client* among clients.
3: The *Key-client* generates a secret value z and encrypts it with its public key $\mathbf{pk}^s$.
4: The *Key-client* sends its encrypted secret value $\bar{z}_{\mathbf{pk}^s} = \text{HEnc}(\mathbf{z}, \mathbf{pk}^s)$ to all clients.
5: **for each** $i \in [1 \dots \mathbf{N}]$ **do in parallel**
6: Client i trains its local model on its own data.
7: Client i encrypts its model's parameters $\mathbf{w}^i$ using the secret value $\bar{z}_{\mathbf{pk}^s}$, *i.e.*, $\bar{w}^i_{\mathbf{pk}^s} = \mathbf{w}^i \cdot \bar{z}_{\mathbf{pk}^s}$.
8: Client i sends its encrypted update $\bar{w}^i_{\mathbf{pk}^s}$ to the aggregation server.
9: The aggregation server computes the encrypted global weight

$$\bar{w}_{\mathbf{pk}^s} = \sum_{i=1}^{n} \bar{w}^i_{\mathbf{pk}^s}.$$

10: The aggregation server sends the encrypted global weight $\bar{w}_{\mathbf{pk}^s}$ to the *Key-client.*
11: The *Key-client* decrypts $\bar{w}_{\mathbf{pk}^s}$ by its secret key $\mathbf{sk}^s$ and the secret value $\mathbf{z}$ to get the global weight $\mathbf{w}$.

$$\mathbf{w} = \frac{1}{\mathbf{z} \cdot \mathbf{N}} \text{HDecrypt}(\bar{w}_{\mathbf{pk}^s}, \mathbf{sk}^s)$$

12: **for each** $i \in [1 \dots \mathbf{N}]$ **do in parallel**
13: The *Key-client* encrypts the updates $\mathbf{w}$ by $\mathbf{pk}^i$, *i.e.*, $\mathbf{w}_{\mathbf{pk}^i}$, and sends the updates to the client i.
14: Client i decrypts $\mathbf{w}_{\mathbf{pk}^i}$ by its secret key $\mathbf{sk}^i$ and applies the updates to its local model.

Each client i holds a pair $(\mathbf{pk}^i, \mathbf{sk}^i)$ of public and secret keys. For each training round, the training process is as follows: First, the system selects a key client among the clients (Line 1). The selected client,*i.e.*, the *key client*, generates a secret value $\mathbf{z}$ and encrypts this value with its public key $\mathbf{pk}^s$ (Line 2). The *key client* then exchanges its encrypted value, *i.e.*, $\bar{z}_{\mathbf{pk}^s} = \text{HEnc}(\mathbf{z}, \mathbf{pk}^s)$ with the other clients (Line 3). Meanwhile, each client trains its model locally. When the training step is finished, they use the ciphertext $\bar{z}_{\mathbf{pk}^s}$ of the key client to encrypt the parameters of their local model (Line 7) and send them to the aggregation server (Line 8). The aggregation server then computes the sum of the clients' parameters (Line 9) and sends it to the key client (Line 10). The

key client decrypts the global updates from the aggregation server and sends the decrypted updates to each client (Line 13). Finally, the clients receive the updates and apply them to their local model (Line 14). After these updates, a new round is started by selecting a new key client and training the local models on the client side.

Application to malware classification We apply our learning protocol to train the deep learning model with N clients. Each client has its own malware dataset, which cannot be shared with others. To identify malware, the clients implement a classifier model on their side. The training process of three clients is illustrated in Figure 5.2. First, each client uses SEMA to generate an SCDG from their local datasets. They train a local model and obtain their model parameters: $\bar{w}^1$, $\bar{w}^2$ and $\bar{w}^3$. After choosing a key client (*e.g.*, client 2 in this first round), the parameters of the trained model are encrypted ($\bar{w}^1_{\mathbf{pk}^2}$,$\bar{w}^2_{\mathbf{pk}^2}$,$\bar{w}^3_{\mathbf{pk}^2}$) using the secret value $\bar{z}_{\mathbf{pk}^s}$ shared by the key client and sent to the aggregation server. The aggregation server securely computes the sum of the encrypted parameters $\bar{w}_{\mathbf{pk}^2}$. It then sends the aggregated parameters to the key client, *i.e.*, client 2, for decryption. The key client decrypts the aggregated parameters and sends them to client 1 and client 3. All clients receive the aggregated parameters and update their local models.

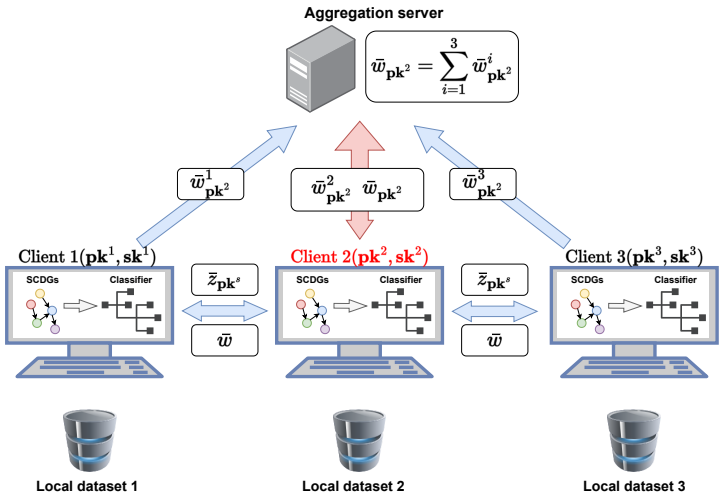


Figure 5.2: Learning with 3 clients and the aggregation server. client 2 is the key client and is replaced by another after each training round.

Learning scenarios In this chapter, we look at three different scenarios:

1. Clients share their dataset directly, without using our federated learning

approach. Therefore, the model is trained on all available data. This scenario is called *Centralized Learning* and is used as a baseline. If each client only uses its own dataset, we call this scenario *Traditional Learning*.

2. Clients learn the complete model together, including the graph auto-encoder $(E(G), D(X))$ and the classifier module $\{\mathbf{F}_i(\cdot)\}$, called *Full-Aggregation Learning*. This is a closed collaboration as **all clients needs to agree on the same model architecture and the malware families considered**.

3. Clients collaboratively learn a part of the model, *i.e.*, the encoder $E(G)$, with each other, called *Partial-Aggregation Learning*. It allows clients to share their feature representation in the form of an encoder $E(G)$, while **avoiding the need to agree on malware families**. Therefore, each client retains flexibility in the implementation of its classification models.

5.4 Evaluation

In this section, we evaluate the performance of our federated learning system. We first present our two datasets and their repartition on each client. We then compare our three learning scenarios on these datasets: *Centralized Learning*, *Full-Aggregation Learning* and *Partial-Aggregation Learning*. The encoder module of each client is parametrized to output a feature vector of size 64. The classifier module is parametrized according to the number of malware families in each local dataset. We use the PyTorch [Pas+19] library version 1.10.2 for the deep learning models and the TenSEAL library version 0.3.8 for encryption and parameter aggregation.

Dataset We collect malware from Cisco and the MalwareBazaar [Abu23] to create two datasets: Dataset-1 and Dataset-2. The malware labels are assigned by AVClass [Seb+16] according to VirusTotal reports. Dataset-1 consists of 2260 malware from 15 families, presented in Table 5.1. It is randomly divided into two sets: the training set of 2034 malware and the test set of 226 malware. The training set is then divided into 3 partitions of 678 malware with equally distributed families and shared to the three clients. All clients use the same exploration strategy, *i.e.*, the CDFS strategy, to extract their SCDGs. We call this configuration *Homogeneous SCDGs*. Dataset-2 is a collection of 1844 malware from 15 families presented in Table 5.2. The full dataset is distributed across the three clients. Each client uses a different exploration strategy to compute SCDGs from Dataset-2. In particular, client 1 successfully extracts

1660 graphs using the BFS strategy. Client 2 successfully extracts 1725 graphs using the CBFS strategy. Client 3 obtains 1662 graphs using the CDFS strategy. Since the clients implement different strategies (with a timeout of 20 minutes), the number of SCDGs extracted from Dataset-2 is different for each client. We call this setup *Heterogeneous SCDGs*. It is designed to investigate whether clients can benefit from unknown exploration strategies applied to a common dataset. The repartition of each dataset between clients is detailed in Table 5.3.

Family	#samples		Family	# samples
autoit	60		sfone	56
bancteian	30		sillyp2p	60
delf	150		simbot	60
ircbot	60		shiz	60
lamer	340		sodinokibi	142
nitol	94		sytro	124
netwire	295		wabot	342
remcosRAT	387			

Table 5.1: Composition of the Dataset-1

Family	#samples		Family	# samples
bancteain	104		remcosRAT	446
delf	109		sfone	32
fickerstealer	97		sillyp2p	269
gandcrab	94		simbot	126
ircbot	26		sodinokibi	51
lamer	173		sytro	77
nitol	71		wabot	134
redlinestealer	35			

Table 5.2: Composition of the Dataset-2

5.4.1 Federated Learning with homogeneous SCDGs

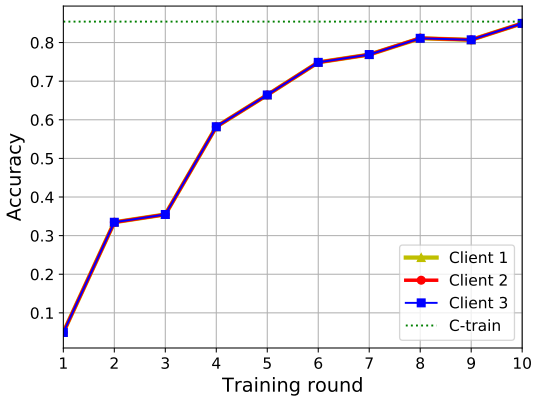
We first consider the homogeneous SCDGs extracted from Dataset-1. In the centralized learning scenario, we train the classifier to identify the 15 malware families using the full dataset. We train the model for 10 epochs and obtain an accuracy of 85.4% on the test set. After each round of training, the clients evaluate their models on the test set. Note that all training scenarios are evaluated on the same test set. Figure 5.3(a) and (b) illustrate the accuracy evolution of each model for each scenario.

For the *Full-Aggregation Learning* (Figure 5.3-a), the performance of all clients is similar and their accuracy can reach 84.96% compared to the accuracy of 85.4% of the *Centralized Learning*. In the *Partial-Aggregation Learning*

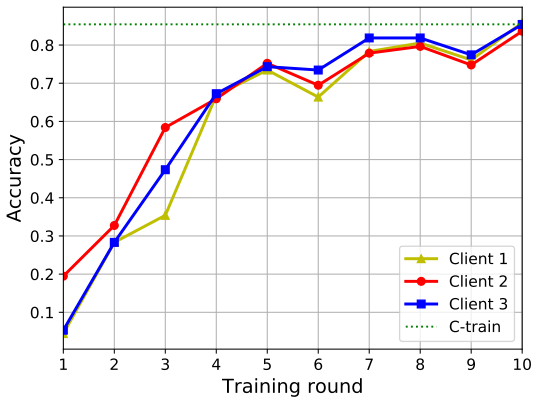
	Partitions	Training set	Test set
Dataset-1	Client-1	678	226
	Client-2	678	
	Client-3	678	
Total		2034	226
Dataset-2	Client-1	1245	415
	Client-2	1293	432
	Client-3	1246	416

Table 5.3: Data distribution among clients

scenario, the accuracies of the clients vary initially but converge at the end of the training. Compared to *Centralized Learning*, the performance of client 1 and client 3 in *Partial-Aggregation Learning* is equal or even better. All results are reported in Table 5.4. The *Full-Aggregation Learning* does not seem to reduce classifiers' accuracies compared to the *Centralized* scenario. However, we observe that learning the SCDG encoder collaboratively with *Partial-Aggregation Learning* seems to be the more interesting option, as it does not require sharing details about the classifiers/families used, while providing good accuracy. The confusion matrix of the classifiers for the three scenarios are represented in Appendix D.1.



(a) Full-aggregation learning



(b) Partial-aggregation learning

Figure 5.3: Accuracy of the client's models at each round using homogeneous SCDGs. The dashed line (C-train) represents the accuracy obtained with centralized learning.

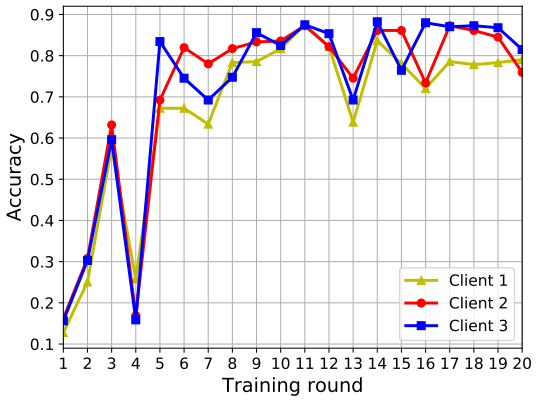
		Accuracy (%)
Centralized Learning		85.4
Full-Aggregation Learning		84.96
Partial-Aggregation Learning	Client-1	**85.84**
	Client-2	83.63
	Client-3	**85.4**

Table 5.4: Comparison of Centralized Learning, Full-Aggregation Learning and Partial-Aggregation Learning on DATASET-1 using homogeneous SCDGs.

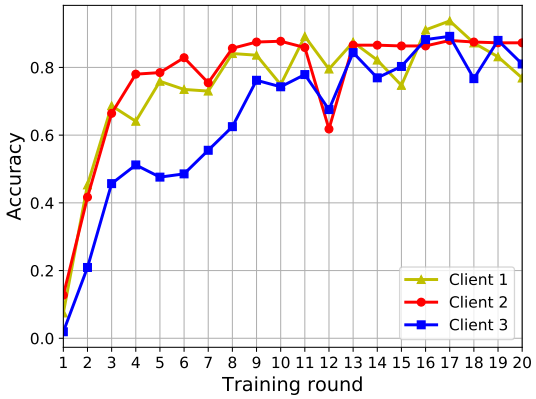
5.4.2 Federated Learning with heterogeneous SCDGs

We then consider that each client uses a different exploration strategy to extract its SCDGs from DATASET-2. Client 1 uses the BFS strategy, client 2 uses the CBFS strategy and client 3 uses the CDFS strategy. In this heterogeneous SCDG setting, each client has its own training and test sets. We first consider a *"Traditional" Learning* scenario where the models are trained separately on each client's training set for 20 epochs. The models are then evaluated on the clients' test sets. In the other two scenarios we consider 20 training rounds. In each round, clients share their local updates and evaluate their new updated model on their own test set. Figure 5.4(a) and (b) illustrate the accuracy evolution of the client's model for each scenario.

In the *Full-Aggregation Learning* scenario, three clients achieve an accuracy of over 87% after 11 training rounds. Compared to *Traditional Learning*, client 2 in *Full-Aggregation Learning* can reach 87.27% accuracy, at the 11-th round while the accuracy in *Traditional Learning* is 81.02%. For *Partial-Aggregation Learning* in Figure 5.4-b, client 1 gets the accuracy of 93.73% after 17 training rounds while client 2 and client 3 get 87.96% and 89.18% respectively. The performances of client 1 and client 2 overtake that of *Traditional Learning* while the degradation of client 3 is about 0.72%. All results are reported in Table 5.5. The benefits of *Full-Aggregation Learning* seem to be less important in this setup and depend on the client's strategy. However, we observe that the *Partial-Aggregation Learning* scenario again provides good accuracies and seems to take advantage of the multiplicity of representations to improve the encoder part of the model. The confusion matrix of the Client 2 classifiers for the three scenarios are represented in Appendix D.1.



(a) Full-aggregation learning



(b) Partial-aggregation learning

Figure 5.4: Accuracy of the client's models at each round using heterogeneous SCDGs.

		Accuracy (%)
Traditional Learning	Client-1	92.05
	Client-2	81.02
	Client-3	89.9
Full-Aggregation Learning	Client-1	87.4
	Client-2	**87.27**
	Client-3	88.22
Partial-Aggregation Learning	Client-1	**93.73**
	Client-2	**87.96**
	Client-3	89.18

Table 5.5: Comparison of Traditional Learning, Full-Aggregation Learning and Partial-Aggregation Learning on DATASET-2 using heterogeneous SCDGs.

5.5 Conclusion

In this chapter, we present a federated learning architecture based on deep learning models for malware classification. We use homomorphic encryption to protect client privacy and compare three different scenarios for model aggregation. We evaluate our solution and show that the federated learning approach provides at least similar performance to a centralized learning approach. Furthermore, our *Partial-Aggregation Learning* scenario represents a promising approach for collaborative learning of malware representations. In the following, we briefly discuss future research directions.

New classifiers In this chapter, our classifier is based on a specific neural network: a graph auto-encoder. While neural networks are commonly used in federated learning [Zha+21], the classifiers proposed in the previous chapters (especially the classifier based on the Weisfeiler-Lehman kernel) could be adapted to our federated learning setup [Xu+21]. We could also explore graph neural networks, which have been successfully applied to malware classification in the literature [Lin+22; Her+22]. Finally, the features used in our models could be extended by considering static and dynamic analysis tools for features extraction [Rey+22].

Semi-Supervised Learning In our architecture, we assume that each client has a labeled dataset to train supervised models. However, it is generally difficult to obtain such a dataset (inaccuracy, time-consuming labeling,...). Semi-supervised learning allows to train models from both labeled and unlabeled data and has been successfully used in Android malware detection [GMD21]. In particular, active learning [Set09] is a semi-supervised learning technique that asks a user to label specific data in order to improve the model. These learning paradigms could be a promising direction of research to limit the need for labeling.

Data heterogeneity In our experiments, we assume that all client datasets come from the same source. This is a strong hypothesis and it would be interesting to consider different sources for each client. Furthermore, we do not consider cleanware, whereas different clients could provide a large data diversity for efficient malware detection. While different data distribution between clients could hinder the learning process [Zha+18], methods have been proposed to address this challenge, such as knowledge distillation [ZHZ21] or new optimisation algorithms [Kar+20].

Robustness The honest-but-curious model is commonly used in the analysis of federated learning systems [PMB14]. However, in the context of mal-

ware analysis, it is important to ensure the security of our system against adversaries with more capabilities [JFK20]. Participants could not follow the protocol [Bha+19] and could even collude with each other [Xie+19] to manipulate the models. Other mechanisms could provide protection against these adversaries, such as a robust distributed gradient algorithm [CL19] or a trust score for local updates [Cao+20].

Conclusion 6

In this thesis, our objective has been to study and improve existing approaches based on symbolic execution and machine learning for malware analysis. While symbolic execution addresses the limitations of static and dynamic malware analysis techniques, it also introduces new challenges such as path explosion or proper modeling of the execution environment. Similarly, machine learning algorithms offer great opportunities to build detection and classification models from malware datasets, but they raise other challenges regarding the use of an appropriate malware representation, the robustness of classifiers, the diversity of the dataset, etc. Our work aimed to propose contributions to address these challenges and improve the automated analysis of malware.

In Chapter 2, we reviewed existing approaches to combine symbolic execution and machine learning in a malware analysis toolchain. We first proposed to improve symbolic exploration with strategies based on uncovered instruction addresses. We then presented different strategies for constructing SCDGs that aim to represent malware behavior. Finally, we showed that existing classifiers neglect parts of the SCDGs extracted from symbolic execution and proposed to use the Weisfeiler-Lehman kernel to overcome this limitation.

In Chapter 3, we introduced SEMA, the implementation of our improved toolchain. After presenting the SEMA modules, we performed experiments to evaluate our different exploration strategies, SCDG strategies and classifiers. While these experiments showed that the Weisfeiler-Lehman kernel combined with CDFS exploration and proper SCDG strategies outperformed previous classifiers, they highlighted the need to adapt the malware representation (*i.e.*, the SCDG) to the classifier.

In Chapter 4, we explored different features and classifiers to detect packing, a common obfuscation strategy. We performed an analysis of the most influential features among our classifiers to better understand their inner workings. We found that features such as the number of standard sections strongly influence the classifier's decision. After comparing the performance of the classifiers, we observed how easily some of them could be bypassed. Indeed, they mistakenly relied too much on features that could be easily altered. This again highlighted the need to evaluate the robustness of machine learning algorithms used for cybersecurity tasks.

Finally, Chapter 5 presented how the federated learning paradigm allows multiple participants to collaboratively train machine learning models for malware classification. This would result in increasing the diversity of the training datasets without sharing them between the participants.

Future work In each chapter, we discussed possibilities for future work to build on the chapter's findings. Some promising directions are highlighted below.

While symbolic execution is helpful to discover malware capabilities, the scalability of our approach remains an important concern. Some preprocessing [Chr+05] of the analyzed binaries, such as unpacking [Che+18], could potentially improve the performance of symbolic execution. Moreover, the development of new exploration techniques based on information related to the execution paths (*e.g.*, number of successors, API call names,...) is an ongoing work that could improve the binary coverage for a given time budget. However, the modeling of the environment is a time-consuming task and becomes very complex when considering process injection or multiple threads. Therefore, an interesting approach would be to further explore the interleaving of symbolic and concrete execution [CYD22] to efficiently guide the exploration towards uncovered code. Other symbolic execution frameworks such as S2E [CKC12], allow full-stack emulation of the operating system and have been successfully applied to malware analysis [Bor+19]. Therefore, adapting our strategies to such a framework could be a promising future work.

Another direction would be to develop hybrid methods that combine classifiers based on static, dynamic, and symbolic analysis. This would require first developing different machine learning classifiers based on these three types of analysis and comparing their performance. Then, by quantifying the uncertainty of these classifiers (*e.g.*, using conformal prediction [Bar+22]), it would be possible to design a robust and efficient malware analysis toolchain. Since the resource costs of each analysis varies, one could first train the classifier based on static features that are easy to extract. If the classifier is uncertain about its classification decision, more advanced features could be extracted to improve its confidence.

While this thesis considers adversarial examples based on static features, our models should also be evaluated with respect to more advanced adversarial examples. Recent research highlights the opportunities and challenges of creating adversarial examples based on dynamic features [Sha+22]. For example, it could be interesting to consider attacks based on the addition/modification of API calls, alteration of program control flow to hinder symbolic analysis, etc. To this end, we are currently investigating dynamic binary lifting [Alt+20] to modify existing binaries by adding specific code (*e.g.*, evasion techniques, "benign" API calls,...). The models should also be evaluated regarding robustness

against temporal and spatial bias [Pen+19].

Finally, exploring the decisions of our classifier with explainability methods could also open up new perspectives. It could allow analysts to learn from our classifiers and improve their other analysis tools (*e.g.*, creating efficient signatures, discovering indicators of compromise,...). It could also be used to identify possible biases/spurious correlations in the classifier's decision process and therefore design attacks [Dem+19] to assess their robustness.

Appendix Chapter 2

A

A.1 gSpan algorithm

Given a dataset of graphs $\mathbb{G}$ and a desired support *min_supp*, gSpan tries to extract all subgraphs that exist in at least *min_supp* graphs of $\mathbb{G}$. If $\mathbb{G}$ represents a set of malware from the same family, the output set of common subgraphs $\mathbb{S}$ represents their signatures. For this purpose, gSpan defines a DFS code of a graph G as an ordered edge sequence constructed from a DFS exploration of G. The original gSpan paper [YH02] defines a way to order these DFS codes, allowing to define a unique minimum DFS code for a graph G ($min(.)$ function in Algorithm 4). Given a DFS code s, c is called a child of s if it expands s with a new edge to create a valid DFS code.

Algorithm 4 gSpan algorithm

```
In: A set of graphs:𝔾
    A minimum support min_supp
Out: A set of common subgraphs 𝕊
Sort labels in 𝔾 by their frequency
Remove infrequent vertices/labels
Relabel remaining vertices and edges
𝕊^1 ← all frequent 1-edge graphs in 𝔾
Sort 𝕊^1 in lexicographical order
𝕊 ← 𝕊^1
for each edge e ∈ 𝕊^1 do
    initialize s with e
    Subgraph_mining(𝔾 ,𝕊, s )
    𝔻 ← 𝔻 − e
    if |𝔾| < min_supp then
        break
Return 𝕊
```

Algorithm 5 *Subgrap_mining* procedure of gSpan algorithm

```
if s ≠ min(s) then
    return;
𝕊 ← 𝕊 ∪ {s}
enumerates s in each graph in 𝔾
and count its children;
for each c; c is a child of s do
    if support(c) ≥ min_supp then
        s ← c
        Subgraph_mining(𝔾 ,𝕊, s )
```

Appendix Chapter 3

B

B.1 GetProcAddress Simprocedure

Malware commonly loads dynamically API calls with `GetProcAddress` to avoid declaring them in the IAT table. It is therefore essential to support dynamic loading in a malware analysis toolchain. The `Simprocedure` implementing this function is depicted in Code B.1.

```
1  class GetProcAddress(angr.SimProcedure):
2      def run(self, lib_handle, name_addr):
3
4          #Module to handle our SimProcedure
5          call_sim = CustomSimProcedure([], [], True,
               True)
6
7          #Name of the requested call
8          name = self.state.mem[name_addr].string.
               concrete
9          proj = self.state.project
10
11         #Check if a symbol exists in the binary
12         symb = proj.loader.find_symbol(str(name))
13
14         #Symbol exists and is already hooked
15         if symb:
16             return symb.rebased_addr
17
18         #Get address of the library and check if it
               was already used
19         lib_addr = self.state.solver.eval(
               lib_handle)
20         if lib_addr in self.state.globals["
               loaded_libs"]:
21             lib = self.state.globals["loaded_libs"
                   ][lib_addr]
22         else:
```

```
23              lib = self.state.mem[lib_handle].string
                    .concrete.decode("utf-8")
24
25          # If lib was not loaded
26          if lib not in SIM_LIBRARIES:
27              proj.loader.requested_names.add(lib)
28              call_sim.loadlibs_proc(call_sim.
                    ddl_loader.load(proj),proj)
29
30          #Allocate an address for the symbol
31          extern = proj.loader.extern_object
32          addr = extern.get_pseudo_addr(name)
33
34          #Link the allocated address to the
                Simprocedure of loaded API
35          if name in call_sim.custom_simproc_windows[
                "custom_package"]:
36              if proj.arch.name == "AMD64":
37                  proj.hook(addr, call_sim.
                        custom_simproc_windows["
                        custom_package"][name](cc=
                        SimCCMicrosoftAMD64(proj.arch))
                        ,)
38              else:
39                  proj.hook(addr, call_sim.
                        custom_simproc_windows["
                        custom_package"][name](cc=
                        SimCCStdcall(proj.arch)),)
40          elif lib in SIM_LIBRARIES:
41              proj.hook_symbol(name, SIM_LIBRARIES[
                    lib].get(name, self.state.arch))
42          else:
43              #Default return value
44              return self.state.solver.BVS("retval_{}
                    ".format(self.display_name), self.
                    arch.bits)
45          return addr
```

Code B.1: Implementation of `GetProcAddress` *Simprocedure*

Appendix Chapter 4

C

C.1 Features extracted for packing detection

The Table C.1 details the 119 static features examined in Chapter 4, along with their respective descriptions. The features are categorized based on their commonality as proposed by Biondi et al. [Bio+19].

ID	Abbreviation	Type	Description
Metadata features			
1	**DLL1**	*Boolean*	DLL can be relocated at load time. *Extracted from DLLs characteristics*
2	**DLL2**	*Boolean*	Code Integrity checks are enforced. *Extracted from DLLs characteristics*
3	**DLL3**	*Boolean*	Image is NX compatible. *Extracted from DLLs characteristics*
4	**DLL4**	*Boolean*	Isolation aware, but do not isolate the image. *Extracted from DLLs characteristics*
5	**DLL5**	*Boolean*	Does not use structured exception (SE) handling. No SE handler may be called in this image. *Extracted from DLLs characteristics*
6	**DLL6**	*Boolean*	Do not bind the image. *Extracted from DLLs characteristics*
7	**DLL7**	*Boolean*	A WDM driver. *Extracted from DLLs characteristics*
8	**DLL8**	*Boolean*	Terminal Server aware. *Extracted from DLLs characteristics*
9	**CHKSUM**	*Integer*	The image file checksum.
10	**ADDR1**	*Integer*	The preferred address of the first byte of image when loaded into memory
11	**ADDR2**	*Integer*	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.

12	**OSVER1**	*Integer*	The major version number of the required operating system.
13	**OSVER2**	*Integer*	The minor version number of the required operating system.
14	**IM_MEM_-SIZE**	*Integer*	The size (in bytes) of the image, including all headers, as the image is loaded in memory.
15	**CODE_SIZE**	*Integer*	The size of the code (.text) section, or the sum of all code sections if there are multiple sections.
16	**COMB_SIZE**	*Integer*	The combined size of an MS DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
17	**INIT_DATA_-SIZE**	*Integer*	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
18	**UNINIT_-DATA_SIZE**	*Integer*	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections
19	**STACKR_-SIZE**	*Integer*	The size of the stack to reserve.
20	**STACKC_-SIZE**	*Integer*	The size of the stack to commit.
21	**ALIGN**	*Integer*	The alignment (in bytes) of sections when they are loaded into memory.
Section features			
22	**STAND_SEC**	*Integer*	The number of standard sections the PE holds
23	**NON_-STAND_SEC**	*Integer*	The number of non-standard sections the PE holds
24	**RATIO_-STAND_SEC**	*Float*	The ratio between the number of standard sections found and the number of all sections found in the PE under analysis
25	**–X_SEC**	*Integer*	The number of Executable sections the PE holds
26	**-W-_SEC**	*Integer*	The number of Writable sections the PE holds
27	**-WX_SEC**	*Integer*	The number of Writable and Executable sections the PE holds

28	**R-X_SEC**	*Integer*	The number of readable and executable sections
29	**RW-_SEC**	*Integer*	The number of readable and writable sections
30	**RWX_SEC**	*Integer*	The number of Writable and Readable and Executable sections the PE holds
31	**CODE_NOT_-X**	*Boolean*	The code section is not executable
32	**X_NOT_-CODE**	*Boolean*	The executable section is not a code section
33	**NOT_CODE**	*Boolean*	The code section is not present in the PE under analysis
34	**EP_NOT_-CODE**	*Boolean*	The entry point is not in the code section
35	**EP_NOT_STD**	*Boolean*	The entry point is not in a standard section
36	**EP_NOT_X**	*Boolean*	The entry point is not in an executable section
37	**RATIO_SEC_-EP**	*Float*	The ratio between raw data and virtual size for the section of the entry point
38	**ZERO_RAW_-SIZE**	*Integer*	The number of sections possessing raw data size zero
39	**VT_GT_RAW**	*Integer*	The number of sections with a virtual size greater than their raw data size.
40	**MAX_RA-TIO_RAW_-VT**	*Float*	The maximum ratio raw data to virtual size among all sections
41	**MIN_RA-TIO_RAW_-VT**	*Float*	the minimum ratio raw data to virtual size among all sections
42	**ADDR_NOT_-ALIGN**	*Boolean*	The address pointing to raw data on disk is not conforming with the file alignment
Entropy features			
43	**ENT_CODE**	*Float* $\in [0;8]$	The byte entropy of code (.text) sections
44	**ENT_DATA**	*Float* $\in [0;8]$	The byte entropy of data section
45	**ENT_RSRC**	*Float* $\in [0;8]$	The byte entropy of resource section

46	**ENT_-HEADER**	*Float* $\in [0;8]$	The byte entropy of PE header
47	**ENT_-GLOBAL**	*Float* $\in [0;8]$	The byte entropy of the entire PE file
48	**ENT_EP_SEC**	*Float* $\in [0;8]$	The byte entropy of the section holding the entry point of the PE under analysis
Entry byte features			
49 - 112	**BYTE1-BYTE64**	*Integer* $\in [0;255]$	Values of 64 bytes following the entry point, each byte correspond to 1 feature position
Import functions features			
113	**DLL_IMP**	*Integer*	The number of DLLs imported
114	**FUNC_IMP_-IDT**	*Integer*	The number of functions imported found in the import directory table (IDT)
115	**MAL_API**	*Integer*	The number of malicious APIs imported (malicious as defined in [ZFA15]). This list includes: "GetProcAddress", "LoadLibraryA", "LoadLibrary", "ExitProcess", "GetModuleHandleA", "VirtualAlloc", "VirtualFree", "GetModuleFileNameA", "CreateFileA", "RegQueryValueExA", "MessageBoxA", "GetCommandLineA", "VirtualProtect", "GetStartupInfoA", "GetStdHandle", "RegOpenKeyExA".
116	**RATIO_-MAL_API**	*Float*	The ratio between the number of malicious APIs imported to the number of all functions imported by the PE
117	**FUNC_IMP_-IAT**	*Integer*	the number of addresses (corresponds to functions) found in the import address table (IAT)
Resource features			
118	**DBG_DIR**	*Boolean*	The debug directory is present or not
119	**NUM_RSRC**	*Integer*	The number of resources the PE holds

Table C.1: Features description.

C.2 Comparison of related works

The Table C.2 presents a detailed comparison of existing approaches to packing detection.

Paper	Algorithm	Features	Comments
Lyda et al. [LH07]	Bintropy with two modes of operation : Entropy of the whole file or inspection of individual entropy section.	Global entropy	They used cleanware not packed and malware packed obtained from a honeypot. Their global entropy threshold of 6.67 allows a 3.8% false positive rate and 0.5% false negative rate. It was not assessed on other datasets. They study also section names with a high entropy on a larger dataset.
Choi et al. [Cho+08]	PHAD, an algorithm based on euclidian distance and custom threshold	8 features related to PE structure.	Their dataset of benign and malicious files is manually analyzed. They used a threshold, estimated on the dataset, to consider a file as packed.

Perdisci et al. [PLL08a] (1)	Naive Bayes Classifier, decision tree (J48 with and without bagging), KNN, Multilayer Perceptron and entropy threshold	9 features : number of standard and non-standard sections; number of executable sections; number of readable, writable and executable sections; number of entries in IAT; PE header entropy; code sections entropy; data sections entropy and entire PE file entropy.	Their dataset is composed of benign non-packed executables, self-packed executables (on benign files) and packed malware from MALFEASE project.
Perdisci et al. [PLL08b] (2)	Bagged decision tree and multilayer perceptrons	Same as [PLL08a] and n-grams (with $n = 2$ or $n = 3$)	Their dataset is made of malware from MALFEASE project as well as benign files (self-packed). Their ground truth is established thanks to know detector such as PEiD. Their objective is packing detection, extracting hidden code and detection of malicious code.

Han and Lee [HLL09]	Check entry-point section properties and entropy threshold.	Entrypoint section writable and entry-point section entropy.	They used a balanced dataset of 400 executables (200 cleanware from windows installation and 200 malware from a honeypot) and an entropy threshold of 6.58.
Shafiq et al. [Sha+09]	KNN, decision trees, Naive Bayes, RIPPER (a rule-based classification algorithm) and SVM	189 features related to structural informations of PE files.	They conducted several experiments to detect malicious files, including the detection of malicious files among benign files, whether they were packed or not. A file is considered packed if it is identified as such by PEiD or Protection ID.
Treadwell et al. [TZ09]	Threshold related to a risk score (weighted sum of features)	8 Features : non-standard section names, flagged section names, entry-point not in code section, TLS section, DLL with no export, low number of imports, abnormal ordinals in import table and bad PE format.	Their dataset is made of packed files (malware) and non-packed files (cleanware and malware). PEiD is used to decide packed status. Their initial goal is to detect obfuscated malware.

Santos et al. [San+11]	KNN, decision trees, random forest and variants of tree based models.	209 structural features of PE files (PE header, sections characteristics, EP section features and entropy)	They used benign and malicious executables from VxHeavens. Some samples are self-packed while others from Zeus malware family are known to be packed with custom packer (not detected with PEiD).
Devi et al. [DN12]	Feature representation after manual analysis	After manual analysis, focus on 4 features : global entropy, size of uninitialized data, header size and raw data size	They focus on feature changes before and after self-packing with UPX on cleanware and malware.
Arora et al. [Aro+13]	Threshold based on score obtained thanks to several heuristics and power distance.	14 features, the five more important being : presence of specific section names, code section not executable, number of section writable and executable, data section executable and IAT in non-standard section.	They used benign non-packed and malicious packed/non-packed samples from windows, honeypot and websites. They observe some features need to be adapted for specific files (DLL, PE built with debug mode and PE built with CLR).

Ugarte-Pedrero et al. [Uga+14]	Bayesian networks, SVM, random forest, MLP and rules based classifiers	Same as [San+11] and n-grams.	Their approach focus on anomaly detection to detect packed samples. The dataset is made of packed samples from Zeus family and manually packed samples (detected as non-packed by PEiD). For the non-packed samples, PEiD is used as well as custom heuristics to build the dataset.
Biondi et al. [Bio+19]	Naive Bayes Gaussian Classifier, Decision Tree, Random Forest and Extra Trees	The 119 features included also in this work	Dataset obtained from Cisco and labeled thanks to several tools : PEiD, YARA rules of VirusTotal and custom engine from Cisco. It highligts the necessity to favor dataset with more labels (even if there is non consensus over them) over reliable labels (with a high consensus). A simple economical analysis is performed, showing decision tree are a good trade-off to balance training cost and efficiency.

Kim et al. [Kim+20]	KNN, logistic regression, linear SVM, Random Forest, MLP, Nayve Bayes, SVM and decision tree.	13 features : matching a known packing signature, entrypoint section characteristics, file entropy and sections characteristics.	Their dataset is labeled thanks to author's previous work. A file is considered packed if : no entrypoint section, matching of known signature, entrypoint section is writeable and entropy within a specific range.
Bergenholtz et al. [Ber+20]	Long short term memory (LSTM) neural network	The first 128 x86 instructions of the sample.	Samples are cleanware from a fresh windows installation packed with several metamorphic packers.
Mantovani et al. [Man+20]	SVM, MLP and RF	Features extracted from several other papers related to PE structure, heuristics, N-grams, Opcodes, statistical features and entropy features.	Malicious binaries from VirusTotal for packed and non-packed samples. To build a ground truth, they develop a tool based on PANDA, inspect and compare their results with PEiD, Manalyze and DIE. The focus of the paper is to illustrate how models does not performs well to detect low entropy packing scheme.

Aghakhani et al. [Agh+20]	SVM, MLP and RF	56,543 features : PE headers, PE sections, DLL & API imports, rich header, Byte & Opcode N-gram and strings.	They used a dataset from AV vendor and EMBER labeled thanks to vendors and VirusTotal. Authors illustrate the current overlap between packing and malware detection when based only on static features.

Table C.2: Summary of related works

Appendix Chapter 5

D

D.1 Confusion matrix

The confusion matrix of the classifiers in the homogeneous SCDG setup with Dataset-1 are shown in Figure D.1, Figure D.2 and Figure D.3. The confusion matrix of the classifiers in the heterogeneous SCDG setup with Dataset-2 are shown in Figure D.4, Figure D.5 and Figure D.6.

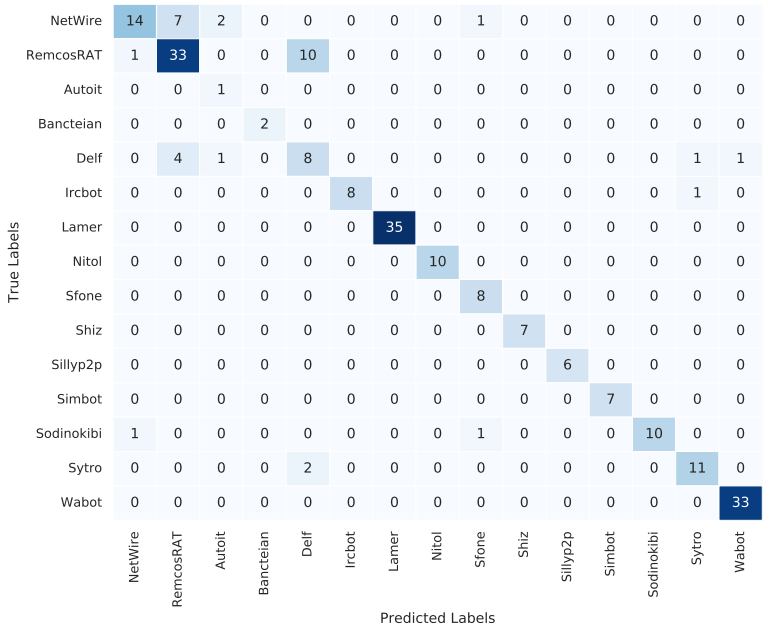


Figure D.1: Confusion matrix on Dataset-1 with *Centralized Learning*

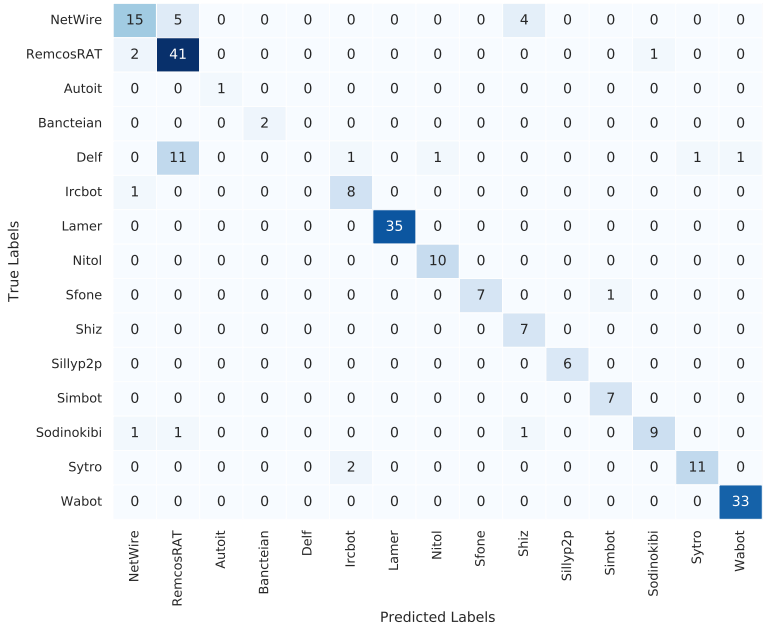


Figure D.2: Confusion matrix on Dataset-1 with *Full-Aggregation Learning*

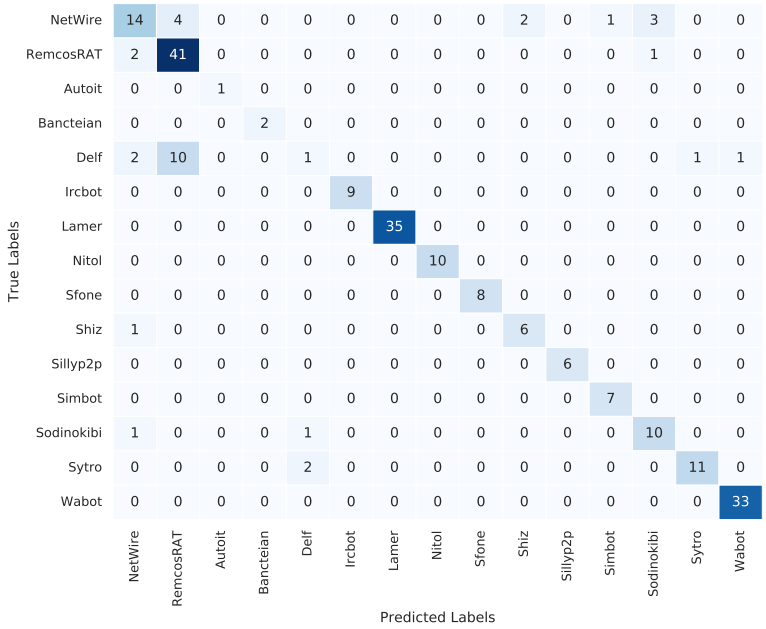


Figure D.3: Confusion matrix on Dataset-1 with *Partial-Aggregation Learning* at Client-1

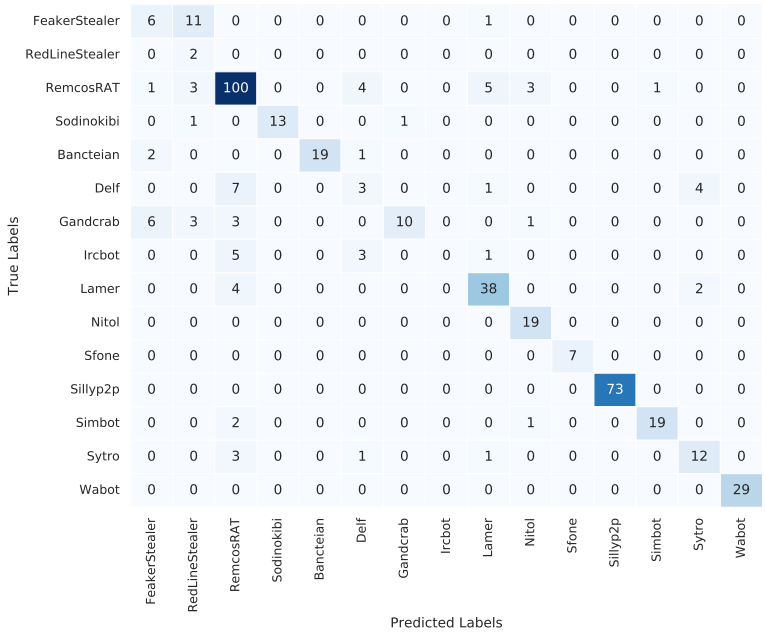


Figure D.4: Confusion matrix of Client 2 classifiers on DATASET-2 with *Traditional Learning*

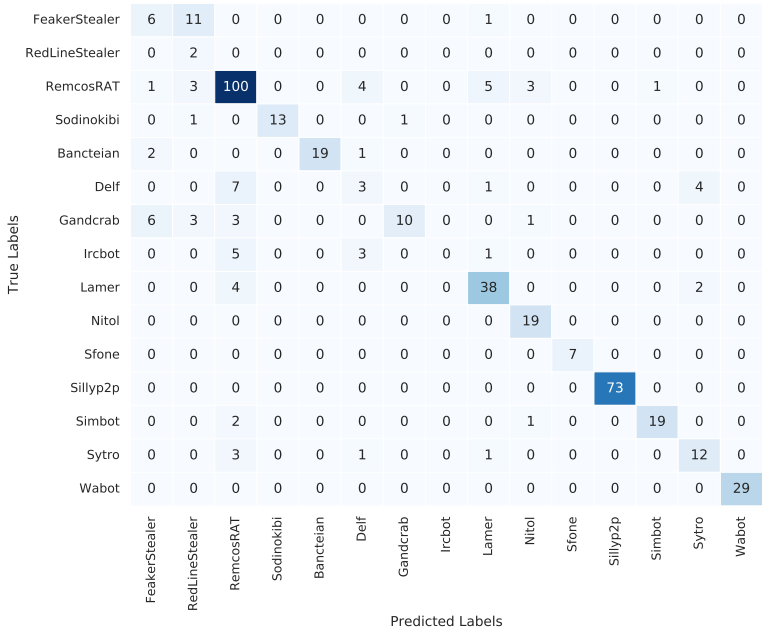


Figure D.5: Confusion matrix of Client 2 classifiers on DATASET-2 with *Full-Aggregation Learning*

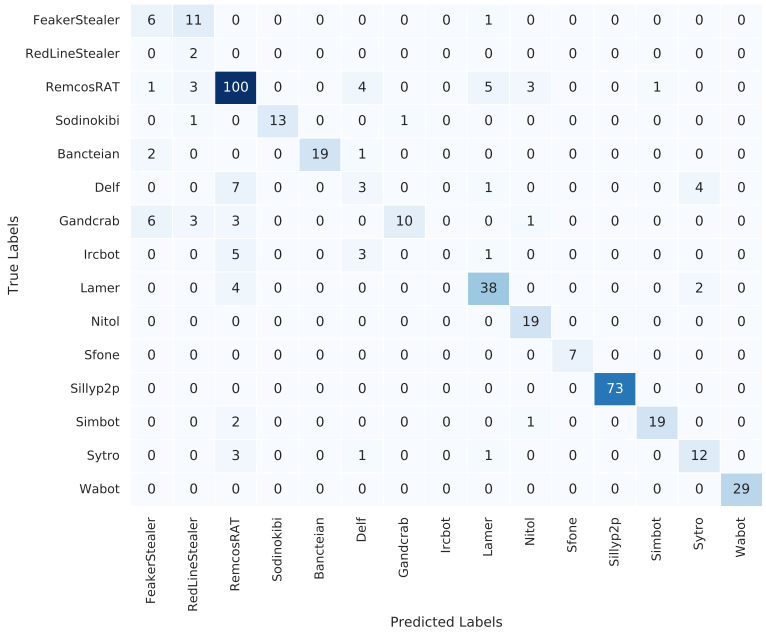


Figure D.6: Confusion matrix of Client 2 classifiers on DATASET-2 with *Partial-Aggregation Learning*

Bibliography

[Aba+16] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep learning with differential privacy". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 308–318.

[Abu23] Abuse.ch. *MalwareBazaar*. 2023.

[Afi+19] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste. "Malware dynamic analysis evasion techniques: A survey". In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–28.

[Agh+20] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel. "When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features". In: *Network and Distributed Systems Security (NDSS) Symposium*. 2020.

[Ahm+13] M. Ahmadi, A. Sami, H. Rahimi, and B. Yadegari. "Malware detection by behavioural sequential patterns". In: *Computer Fraud & Security* 2013.8 (2013), pp. 11–19.

[Ahm+16] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. "Novel feature extraction, selection and fusion for effective malware family classification". In: *Proceedings of the sixth ACM conference on data and application security and privacy*. 2016, pp. 183–194.

[Alt+20] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, et al. "BinRec: dynamic binary lifting and recompilation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[Ama] G. Amato. *peframe version 6.0.3*. `https://github.com/guelfoweb/peframe`. (Visited on 11/01/2019).

[And+11] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. "Graph-based malware detection using dynamic analysis". In: *Journal in computer Virology* 7 (2011), pp. 247–258.

[Ank+99] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. "OPTICS: Ordering points to identify the clustering structure". In: *ACM Sigmod record* 28.2 (1999), pp. 49–60.

[Aon+16] Y. Aono, T. Hayashi, L. Trieu Phong, and L. Wang. "Scalable and Secure Logistic Regression via Homomorphic Encryption". In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. 2016, pp. 142–144.

[Aon+23] S. Aonzo, Y. Han, A. Mantovani, and D. Balzarotti. "Humans vs. machines in malware classification". In: *Proc. of USENIX-23* (2023).

[AR18] H. S. Anderson and P. Roth. "Ember: an open dataset for training static pe malware machine learning models". In: *arXiv preprint arXiv:1804.04637* (2018).

[Aro+13] R. Arora, A. Singh, H. Pareek, and U. R. Edara. "A heuristics-based static analysis approach for detecting packed pe binaries". In: *International Journal of Security and Its Applications* 7.5 (2013), pp. 257–268.

[Arp+22] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. "Dos and don'ts of machine learning in computer security". In: *Proc. of the USENIX Security Symposium*. 2022.

[ASL12] B. Anderson, C. Storlie, and T. Lane. "Improving malware classification: bridging the static/dynamic gap". In: *Proceedings of the 5th ACM workshop on Security and artificial intelligence*. 2012, pp. 3–14.

[Asq16] M. Asquith. "Extremely scalable storage and clustering of malware metadata". In: *Journal of Computer Virology and Hacking Techniques* 12.2 (2016), pp. 49–58.

[Avg+14a] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. "Automatic exploit generation". In: *Communications of the ACM* 57.2 (2014), pp. 74–84.

[Avg+14b] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing symbolic execution with veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 1083–1094.

[Avl+21] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitras. "When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World." In: *USENIX Security Symposium*. 2021, pp. 3487–3504.

[AZ20] E. Amer and I. Zelinka. "A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence". In: *Computers & Security* 92 (2020), p. 101760.

[AZE21] E. Amer, I. Zelinka, and S. El-Sappagh. "A multi-perspective malware detection approach through behavioral fusion of api call sequence". In: *Computers & Security* 110 (2021), p. 102449.

[Bal+17] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu. "Assisting malware analysis with symbolic execution: A case study". In: *ICCL*. 2017, pp. 171–188.

[Bal+18] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. "A survey of symbolic execution techniques". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[Bar+22] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro. "Transcending transcend: Revisiting malware classification in the presence of concept drift". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 805–823.

[Bat+13] M. Bat-Erdene, T. Kim, H. Li, and H. Lee. "Dynamic classification of packing algorithms for inspecting executables using entropy analysis". In: *2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*. IEEE. 2013, pp. 19–26.

[BCD22] L. Borzacchiello, E. Coppa, and C. Demetrescu. "SENinja: A symbolic execution plugin for Binary Ninja". In: *SoftwareX* 20 (2022), p. 101219.

[BCE08] P. Boonstoppel, C. Cadar, and D. Engler. "RWset: Attacking path explosion in constraint-based test generation". In: *TACAS*. Springer. 2008, pp. 351–366.

[BDL22] C.-H. Bertrand Van Ouytsel, K. H. T. Dam, and A. Legay. "Symbolic Analysis Meets Federated Learning to Enhance Malware Identifier". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security ARES*. 2022, pp. 1–10.

[BDL23] C.-H. Bertrand Van Ouytsel, K. H. T. Dam, and A. Legay. "Analysis of machine learning approaches to packing detection". In: *Computers & Security* (2023), p. 103536.

[Bek+15] D. Bekerman, B. Shapira, L. Rokach, and A. Bar. "Unknown malware detection using network traffic classification". In: *2015 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2015, pp. 134–142.

[Ber+] C.-H. Bertrand Van Ouytsel, C. Crochet, K. H. T. Dam, and A. Legay. *SEMA*. `https://github.com/csvl/SEMA-ToolChain`.

[Ber+20] E. Bergenholtz, E. Casalicchio, D. Ilie, and A. Moss. "Detection of metamorphic malware packers using multilayered LSTM networks". In: *International Conference on Information and Communications Security*. Springer. 2020, pp. 36–53.

[Ber+21] C.-H. Bertrand Van Ouytsel, O. Bronchain, G. Cassiers, and F.-X. Standaert. "How to fool a black box machine learning based side-channel security evaluation". In: *Cryptography and Communications* 13.4 (2021), pp. 573–585.

[Ber+23] C.-H. Bertrand Van Ouytsel, C. Crochet, K. H. T. Dam, and A. Legay. "Tool Paper-SEMA: Symbolic Execution Toolchain for Malware Analysis". In: *Risks and Security of Internet and Systems: 17th International Conference, CRiSIS 2022, Revised Selected Papers*. Springer. 2023, pp. 62–68.

[Bha+19] A. N. Bhagoji, S. Chakraborty, P. Mittal, and S. Calo. "Analyzing federated learning through an adversarial lens". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 634–643.

[Big+13] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. "Evasion attacks against machine learning at test time". In: *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Proceedings, Part III 13*. Springer. 2013, pp. 387–402.

[Bio+18] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf. "Tutorial: An overview of malware detection and evasion techniques". In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, (ISoLA 2018)*. Springer. 2018, pp. 565–586.

[Bio+19] F. Biondi, M. A. Enescu, T. Given-Wilson, A. Legay, L. Noureddine, and V. Verma. "Effective, efficient, and robust packing detection and classification". In: *Computers & Security* 85 (2019), pp. 436–451. ISSN: 0167-4048.

[BJH13] U. Baldangombo, N. Jambaljav, and S.-J. Horng. "A Static Malware Detection System Using Data Mining Methods". In: (Aug. 13, 2013).

[BL21] C.-H. Bertrand Van Ouytsel and A. Legay. "Detection and classification of malware based on symbolic execution and machine learning methods". In: *Cybersec & AI* (2021).

[BL23] C.-H. Bertrand Van Ouytsel and A. Legay. "Malware Analysis with Symbolic Execution and Graph Kernel". In: *Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings*. Springer. 2023, pp. 292–310.

[BN06] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[Boe+23] F. Boenisch, A. Dziedzic, R. Schuster, A. S. Shamsabadi, I. Shumailov, and N. Papernot. "When the curious abandon honesty: Federated learning is not private". In: *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2023, pp. 175–199.

[Boj+17] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. "Enriching word vectors with subword information". In: *Transactions of the association for computational linguistics* 5 (2017), pp. 135–146.

[Bom+20] A. Bommert, X. Sun, B. Bischl, J. Rahnenführer, and M. Lang. "Benchmark for filter methods for feature selection in high-dimensional classification data". In: *Computational Statistics & Data Analysis* 143 (2020), p. 106839.

[Bor+19] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu. "Reconstructing C2 servers for remote access trojans with symbolic execution". In: *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer. 2019, pp. 121–140.

[Bru+08] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. "Automatically identifying trigger-based behavior in malware". In: *Botnet Detection*. 2008.

[Bur+10] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. "SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics". In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. USA: USENIX Association, 2010, p. 15.

[Cao+20] X. Cao, M. Fang, J. Liu, and N. Z. Gong. "Fltrust: Byzantine-robust federated learning via trust bootstrapping". In: *arXiv preprint arXiv:2012.13995* (2020).

[Car+19] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. Goodfellow, A. Madry, and A. Kurakin. "On evaluating adversarial robustness". In: *arXiv preprint arXiv:1902.06705* (2019).

[Cav+23] L. Cavallaro, J. Kinder, F. Pendlebury, and F. Pierazzi. "Are Machine Learning Models for Malware Detection Ready for Prime Time?" In: *IEEE Security & Privacy* 21.2 (2023), pp. 53–56.

[CDE+08] C. Cadar, D. Dunbar, D. R. Engler, et al. "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[Cha+12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. "Unleashing mayhem on binary code". In: *2012 IEEE Symposium on Security and Privacy*. IEEE. 2012, pp. 380–394.

[Che+18] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J. Marion. "Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*. ACM, 2018, pp. 395–411.

[Cho+08] Y.-s. Choi, I.-k. Kim, J.-t. Oh, and J.-c. Ryou. "PE file header analysis-based packed PE file detection technique (PHAD)". In: *International Symposium on Computer Science and its Applications*. IEEE. 2008, pp. 28–31.

[Cho+21] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi. "EVA Improved: Compiler and Extension Library for CKKS". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2021, pp. 43–55.

[Chr+05] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith. *Malware normalization*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 2005.

[CJK07] M. Christodorescu, S. Jha, and C. Kruegel. "Mining specifications of malicious behavior". In: *FSE*. 2007, pp. 5–14.

[CKC12] V. Chipounov, V. Kuznetsov, and G. Candea. "The S2E platform: Design, implementation, and applications". In: *ACM Transactions on Computer Systems (TOCS)* 30.1 (2012), pp. 1–49.

[CL19] X. Cao and L. Lai. "Distributed gradient descent algorithm robust to an arbitrary number of byzantine attackers". In: *IEEE Transactions on Signal Processing* 67.22 (2019), pp. 5850–5864.

[Cla] ClamAV. *ClamAVNet*. `https://www.clamav.net`. November 2019.

[Coz+18] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. "Understanding linux malware". In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 161–175.

[Coz+20] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti. "The tangled genealogy of IoT malware". In: *Annual Computer Security Applications Conference*. 2020, pp. 1–16.

[CRT18] Y.-R. Chen, A. Rezapour, and W.-G. Tzeng. "Privacy-preserving ridge regression on distributed data". In: *Information Sciences* 451 (2018), pp. 34–49.

[CTC18] A. Calleja, J. Tapiador, and J. Caballero. "The malsource dataset: Quantifying complexity and code reuse in malware development". In: *IEEE Transactions on Information Forensics and Security* 14.12 (2018), pp. 3175–3190.

[CW17] N. Carlini and D. Wagner. "Towards evaluating the robustness of neural networks". In: *2017 ieee symposium on security and privacy (sp)*. Ieee. 2017, pp. 39–57.

[CX11] S. Cesare and Y. Xiang. "Malware variant detection using similarity search over sets of control flow graphs". In: *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE. 2011, pp. 181–189.

[CYD22] E. Coppa, H. Yin, and C. Demetrescu. "Symfusion: hybrid instrumentation for concolic execution". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–12.

[Dam+22] K. H. T. Dam, T. Given-Wilson, A. Legay, and R. Veroneze. "Packer classification based on association rule mining". In: *Applied Soft Computing* 127 (2022), p. 109373.

[Dam+23] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge. "Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 60–74.

[DBL22a] A. D'Hondt, C.-H. Bertrand Van Ouytsel, and A. Legay. "PackingBox: Playing with Executable Packing". In: *BlackHat Europe*. 2022.

[DBL22b] A. D'Hont, C.-H. Bertrand Van Ouytsel, and A. Legay. *Packing Box : Study executable packing easy with this dedicated platform.* `https://github.com/packing-box/docker-packing-box`. 2022.

[DBL23] A. D'Hondt, C.-H. Bertrand Van Ouytsel, and A. Legay. "Experimental Toolkit for Manipulating Executable Packing". In: *Risks and Security of Internet and Systems: 18th International Conference, CRiSIS 2023, To appear*. Springer. 2023.

[Dem+19] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando. "Explaining vulnerabilities of deep learning to adversarial malware binaries". In: *arXiv preprint arXiv:1901.03583* (2019).

[Dem+21] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli. "Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection". In: *ACM Transactions on Privacy and Security (TOPS)* 24.4 (2021), pp. 1–31.

[DGL21] K. H. T. Dam, T. Given-Wilson, and A. Legay. "Unsupervised behavioural mining and clustering for malware family identification". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 374–383.

[DHo+23] A. D'Hondt, R. Jennes, S. Martinez Balbuena, C.-H. Bertrand Van Ouytsel, and A. Legay. "Packing-Box: Breaking Detectors and Visualizing Packing". In: *BlackHat Europe*. 2023.

[DN12] D. Devi and S. Nandi. "Pe file features in detection of packed executables". In: *International Journal of Computer Theory and Engineering* 4.3 (2012), p. 476.

[ED07] D. Engler and D. Dunbar. "Under-constrained execution: making automatic code destruction easy and scalable". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 1–4.

[ENI23] ENISA. *ENISA Threat Landscape report*. 2023. URL: `https://www.enisa.europa.eu/publications/enisa-threat-landscape-2023`.

[Est+96] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise". In: *kdd*. Vol. 96. 34. 1996, pp. 226–231.

[Far+15] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor. "AndroSimilar: Robust signature for detecting variants of Android malware". In: *J. Inf. Secur. Appl.* 22 (2015), pp. 66–80.

[Fer08] P. Ferrie. "Anti-unpacker tricks–part one". In: *Virus Bulletin* 4 (2008).

[Geh+23] T. Gehlhar, F. Marx, T. Schneider, A. Suresh, T. Wehrle, and H. Yalame. "SAFEFL: MPC-friendly Framework for Private and Robust Federated Learning". In: *Cryptology ePrint Archive* (2023).

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed automated random testing". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[GMD21] R. Gálvez, V. Moonsamy, and C. Diaz. "Less is More: A privacy-respecting Android malware classifier using federated learning". In: *Proceedings on Privacy Enhancing Technologies* 4 (2021), pp. 96–116.

[GR22] B. Ghimire and D. B. Rawat. "Recent Advances on Federated Learning for Cybersecurity and Cybersecurity for Federated Learning for Internet of Things". In: *IEEE Internet of Things Journal* (2022).

[Gri+20] F. Gritti, L. Fontana, E. Gustafson, F. Pagani, A. Continella, C. Kruegel, and G. Vigna. "Symbion: Interleaving symbolic with concrete execution". In: *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE. 2020, pp. 1–10.

[GSS15] M. Ghiasi, A. Sami, and Z. Salehi. "Dynamic VSA: a framework for malware detection based on register contents". In: *Engineering Applications of Artificial Intelligence* 44 (2015), pp. 111–122.

[GTG07] M. N. Gagnon, S. Taylor, and A. K. Ghosh. "Software protection through anti-debugging". In: *IEEE Security & Privacy* 5.3 (2007), pp. 82–84.

[He+21] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev. "Learning to explore paths for symbolic execution". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 2526–2540.

[Her+22] J. D. Herath, P. P. Wakodikar, P. Yang, and G. Yan. "Cfgexplainer: Explaining graph neural network-based malware classification from control flow graphs". In: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2022, pp. 172–184.

[HLL09] S. Han, K. Lee, and S. Lee. "Packed PE file detection for malware forensics". In: *2009 2nd International Conference on Computer Science and Its Applications, CSA 2009*. 2009, p. 5404211.

[Hof06] M. Hofmann. "Support vector machines-kernels and the kernel trick". In: *Notes* 26.3 (2006), pp. 1–16.

[hor] horsicq. *Detect-It-Easy version 2.06*. `https://github.com/horsicq/Detect-It-Easy`. (Visited on 11/01/2019).

[Hsu+20] R.-H. Hsu, Y.-C. Wang, C.-I. Fan, B. Sun, T. Ban, T. Takahashi, T.-W. Wu, and S.-W. Kao. "A Privacy-Preserving Federated Learning System for Android Malware Detection Based on Edge Computing". In: *2020 15th Asia Joint Conference on Information Security (AsiaJCIS)*. 2020, pp. 128–136.

[HWC13] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*. Vol. 751. John Wiley & Sons, 2013.

[INR] INRIA. *quickSpan, A Fast, Robust gSpan Implementation*.

[Isl+13] R. Islam, R. Tian, L. M. Batten, and S. Versteeg. "Classification of malware based on integrated static and dynamic features". In: *Journal of Network and Computer Applications* 36.2 (2013), pp. 646–656.

[Jam+18] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore, and G. R. K. Rao. "Dynamic malware analysis using cuckoo sandbox". In: *ICICCT*. IEEE. 2018, pp. 1056–1060.

[Jeo+10] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee. "Generic unpacking using entropy analysis". In: *2010 5th International Conference on Malicious and Unwanted Software*. IEEE. 2010, pp. 98–105.

[JFK20] M. S. Jere, T. Farnan, and F. Koushanfar. "A taxonomy of attacks on federated learning". In: *IEEE Security & Privacy* 19.2 (2020), pp. 20–28.

[Kal+18] M. Kalash, M. Rochan, N. Mohammed, N. D. Bruce, Y. Wang, and F. Iqbal. "Malware classification with deep convolutional neural networks". In: *2018 9th IFIP international conference on new technologies, mobility and security (NTMS)*. IEEE. 2018, pp. 1–5.

[Kan20] C. Kanipe. *Modality*. `https://github.com/0xchase/modality`. 2020. (Visited on 10/28/2023).

[Kar+20] S. P. Karimireddy, S. Kale, M. Mohri, S. Reddi, S. Stich, and A. T. Suresh. "Scaffold: Stochastic controlled averaging for federated learning". In: *International conference on machine learning*. PMLR. 2020, pp. 5132–5143.

[Kas21] KasperskyLab. *Machine Learning Methods for Malware Detection*. 2021. URL: `https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf`.

[KB14] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Kho+15] P. Khodamoradi, M. Fazlali, F. Mardukhi, and M. Nosrati. "Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms". In: *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*. IEEE. 2015, pp. 1–6.

[Kim+20] J.-W. Kim, J. Namgung, Y.-S. Moon, and M.-J. Choi. "Experimental Comparison of Machine Learning Models in Malware Packing Detection". In: *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE. 2020, pp. 377–380.

[Kin76] J. C. King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[KKK15] Y. Ki, E. Kim, and H. K. Kim. "A novel approach to detect malware based on API call sequence analysis". In: *International Journal of Distributed Sensor Networks* 11.6 (2015), p. 659101.

[Kon+16] J. Konevcny, H. B. McMahan, D. Ramage, and P. Richtárik. "Federated optimization: Distributed machine learning for on-device intelligence". In: *arXiv preprint arXiv:1610.02527* (2016).

[KPP20] A. Kim, A. Papadimitriou, and Y. Polyakov. "Approximate homomorphic encryption with reduced approximation error". In: *Cryptology ePrint Archive* (2020).

[KPY07] M. G. Kang, P. Poosankam, and H. Yin. "Renovo: A hidden code extractor for packed executables". In: *Proceedings of the 2007 ACM workshop on Recurring malcode*. 2007, pp. 46–53.

[Küc+21] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti. "Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes." In: *NDSS*. 2021.

[Kuz+12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. "Efficient state merging in symbolic execution". In: *Acm Sigplan Notices* 47.6 (2012), pp. 193–204.

[KV15] D. Kirat and G. Vigna. "MalGene: Automatic Extraction of Malware Analysis Evasion Signature". In: *SIGSAC*. Ed. by I. Ray, N. Li, and C. Kruegel. ACM, 2015, pp. 769–780.

[KVK11] D. Kirat, G. Vigna, and C. Kruegel. "Barebox: efficient malware analysis on bare-metal". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, pp. 403–412.

[KW18] V. Kotov and M. Wojnowicz. "Towards generic deobfuscation of windows API calls". In: *arXiv preprint arXiv:1802.04466* (2018).

[Kwi] I. Kwiatkowski. *Manalyze*. `https://github.com/JusticeRage/Manalyze`. (Visited on 11/01/2019).

[LH07] R. Lyda and J. Hamrock. "Using Entropy Analysis to Find Encrypted and Packed Malware". In: *IEEE Security & Privacy* 5.2 (2007), pp. 40–45.

[LH20] K.-Y. Lin and W.-R. Huang. "Using federated learning on malware classification". In: *2020 22nd International Conference on Advanced Communication Technology (ICACT)*. IEEE. 2020, pp. 585–589.

[Lin+15] C.-T. Lin, N.-J. Wang, H. Xiao, and C. Eckert. "Feature selection and extraction for malware classification." In: *J. Inf. Sci. Eng.* 31.3 (2015), pp. 965–992.

[Lin+22] X. Ling, L. Wu, W. Deng, Z. Qu, J. Zhang, S. Zhang, T. Ma, B. Wang, C. Wu, and S. Ji. "Malgraph: Hierarchical graph neural networks for robust windows malware detection". In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 1998–2007.

[Lin+23] X. Ling, L. Wu, J. Zhang, Z. Qu, W. Deng, X. Chen, Y. Qian, C. Wu, S. Ji, T. Luo, et al. "Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art". In: *Computers & Security* (2023), p. 103134.

[Lin17] Y. Lin. "Symbolic Execution with Over-approximation". PhD thesis. University of Melbourne, Parkville, Victoria, Australia, 2017.

[Luc+23] S. Lucca, C. Crochet, C.-H. Bertrand Van Ouytsel, and A. Legay. "On Exploiting Symbolic Execution to Improve the Analysis of RAT Samples with angr". In: *Foundations and Practice of Security: 16th International Symposium, FPS 2023, Revised Selected Papers*. Springer. 2023.

[Ma+11] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. "Directed symbolic execution". In: *Static Analysis: 18th International Symposium SAS, Proceedings 18*. Springer. 2011, pp. 95–111.

[MAM15] A. Mohaisen, O. Alrawi, and M. Mohaisen. "AMAL: high-fidelity, behavior-based automated malware analysis and classification". In: *computers & security* 52 (2015), pp. 251–266.

[Man+20] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti. "Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem." In: *NDSS*. 2020.

[Man+23] M. Mansouri, M. Onen, W. B. Jaballah, and M. Conti. "Sok: Secure aggregation based on cryptographic schemes for federated learning". In: *Proc. Priv. Enhancing Technol* 1 (2023), pp. 140–157.

[Mas+12] F. Massicotte, M. Couture, H. Normandin, and F. Michaud. "A testing model for dynamic malware analysis systems". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 826–833.

[MBN09] T. Mandl, U. Bayer, and F. Nentwich. "ANUBIS ANalyzing unknown BInarieS the automatic way". In: *Virus bulletin conference*. Vol. 1. 2009, p. 02.

[McM+17] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas. "Communication-efficient learning of deep networks from decentralized data". In: *Artificial intelligence and statistics*. PMLR. 2017, pp. 1273–1282.

[ME20] W. E. Marcilio and D. M. Eler. "From explanations to feature selection: assessing SHAP values as feature selection mechanism". In: *2020 33rd SIBGRAPI conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE. 2020, pp. 340–347.

[Mel+19] L. Melis, C. Song, E. De Cristofaro, and V. Shmatikov. "Exploiting unintended feature leakage in collaborative learning". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 691–706.

[Mica] Microsoft. *PE Format (Microsoft Docs)*. URL: `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format` (visited on 06/06/2023).

[Micb] Microsoft. *Programming reference for the Win32 API*. URL: `https://learn.microsoft.com/en-us/windows/win32/api/` (visited on 02/21/2023).

[Min+15] J. Ming, D. Xu, L. Wang, and D. Wu. "Loop: Logic-oriented opaque predicate detection in obfuscated binary code". In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 2015, pp. 757–768.

[MKK07] A. Moser, C. Kruegel, and E. Kirda. "Limits of static analysis for malware detection". In: *ACSAC*. IEEE. 2007, pp. 421–430.

[Mol20] C. Molnar. *Interpretable machine learning*. Lulu. com, 2020.

[Mur+22] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim. "File Packing from the Malware Perspective: Techniques, Analysis Approaches, and Directions for Enhancements". In: *ACM Computing Surveys* 55.5 (2022), pp. 1–45.

[Nal20] Nalen98. *AngryGhidra*. `https://github.com/Nalen98/AngryGhidra`. 2020. (Visited on 10/28/2023).

[Nas+18] S. Naseer, Y. Saleem, S. Khalid, M. K. Bashir, J. Han, M. M. Iqbal, and K. Han. "Enhanced network anomaly detection based on deep neural networks". In: *IEEE access* 6 (2018), pp. 48231–48246.

[Nat+11] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. "Malware images: visualization and automatic classification". In: *Proceedings of the 8th international symposium on visualization for cyber security*. 2011, pp. 1–7.

[Ngu+19] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A. Sadeghi. "DÏoT: A Federated Self-learning Anomaly Detection System for IoT". In: *39th IEEE International Conference on Distributed Computing Systems, ICDCS*. 2019, pp. 756–767.

[Nik+13] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. "Privacy-preserving ridge regression on hundreds of millions of records". In: *2013 IEEE symposium on security and privacy*. IEEE. 2013, pp. 334–348.

[Nou+21] L. Noureddine, A. Heuser, C. Puodzius, and O. Zendra. "SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution". In: *CODASPY'21: Eleventh ACM Conference on Data and Application Security and Privacy*. 2021.

[NP17] S. D. Nikolopoulos and I. Polenakis. "A graph-based model for malware detection and classification using system-call groups". In: *Journal of Computer Virology and Hacking Techniques* 13.1 (2017), pp. 29–46.

[NV21] G. Nikolentzos and M. Vazirgiannis. "Learning Structural Node Representations Using Graph Kernels". In: *IEEE Trans. Knowl. Data Eng.* 33.5 (2021), pp. 2045–2056.

[Or-+19] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach. "Dynamic malware analysis in the modern era—A state of the art survey". In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–48.

[Pal+13] S. Palahan, D. Babić, S. Chaudhuri, and D. Kifer. "Extraction of statistically significant malware behaviors". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. 2013, pp. 69–78.

[Par+21] K. Park, B. Sahin, Y. Chen, J. Zhao, E. Downing, H. Hu, and W. Lee. "Identifying behavior dispatchers for malware analysis". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 2021, pp. 759–773.

[Pas+19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in neural information processing systems* 32 (2019).

[Ped+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PEI] PEID. *PEiD, version 0.95*. `https://appnee.com/peid/`. (Visited on 11/01/2019).

[Pen+19] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. "{TESSERACT}: Eliminating experimental bias in malware classification across space and time". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 729–746.

[Pho+18] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai. "Privacy-Preserving Deep Learning via Additively Homomorphic Encryption". In: *IEEE Transactions on Information Forensics and Security* 13.5 (2018), pp. 1333–1345.

[Pie+20] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. "Intriguing properties of adversarial ml attacks in the problem space". In: *2020 IEEE symposium on security and privacy (SP)*. IEEE. 2020, pp. 1332–1349.

[Pit+19] N. Pitropakis, E. Panaousis, T. Giannetsos, E. Anastasiadis, and G. Loukas. "A taxonomy and survey of attacks against machine learning". In: *Computer Science Review* 34 (2019), p. 100199.

[PLL08a] R. Perdisci, A. Lanzi, and W. Lee. "Classification of packed executables for accurate computer virus detection". In: *Pattern Recognition Letters* 29.14 (Oct. 15, 2008), pp. 1941–1946. ISSN: 0167-8655.

[PLL08b] R. Perdisci, A. Lanzi, and W. Lee. "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables". In: *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE. 2008, pp. 301–310.

[PMB14] A. Paverd, A. Martin, and I. Brown. "Modelling and automatically analysing privacy properties for honest-but-curious adversaries". In: *Tech. Rep* (2014).

[Puo+21] C. Puodzius, O. Zendra, A. Heuser, and L. Noureddine. "Accurate and Robust Malware Analysis through Similarity of External Calls Dependency Graphs (ECDG)". In: *CODASPY*. 2021, pp. 1–12.

[Qia+13] Y. Qiao, Y. Yang, L. Ji, and J. He. "Analyzing malware by abstracting the frequent itemsets in API call sequences". In: *2013 12th IEEE international conference on trust, security and privacy in computing and communications*. IEEE. 2013, pp. 265–270.

[Raf+17] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. "Malware detection by eating a whole exe". In: *arXiv preprint arXiv:1710.09435* (2017).

[Ram+12] K. Raman et al. "Selecting features to classify malware". In: *InfoSec Southwest* 2012 (2012), pp. 1–5.

[Reg09] O. Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[Rey+22] V. Rey, P. M. S. Sánchez, A. H. Celdrán, and G. Bovet. "Federated learning for malware detection in iot devices". In: *Computer Networks* (2022), p. 108693.

[RH20] T. Rezaei and A. Hamze. "An Efficient Approach For Malware Detection Using PE Header Specifications". In: *2020 6th International Conference on Web Research (ICWR)*. 2020, pp. 234–239.

[RP] F. Roth and T. Patzke. *Sigma*.

[Rua+21] N. Ruaro, K. Zeng, L. Dresel, M. Polino, T. Bao, A. Continella, S. Zanero, C. Kruegel, and G. Vigna. "SyML: Guiding symbolic execution toward vulnerable states through pattern learning". In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. 2021, pp. 456–468.

[RV15] J. Raphel and P. Vinod. "Information theoretic method for classification of packed and encoded files". In: *Proceedings of the 8th International Conference on Security of Information and Networks*. 2015, pp. 296–303.

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *2010 IEEE symposium on Security and privacy*. IEEE. 2010, pp. 317–331.

[Sai+18] N. B. Said, F. Biondi, V. Bontchev, O. Decourbe, T. Given-Wilson, A. Legay, and J. Quilbeuf. "Detection of mirai by syntactic and behavioral analysis". In: *ISSRE*. IEEE. 2018, pp. 224–235.

[San+11] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas. "Collective classification for packed executable identification". In: *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*. 2011, pp. 23–30.

[Sax+09] P. Saxena, P. Poosankam, S. McCamant, and D. Song. "Loop-extended symbolic execution on binary programs". In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. 2009, pp. 225–236.

[Sch+00] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. "Data mining methods for detection of new malicious executables". In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE. 2000, pp. 38–49.

[Seb+16] M. Sebastian, R. Rivera, P. Kotzias, and J. Caballero. "Avclass: A tool for massive malware labeling". In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2016, pp. 230–253.

[Seb+20] S. Sebastio, E. Baranov, F. Biondi, O. Decourbe, T. Given-Wilson, A. Legay, C. Puodzius, and J. Quilbeuf. "Optimizing symbolic execution for malware behavior classification". In: *Computers & Security* (2020), p. 101775.

[Séb19] C. Sébastien. *Portable Freeware Dataset*. Jan. 2019.

[Set09] B. Settles. *Active learning literature survey*. University of Wisconsin-Madison Department of Computer Sciences, 2009.

[SH12] M. Sikorski and A. Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[Sha+09] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq. "Pe-miner: Mining structural information to detect malicious executables in realtime". In: *International workshop on recent advances in intrusion detection*. Springer. 2009, pp. 121–141.

[Sha+22] A. Shafiei, V. Rimmer, I. Tsingenopoulos, L. Desmet, and W. Joosen. "Position Paper: On Advancing Adversarial Malware Generation Using Dynamic Features". In: *Proceedings of the 1st Workshop on Robust Malware Analysis*. 2022, pp. 15–20.

[She+11] N. Shervashidze, P. Schweitzer, E. J. Van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. "Weisfeiler-Lehman graph kernels." In: *Journal of Machine Learning Research* 12.9 (2011).

[Sho+16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *S&P*. 2016.

[Sig+20] G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis. "GraKeL: A Graph Kernel Library in Python". In: *Journal of Machine Learning Research* 21.54 (2020), pp. 1–5.

[SMA05] K. Sen, D. Marinov, and G. Agha. "CUTE: A concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[Smi+20] M. R. Smith, N. T. Johnson, J. B. Ingram, A. J. Carbajal, B. I. Haus, E. Domschot, R. Ramyaa, C. C. Lamb, S. J. Verzi, and W. P. Kegelmeyer. "Mind the gap: On bridging the semantic gap between machine learning and malware analysis". In: *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*. 2020, pp. 49–60.

[Son+22] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. "MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 2022, pp. 990–1003.

[SPB22] A. Sotgiu, M. Pintor, and B. Biggio. "Explainability-Based Debugging of Machine Learning for Vulnerability Discovery". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES '22. New York, USA: Association for Computing Machinery, 2022.

[SS21] J. Singh and J. Singh. "A survey on machine learning-based malware detection in executable files". In: *Journal of Systems Architecture* 112 (2021), p. 101861.

[Sun+10] L. Sun, S. Versteeg, S. Boztaş, and T. Yann. "Pattern recognition techniques for the classification of malware packers". In: *Australasian Conference on Information Security and Privacy*. Springer. 2010, pp. 370–390.

[Sun+22] Q. Sun, E. Abdukhamidov, T. Abuhmed, and M. Abuhamad. "Leveraging spectral representations of control flow graphs for efficient analysis of windows malware". In: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*. 2022, pp. 1240–1242.

[Tra+18] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. "Chopped symbolic execution". In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 350–360.

[Tru+20] S. Truex, L. Liu, K.-H. Chow, M. E. Gursoy, and W. Wei. "LDP-Fed: Federated learning with local differential privacy". In: *Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking*. 2020, pp. 61–66.

[TZ09] S. Treadwell and M. Zhou. "A heuristic approach for detection of obfuscated malware". In: *IEEE International Conference on Intelligence and Security Informatics, ISI*. IEEE, 2009, pp. 291–299.

[UAB19] D. Ucci, L. Aniello, and R. Baldoni. "Survey of machine learning techniques for malware analysis". In: *Computers & Security* 81 (2019), pp. 123–147.

[UG23] C. Unger and E. Gustafson. *Throwing a tantrum, part 1: angr internals*. 2023. URL: `https://angr.io/blog/throwing_a_tantrum_part_1/`.

[Uga+12] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P. G. Bringas. "Countering entropy measure attacks on packed software detection". In: *2012 IEEE Consumer Communications and Networking Conference (CCNC)*. IEEE. 2012, pp. 164–168.

[Uga+14] X. Ugarte-Pedrero, I. Santos, I. Garcia-Ferreira, S. Huerta, B. Sanz, and P. G. Bringas. "On the adoption of anomaly detection for packed executable filtering". In: *Computers & Security* 43 (2014), pp. 126–144.

[Uga+15] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 659–673.

[VHH22] M. Venkatasubramanian, A. Habibi Lashkari, and S. Hakak. "Federated Learning Assisted IoT Malware Detection Using Static Analysis". In: *Proceedings of the 2022 12th International Conference on Communication and Network Security*. 2022, pp. 191–198.

[Vin+10] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol, and L. Bottou. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." In: *Journal of machine learning research* 11.12 (2010).

[Vira] VirusTotal. *VirusTotal report*. `https://www.virustotal.com/gui/stats`.

[Virb] VirusTotal. *VirusTotal: YARA in a nutshell (2019)*. URL: https://virustotal.github.io/yara/ (visited on 01/14/2021).

[Vir+20] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272.

[Wan+19] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi. "Beyond inferring class representatives: User-level privacy leakage from federated learning". In: *IEEE conference on computer communications (INFOCOM)*. IEEE. 2019, pp. 2512–2520.

[Wei+20] K. Wei, J. Li, M. Ding, C. Ma, H. H. Yang, F. Farokhi, S. Jin, T. Q. Quek, and H. V. Poor. "Federated learning with differential privacy: Algorithms and performance analysis". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3454–3469.

[Wil45] F. Wilcoxon. "Individual Comparisons by Ranking Methods". In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 00994987.

[Xia+19] L. Xiaofeng, J. Fangshuo, Z. Xiao, Y. Shengwei, S. Jing, and P. Lio. "ASSCA: API sequence and statistics features combined architecture for malware detection". In: *Computer Networks* 157 (2019), pp. 99–111.

[Xie+19] C. Xie, K. Huang, P.-Y. Chen, and B. Li. "Dba: Distributed backdoor attacks against federated learning". In: *International conference on learning representations*. 2019.

[XMW17] D. Xu, J. Ming, and D. Wu. "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping". In: *SP*. 2017, pp. 921–937.

[Xu+21] R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, J. Joshi, and H. Ludwig. "Fedv: Privacy-preserving federated learning over vertically partitioned data". In: *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*. 2021, pp. 181–192.

[YBK13] G. Yan, N. Brown, and D. Kong. "Exploring discriminatory features for automated malware classification". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 10th International Conference (DIMVA 2013)*. Springer. 2013, pp. 41–61.

[YH02] X. Yan and J. Han. "gspan: Graph-based substructure pattern mining". In: *2002 IEEE International Conference on Data Mining, 2002*. IEEE. 2002, pp. 721–724.

[Yi+15] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao. "Postconditioned symbolic execution". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10.

[Yon12] J. Yonts. "Attributes of malicious files". In: *SANS Institute InfoSec Reading Room* (2012).

[ZFA15] M. Zakeri, F. Faraji Daneshgar, and M. Abbaspour. "A static heuristic approach to detecting malware targets". In: *Security and Communication Networks* 8.17 (2015), pp. 3015–3027.

[Zha+18] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra. "Federated learning with non-iid data". In: *arXiv preprint arXiv:1806.00582* (2018).

[Zha+19] Y. Zhao, J. Chen, D. Wu, J. Teng, and S. Yu. "Multi-Task Network Anomaly Detection Using Federated Learning". In: *Proceedings of the Tenth International Symposium on Information and Communication Technology*. 2019, pp. 273–279.

[Zha+20] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu. "{BatchCrypt}: Efficient homomorphic encryption for {Cross-Silo} federated learning". In: *USENIX annual technical conference (USENIX ATC 20)*. 2020, pp. 493–506.

[Zha+21] C. Zhang, Y. Xie, H. Bai, B. Yu, W. Li, and Y. Gao. "A survey on federated learning". In: *Knowledge-Based Systems* 216 (2021), p. 106775.

[ZHZ21] Z. Zhu, J. Hong, and J. Zhou. "Data-free knowledge distillation for heterogeneous federated learning". In: *International conference on machine learning*. PMLR. 2021, pp. 12878–12889.

[ZLH19] L. Zhu, Z. Liu, and S. Han. "Deep leakage from gradients". In: *Advances in Neural Information Processing Systems* 32 (2019).

[ZQW20] Z. Zhang, P. Qi, and W. Wang. "Dynamic Malware Analysis with Feature Engineering and Feature Learning". In: *IAAI*. 2020, pp. 1210–1217.