Universidad de Deusto

New perspectives in classification, complexity analysis and unpacking of run-time packers

Dissertation submitted by
Xabier Ugarte-Pedrero
for the degree of
doctor of philosophy

Supervised by
Dr. Igor Santos Grueiro
and
Dr. Pablo García Bringas

Bilbao, February 2015

Universidad de Deusto

New perspectives in classification, complexity analysis and unpacking of run-time packers

Dissertation submitted by
Xabier Ugarte-Pedrero
for the degree of
doctor of philosophy

Supervised by
Dr. Igor Santos Grueiro
and
Dr. Pablo García Bringas

Author | Co-advisor | Co-advisor

Bilbao, February 2015

A todos esos gigantes...

Abstract

Run-time packers are widely used by malware writers in order to hinder reverse engineering and automated analysis. These tools consist in the encryption of the original program, which is restored at runtime and afterwards executed. When these tools became popular for malware protection, the research community focused the efforts on the detection and generic unpacking of this type of obfuscations. Researchers quickly shifted their attention to other problems in this domain. The current malware landscape evidences that the problem is not yet completely solved. Malware authors keep protecting their samples with run-time packers and available on-line malware analysis services do not provide any information about the packer used for protection beyond the result provided by signature-based detection methods. Malware writers typically scramble well-known versions of packers or implement their own custom unpackers. The fact that malware writers still make an effort to implement these techniques highlights that they are still effective to protect binaries.

In this dissertation we combine different static techniques for packed binary classification. First, we propose the use of structural features extracted from Portable Executable files in order to discriminate packed from non-packed binaries, and study the performance differences among several feature-sets for this classification task. In this context, we also propose the application of anomaly detection under the intuition that common compilers produce binaries following standards and conventions, making the set of non-packed binaries easier to model than packed binaries.

Second, although the research community has published successful solutions to generically unpack malware, this technique is still employed to protect samples. Following this idea, we

raise a series of questions: What is the actual complexity of current run-time packers? Do these packers violate the assumptions made by previous approaches? What has been the evolution of the packer landscape over the years? In order to answer these questions we focus on studying the structural complexity of run-time packers. To this aim we design and develop a complete framework based on a dynamic analysis platform to record and analyse many different system events related to run-time packer behaviour. Moreover, we propose a taxonomy that combines several dimensions of complexity into one single score. This framework allowed us to perform the first longitudinal study on run-time packer complexity over custom-packed binaries, collected by the Anubis on-line sandbox since 2007.

Finally, we focus on the technical challenges and limitations involved to apply multi-path exploration to the unpacking domain. It is well-known that multi-path exploration presents severe limitations for the analysis of highly obfuscated software and does not scale to large programs. In order to unpack binaries that partially reveal their code on demand (the most complex type of packer represented in our taxonomy), the first solution that comes to mind is multi-path exploration. Our research describes some of these limitations and proposes a set of domain-specific optimisations and a heuristic that combined together improve the feasibility of multi-path exploration for unpacking.

Resumen

Los autores de malware a menudo utilizan empaquetadores para proteger sus muestras contra los intentos de ingeniería inversa y el análisis automatizado. Estas herramientas se basan en el cifrado del contenido real del ejecutable, que después es descifrado y ejecutado una vez cargado en el sistema. Cuando el uso de este tipo de técnicas creció notablemente, la comunidad investigadora centró su esfuerzo en su detección y desempaquetado, obteniendo resultados satisfactorios. Consecuentemente, la comunidad rápidamente dirigió su atención a otros problemas. Sin embargo, el panorama actual en el ámbito de la detección de malware evidencia que el problema aún no tiene una clara solución. Los autores de malware continúan protegiendo sus muestras mediante empaquetadores y los servicios on-line disponibles de análisis de malware, así como las diferentes soluciones antivirus no proporcionan gran información acerca del empaquetador utilizado para proteger las muestras más allá de una simple detección por medio de firmas. Los autores de malware comúnmente alteran empaquetadores conocidos o implementan sus propios empaquetadores a medida. El hecho de que sigan volcando sus esfuerzos en este tipo de técnicas revela que aún resultan efectivas para proteger los binarios.

En esta tesis proponemos el uso de características estructurales extraídas de las cabeceras *Portable Executable*, y evaluamos el rendimiento de este conjunto de características respecto a las heurísticas clásicas de detección de empaquetado. Además, proponemos y evaluamos la aplicación de un método basado en detección de anomalías bajo la intuición de que los binarios generados por compiladores comunes respetan los estándares y convenciones, resultando más fácil modelar este conjunto de instancias que el universo de empaquetadores conocidos y desconocidos.

En segundo lugar, a pesar de que la comunidad investigadora ha publicado varias soluciones para el desempaquetado genérico con cierto grado de efectividad, esta técnica sigue siendo utilizada para proteger las muestras. Siguiendo esta idea, planteamos una serie de cuestiones: ¿Cuál es la complejidad real de los empaquetadores? ¿Violan estos empaquetadores las asunciones de partida de las soluciones propuestas previamente? ¿Cuál ha sido la evolución de los empaquetadores a lo largo de los últimos años? Para poder responder a estas preguntas, desarrollamos un *framework* basado en análisis dinámico para monitorizar y analizar diferentes eventos del sistema relacionados con el proceso de desempaquetado. Además, proponemos una taxonomía capaz de aunar diferentes métricas de complejidad en un único valor. Este *framework* nos ha permitido llevar a cabo el primer estudio longitudinal de la complejidad de empaquetadores hechos a medida, recogidos en la plataforma pública de análisis de malware Anubis, en funcionamiento desde 2007.

Finalmente, enfocamos la investigación hacia los retos técnicos y limitaciones prácticas de la aplicación de técnicas de exploración por múltiple flujo en el dominio del desempaquetado. Es bien sabido que este tipo de técnicas presentan importantes limitaciones a la hora de analizar programas altamente ofuscados o de gran tamaño. De cara al desempaquetado de muestras que revelan su código de manera parcial y bajo demanda, la ejecución por múltiple flujo parece una solución evidente para permitir la exploración y desempaquetado de todo su código. Nuestra investigación recoge y analiza estas limitaciones, y propone un conjunto de optimizaciones específicas a este dominio y una heurística que combinados, permiten mejorar la viabilidad de la exploración por múltiple flujo para el desempaquetado.

Agradecimientos

Dicebat Bernardus Carnotensis nos esse quasi nanos, gigantium humeris incidentes, ut possimus plura eis et remotiora videre, non utique proprii visus acumine, aut eminentia corporis, sed quia in altum subvenimur et extollimur magnitudine gigantea.

Decía Bernardo de Chartres que somos como enanos a los hombros de gigantes. Podemos ver más, y más lejos que ellos, no por la agudeza de nuestra vista ni por la altura de nuestro cuerpo, sino porque somos levantados por su gran altura.

Juan de Salisbury, *Metalogicon*, *1159 (III, 4)*

No estaría escribiendo estas lineas de no ser por todos los *gigantes* que habéis contribuido en tantos aspectos a lo largo de estos años.

Ante todo, me gustaría dar mi agradecimiento a mis directores de tesis, Pablo e Igor. Más allá de vuestra contribución a esta investigación, y todos vuestros consejos y enseñanzas: Pablo, gracias por darme ese empujón para embarcarme en esta aventura; Igor, gracias por contagiarme tu pasión por la ciencia.

Durante este periodo, he tenido la oportunidad de trabajar en un equipo con no solo una calidad técnica y científica excelente, sino una dimensión humana inigualable. Todos vosotros habéis contribuido en la calidad científica de esta tesis, aportando ideas, proponiendo retos, y acompañándome en este camino. Al equipo *binario*, Ivan e Iskander, gracias por ayudarme tantas y tantas veces *con los unos y los ceros*. Al *Brownie team*, y a todos

los que hacéis posible que el S^{3}Lab siga adelante día a día, gracias por vuestro esfuerzo, dedicación, y ganas de hacer las cosas bien. A todo el equipo al completo, gracias por tender vuestra mano cada vez que he necesitado ayuda o consejo.

Fuera del laboratorio, son muchas las personas e instituciones que han hecho posible este trabajo. Me gustaría agradecer a la Universidad de Deusto y todo el equipo de Deustotech su trabajo diario para hacer esto posible. Todos vosotros estáis tras cada tesis, publicación, y proyecto. Al Gobierno Vasco, por financiar esta investigación a través de una beca predoctoral, y al proyecto SysSec, por financiar mi estancia en el centro Eurecom. A S21Sec, por colaborar en las primeras fases de esta investigación, y proporcionarnos medios y materiales necesarios. A Joxean Koret, por proporcionarme el conjunto de muestras que necesitaba, dándo un empujón a esta investigación. To Mariano, for his invaluable assistance with the Anubis database and all the useful discussion, and Davide, for his invaluable support on the second part of this project.

During four months of my life I had the opportunity to meet a group of wonderful people in the French Riviera. Not only you contributed to this thesis, but also accepted me as one more member of your team. Such an opportunity is not something one easily forgets. Davide, Aurelien, Leyla, Mariano, Onur, Merve, Andrei, Antonio... mille grazie, teşekkür ederim!

Me gustaría agradecer a todas las personas que me habéis acompañado personalmente en este periodo, y a lo largo de mi vida. Fernan, Pablo, gracias por todo lo que hemos compartido, y por escucharme tantas veces. No puedo olvidarme de todos los compañeros de comidas de los jueves, un respiro de aire fresco cada semana. A tantos amigos de colegio y universidad que a pesar de mis ausencias me acogéis de nuevo cada vez. A mis padres, Javi, Pepi, que tanto me habéis enseñado. A Pedro, Rosa, Manolo, y Espe, mis aitites, y a toda mi familia, tíos, primos, que tanto me habéis dado. A Itzi, por acompañarme durante esta tesis, entendiendo y aceptando todo lo que ha supuesto para ambos.

A todos vosotros, *gigantes*, eskerrik asko.

Publications

Some of the contributions exposed in this dissertation have been addressed in the following publications:

International Journals

1. *Collective Classification for Packed Executable Identification.*

   X. Ugarte-Pedrero, I. Santos, C. Laorden, B. Sanz, and P.G. Bringas.

   International Journal of Computer Systems Science & Engineering **28**(1), 25–36. ISSN: 0267-6192.

   Impact Factor: 0.235 (JCR 2013)

2. *On the adoption of anomaly detection for packed executable filtering.*

   X. Ugarte-Pedrero, I. Santos, I. García-Ferreira, S. Huerta, B. Sanz, and P.G. Bringas.

   Computers & Security **43** 126–144. ISSN: 0167-4048.

   Impact Factor: 1.172 (JCR 2013)

International Conference Proceedings

3. *Structural Feature based Anomaly Detection for Packed Executable Identification.*

   X. Ugarte-Pedrero, I. Santos and P.G. Bringas.

   In "Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS)", pages 230–237 (2011)

4. *Boosting Scalability in Anomaly-based Packed Executable Filtering.*

   X. Ugarte-Pedrero, I. Santos and P.G. Bringas.

   In "Proceedings of the 7^{th} International Conference on Information Security and Cryptology (INSCRYPT)" (2011)

5. *Collective Classification for Packed Executable Identification.*

   I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden and P.G. Bringas.

   In "Proceedings of the 8^{th} Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)", pages 23-30 (2011)

Other related publications

These publications are also result of the research conducted during this dissertation but were finally not included in the document.

6. *Countering Entropy Measure Attacks on Packed Software Detection.*

   X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden and P.G. Bringas.

   In "9^{th} Annual IEEE Consumer Communications and Networking Conference - Security and Content Protection (CCNC)" (2012)

7. *Semi-supervised Learning for Packed Executable Detection.*

   X. Ugarte-Pedrero, I. Santos, P.G. Bringas, M. Gastesi and J.M. Esparza.

   In "Proceedings of the 5^{th} International Conference on Network and System Security (NSS)", pages 342–346 (2011)

Contents

Figure Index

Table Index

« (Why climb mount everest?) Because it's there... Everest is the highest mountain in the world, and no man has reached its summit. Its existence is a challenge. The answer is instinctive, a part, I suppose, of man's desire to conquer the universe.»

George Mallory (1886–1924)

Chapter

1

Introduction

Malware is the term used to designate any computer software coded with malicious intentions. Current malicious software differs completely from the first viruses in the decade of the 70's. At that time, Creeper [Vir12] was expanding through ARPANET, copying itself from system to system, showing the following message: "I'm the Creeper: Catch me if You Can". Whereas first malware creators were people looking for fame and self-pride, now they are professional and well-organised groups of experts that intend to obtain economic profit [Pan12], or even state sponsored teams with diverse intentions[1].

Besides, the essential nature of malware has evolved together with authors' motivations. In the past, a few malicious programs were capable of infecting hundreds or even thousands of computers. These programs sometimes displayed annoying messages to the user, blocked the computer or even destroyed the files in it. In almost all the cases the user was warned by the effects of the software and knew that the machine was infected. Now, on the contrary, thousands of malware samples are discovered every day. According to the reports provided by different antivirus companies [McA14], the number of malware samples keeps growing every year. This change has been motivated by malware writers that need to hide the effects of the

[1] https://www.eff.org/issues/state-sponsored-malware

infection in order to remain in the system as much time as possible[1]. In this way, they can maximise the profit obtained from their creations. The daily number of new samples collected by anti-malware companies seriously affects the capacity of current anti-malware solutions to deal with the problem of detecting and mitigating current threats.

Furthermore, the sophistication of the latest malware creations goes far beyond the illicit obtention of economic profit: Stuxnet [Sym11b] and Duqu [Sym11a] have been designed to carry out tasks such as industrial espionage, or even interfere in critical industrial processes. Several organised groups have been involved in international cyber-espionage operations (for more information please refer to APT-1 report by Mandiant [Man13]). In fact, in 2012 McAfee identified the cyber-war as one of the main security threats [McA12b].

All these events have lead the security industry to a permanent cat and mouse game in which malware creators discover new attack vectors while the security industry tries to protect systems. Meanwhile, security companies have traditionally relied on signature based detection methods that, although simple and efficient, fail to confront the enormous quantity of malware samples released each day. As a consequence, proactive detection methods have been incorporated to anti-malware solutions in order to analyse program behaviour. Unfortunately, some of these approaches have resulted into intolerable false positives and a significant computational resource consumption [SKH11].

One of the most common techniques employed for malware obfuscation is packing. This technique consists in hiding the malicious payload (the actual behaviour of the executable file) using codification, compression, cryptographic, or even virtualization-based methods. In this way, when the file is statically analysed, (i.e., analysed avoiding its execution) the code is not visible. In the moment the binary is loaded and executed, the original code is decrypted and loaded into memory in order to execute it. This technique makes signature-based detection techniques useless [LH07].

When the adoption of packing techniques started to grow, anti-malware companies incorporated mechanisms to detect packed samples. Commercial solutions typically employ two approaches to deal with the packer problem. If they can identify the packer by its signature, they apply an unpacking routine specifically designed to reveal the code protected by each packer. Otherwise, they run the sample under a generic unpacking

[1] With the exception of certain types of malware, like ransom-ware.

engine. Nevertheless, nowadays, there is a considerable number of different packers and versions available on the internet. This situation allows malware creators to build their own custom protection schemes combining the existing ones or even writing them from scratch. It is estimated that every month between 10 and 15 new packers are created [Ste06a]. Recent reports [McA13] claim that new protection engines are discovered every day. Furthermore, the 35% of packed malware is protected with routines designed and coded by the author, avoiding commercial (and thus well-known) packers [MP10, GFC08]. All these facts make traditional approaches clearly insufficient for the current characteristics of malware.

According to the anti-malware tests «AVTest GMBH» [BM06], in 2006 the detection rates of the 8 most extended anti-malware solutions was between 10% and 80% for packed malware. The situation today has not improved significantly. A recent report by Pentest Partners [Mun14] shows that (i) an unpacked Meterpreter shell was detected by 33/51 AVs, (ii) when the sample was packed with UPX, it was detected by 37/51 AVs (many of them detected the packer), (iii) the shell protected by PolyCrypt was detected by 20/51 of the AVs, and (iv) only 3/51 detected the program when it was packed by Veil-Evasion, using a custom template.

Considering the packer ecosystem, some solutions detect any packed sample as malware[1]. According to the report from Pentest Partners, when the Windows notepad was submitted to VirusTotal protected by UPX, 17/51 AVs detected it as malware. However, packers are legitimate tools that are sometimes used by legitimate software (according to Guo et al. [GFC08], up to a 35% of packed software falls into the legitimate category). This fact makes necessary to perform a deeper analysis of software samples to determine if the content of the binary is malicious or not. This introspection requires, in most cases, unpacking the sample to access the original code.

Finally, recent studies [SM14] demonstrate that it is possible to systematically develop techniques to avoid heuristic detection and generic unpacking engines implemented in current antivirus systems. Furthermore, these products are generally complex solutions prone to software errors that can be exploited to gain control of the system [Kor14].

[1] These tools also generally maintain white-lists in order to avoid false positives.

1.1 Packer predominance statistics

The packer predominance has grown in the last decade: the rate of collected packed malware has increased from a 29% in 2003, a 35% in 2005, to an 80% in 2007 [Mas05, Res07]. In year 2009, McAfee [McA09] reported that 80% of the malware collected was packed. Other reports show that, although off-the-self packers are still extensively used by malware authors, up to a 35% of the malware is protected with custom packers [MP10], designed and implemented by malware authors themselves. Some studies even highlight the fact that a substantial part of the software packed (35%) is not considered malicious [GFC08]. In fact, a considerable amount of packed malware is protected with packers available on the internet [MP10], sometimes offered as legitimate software protection tools used to difficult reverse engineering. Nevertheless, it is not clear how many legitimate products are protected with this kind of techniques. Anti-malware solutions generally have to previously unpack the samples under analysis. Also, the numbers provided in different reports by security companies vary, and it is not clear how they categorise the samples as packed or not packed. For instance, Bayer et al. analysed in 2009 a dataset of 901.294 unique samples (based on their MD5 hash function), covering a total of 1.167.542 submissions [Bay09] to Anubis[1]. According to these data, only a 40.64% were packed samples (including polymorphic worms), a rate far below the 80% reported by MacAfee [GFC08] one year before. A recent study [NR14] states that only a 37.53 % of the malware analysed is packed. Although the industry has adopted different techniques to fight the packer problem, these discrepancies highlight that (i) there is not a consensus on this matter and (ii) the existing methods do not provide a precise detection. Many of the on-line services that provide this kind of information rely on signature-based detection engines such as PEiD[2], TRiD[3] or F-Prot[4] (in the case of VirusTotal[5]), or Sigbuster[6], a tool with a proprietary signature database (in the case of Shadowserver). The packing strategy enables malware writers to produce thousands of different malware samples based on the same source code. The necessity to obtain the original version of the binary, and the

[1] https://anubis.iseclab.org/
[2] https://github.com/sroberts/peid4yara
[3] http://mark0.net/soft-trid-e.html
[4] http://www.f-prot.com/
[5] https://www.virustotal.com/
[6] http://www.teamfurry.com/

great diversity of packing tools and techniques developed in the last years make the unpacking task a significantly problematic challenge for the anti-malware industry.

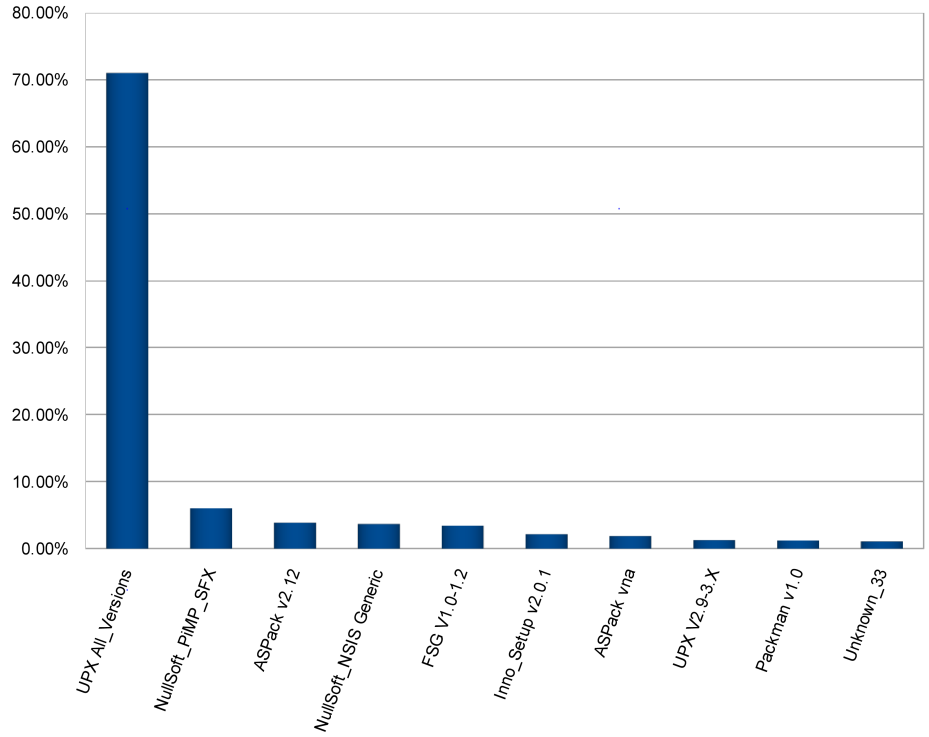


Figure 1.1: Packers employed to protect the samples collected by the Shadowserver platform from the 2^{nd} of November 2013 to the 2^{nd} of November, 2014.

Figure 1.1 shows the packers most commonly used among the malware samples collected by the Shadowserver analysis platform[1], from the 2^{nd} of November 2013 to the 2^{nd} of November, 2014. It covers 32,055,260 samples, and it can be observed that the most commonly used packer is UPX (in all its possible versions), with a 71.01% of packed samples. In fact, UPX is still a widely used packer. Many malware authors slightly modify this simple packer in order to confuse signature based systems, trying, for instance, to modify the samples so that they resemble other packers.

[1] `https://www.shadowserver.org/wiki/pmwiki.php/Stats/PackerStatistics`

1.2 Challenges of the packer problem

Considering the limitations of current solutions, we identify several challenges regarding the packer problem.

1.2.1 Packer detection

The first challenge for an efficient treatment of samples is to correctly determine whether the sample under analysis is packed or not. Several approaches [STF09, PLL08b] have proposed sample filtering techniques to determine how a given sample should be treated (i.e., is it necessary to run the sample in a computationally complex dynamic unpacking system before scanning the binary for known signatures?).

There are several limitations that must be addressed in order to build efficient and effective packed software classification systems:

- **Detection rate.** Packed software detection has been addressed by several authors obtaining considerably sound results. In general, these systems are designed to provide a binary value: packed or not packed. We can consider different approaches for the deployment of such filters. On the one hand, false positives may incur into a higher computational cost (i.e., non-packed samples might be processed in a generic unpacking system). On the other hand, false negatives imply a risk of not processing the sample adequately or not discovering the malicious hidden code. As a result, different environments may require a different trade-off between false positives and false negatives.

- **Dependence on heuristics.** Many of the approaches rely on heuristics that are now well-known by both the security industry and malware writers (e.g., high entropy values, number of sections or characteristics of the sections). Although simple to implement, malware writers can easily evade these heuristics by adapting their binaries to adopt the characteristics of non-packed software.

- **Detection of new packers.** Malware writers try to produce binaries that remain undetected. To this aim, they modify existing packers in order to evade signature scanning based filters, or even create new packers either by modifying existing ones or programming them from scratch (i.e., custom packers). Methods based on machine learning have been proposed in order to build more robust detection systems. Unfortunately, if supervised machine-learning techniques are

applied, the resulting classifiers may be susceptible of adapting the solution to the set of known packers. In this way, these solutions in practice may present limitations to detect new packers and require training the classifiers periodically, which implies additional costs.

- **Labelling efforts.** Finally, another limitation of current approaches is the necessity of labelling a high amount of data to train the models. In general, the size and diversity of the dataset affects considerably the results achieved by machine-learning models. Labelling software samples requires the efforts of a software analyst that will need to inspect each sample in order to determine if it is packed or not. This process not only implies an economic expense, but also relies on the capacity of the expert to correctly label samples. As depicted in Section 1.1, there is not a consensus on the number of packed samples in the analysed datasets, and there is no publicly available dataset in which the packed samples are labelled beyond classic signature based detection. Also, there are no tools available for the analysis of the packer structure, and researchers often have to resort to signature based detectors and heuristics.

1.2.2 Recognition and analysis of the unpacking routine

Packer detection constitutes a challenge by itself, and it is in fact a critical aspect in malware analysis. Nevertheless, in many occasions the identification of the packer used to protect the sample makes the process more efficient by allowing the application of specific procedures to obtain the original code in a fast and reliable way.

The most simple approach is to use specific unpacking routines designed to obtain the original code for each specific packer, and then proceed with the detection of malicious code. In order to identify the packing engine, these approaches scan the samples to identify patterns, known as signatures. The main limitations of these approaches are the need to generate useful signatures to identify packers, and the efforts needed to design and code specific unpacking routines for each different packer. Despite these limitations, this approach is a very successful solution to deal with software protected by known packers.

A few alternative packer identification approaches have been proposed in the literature. Despite their similarity to malware detection, packer identification exhibits certain particularities that make it an interesting re-

search topic to be addressed.

Nevertheless, given the wide variety of packers available and the predominance of custom-packers, this approach is not enough to deal with the increasing number of malware samples discovered every day.

Up to date, most of the approaches have focused on developing packer detection and identification methods, or providing different approaches and solutions for generic unpacking. Nevertheless, few approaches have focused on analysing and understanding the structure of the packer. Despite this kind of approaches do not directly address the packer problem, they can improve the understanding of the samples, giving the analyst valuable information for the reverse engineering process.

1.2.3 Generic unpacking of increasingly complex packers

Specific unpacking is an efficient approach to deal with known packers. However, current trends in malware make this approach insufficient to tackle with the great diversity of packers. Stepan et al. [Ste06a] estimated that every month between 10 and 15 new packers are created. In some cases, the source code is available for anyone to modify it and to generate variants. In addition, the 35% of malware implements its own protection mechanisms (custom packers), instead of using known packers [MP10, GFC08].

Given this background, one of the most complex challenges in software unpacking is dealing with binaries protected by unknown packers. The main alternative to specific unpacking is generic unpacking. This approach consists in executing the sample in a controlled environment, monitoring its execution, and obtaining the original code once it is unprotected by the unpacking routine that includes the sample. This technique is a computationally complex task that involves several challenges:

- **Heterogeneous unpacking routines.** The diversity of packing techniques that are employed by current malware makes difficult to define generic mechanisms and rules to unpack software. Many approaches are based on the execution of the sample and the detection of the moment in which the execution flow jumps from the unpacking routine to the original code. This is known as *tail-jump*. Some methods implement heuristic rules to detect this point in the execution. Other methods, in contrast, use statistical approaches.

- **Partial code revelation and trigger-based behaviour.** The main limitation of dynamic approaches is the capacity of any sample to evade analysis by partially revealing the code. Dynamic approaches can only monitor one execution path, given the conditions imposed by the environment in which it is executed. For instance, a certain malicious routine may only be executed on certain dates. If we dynamically analyse the sample in a different date, the monitored behaviour would be considered benign. In some occasions, the environment in which the sample is analysed is not adequate (e.g. it lacks network connectivity, incompatible software or hardware...). This limitation can affect the unpacking of samples in two different ways. Some samples only trigger the unpacking of the original payload under certain circumstances. In order to unpack the sample in a dynamic analysis engine, it is necessary to trigger the execution path that drives to the unpacking routine. Finally, an special kind of packers apply partial code revelation (i.e., shifting-decode-frames). These packers unpack the hidden code on demand before its execution and sometimes even protect the code again. In these cases, if only certain code paths are revealed, the entire content of the binary will not be unpacked.

  There are two main alternatives to deal these situations:

  - *Transparency.* This alternative tries to provide a transparent analysis platform. This enables to execute the sample in an environment as close as possible to a real machine with network connectivity. The monitoring mechanisms must be implemented at kernel level to avoid possible detection by the sample. This makes the implementation of such systems a technically difficult process.
  - *Multi-path Exploration.* In contrast, multi-path exploration approaches try to adapt the system to the conditions that guard the different execution paths of the binary. This approach enables to explore the different execution paths of a sample, and thus, the behaviour it may show under different circumstances.

    One of the limitations of these approaches is their high computational cost due to the number of possible paths that a binary may have. For this reason, it is important to study how to apply these techniques to the unpacking problem, and to propose heuristics to optimise the exploration process.

1.3 Hypothesis and objectives

Given the limitations described in this chapter, we formulate the following hypothesis to test during this dissertation:

> «*It is possible to classify and measure the complexity of run-time packers using static and dynamic analysis techniques and to recover the code of binaries protected with partial code revelation.*»

Considering this hypothesis, we define the general objective of this dissertation:

General objective 1.1 *Improve the packer analysis process from different perspectives in order to enable malware analysis.*

Besides, this objective can be divided into the following specific objectives:

Specific objective 1.1 *Improve packed software filtering systems providing alternative classification techniques.*

Specific objective 1.2 *Provide a deep understanding of the structural complexity of run-time packers.*

Specific objective 1.3 *Study the limitations and propose heuristics to enable the application of multi-path exploration techniques for the generic unpacking of samples that implement partial code revelation.*

Given these objectives, we define the following operational objectives that will guide the research during this dissertation.

Packer detection

Operational objective 1.1 *Propose and evaluate new representation methods capable of differentiating packed and non-packed samples.*

Operational objective 1.2 *Develop and evaluate classification methods to improve the limitations of current approaches.*

Packer structural complexity analysis

Operational objective 1.3 *Develop a system capable of capturing all the system events associated to the unpacking of complex packers.*

Operational objective 1.4 *Develop a model to analyse all the system events produced during the unpacking of a sample.*

Operational objective 1.5 *Propose a taxonomy capable of measuring the structural complexity of a packer from different perspectives.*

Multi-path generic unpacking

Operational objective 1.6 *Study the limitations of current multi-path exploration techniques based on symbolic execution in order to deal with the unpacking of samples.*

Operational objective 1.7 *Build a multi-path exploration based generic unpacking system.*

Operational objective 1.8 *Propose and evaluate a heuristic to guide multi-path exploration in the context of software unpacking in order to improve its feasibility.*

1.4 Research methodology

This section describes the research methodology followed during this dissertation. This methodology is divided in several stages, according to the *sand clock* research model [TD05, Der03] that drives the research from the generality (identification of the area, literature review), to the specificity (research question statement, experimentation, observation, analysis and interpretation of the results) and back to generality (discussion, conclusions and future work). In addition, the process must be cyclic, using the conclusions obtained to refine the methods proposed or even reformulate the hypotheses, if necessary.

1. **Identification of the research area and open problems.** The first step is to choose the research area and to identify the open problems. In some occasions, we cannot determine the limitations of previously proposed solutions. To this aim, we will obtain and review the most recent and relevant publications in the area in order to identify specific problems that should be addressed by the research community.

2. **Literature review.** Once the research topic has been selected, it is necessary to review the most recent scientific publications. Latest contributions, surveys, or even not scientific technical reports are useful to gain a deep understanding of the area and the problems that should be addressed. It is important to evaluate the impact and relevance of each of the contributions, critically interpret the results and consider the limitations and weaknesses of current approaches.

3. **Hypothesis, objectives and planning.** Afterwards, once the current limitations have been defined, we will delimit one or several research hypotheses. We will then define the objectives that will guide our work in order to validate the hypothesis. Finally, we will develop a work plan and define deadlines and milestones.

4. **Development of our approach.** Once the work plan is defined, we will follow it in order to develop our original approach. In some occasions, we will have to study the technical viability of our approach, or even learn different skills in order to be able to develop it.

5. **Evaluation and confirmation or rejection of the hypothesis.** Once the method has been developed, the next step is to design the necessary experiments in order to validate the hypothesis. This phase requires a thorough work, considering possible biases and risks to the validity of the experiments. Once we ensure that the experiments are adequate to validate the hypothesis, we can proceed to their execution. If the results obtained are consistent with the proposed hypothesis we will confirm it. On the contrary, if the results are not consistent, we will redesign our approach or even reformulate the original hypothesis.

6. **Presentation of the results and publication.** Once the first results are obtained, it is convenient to present them to the scientific community in order to obtain feedback from experts in the field. If the

feedback is precise enough, we will thoroughly analyse the comments and correct our approach or our experimentation, if necessary.

1.5 Structure of the document

The rest of this document is structured as follows. Chapter 2 introduces the concepts and definitions involved in the packer problem domain, and describes the most relevant publications in the area. Then, the contributions of this dissertation are divided into two different parts. Part I deals with static analysis techniques for the efficient filtering of packed binaries. In this context, Chapter 3 evaluates the performance of different supervised machine-learning approaches over different feature-sets for the classification of packed binaries, while Chapter 4 proposes an approach based on anomaly detection for the detection of packed binaries. Part II, in contrast, deals with dynamic analysis techniques for run-time packer complexity analysis and unpacking. Chapter 5 describes a system capable of recording all the system events involved in the unpacking process and a model capable of measuring the structural complexity of run-time packers from different perspectives. Chapter 6, describes our implementation of a multi-path exploration engine and the heuristic designed to improve its efficiency for the unpacking of samples. Finally, Chapter 7 summarises the conclusions of this research, and outlines avenues for future work.

«The true mountaineer is a wanderer... a man who loves to be where no human being has been before, who delights in gripping rocks that have previously never felt the touch of human fingers... Equally, whether he succeeds or fails, he delights in the fun and jollity of the struggle.»

Albert Frederick Mummery
(1855–1895)

Chapter 2

Literature review

Malware has experienced a constant evolution in the last years, both in complexity and hiding capability. As a consequence, the number of different samples and infections has grown, making the anti-malware industry and the scientific community focus their efforts on looking for new malware detection methods. The key to understand the growth of scientific research and publication in the last years in this area is the fact that, in spite of the excellent approaches proposed, the battle between the attackers and defenders is a never-ending story of success and fail.

This chapter gathers the efforts dedicated in the last years by the scientific community and the security industry in the area of malware detection, and more specifically in packed software analysis. Section 2.1 presents some basic concepts and definitions related to this topic. In Section 2.2 we propose a classification of the different tasks related to the analysis of packed malware. Section 2.3 describes the techniques employed by current anti-malware solutions. Section 2.4 deals with the methods proposed for packed binary detection. While Section 2.5.2 collects the approaches based on specific unpacking, Section 2.5.3 details the different generic unpacking methods published in the last years. Finally, Section 2.7 summarises the aspects discussed during the chapter.

2.1 Concepts and definitions

This section explains some relevant concepts and definitions in the area of malware detection. The concepts are organised as follows: first we define malware, its kinds and characteristics. Secondly, we tackle with software obfuscation, going into greater detail with packing. Finally, we deal with malware detection and unpacking.

2.1.1 Malware

The term *computer virus* was coined by Fred Cohen [Coh86] in 1986.

Definition 2.1 ***Computer Virus (Cohen)***
«We define a computer virus as a program that can infect other programs by modifying them to include a possibly evolved copy of itself.»

However, this definition does not mention anything about computer virus' intentions. Later on, the term malicious software, or malware, was used to refer to the intentions behind software [McA12a]:

Definition 2.2 ***Malware (McAfee Virus Glossary)***
«Malware is a generic term used to describe malicious software such as viruses, Trojan horses, spyware, and malicious active content.»

This definition suggests that malware does not necessarily behave like a virus: it can also adopt other different forms such as worms, or trojans. Accordingly, nowadays different terms for malicious software are used depending on its intentions and propagation mechanism:

- **Virus.** This term has been previously defined as a program able to copy or insert a version of itself into another program. There are many kinds of viruses: *Boot Sector Viruses* affect the Master Boot Record (MBR) of hard disks; other viruses infect files, stay in memory, or are programmed as Microsoft Office macros.
- **Worms.** Worms are sophisticated pieces of replicating code that use their own program to spread without user interaction [Gri01]. Worms normally spread over a network, but can also spread through removable devices.
- **Trojans.** A trojan horse is a non-replicating computer program masqueraded as a different program, hiding its real intentions [Gri01].

- **Spyware.** Spyware is defined as a type of software installed on a computer that monitors the activity of the user and sends it to a third party [SU04]. This illicit activity can have the intention to mine the user preferences and send directed marketing (*Ad-ware*) or even to steal credentials or credit-card numbers using *Key-Loggers*.

- **Rootkits.** Modern rootkits are defined as a set of programming tools that allow an intruder to hide the fact that a system has been compromised, hiding files, processes, registry keys and other objects, ensuring the continuity of the compromise [DCKB07].

2.1.2 Metamorphism, encrypted malware, oligomorphism and polymorphism

One of the techniques employed by malware creators to evade detection is metamorphism. Chouchane et al. [CL06] define metamorphism as:

Definition 2.3 ***Metamorphism (Chouchane and Lakhotia)***
«The main goal of metamorphism is to change the appearance of a malicious program while keeping its functionality and many metamorphic transformations can be used (or combined) to achieve this goal.»

Anyhow, common malware samples do not base their hiding capabilities exclusively on metamorphism. A different technique employs cryptography to hide the code. Skulason [Sku90] defines encrypted viruses:

Definition 2.4 ***Encrypted Viruses (Skulason)***
«Usually, an encrypted virus consists of two parts; the decryptor and the encrypted main body of the virus. The decryptor executes when an infected program runs, and decrypts the virus body.»

According to Skulason, the encryption of malware follows several objectives [Sku90]:

- **Evade static analysis**. Some programs analyse the executable code to discover suspect patterns related to known malicious samples. Code encryption is used to hide the original code and prevent static analysis.

- **Delay the infection process**. It can make the analysis process take larger computation efforts and processing time. Nevertheless, current computers and systems are not affected significantly by these routines.
- **Avoid the unauthorised modification of malware**. Malware encryption prevents other authors from modifying existing malware binaries to create new variants.
- **Evade detection**. An encrypted virus cannot be detected by signature scanners when it is encrypted. The only part of the code that has the same signature along all the infections is the decryption algorithm, which could correspond to any piece of software, either malicious or legitimate.

Nevertheless, these techniques remain insufficient to effectively evade detection. If the decryption routine is an invariant code, signature scanning can be applied to these routines. Oligomorphic malware tries to change that code routine in each generation [RMI11].

Definition 2.5 ***Oligomorphic Virus (Rad et al.)***

«Oligomorphic viruses are willing to substitute the decryptor code in new offspring. The easiest method to apply this idea is to provide a set of different decryptor loops rather than one.»

Oligomorphic engines are simple versions of polymorphic engines. Jacob, Debar and Filiol [JDF08] formulated the following definition of polymorphic malware:

Definition 2.6 ***Polymorphic malware (Jacob, Debar and Filiol)***

«Polymorphic malware encrypt their entire code in order to conceal any potential signature. A simple variation of the ciphering key modifies totally their byte sequences. A decryption routine is then required to recover the original code and execute it. This routine must possess its own mutation facilities to avoid becoming a signature on its own.».

In the case of polymorphic software, the decryption routine is obfuscated in each generation to evade detection. The difference between oligomorphism and polymorphism resides in the mutation engine for the decryption routine. Polymorphic malware can transform the decryption routine innumerable times, while oligomorphic malware only presents a limited number of versions [RMI11].

2.1.3 Packers

Packers are tools employed to compress or encrypt the code inside an executable file [CKJ+05]. These tools can implement different techniques to evade detection: encryption, oligomorphism, or polymorphism. In this section we define and clarify some concepts related to this obfuscation method. In this way, we find several definitions in the literature:

According to Sun [Sun12], the behaviour of packers is defined as follows:

Definition 2.7 ***Packer behaviour (L. Sun)***

«Files, or groups of files, are encrypted/compressed into another file so that they take up less space and hinder signature based malware detection.»

Similarly, Martignoni et al. [MCJ07] refer to software packing in the following way:

Definition 2.8 ***Packer (Martignoni, Christodorescu and Jha)***

«Such programs consist of a decompression or decryption routine that extracts the garbled payload from memory and then executes it. We use the term packed and its variations to refer to malware whose payload is either compressed or encrypted.»

In addition, Martignoni et al. [MCJ07] specify how packed executables work:

«This unpacking routine can be invoked once, in which case the whole payload is extracted to memory in a single step, or multiple times, when parts of the payload are extracted to memory at different times.»

Nevertheless, these definitions are excessively simplistic. According to Rolles [Rol09], not all the packers extract the entire original code into memory before executing. Sometimes, packed binaries do not even compress or encrypt the code:

«Some protections utilise multiple executables: some will unpack the executable into a new process, some operate with a two-process model in which one executable debugs a modified version of the original. Some protections drastically obscure the relationship of the protected executable with shared libraries on the system. [...] some protections never restore portions of the protected code; instead, they translate the code into a different language and execute it at run-time inside of a custom, often obfuscated interpreter.»

It is important to highlight that, although most of the malware collected by anti-malware companies is packed, this technique is also used to protect legitimate software. Many computer programs are protected against reverse engineering attacks in order to prevent piracy and preserve the copyright of the author. Guo, Ferrie and Chiueh [GFC08] obtained interesting conclusions during the collection of a dataset:

> «*Not all packed programs are malware. We took a random sample of tens of thousands of executable files that were collected over a period of several months and were packed by packers that Symantec recognises and knows how to unpack, and ran a set of commercial antivirus (AV) scanners from multiple vendors against them. About 65% of these executable files are known malware. The remaining 35% most likely falls into the goodware category [...]. Clearly, the use of packers to protect goodware is quite common too.*»

2.1.3.1 **Description of a simple packer**

Basic packers show a similar operating scheme. These packed binaries consist of a compressed or encrypted payload (i.e. the sensitive code and data that must be protected), and a routine designed to reveal the protected data. This routine typically operates by decompressing or deciphering the payload, placing it into memory, and then executing it. One of the most common and simple packers available is named UPX [Bay09].

More concretely, UPX does not encrypt the original code, it compresses it in such a way that the final executable presents a lower file size. This packer creates a binary with 3 sections[1]. The first section, namely `UPX0` is a virtual section without any physical data. This means that, when the executable is loaded[2], some memory space is reserved for the section so that it can be used later on. More concretely, this space will be used by the decompression routine to store the original code before executing it. Section `UPX1` contains the aforementioned decompression routine and all the compressed code and data. The entry point to the executable (i.e. the first memory address that will be executed once a process is created and

[1] Windows PE (Portable Executable) binaries are usually divided into a variable number of sections, organising the code, data, DLL and function import and export information, relocation data, and any other resources the program may need.

[2] The operating system has a component called loader that is responsible for copying the binary from disk to memory and creating a process to execute the code.

the binary is successfully loaded), points to the decompression routine in section UPX1.

Finally, UPX2 contains the import and export information of the executable. This information consists of several tables containing the list of DLLs that must be loaded by the system for the correct functioning of the binary, and the functions inside these DLLs that are used by the binary. The system, after loading the necessary DLLs, updates these tables so that the code can correctly call to the imported functions independently of the position in memory where they are loaded. Section UPX2, in addition, may contain some other resources of the original binary. This packer is so simple that resources, import and export information is not compressed or protected. The first two sections, UPX0 and UPX1 have read, write, and execute permissions, in order to facilitate the unpacking routine the process of decompressing, writing, and then executing the original code. Nevertheless, the original code may contain instructions to check these permissions and avoid execution if it finds that the permissions have been modified. For this reason, the packer inserts some code into the unpacking routine that is responsible for altering the flag that enables to write over the PE header, allowing the following instructions to change the permission flags associated to the different sections, and finally resetting back the first flag in the header. In this way, applications that implement this constraints will run normally even if they are packed [GFC08].

2.1.3.2 **Packer evolution**

The packers used to protect software have evolved considerably. The battle derived from obfuscation and de-obfuscation has contributed to this evolution. On the one hand, legitimate software developers protect their products against cracking methods to ensure that users pay the corresponding licenses. Besides, crackers find new methods to bypass these security barriers. On the other hand, malware creators protect their malicious software with more and more resilient techniques to avoid detection. Malware researchers develop new methods to unveil the real content of the binaries, forcing malware writers to develop new protection measures. It is important to highlight that, despite of the widespread use of these protection tools for obfuscating malware, many of these products are commercialised as legitimate software protection engines (e.g.Themida [1]).

[1] http://www.oreans.com/es/themida.php

- **First generation: Compressors**

  First packers had a very simple objective, which was to reduce the size of the executable files in order to facilitate their transmission over the network. In this way, several packers were developed to compress (or rarely encrypt) the binaries, inserting a decompression routine to extract the original code before executing it. Ultimate Packer for eXecutables (UPX), Ultimate Packer (UPack), ASPack, PECompact, FSG, and MEW are packers that belong to this generation [Sun12].

- **Second generation: Protectors**

  The intention behind this kind of packers differs from the older compressors. In this case, there is an actual intention to conceal the payload of the binary from analysts or reverse engineers. Malware authors soon began to use cryptographic techniques instead of plain compression. Some examples of this kind of packers are Yoda's, ASProtect, Armadillo, Morphine, EXECryptor, Obsidium, Enigma, or Themida [Sun12].

  In addition, these packers generally use anti-reversing[1] techniques, providing additional protection aimed at preventing analysts from obtaining the actual code.

- **Third generation: Virtualization-based packers**

  This new generation of packers has adopted a different approach for software protection [Sun12]. Themida, VMProtect or ExeCryptor are protection engines that transform the original code, translating each instruction to a different instruction set, or even to a different hardware architecture that is virtually implemented in the stub of the resulting executable file. Thus, the original code is never present, neither in the file or in memory, because it is transformed to a completely different emulation-based program semantically equivalent. The stub of the packed binary contains an interpreter that implements the desired functionality for each virtual instruction. Furthermore, some of these tools allow to randomise the numeric code used for each instruction (sometimes called byte-code), or even choose between differ-

[1] An anti-reversing technique is any method designed to evade reverse-engineering, either by obfuscating the code, or by including code to detect certain tools or methods commonly used by reverse engineers (e.g. debuggers, virtual machines, sandboxes, memory dumps, and so on).

ent virtual processor architectures, making impossible to generalise the behaviour from one packed sample to another [BK09].

- **Target specific protection**

  The raise of targeted malware has originated a different kind of protection. Royal et al. [SRL12] proposed an obfuscation approach based on the techniques implemented by the Flashback botnet targeting Mac OSX systems. This approach is inspired by DRM software, and consists in infecting the machine in a first stage, and then downloading a payload in a second stage, encrypting the code with a key derived from the system hardware properties or configuration. In this way, once the machine is infected, the malicious binary can only be executed in the target machine, defeating automated malware analysis platforms. Sharif et al. [SLGL08] proposed an approach to encrypt portions of the code that are triggered conditionally using keys derived from the conditions that must be met to execute it. In this way, unless the binary is executed in the appropriate system, the payload will never be revealed. Both approaches implement a system specific encryption and effectively evade the analysis of the sample outside the system. The Gauss malware [Lis13] is an example of this protection technique. In this case, one of the modules was encrypted with a key derived from certain configuration parameters. Once the parameters were fetched for the system, the MD5 hashing algorithm was applied 10,000 times to the key. The result was used to decrypt the payload, making infeasible the decryption of the content by brute-force.

2.1.3.3 Techniques used by packers

Although UPX or simple compressors do not represent any inconvenient for analysts or anti-malware analysis engines, there are several techniques used by certain packers that complicate software analysis in different ways.

- **Multi-layer packing**

  «*Multi-layer packing uses a combination of potentially different packers to pack a given binary [GFC08].*»

  This makes possible to generate many different packed binaries from the same original file. Nonetheless, in practice, some packers produce

as a result a binary that cannot be packed by other packers. The reason is that the resulting file does not strictly follow the standard specifications for Windows PE files, and sometimes includes obfuscating techniques, that, despite allowing the system to load and execute the program, can make other packers fail trying to parse the headers of the file. Guo et al. suggest that the mere presence of this technique is a good indicator of malicious code [GFC08], assuming that legitimate software will not use this technique. However, these assumptions can be harmful for commercial solutions that may throw false positives annoying the users that may want to execute a legitimate application.

- **Incremental unpacking**

  Some packed binaries reveal the code incrementally, decrypting only the code frames they are about to execute [SYS+08]. This technique has 3 main implications for software analysis. First, as the code is not present until the last instruction is executed, any memory analysis previous to this moment may be useless. Second, as the code must be run until the last instruction, the potential malicious software will have carried out its malicious actions such as infection or network propagation. Finally, if the binary contains conditionally triggered code (i.e. code that is only executed under certain circumstances), the real payload may not be revealed even if the execution of the process finishes. The Shrinker packer implements this technique.

- **Shifting decode frames**

  Shifting decode frames (also named partial code revelation [GFC08]) is a variation of incremental packing that goes one step further. The binaries protected with this technique decrypt each code frame before executing it and encrypt it back when the execution flow jumps to a different frame. In this way, the executed code is never completely available in memory even when the process finishes its execution [Böh08]. Armadillo is a good example of this kind of packing technique [GFC08]. To this end, Armadillo activates a page-level protection flag that will throw an exception when the memory page is accessed. As a consequence, the unpacking stub will decrypt the memory page and the execution will continue until it reaches a different memory page, protecting the previous page again. Variations of this approach have been also studied in the literature [BLB11] relying on alternative implementation approaches like code instrumentation.

- **Passive anti-unpacking techniques**

  «*Passive anti-unpacking techniques are intended to make disassembly difficult, which in turns makes it difficult to identify and reverse the unpacking algorithm [GFC08].*»

  – *Assembly obfuscation.* The majority of packers obfuscate the code of their unpacking routines applying different anti-disassembly techniques in order to confuse common disassemblers. These techniques hinder the reverse engineering of the algorithms used for unpacking the payload.

  – *Stolen bytes.* This method obfuscates memory addresses such as the *Original Entry Point*, the entry point of functions inside the code, or Windows API functions. To this end, the instructions present in the memory addresses that are the destiny of a function call are copied to a different location in memory, changing the address in the corresponding call instructions. In some occasions, the original memory space where the function was located is overwritten with other data. The copied code may be also obfuscated with independent instruction reordering, NOP insertion, or garbage byte insertion [Böh08]. Themida is a good example of this behaviour. Among other options, it identifies functions in the protected code and allows to relocate them, splicing the code in different points in memory and even replacing the code by a mutated version.

- **Active anti-unpacking techniques**

  «*Active techniques are intended to protect the running binary against having the fully unpacked image intercepted and extracted [GFC08].*»
  Some legitimate packers, such as Enigma or Themida, use these techniques.

  – *Anti-dumping.* One of the most commonly used techniques to unpack software is named *run and dump*. This technique consists of running the binary in a controlled environment, waiting until the original code is decrypted and loaded into memory, and then dumping the process memory to a new file [BK09].

    The anti-dumping methods try to avoid the dump of memory to disk. To this end, one possible implementation is to change the

image size parameter (i.e. size of the code and data of the process) that is stored in the PEB[1]. This alteration complicates certain tasks such as attaching a debugger to the process or dumping the correct number of memory pages. Shifting decode frames can be classified as an anti-dumping method, considering that it avoids the presence of all the original code in memory.

- *Anti-intercepting.* Some unpackers try to stop execution of the process when a previously written memory page is executed. This process is called interception. A possible anti-intercepting technique forces the interceptor to stop the execution writing on any page in memory and executing it afterwards, before the original code is present in memory. In this manner, the target process can detect the presence of an interceptor. Another technique is based on the detection of modifications of the write permissions of memory pages (the interceptor may modify these permissions to detect the execution of a memory page). To detect this modification, the target process can use a kernel function to write on a memory page in user mode. If the kernel function returns an error code, the write permissions were likely modified [Fer08].
- *Process creation.* Some binaries create a process and inject the original code into its memory, or directly create a new file in disk to launch it afterwards in order to evade detection by debuggers or emulators that only monitor a unique process [SYS+08].
- *Anti-debugging.* The operating system provides several manners to detect the presence of a debugger. One of these methods is the Windows API function IsDebuggerPresent(). Another way to detect an attached debugger is to check certain flags of the NtGlobalFlag field (*heap tail checking*, *heap free checking*, *heap parameter checking*). These flags are set when the process is being debugged. Both the return value of the Windows API function or the flags in NTGlobalFlags can be manually reset to avoid the detection of the debugger [GFC08]. Execryptor, Themida, or MSLRH use anti-debugging techniques.

  Timing attacks are based on the comparison of the execution time of a certain block of instructions. For example, the x86 instruction RTDSC can be used to count the time elapsed from one

[1] Process Environment Block (PEB) is a system structure that contains the necessary information about every process under execution in the system.

point of the execution to another. If a debugger is present and the analyst is, for example, stepping over instructions, the time elapsed will be much longer than the time required by any system to execute the code.

Besides, debugging breakpoints are installed in the memory of a process by modifying the first byte of the target instruction, changing it by the INT3 instruction that will produce an interruption captured by the attached debugger. A process can scan its own code to find INT3 instructions used by the debugger to stop execution or avoid the unpacking of the original code. A different way to implement this approach is to embed a code checksum in the file and ensure in a certain point of the execution that the memory checksum matches it. If any modification to the code is made (like INT3 interrupts), the checksum will be different, and the execution will be stopped.

Finally, another anti-debugger technique used by both malicious and legitimate software is to scan the active processes in the system to check for certain strings corresponding to debuggers or reverse-engineering tools.

Another substantially different approach tries not only to detect the debugger, but also to interfere with the analysis tasks. A process can, at a certain point, generate exceptions that will be captured by a debugger. This can, on the one hand, confuse the analyst or the tool debugging the process, and on the other hand, redirect the execution to some malicious code installed as an exception handler. This code will be executed only if there is no debugger present, or if the analyst passes the exception to the process after capturing it. In this fashion, the program may insert INT3 instructions in its own code to confuse the analyst in the task of tracing the execution of the process. Another possible approach is to use invalid operational codes, or division by zero. Another technique that can be used to avoid the use of a debugger is to modify certain structures in the PE header in such a way that the system loader will execute the files correctly, but the debugger will find inconsistencies and reject the sample because of its errors in the format of the headers. This vulnerability occurs because the Windows Loader is not as restrictive with the PE headers formats as the PE header specification.

- **Anti-virtual-machine, anti-emulation**

  A very common technique is to execute suspicious binary files into a controlled environment like a virtual machine or emulator [GFC08]. This approach allows the analyst to monitor the execution of the sample (i.e. observe its interaction with the system, or its behaviour), and isolate the machine to prevent the sample from infecting the guest machine or spreading through the network.

  Dynamic analysis systems can resolve some of the limitations of static analysis, but, with the exception of multiple-path execution engines, only allow the analyst to observe one execution path. If, for any reason, the malware sample does not show its real behaviour and applies any conditional code revelation technique, the dynamic analysis will not be able to provide useful results. In order to take advantage of this limitation, malware authors have developed different techniques to detect the presence of virtual-machines and emulators [Fer07, RKK07].

  Virtual machines use an isolated memory and disk space and execute a complete operating system that runs independently of the host machine. Virtual machines, in addition, allow to execute machine instructions directly on the physical processor, thanks to an extended instruction set provided by Intel Virtualization Technology[1]. For this reason, virtual-machine based environments are considered more efficient than pure software emulators, which provide a software implementation of the guest architecture. In contrast, emulators allow a complete control over the execution, allowing low-level instrumentation with any granularity. This approach is useful to implement advanced dynamic analysis systems such as taint-analysis engines, or multi-path exploration engines [SBY+08]. In addition, emulators are more difficult to detect [BK09].

  There are two main approaches to detect a virtual-machine or an emulator. First, a sample under analysis can check the presence of files, registry keys, environment variables or specific processes characteristic of certain virtualization systems. On the other hand, it is possible to detect semantic differences between the real implementation of

[1] Intel Virtualization Technology is an extension of Intel 32 and 64 bits architectures that provides hardware resources to implement virtualised environments: `http://ark.intel.com/Products/VirtualizationTechnology`

some specific machine operations, and the implementation given by an emulator or virtual-machine [BCK+10]. Timing-attacks can also discover differences between the execution of the sample in native machines and the execution in controlled environments.

Together with detection, Ferrie distinguishes 3 different attacks to virtual-machines or emulators [Fer07]:

- *Virtual-machine detection.* Virtual-machine detection exploits the main limitation of dynamic analysis systems, making the execution flow of the sample to be subject to conditions based on the detection of this kind of systems.
- *Denial of Service.* These attacks are more aggressive, trying to exploit software vulnerabilities in these systems to force their termination or to make the analysis infeasible.
- *Virtual-machine evasion.* Finally, although very uncommon, a malware sample may eventually take advantage of security flaws to evade the guest-system and infect the host machine or other machines in the network.

2.1.3.4 **Other common features of packed binaries**

Mody et al. [MMF10] enumerate the characteristics shared among packed binaries. Not all packers implement all the features listed, but they commonly combine different approaches to make the obfuscation more resilient.

- **Polymorphic/junk code.** Some packers (e.g., TeLock, ACProtect, Obsidium, Morphine) obfuscate unpacking routines.
- **Excessive branching.** Some protectors create parallel execution paths (i.e., parallel branching) that have the same functionality. These paths are chosen according to the values adopted by insignificant flags during execution. This approach makes difficult the task of setting breakpoints to debug and reverse-engineer the code. One example of this kind of packer is Execryptor. Other packers distribute the code of certain functionalities over all the memory space, linking the code blocks with unconditional jumps and avoiding the existence of patterns in the code. Obsidium and PESpin use this technique.
- **Excessive loops.** Morphine implements loops that do not implement any useful functionality. Sometimes, these loops can be employed to detect the presence of emulators (delays in execution).

- **Unusual control transfer.** Exceptions are employed by several packers: Obsidium, Armadillo, ASProtect, Enigma, PECompact. These methods consist of installing Structured Exception Handlers (SEH) or Vectored Exception Handlers (VEH) with the desired functionality, and provoking exceptions intentionally to change the natural control flow at a certain point. This resource should only be employed to treat execution errors.

- **Unnecessary use of unusual instructions.** The excessive use of MMX or floating point instructions can be considered suspicious. In any way, these instructions are totally legitimate. NTkrnl, XTreamLock and KKrunchy packers use this instructions to complicate the analysis of samples.

- **Use of unusual structural features.** Many packers present unusual structural features such as unaligned, overlapping sections or non-standard section names and permissions. Other packers use certain parts of the PE header to hide data or code to reduce the size of the file, or simply modify common values in the header that are rarely modified by legitimate software. Despite there are not practical motivations to do these changes, many common packers implement them. For example, UPX, which is one of the most common packers, names the sections `UPX0`, `UPX1` and `UPX2`.

- **Faking other packers.** Some packing engines substitute different features like section names with strings or signatures typical of other packers in order to confuse the analyst or automatic analysis platforms. NSAnti is an example of this technique.

- **Limited compatibility with different environments.** Whereas legitimate applications are compatible with many different environments, malicious applications are not always implemented in this way.

- **Use of undocumented APIs or instructions, or non-standard use of APIs.** Some packers like SVK-Protector or Obsidium call to rare API functions that are not common in legitimate software.

- **Destruction of data in the original file.** Some packers eliminate some data of the original file like digital signatures, versions, or even icons.

- **Watermarks, user licences.** Some packers like VMProtect include licence information or watermarks.
- **Poor programming standards.** Some non-commercial packers tend to present low quality code and sometimes can present software errors.

Finally, Roundy and Miller [RM13] recently elaborated an interesting survey on binary-code obfuscations found in prevalent packer tools. For the sake of simplicity we do not cite all the techniques mentioned in the survey, but we rather point it out as an interesting reference on this kind of techniques.

2.1.4 Unpacking techniques

Software packing stands as a problem when it comes to analyse the real content of a sample. Both current malware and legitimate software have the same objective: conceal the real code, and prevent reverse-engineering. For this reason, it is necessary to unpack software in order to make a complete analysis of the samples.

In this way, the packer problem is usually handled treating the sample in different stages. According to Sun [Sun12], we can distinguish 3 different challenges:

- **Packer detection.** The first problem to tackle is to determine if a certain sample is packed or not, in order to continue with the analysis process or to unpack it first.
- **Packer identification.** Another aspect to consider is the identification of the packer that protected the sample. Some packers are very common and widely used to protect both legitimate and malicious software. Signature scanning methods can be applied for packer detection in the same way as it is applied to malware detection. Nevertheless, due to the obfuscations some packers may apply, this approach presents many limitations. It is well-known that malware writers usually modify existing versions of packers in order to evade detection. Considering these limitations, identifying the packer used is an interesting challenge. It makes possible to develop specific unpacking methods for common packers. Specific unpacking routines are usually the most efficient and effective method to obtain the original code of a sample.

- **Unpacking.**

  The following definition describes the objective of unpacking according to Sun [Sun12]:

  «*The objective of PE unpacking is to remove the PE packer's unpacking stub and restore the original executable.*»

  Nonetheless, Sun [Sun12] also states:

  «*However, in the AV industry, sometimes a dumped memory of unpacked file which exhibits malicious code is sufficient.*»

  Academic community and antivirus industry have proposed different approaches for software unpacking.

2.1.4.1 Static unpacking

Static unpacking approaches have traditionally relied on specific decompression or decryption routines for well-known packers, avoiding to execute the sample. Nevertheless, some other approaches have applied static techniques for generic unpacking [CDKT09].

2.1.4.2 Dynamic unpacking

The most common approach adopted by the community in order to deal with the unpacking of samples is to implement dynamic execution environments in order to monitor the unpacking behaviour of the sample.

Normally, the execution of potentially malicious code is performed in controlled environments like virtual machines or emulators. We can distinguish 3 different environments [Fer07]:

- **Hardware assisted virtual machines**

  Virtual machines use instructions and hardware mechanisms of the processor specifically implemented for virtualization, allowing to execute most of the instructions of the guest machine in the host processor. Intel VT is an example of this technology. The effects of the execution do not affect the host system, making the approach almost transparent. Ether [DRSL08], V2E [YJZY12] and Spider [DZX13] are examples of this kind of approach.

– **Reduced-privilege virtual machines**

  This kind of tools emulate data structures and registers using software resources, while instructions are executed over the real machine to enhance the efficiency with respect to pure emulated machines. These virtual machines execute the guest operating system code with reduced privileges over the host operating system. ZeroWine[1] is a malware analysis sandbox developed over Wine[2], an implementation of the Windows API system for UNIX based systems.

– **Emulators**

  Emulators are pure software virtualised environments that implement a complete CPU including registers and necessary data structures. Their main disadvantage is their low efficiency, although they offer a complete control of the system. Some approaches have based their monitoring engines on this technique: Bitblaze [SBY+08], Anubis [BKK06] or Pandora's Bochs [Böh08].

2.1.4.3 **Generic unpacking**

Generic unpacking is intended to recover the protected code for any possible packer without any knowledge of the encryption of compression algorithm used.

2.1.4.4 **Heuristic approaches**

Some solutions use heuristics to determine when the unpacking stub finishes unpacking the original code. Martignoni et al. [MCJ07] use the following heuristic to determine the moment in which the unpacking stage is completed:

«[...]if the current execution trace indicates unpacking (i.e., memory pages were written and then executed), and if the program is about to invoke a dangerous system call, then we assume that an unpacking stage has completed and we invoke the malware detector.»

[1] `http://zerowine.sourceforge.net/`
[2] `https://www.winehq.org/`

2.1.4.5 Static analysis vs. dynamic analysis

When the packer used to protect a sample is unknown, unpacking the binary requires analysing the code or the behaviour of the sample, either manually or automatically. This analysis can be performed using two different approaches: static and dynamic analysis.

Each of the approaches present advantages and disadvantages. Dynamic analysis executes the sample in a controlled environment, allowing to monitor its execution. A sample can be traced at different abstraction levels, registering certain events (e.g. Windows API calls, system calls, interaction with Windows Registry, network traffic, processes and thread created, mutexes created) or even registering every machine instruction executed. Typically, the observed behaviour will provide a view of only one execution path. If other execution paths are triggered by certain events (e.g. presence of certain applications or configuration in the system, connectivity with a network server, or even events such as date or time), the alternative execution paths will not be observed by the analyst. Despite of this inconvenient, dynamic analysis solvents some of the limitations present in static analysis.

Static analysis, on the contrary, involves any method that enables the analyst to gather information about the binary without executing it. Static analysis provides high-level information such as file format, imported libraries or printable strings, and low-level information such as machine code (disassembly) or source code (decompilation) [Böh08].

This approach allows the analyst to obtain a general view of the contents of the file. Static disassembly does not depend on a single execution path, and allows the exploration of all the different execution paths. The main limitation of static analysis is its complexity. First, the reverse engineering process requires a great effort for a software analyst. The complexity of current architectures and the low level view of assembly language requires the experience of an expert in the field. In addition, obfuscation methods such as anti-disassembly tricks, self-modifying code or packers can complicate or even impede static analysis confusing automatic analysis tools such as disassemblers.

Disassemblers extract from the original code of the file (machine code) the equivalent mnemonics for the machine operations and the param-

eters used in such operations. This process presents several limitations. First, only a part of the content of the file is code. The rest may be data or different resources that should not be disassembled. There is no method to determine which parts of the content are code, and which are data. In fact, the PE file format used in Microsoft Windows systems specifies that the code should be stored in a section named *.code*. Nevertheless, the system loader is permissive with this specification and it actually allows to store code in any part of the file. One possible way to determine the place where the code starts is to start disassembling the file at the entry point (i.e. first address executed) specified in the PE header.

Second, the CISC architecture that today dominates the market (x86-64 architectures) complicates the disassembly task. The size of each instruction (operational code and operands) varies from 1 byte to 15 bytes. On the contrary, RISC architectures normally employ the same number of bytes for each instruction. The consequence is that, in a CISC architecture a single error will affect the whole disassembly process, while in RISC architecture it may affect only 1 instruction.

To deal with these difficulties, several disassembly methods have been proposed [Böh08]:

- **Linear Sweep**

  This method disassembles the code section of the file, instruction by instruction in sequential order. It does not consider jumps in the execution flow or code blocks. In this manner, it performs poorly when data blocks are embedded into the code, or when some memory positions are not used between blocks of code. In these cases, these bytes will be disassembled as instructions, resulting into invalid instructions until an invalid operational code is found. This method is fast but can be easily mislead.

- **Recursive traversal**

  Another approach to disassemble a binary is to start the process for the bytes pointed by the entry point (i.e. first instruction that will be executed) and then continuing until the first procedure call or jump to another address is found. The process then continues disassembling the destiny address. In this way the disassembler identifies the basic code blocks of the program (i.e. pieces of code with a single point of entry and a single point

of return). The inconvenient of this method is that it cannot always calculate the destiny address in every call or jump. These addresses can be contained in registers or memory, and be the result of previous operations. This problem adds a great complexity considering that the addresses sometimes depend on input values.

– **Speculative disassembly**

  The method proposed by Cifuentes et al. [CVERL02] is based on Recursive Transversal but, in addition, tries to disassemble all the blocks of bytes that are not contained in any basic block. If it finds an invalid operational code, it stops the analysis and discards the code block.

– **Static disassembly of obfuscated binaries**

  Kruegel et al. [KRVV04] proposed a method to statically disassemble obfuscated binaries. The first step was to identify the functions or procedures finding patterns relative to function prologues (i.e. instructions inserted by compilers to set up the local stack frame). Once identified, they located the intra-procedural branches (i.e. branches to addresses inside the functions). With this information they built an intra-procedural control flow graph (CFG), and reduced the graph using an algorithm in order to obtain the real CFG. In addition, they used an statistical approach based on common instructions to disassemble byte blocks for which no instruction had been disassembled.

2.1.4.6 Decidability of the detection of the unpack-execute process

Malware analysis generally requires to identify the existence of an unpack-execute process. Royal et al. [RHD+06] formally express this property at the end of their publication describing Polyunpack, a dynamic generic unpacker.

They define a packed binary as a program P that has at least one immutable code section (unpacking routine) and optionally one or more data sections. During its execution, P can write data over any of its data sections and then execute them. This property allows to formulate the packed binary as a *Universal Turing Machine* (UTM) that simulates another UTM in its input tape: the immutable code section

corresponds to the immutable control states of a UTM, while, the mutable data sections correspond to the mutable input tape of a UTM. In this way, Royal et. al. formally prove that determining if a UTM simulates another UTM in its input tape is an undecidable problem.

More recently, Bueno et al. [BCSB13] have demonstrated that under certain assumptions (boundedness of space and time), detecting traditional unpacking behaviour is an NP-complete problem. Nevertheless, problems like obtaining a sound disassembly of highly obfuscated software, statically determining the values that can be adopted by variables, or achieving a complete dynamic exploration of all the code-paths in a binary are open research problems.

2.2 Classification of unpacking techniques

In this dissertation we follow a classification for unpacking methods that reflects the different phases that a sample analysis may require. Regarding unpackers, we divide them into 2 categories: specific unpackers (designed for known packers) and generic unpackers (designed for unknown packers). In any way, there are two additional tasks to consider as part of the unpacking process. On the one hand, packer detection tries to determine if a sample or a certain piece of data is packed. Besides, packer identification tries to identify the packer (or the packer family) used to protect a sample, if present.

2.2.1 Packer detection and identification

Malware analysis requires to determine if a sample is packed or not in different phases. This information is necessary to conduct a complete sample analysis. Packer detection is generally applied in two different moments of the sample analysis workflow:

- **Sample filtering.** According to statistics, up to an 80% of the malware is packed [McA09]. Nevertheless, depending on the nature of the dataset, this percentage may vary. Considering that unpacking software is generally a computationally expensive task, sample filtering can reduce the number of samples to unpack, making the analysis more efficient.

- **Generic unpacking.** Packer detection is a key task in the generic unpacking process. When a sample is running an unpacking process, it is necessary to determine when the sample may be fully unpacked, meaning that the unpacking process succeeded. Besides, it is also important to determine if, after unpacking a sample, there are more layers of protection remaining.

Packer identification goes one step further. Several studies reveal that up to the 65% of packed binaries are protected using packers available on the internet [MP10, GFC08]. These packers could be considered known packers, while the rest (35%) could be considered custom made packers, designed by software authors to protect their creations. Identifying known packers is a very useful technique to efficiently manage samples. It allows analysis systems to apply specific unpacking methods to unprotect the samples or, in the worst case, it can provide the analyst with very useful information about the sample.

Figure 2.1: Known and unknown packers.

Figure 2.1 shows the universe of executable files. Let N be the number of known packers and M the total number of existing packers such that $N <= M$. If we assign a numerical value i to each packer, every known packer would satisfy $0 < i <= N$, whereas any unknown packer would satisfy $N < i <= M$.

In this way we can define $\mathcal{P}$ as the set of packed executables. The set containing the binaries protected by known packers is denoted $\mathcal{K}$, and is defined as the union of all the sets containing the executables protected by all the known packers. Similarly, the set of unknown packers $\mathcal{U}$ would be defined as the union of all the sets containing the executables protected by unknown packers.

$$\mathcal{K} = \{\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \cup ... \cup \mathcal{P}_N\} \tag{2.1}$$

$$\mathcal{U} = \{\mathcal{P}_{N+1} \cup \mathcal{P}_{N+2} \cup \mathcal{P}_{N+3} \cup ... \cup \mathcal{P}_M\} \tag{2.2}$$

were $\mathcal{P}_i$ is the set containing all the executables protected by packer i. $\mathcal{K}$ and $\mathcal{U}$ are disjoint sets such that $\mathcal{K} \cap \mathcal{U} = \emptyset$. If we assume that $\mathcal{K} \neq \{\emptyset\}$ and $\mathcal{U} \neq \{\emptyset\}$, $\mathcal{K}$ and $\mathcal{U}$ would satisfy $\mathcal{K} \subsetneq \mathcal{P}$ and $\mathcal{U} \subsetneq \mathcal{P}$. In any case, $\mathcal{P} = \mathcal{K} \cup \mathcal{U}$. We also define the set $\mathcal{NP}$ as the set of executables that are not protected by any packer at all. In this way, $\mathcal{U} = \mathcal{P} \cup \mathcal{NP}$.

Considering this distribution for executable files, we define a packer detector as a function that, for any sample, returns a dichotomous outcome: 0 if the sample is not packed, and 1 if the sample is packed.

$$d : S \longrightarrow C \mid d(s) = \left\{ \begin{array}{ll} 1 & \text{if } \exists i.i \leq M.s \in \mathcal{P}_i \\ 0 & \text{if } \forall i.i \leq M.s \notin \mathcal{P}_i \end{array} \right\} \tag{2.3}$$

were S is the domain of executable samples and C is the domain that comprehends the possible classes of the sample, in this case $\{0, 1\}$.

Likewise, we define a packer identifier as a function that, for any sample, returns the packer that protected it.

$$i : S \longrightarrow C \mid i(s) = \left\{ \begin{array}{ll} i & \text{if } \exists i.i \leq N.s \in \mathcal{P}_i \\ 0 & \text{if } \forall i.i \leq N.s \notin \mathcal{P}_i \end{array} \right\} \tag{2.4}$$

were S is the domain of executable samples and C is the domain that comprehends the possible classes of the sample, in this case $\{1, 2, 3...N\}$.

At this point, we must notice the equivalence between a packer detector and a packer identifier. We can observe that a packer identifier works as a packer detector when N approximates to M, if we simplify the output of the identifier.

$$d' : S \longrightarrow C \mid d'(s) = \left\{ \begin{array}{ll} 1 & \text{if } \exists i.i \leq N, N \doteq M.s \in \mathcal{P}_i \\ 0 & \text{if } \forall i.i \leq N, N \doteq M.s \notin \mathcal{P}_i \end{array} \right\}, d' \equiv d \tag{2.5}$$

We can classify packing detection and identification methods following two different criteria: static or dynamic, and the data analysis technique employed.

Regarding the execution of the sample, we can divide packing detection and identification approaches into static and dynamic. In this case, the division is not as clear as for malware analysis. For example, static detection could be applied as a part of a dynamic unpacking scheme. In this case,

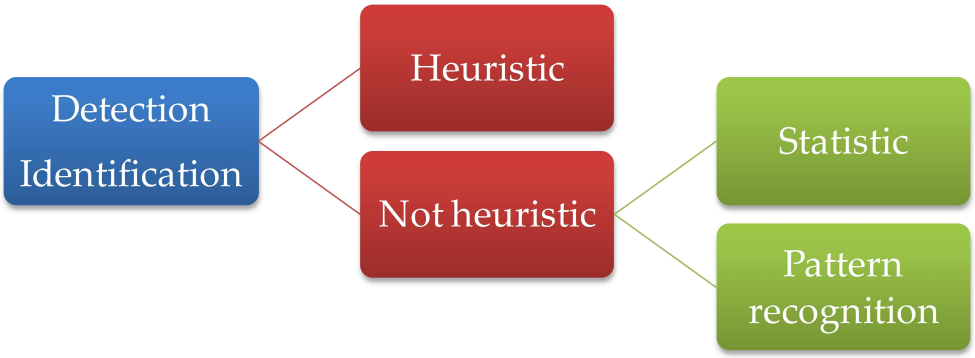


Figure 2.2: Classification proposed for packer detection and identification approaches.

although the general approach is dynamic, the detection process would not be dynamic if the data it employs for analysis is strictly static. Lets suppose that a dynamic unpacking method executes the sample under analysis into a controlled environment, waiting until a moment in which the execution jumps from the unpacking routine to the original code (also called *tail-jump*). In order to detect the tail jump, the unpacker could analyse the sections of the binary prior to any execution, measuring the information-entropy. The information-entropy measures the quantity of information of a certain block of data, as well as its randomness. If the data analysed is compressed or encrypted, then it may present a high randomness. A jump to a section that was presumably compressed before execution may be a good indicator of the *tail-jump*. Besides, the system could use a different approach, considering as unpacked any memory position written and afterwards executed. The difference between the two methods is that, in the first case, the data employed for packer detection is obtained statically, while in the second case, it is strictly derived from the execution of the sample.

Both static and dynamic approaches can be theoretically applied for sample classification or as part of generic unpacking methods. Nevertheless, due to the computational cost of dynamic approaches, dynamic detection would not be applicable in certain circumstances like sample filtering, and it is practically infeasible.

Besides, Figure 2.2 shows how the data analysis techniques employed for packing detection and identification can be classified into different categories:

- **Signature scanning.** Signature scanning is a traditional malware detection technique that consists of searching for certain sequences of bytes in the binaries. The sequences, which are usually stored in a database, represent fragments of code or data that are characteristic of a certain packer and that are not normally present in non-packed binaries.

- **Statistical approaches.** Statistical approaches, in contrast, do not depend on any signature database. These approaches are based on the analysis of the byte distribution in the executable in order to differentiate between packed and non-packed executables. Randomness analysis, byte frequency, or entropy analysis are examples of this approach.

- **Heuristic approaches.** Heuristic approaches are based on the knowledge acquired by analysts on how software packers work. These heuristics can be considered as rules of thumb that apply to many packers in the wild.

- **Pattern recognition.** Finally, other approaches are based on pattern recognition techniques. These methods try to build mathematical models to classify samples using any kind of data from the executable. A common technique for pattern recognition is machine-learning.

2.2.2 Unpackers

Regarding unpacking, we must consider two main categories: specific and generic unpacking. Apart from this division, we must also consider the technique employed for the analysis: static or dynamic. Figure 2.3 shows the possible combinations.

2.2.2.1 Specific unpacking

Specific unpacking is applied when the packer used to protect the sample has been identified, and it is possible to apply an specific routine (script or program) that has been developed to unprotect the sample. The development of these routines requires a previous reverse engineering process in order to understand how the unpacking routine works. An specific unpacker can be either static or dynamic.

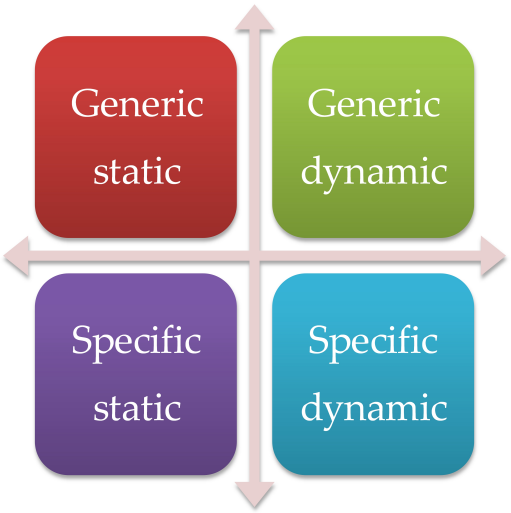


Figure 2.3: Classification proposed for unpacking approaches.

2.2.2.2 Generic unpacking

Generic unpacking is applied to unprotect packed samples for which there is no specific unpacking method available. In this way, it is a method applied to samples packed by unknown packers or custom-made packers (*custom-packers*). Both static and dynamic approaches can be applied. Nevertheless, static approaches present a high complexity, and thus, the majority of generic unpackers proposed have been designed as dynamic systems. Dynamic generic unpacking systems usually execute the sample in a controlled environment, monitoring its execution [Szo05] and employ different heuristic rules to determine the point in which the execution transitions from the unpacking routine to the original code, or *Original Entry Point (OEP)*. In this moment the memory of the process can be analysed or dumped for a posterior analysis.

Some of the possible heuristics are:

- **Instruction pointer rule.** This heuristic takes two assumptions: (i) the unpacking routine unprotects all the original code before executing it, and (ii), when the unpacking routine finishes, the execution jumps to a different section (the instruction pointer, thus, will point to a different section).
- **Stack pointer rule.** The stack pointer rule considers that common packers use the stack to temporarily save the initial execution context to recover it before continuing with the execution of the original code.

- **Compiler signatures.** Some compilers (Microsoft Visual C++, GNU GCC, Dephi) generate specific code on the entry point of the binary that can be identified by certain signatures. When the code is unpacked, these signatures are revealed and an unpacker can presumably detect the original entry point of the binary.

- **Behaviour rules.** Many binaries exhibit a similar behaviour that are not common in unpacking routines, such as calling to certain Windows API functions (e.g., CreateWindowA). If a packed binary calls such API functions it may be already executing the original code.

2.3 Antivirus solutions

Antivirus solutions are the main shield to protect systems against current malware. In contrast to other tools or approaches, these products have strong requirements and generally need to provide real-time protection. Unfortunately, some of the techniques proposed in the literature present high computational requirements and cannot be deployed in end-user machines. Antivirus tools have traditionally relied on signature scanning methods to identify malicious software in the system [Szo05]. The effectiveness of this method depends on the completeness of the signature database. These databases must cover the possible variants of each malware family in order to detect any mutations. In addition, the solutions should not unintentionally flag legitimate software as malware (i.e., false positive), preventing the user from installing legitimate applications.

The signature detection systems used by first-generation anti-malware solutions are also called *string signatures*. These signatures consist of sequences of bytes present in a certain malware sample, that are, ideally, not present in any legitimate binary of the system. There exist special cases that try to add some flexibility to the scanner [RMI11]: *wildcards* (bytes undefined inside the signature string similarly to regular expressions), *mismatches* (presence of a concrete number of contiguous bytes in the binary) or *generic degree* (unique signature for all the variants of a malware family).

A common technique to avoid raising false positives is the use of bookmarks [RMI11] (e.g. size of the malicious code, position of the code with respect to a certain point in the binary) in order to avoid casual coincidences in legitimate binaries. In order to improve the efficiency of these methods, some anti-malware solutions use other techniques such as *hashing* or *Top Tail Scanning* (i.e., selective scanning of certain memory regions [CO07],

as well as *Entry Point* and *Fixed-Point scanning* (i.e., starting the scanning process at a certain point of the binary in which it is more likely to find the malicious code [Szo05]).

Unfortunately, the generation of signatures is a hard work considering the current malware trends [Zha08]. Malicious software developers automatise the obfuscation of their malware creations to evade detection systems. As a consequence the number of samples has grown in an uncontrolled way in the last years.

The second generation of anti-malware solutions applied advanced techniques to enhance the effectiveness in the presence of more complicated malicious software. In this way, *smart scanning* tries to solve problems like junk-code insertion [RMI11] (i.e., instructions that do not modify the behaviour of the sample but change the signature). Other methods of this generation are *Nearly Exact Identification* (i.e., using composed signatures to reduce false positives or to distinguish the variants) and *Exact Identification* (i.e., identify all the bytes that do not vary among the samples of the same family) [RMI11]. Another approach implemented by this generation is heuristic detection, applying certain rules based on the knowledge about common characteristics among malware (e.g., structure, behaviour, or other attributes). This heuristics can be applied as part of an static or dynamic analysis process [Nac98].

Occasionally, anti-malware solutions have to apply specific algorithms to detect malware that cannot be detected by the signature scanning engine [RMI11]. In order to optimise these algorithms as well as signature scanning, some solutions filter the files to analyse to avoid all the files that are not susceptible of being infected by a certain malware [Szo05].

Commercial solutions follow several objectives [GFC08]:

- **Effective.** It must obtain the original code from the protected sample.
- **Generic.** It should be able to deal with as many packers as possible.
- **Safe.** The unpacking process should not damage the system at any moment.
- **Portable.** Ideally, it should be compatible with different systems or platforms.

In order to comply with this requirements, the anti-malware industry employs different approaches [GFC08]:

- **Traditional approach.** It is based on specific unpacking. It tries to identify the packer and, if possible, applies an specific unpacking routine to it. It requires to reverse engineer the most common packers but it is efficient and effective for known protectors. Its main disadvantage is that it is useless for unknown packers. The Sympack library by Symantec was an example of this approach [GFC08]. Another approach is X-ray scanning, that applies crypto-analysis techniques to recover the original code without executing the sample or applying specific unpacking routines [Szo05].

- **Controlled execution.** This approach executes the packed software in a controlled environment to analyse the original code once unpacked. Anti-malware solutions typically employ emulators and heuristics to determine when the sample is unpacked.

- **Memory inspection.** Finally, a different approach analyses the code in memory periodically or when certain events occur, in order to find unpacked malicious code. Eraser Dump was an example of this approach developed by Symantec [GFC08].

Software packing is a legitimate technique employed by software authors to protect their products. For this reason, it is not adequate to flag any packed binary as malware. Nevertheless, the IEEE ICSG suggests to use commercial packers with licensing capabilities, applying *watermarking* techniques or *taggants* to include encrypted information about the license and the legitimacy of the packer and the software protected. Nevertheless, Mody et al. [MMF10] identify several inconveniences of this approach:

- It can only be applied to commercial packers, and, unfortunately, Guo et al. [GFC08] found a relevant number of legitimate software samples protected with *custom-packers*.

- It would require to coordinate all the packer developers to ensure that all packers meet the standards related to software certification.

- A malware author may steal a valid licence to sign malware. This risk should be mitigated implementing a *black-listing* mechanism to revoke licences.

- The *watermarking* method would be susceptible of sabotage by malware authors in order to bypass security measures.

- Some executable formats different from Microsoft PE executables may present limitations.

Finally, recent reports [McA13] highlight the fact that the number of malicious samples signed is increasing.

2.4 Packer detection and identification

Several authors have proposed different methods to statically detect packed software. These approaches can be classified into statistic, heuristic, and pattern-recognition approaches. Many solutions combine several aspects of them. Anti-malware solutions like McAfee [McA08] or well-known tools such as PEiD[1] leverage statistic detection techniques such as entropy analysis.

Statistic detection is based on the analysis of the byte distribution in the binary, that usually changes for compressed or encrypted data (note that not all the packers may use this technique to protect the code). As an example, when data is compressed or encrypted, it usually presents a high information entropy. This feature can be used for the detection of packers. Bintropy is a tool proposed by Lyda and Hamrock which analyses binaries to determine if they are packed or not [LH07]. To this aim, they calculated average and maximum entropy of 512 byte long blocks for a set of packed and non-packed executables and determined the entropy interval in which samples were considered packed with a confidence of 99.9%. Although the results of this approach are outstanding, the dataset used for the experiments consists of samples protected very simple packers (MEW, UPX and Morphine), and the well-known cryptographic tool (PGP). Another measure commonly used is byte frequency. Packed binaries tend to present an almost random distribution of bytes, which means that all the possible bytes (values from 0 to 255) have a similar frequency in the file [Sun12].

More recently, some authors have combined pure statistical approaches with heuristics for packed binaries. Perdisci et al. [PLL08a] proposed a filter based on heuristics and entropy analysis to detect packed binaries. More concretely, they tested several machine-learning classifiers trained with a dataset formed by packed and non-packed executables and the following features: number of standard and not standard sections, number of executable sections, number of sections with read, write and execution permission, number of entries in the IAT, and entropy of the header,

[1] http://www.aldeid.com/wiki/PEiD

code sections, data sections, and entropy of the file. Later, the same authors proposed McBoost [PLL08b], a complete analysis system that added a new approach to the previous work: n-gram analysis. In this work, 3 different classifiers were combined: (i) the classifier proposed in previous work [PLL08a], (ii) a classifier based on n-gram analysis for code sections, and (iii) a similar classifier, based on n-grams of the complete file.

Table 2.1: Comparison of the different approaches proposed for packer detection and identification: the work of Perdisci et al. [PLL08b, PLL08a], PE-Probe [STF09] and PE-Miner [STMF09]. For each of the approaches, we specify the feature-set proposed (i.e., heuristics, statistical properties or structural characteristics), the type of classification and the classification method. P/NP refers to classification into Packed and Non-packed categories, while MW/GW refers to classification between Malware and Goodware (legitimate software).

Features	**Perdisci et al.**	**MacBoost**	**PE-Probe**	**PE-Miner**
Use of heuristics	✓	✓	✓	×
Statistical properties	×	✓	×	×
Structural characteristics	×	×	×	✓
Classification type	P/NP MW/GW	P/NP MW/GW	P/NP	MW/GW
Classification Method	Machine Learning	Machine Learning	Machine Learning	Machine Learning

Shafiq et al. proposed PE-Probe [STF09] as an optimization of PE-Miner [STMF09]. In this case, they used a classification system to detect packed files based on the features used by Perdisci et al. [PLL08a, PLL08b]. Then, they applied different classification methods to differentiate between malware and legitimate software, using different classifiers and feature-sets for packed and non-packed files. This last classifiers were based on structural features extracted from the headers of the binaries. Nevertheless, they did not apply the whole set of structural features for the classification of packed and non-packed files.

Choi et al. [CKOR08] proposed in PHAD a packer detection system based on 8 heuristics (standard and non-standard sections, read/write/ex-

Table 2.2: Comparison of the different approaches proposed for packer detection and identification: Bintropy [LH07], Reform [Sun12] and PHAD [CKOR08]. For each of the approaches, we specify the feature-set proposed (i.e., heuristics, statistical properties or structural characteristics), the type of classification and the classification method. P/NP refers to classification into Packed and Non-packed categories, while MW/GW refers to classification between Malware and Goodware (legitimate software).

Features	**Bintropy**	**Reform**	**PHAD**
Use of heuristics	×	×	✓
Statistical properties	✓	✓	×
Structural characteristics	×	×	×
Classification type	P/NP	Packer identification	P/NP
Classification technique	Statistical	Mach. Learning Vector Space Model	Vector Space Model

ecute permissions of sections, size of sections and entry-point). Some of the heuristics had already been used in previous work. The 8 values selected conformed a multidimensional vector that adopted a value of 0 for any non-packed sample. The authors conducted an statistical validation of this assumption, but the dataset used to this end was not very representative of current malware. Any sample that presented a significant deviation to that representation according to the Euclidean distance between vectors was considered as packed.

Park et al. [PRM11] proposed in BinStat a method based on n-gram analysis and different statistical and information-theory based metrics over the raw data of the binary in order to discriminate packed and non-packed binaries. Their approach applied decision trees in order to provide a binary output (packed or not packed), and did not rely on heuristics or other features.

More recently, Jacob et al. [JCN+13] proposed an approach that aimed to group samples by their similarity even in cases in which packer tools used to protect them. Their system followed the assumption that the original code protected by certain compression or encryption algorithms still

maintains similarities. They also proposed a series of tests that allow to distinguish between different packing approaches (i.e., compression, encryption). They also applied a reduced set of structural features for prefiltering samples from a database in order to reduce the number of samples for a more time-consuming comparison stage.

Current antivirus solutions also implement methods to detect packed binaries. MacAfee, for example, measures the information entropy of the different sections of the binary, together with the presence of code loops near the entry-point (possible unpacking routines) and jump instructions to memory sections with write permissions [McA08].

Regarding packer identification, the majority of the solutions are based on signature-scanning. This method allows to efficiently identify most known packers, but presents the same limitations as malware analysis. In general, signature-scanning is vulnerable to polymorphic and metamorphic transformations. Any packer which applies this kind of techniques to the unpacking routine could potentially bypass signature-based detectors. One well-known tool to detect packers is PEiD, which provides a signature database that covers most common packers. In addition, it provides entropy analysis, generic original entry point searching and a plugin infrastructure to develop, for example, unpacking tools. In this manner, Naval et al. proposed in SPADE [NLGV12] an approach to automatically generate packer signatures using sequence alignment methods largely used in bioinformatics and DNA sequencing. Although they achieved sound results, their experimental set was limited to 7 simple packers and did not address advanced techniques such as metamorphism.

Besides, Li Sun [Sun12] proposed in her PhD dissertation an alternative representation method to identify packers based on an statistical approach. This representation measured the randomness of the byte distribution along the executable, creating a randomness profile of each binary. Afterwards, she employed different methods to identify the packer used taking as input the representation proposed. First, she measured the similarity between executable representations using Euclidean and Cosine distances, identifying the most similar packer representation to the sample under analysis. Second, she employed machine-learning algorithms *Naive Bayes*, *Sequential Minimal Optimization*, *K-Nearest Neighbour*, *Best-first Decision Tree*) to train classifiers for packer identification. The inconvenient of this approach is its dependence on the position of the encrypted payload inside the binary.

Bat-Erdene et al. [BEKLL13] later proposed as an alternative a model for

packer algorithm classification based on Symbolic Aggregate Approximation and entropy analysis. Similarly, Ban et al. [BIG+13] proposed a pattern recognition approach based on SVM classification models with the same objective.

Dynamic approaches have higher computational requirements than any static technique. Although statistical approaches require more complex calculations than simple heuristics, these techniques are sometimes less evident to circumvent. Besides, pattern recognition approaches require complex computations to build classification models but once trained, these models are fast for classification.

For this reason, dynamic approaches are not commonly used to detect packed binaries. Several authors [DCT08, GMRP09, MR13, AMDB07] have addressed the formalisation of self-modifying code and other unpacking behaviours modelling not only static but also dynamic execution features. In contrast, dynamic unpacking systems commonly apply heuristic rules that implicitly determine if the sample is packed or not. A typical heuristic for dynamic unpacking is the «written then executed »rule. Omniunpack [MCJ07] and Renovo [KPY07] monitor memory writes and memory execution, at instruction and memory page level respectively.

A few approaches have focused on understanding the inner structure of the packer. Debray and Patel [DP10] proposed a method to post-process the execution trace of a packed binary in order to extract the unpacker. Vera [QL09], in contrast, is a tool focused on the visualization of execution traces to aid reverse engineering. Although the tool has been applied to understanding packers, the granularity of the visualization may not be appropriate for understanding complex packers.

Finally, Table 2.1 and Table 2.2 show a comparison between the different approaches proposed in the literature for packer detection and identification. Apart from signature scanning based approaches, not many authors have focused on the problem of identifying the packer that protected the sample.

Regarding packer detection, the different static approaches described apply different classification techniques and feature-sets for packed file filtering. Heuristic values, entropy, n-grams, and byte randomness are some of the features considered in previous work. Nevertheless, none of them has performed a complete study of the capacity of structural features to differentiate packed and non-packed malware. In the case of PE-Miner these features were applied to discriminate malware and legitimate software. In addition, the majority of the methods proposed are based on su-

pervised machine-learning techniques. However, other classification approaches can be explored such as anomaly detection, semi-supervised or unsupervised learning.

2.5 Unpacking

This section reviews the efforts from both the academic world and the security industry in the field of unpacking malware. We divide these approaches into 3 categories: manual unpacking (Section 2.5.1), specific unpacking (Section 2.5.2) and generic unpacking (Section 2.5.3).

2.5.1 Manual unpacking

Manual unpacking requires to reverse-engineer the sample in order to understand the structure of the unpacking routine. Static and dynamic tools help the analyst in this process. The unpacking process can be performed both statically or dynamically. Static approaches require to know the compression or cryptographic algorithm employed to protect the code and data, and, in case of cryptographic approaches, the key employed. In these cases it is possible to apply the same decompression or decryption algorithm to obtain the original code.

Nevertheless, this approach is not very common. The most common technique is to execute the samples in a controlled environment like a debugger stopping the execution when the unpacking routine is finished. In this moment it is possible to dump the memory of the process to obtain the original code. Determining the moment in which it is possible to obtain the original code requires knowledge and experience. There are also certain heuristics that can be applied in order to discover the original entry point (described in Section 2.2.2.2).

IDA Pro[1] is a powerful disassembler and debugger for reverse engineering binaries. Besides, OllyDbg[2] is a debugger commonly used for reverse engineering that allows to execute the sample and to install plugins to dump the memory or even to reconstruct the Import Address Table (IAT). Universal PE Unpacker [Van05] is a plugin for IDA Pro that allows dynamic unpacking in a semi-automatic fashion. It allows to introduce a range of memory addresses where the encrypted code is supposed to be, and once

[1] http://www.hex-rays.com/idapro/

[2] http://www.ollydbg.de/

the execution reaches that point, it stops and dumps the memory range under analysis with the original code.

2.5.2 Specific unpacking

A considerable number of malware samples are protected with common packers. Specific unpacking tries to unprotect software packed with these packers applying an specific routine designed for each packer. This approach avoids processing every sample in a more complex (and sometimes less accurate) generic unpacking system. Although some packers like UPX or UPack distribute their own unpackers, in most occasions it is necessary to code such unpacking routines. This task requires a reverse engineering effort to understand how the packing algorithm works. This approach is not applicable to custom-packers or modified versions of known packers, since the number of different techniques is too high to design a new routine each time. Nevertheless, despite of its limitations, it is a very efficient method commonly used by anti-malware solutions.

One common technique is to implement specific unpackers as scripts for debuggers like OllyDbg, that allows scripting in OllyScript language, Immunity Debugger[1], that has a python plugin interface, or IDA Debugger[2]. Debuggers provide an infrastructure to monitor and control the execution of a program, and scripts allow to drive the execution towards unpacking. Nonetheless, these debuggers are sensitive to anti-debugger techniques, making necessary to explicitly bypass those tricks.

Other frameworks allow to create unpacking routines directly and provide both debugging (PyDbg[3]) or emulation capabilities (PyEmu[4]).

Besides, Fast Universal Unpacker (FUU)[5] is an open-source tool based on the TitanEngine SDK by Reversing Lab[6]. This framework simplifies different tasks related with unpacking such as memory dump or Import Address Table (IAT) reconstruction. FUU allows to detect most common packers by signature scanning, and allows to implement plugins to unpack them. In 2010, Reversing Lab published a similar tool named TitanMist [Lab10] which similarly detects common packers by signature scan-

[1] http://www.immunityinc.com/products-immdbg.shtml
[2] http://www.hex-rays.com/idapro/
[3] https://github.com/OpenRCE/pydbg
[4] https://code.google.com/p/pyemu/
[5] http://code.google.com/p/fuu/
[6] http://www.reversinglabs.com/products/TitanEngine.php

ning and applies specific unpacking routines programmed in different languages: Python, Lua, or Titanscript.

Finally, Pin is a dynamic binary instrumentation tool developed by Intel that allows dynamic program analysis in user space in both Linux and Windows platforms. This framework allows to dynamically inject code in the binary that will be called at certain events defined by the developer. It does not require to modify the source code of the application under analysis. It can be used, as in the previous examples, to develop custom unpacking routines for certain packers controlling the execution to recover the original code[1].

2.5.3 Generic unpacking

Manual unpacking and more specially specific unpacking require a great reverse engineering effort. A software analyst may require too much time to understand how the packer works, incurring into an excessive cost, due to the quantity and diversity of packers in the wild. For this reason, in recent years, several authors have proposed different approaches for generic unpacking.

2.5.3.1 Static approaches

There are not many static generic approaches due to their intrinsic complexity. In Section 2.3 we mentioned the *X-Ray scanning* technique [Szo05], that consists of exploiting vulnerabilities in cryptographic algorithms in order to obtain the original code without executing any unpacking routine. This approach, unfortunately, is very dependant on the algorithm used to protect the code. Some protection schemes rely on simple encryption techniques and can be reliably crypto-analysed [WBR13]. Besides, a packer may not use any cryptographic algorithm, but custom-designed transformation techniques.

Coogan et al. [CDKT09] proposed a solution based on the static analysis of the sample to extract the unpacking routine from the binary in order to use it to decrypt the protected code and data.To this end, they applied flow control analysis and alias analysis techniques to identify potential transition points from the unpacking routine to the original code. Once the tail-jump is located, they applied *backward slicing* to extract the routine and

[1] `http://jbremer.org/malware-unpacking-level-pintool/`

execute it as part of an unpacking tool. The main limitation of this approach is that static analysis is very sensitive to anti-reverse engineering attacks, anti-disassembly attacks and obfuscation techniques. In addition, there are limitations associated with current architectures such as indirect jump analysis, that make static analysis a very complex task. In addition, it relies on a rather simplistic model about the inner structure of the unpacking routine. More complex unpacking routines could mislead static analysis and evade this kind of systems.

2.5.3.2 **Dynamic approaches**

Due to the inherent complexity of static analysis, the majority of generic unpackers are based on a dynamic approach. Table 2.3, Table 2.4, and Table 2.5 show some of the most important solutions proposed in the literature for dynamic generic unpacking and compare their main features. For each generic unpacker, we specify the approach it employs (heuristic or statistic), the granularity of the analysis and its resilience to trigger-based behaviour techniques. In addition, it indicates the capacity of each unpacker to deal with multi-layer packing, incremental packing, shifting-decode-frames, and unpacking based on multiple processes.

Some of the solutions have been implemented as plugins of well-known open-source tools such as OllyDbg, IDA Pro or QEMU. OllyBonE [Ste06b] is a plugin for OllyDbg that analyses memory pages written and afterwards executed, allowing to install interruptions at memory page level. To this aim, a kernel driver implements a memory protection mechanism and generates an INT1 interruption that can be handled by OllyDbg. PAX PAGEXEC[1] is a tool that implements a similar approach to avoid the execution of injected code in buffer overflow attacks. Its main limitation is its sensitiveness to anti-debug techniques and its simplicity, that makes it unable to deal with more complex packers.

Saffron [QS07] is built over OllyBonE, using Intel PIN binary instrumentation to monitor the execution of the binaries. The memory is dumped each time the control is transferred to memory pages dynamically created or modified.

Justin is a system proposed by Guo et al. [GFC08] that analyses binaries under execution following two different approaches. First, it registers all the memory pages that are created and analyses them with an anti-malware engine when the execution control flow jumps to any of them. Neverthe-

[1] http://pax.grsecurity.net/docs/

Table 2.3: Comparison of the different generic unpacking methods proposed: Ollybone [Ste06b], Saffron [QS07] and Justin [GFC08].

Feature	**Ollybone**	**Saffron**	**Justin**
Heuristic	✓	✓	✓
Statistic	×	×	×
Granularity	Page	Instruction	Page
Addresses anti-debug	×	✓	Partially
Addresses anti-emu,anti-vm	×	×	Partially
Addresses trigger based behaviour	×	×	×
Addresses multi-layer packing	×	×	×
Addresses incremental packing	×	×	×
Addresses shifting-decode frames	×	×	×
Addresses multiple processes	×	×	×

less, this technique is not compatible with commercial solutions that only scan certain parts of the files. The second approach tries to heuristically determine the tail-jump (i.e., end of the unpacking routine). To this end, it monitors several events that are characteristic of unpacking routines: control transfers to created or modified memory pages at execution time, stack restoration and command-line parameter fetching routines. The limitations of Justin are its lack of portability and its dependence on certain heuristics. Additionally, it relies on several assumptions that are not com-

mon for all the packers: same memory space for packed binary and the resultant unprotected code, and complete unpacking before the execution of the original code. In addition, the approach cannot deal with more advanced techniques such as multi-layer unpacking, incremental unpacking or shifting-decode-frames.

Christodorescu et al. proposed in Malware Normalization [CKJ+05] several algorithms for the normalization of software. Among them there is a malware unpacking technique that takes two premises: the packed code does not present self-modification routines (i.e. once unpacked, it remains unmodified) and the execution flow is independent of the system state and the environment in which the code is executed. It monitors memory write operations to locate the Original Entry Point (OEP), but the authors affirm that it is not always correctly located. In a first phase, it executes the program to locate the OEP and in the second phase, it uses the information about memory write operations to obtain the original code. This approach, in addition, faces multiple layer packers by applying repeatedly the unpacking algorithm. It is not designed to deal with anti-debug, anti-emulation or anti-virtual machine techniques.

Pandora's Bochs is based on the Bochs emulator, an x86 architecture software emulator that allows code instrumentation at CPU level [Böh08]. The system collects information regarding memory structures, execution control flow jumps and API function calls. With this information, it implements several heuristics to determine the OEP and the moment at which the memory must be dumped. The heuristics are refined to enable unpacking of multi-layer packers. It measures the unpacking progress and stablishes time-outs for the periods in which no progress is made. More concretely, it assumes the following premises: it considers only one process (some packers inject the original code to a different process), unpacking is not performed at the kernel execution level, the code remains in the limits of the loaded image in memory, the OEP is executed after a jump instruction and the original code is not obfuscated. Its main limitations are its dependency on heuristics and its computational cost. In contrast, it addresses anti-virtual-machine techniques.

Polyunpack is a tool developed by Royal et al. [RHD+06] that compares the instruction trace generated during the execution of a binary file with an static model of the binary previously obtained. In this way, they propose the algorithm *EXTRACTUNPACKCODE* to obtain the code revealed during execution. In addition, it excludes the code corresponding to imported libraries to avoid considering them as dynamically generated code. Al-

though Polyunpack implements some measures to prevent *anti-debugging* techniques, it does not directly address trigger-based behaviour. In order to unpack binaries protected by multiple layers of packing, it must reconstruct a binary with the extracted code and re-execute it again.

Table 2.4: Comparison of the different generic unpacking methods proposed: Malware Normalization [CKJ+05], Pandora's Bochs [Böh08] and Polyunpack [RHD+06]

Feature	**Malware normalization**	**Pandora's Bochs**	**Polyunpack**
Heuristic	×	✓	×
Statistic	×	×	×
Granularity	Instruction	Instruction	Instruction
Addresses anti-debug	Not relevant	Not relevant	×
Addresses anti-emu,anti-vm	×	Partial	×
Addresses trigger based behaviour	×	×	×
Addresses multi-layer packing	✓	✓	✓
Addresses incremental packing	×	×	×
Addresses shifting-decode frames	×	×	×
Addresses multiple processes	×	✓	×

Renovo [KPY07] monitors the accesses to memory of each instruction and considers unpacked code all the memory positions that have been writ-

ten and then executed. It dumps the memory in this point and flags it as a possible OEP. Afterwards, it cleans the memory state and repeats the process to face multi-layer packers. A time-out of 4 minutes is established. If this time-out expires and no memory position has been written and the executed, then the unpacking concludes. The authors consider that this approach is heuristic-free. Nevertheless, the approach does not allow to identify the memory dumps containing the original code. A positive aspect about Renovo is its capacity to deal with unpacking routines based on multiple processes. In contrast, it does not directly address trigger-based behaviour techniques.

OmniUnpack [MCJ07] is a tool implemented over OllyBonE [Ste06b], providing a kernel driver that allows break-on-execute at memory page level. It invokes an anti-malware engine to scan modified memory pages when the unpacking routine is considered to have finished. To this end, it assumes that this phase ends when a dangerous system call is performed and the memory page which is under execution has been over-written. This approach is more efficient than its predecessors as it only invokes the anti-malware engine once. As the heuristic is implemented by means of a kernel-driver that is installed in a real environment, the system is resilient to *anti-debugging*, *anti-emulation*, and *anti virtual-machine* techniques. Moreover, authors establish 3 conditions that a signature-scanning based anti-malware engine should satisfy: (i) signatures must characterise the actual malicious code and not the unpacking routines of the binary; (ii) signatures must represent the code fragments in the instant previous to the dangerous system call; (iii) the code to identify should have not been modified in the time elapsed from the start of the execution of the malicious code and the system call. To this end, they modified the well-known tool Clam-AV to satisfy these conditions.

Eureka [SYS+08], in a similar way to previously described unpackers, allows a binary to execute while it monitors execution at system call level (*coarse-grained*) instead of instruction level (*fine-grained*), in order to reduce the overhead of the system and improve its efficiency. It considers different events as a possible end of the unpacking routine, and then dumps the code present in memory. On the one hand, the heuristic approach consists of dumping the memory contents when the NtTerminateProcess system function is called. It assumes that a binary that finishes execution will have all the original code present in memory. This approach is valid for simple packers and incremental packers, but unfortunately packers based on the shifting-decode-frames technique, or packers which erase the code

after executing it are resilient to this approach. Besides, it monitors the execution of the system call NtCreateProcess to capture created processes which may be injected with the original code and then executed. On the other hand, the statistical approach analyses bi-gram frequency, measuring the frequency of most common operational codes such as push and call, to determine if a region of code is unpacked or protected.

Table 2.5: Comparison of the different generic unpacking methods proposed: Renovo [KPY07], Omniunpack [MCJ07] and Eureka [SYS+08].

Feature	**Renovo**	**Omniunpack**	**Eureka**
Heuristic	×	✓	✓
Statistic	×	×	✓
Granularity	Instruction	Page	System call
Addresses anti-debug	Not relevant	✓	Not relevant
Addresses anti-emu,anti-vm	×	✓	×
Addresses trigger based behaviour	×	×	×
Addresses multi-layer packing	✓	✓	✓
Addresses incremental unpacking	×	×	Partially
Addresses partial code revelation	×	×	×
Addresses multiple processes	✓	×	✓

Kim et al. proposed in Quantification of Code Revelation [KIE+09] an approximation to the development of a generic unpacker, based on the

quantification of the hidden code revealed by a binary under execution. They propose the execution of the binary using Dynamic Binary Instrumentation (DBI) techniques and implement a shadow state for the memory to record the write operations and executions over each memory address, monitoring written-then-executed and executed-then-written positions. In this way, they quantify the code revealed and determine the OEP using a heuristic approach, considering that it is located in the first positions modified and then executed.

Other approaches [JCL+10, CX10] have focused on measuring the entropy variation during the unpacking process in an effort to locate the original entry point, and thus the correct moment to dump the memory contents of the binary. More specifically, Cesare and Xiang [CX10] leveraged application level emulation in order to run the binary, instead of applying whole system emulation. A refinement for OEP detection was proposed by Wu et al. [WCZ09], based on the intuition that, when the original code is executed, the stack pointer should be equal to the state when the binary is loaded in memory.

Some of the approaches presented (e.g. Renovo [KPY07]) are focused on obtaining one or several snapshots of the process memory. In some cases, when several snapshots are available because the packer presents several unpacking phases, it is important to distinguish which snapshot contains a dump of the original binary. Kawakoya, Iwamura and Itoh [KII10] propose a method to mine memory accesses in order to determine the dump which most likely contains the original code.

In a different fashion, Gnaedig et al. [GKRW10] proposed a model for eliminating self-modifying code (i.e., unpacker code) from execution traces in order to obtain a clean version of the unpacked code.

Some approaches adopt a hybrid technique for code unpacking. In this way, Caballero et al. [CJMS09] proposed a mixed dynamic and static approach consisting on hybrid disassembly and data-flow analysis to extract self-contained *transformation functions*, identifying code and data dependencies, and extracting the function interface (input and output parameters). In particular, Caballero et al. applied cryptographic function extraction to identify the unpacking routine of the ZBot malware, and employed it to statically unpack a different sample of the same family. Kolbitsch et al. proposed another approach designed to extract proprietary *gadgets* from the binary.

Finally, a special category of protection engines (virtualization based obfuscators) have drawn researchers' attention. Rolles [Rol09] studied the

VMProtect packer describing a method to recover the original code of the binary (translated by the packer to a different target architecture). Besides, Sharif et al. [SLGL09] proposed a method based on QEMU and different binary analysis techniques (abstract variable binding, clustering of memory reads and dynamic tainting) in order to extract the syntax and semantics of the byte-code language translated by the protection tool. Later, Coogan et al. [CLD11] proposed a set of methods for reverse engineering and deobfuscating virtualization based obfuscators analysing their execution traces. Ghosh et al. [GHD12] proposed the use of replacement attacks to modify the virtual-machine or emulator running the protected process in order to analyse or modify it. Kinder et al. [Kin12] proposed a series of optimizations for the static analysis of virtualization-based protectors over a toy-example. These protection engines are a completely different challenge and require other techniques in order to recover the original code.

The methods described demonstrate their effectiveness and efficiency in different ways. Some unpackers have been designed to deal with more complex techniques, while others, less sophisticated, can only unpack simple packers. Eureka is one of the most advanced unpackers, as it combines several techniques to identify the unprotected code. Nevertheless, due to the great diversity of packers, generic unpacking is still a challenging research field. On the one hand, understanding the packer structure can be beneficial for the malware analysis process. Current solutions have focused on providing sound mechanisms to dump potentially original code, but generally do not report useful information about the packer. On the other hand, although many unpackers include methods to deal with anti-debug, anti-emulation, or anti-virtual-machine techniques making the environment as transparent as possible, other techniques such as multi path exploration have not been explored in the domain of unpacking. Although several publications have addressed multi-path exploration for malware analysis, it is still unclear how they should be applied to the unpacking problem, which limitations they present, and which heuristics can be employed to improve their feasibility. Specifically, these techniques can help unpacking tools to ensure that all code paths are explored, dealing with trigger-based behaviour and thus addressing one of the main shortcomings of dynamic analysis systems.

2.6 Other relevant contributions

Finally, we describe other relevant contributions that do not directly address the unpacking problem but have a special relevance for the research presented in this dissertation. First, we focus on analysis transparency, and then describe previous work on trigger-based behaviour. Both of the research topics are related to the main limitation of dynamic analysis platforms (i.e., their incapacity to explore multiple paths).

2.6.1 Transparency

Some researchers have focused their efforts on ensuring the resilience and transparency of dynamic execution platforms. As we mentioned earlier, the main disadvantage of these platforms is their incapacity to explore several execution paths. Malware writers have traditionally taken advantage of this weakness to implement different techniques to evade detection.

Automatic dynamic analysis of malware generally requires to run whole isolated systems (i.e., whole system emulation). Application level emulation has been successfully applied for simple tasks, but unfortunately malware samples generally use operating system resources in unpredictable ways, create several processes, inject code to other processes or even interact directly with the operating system kernel.

Some years ago, whole system emulation was too resource consuming for large-scale malware analysis. Nevertheless, different techniques were proposed to improve the efficiency of such systems. Stepan [Ste05] proposed a technique named *dynamic translation* that boosts the performance of emulators significantly. This technique consists of disassembling and translating code blocks to the host architecture to execute them on the real machine and emulating system resources such as memory, I/O, and hardware. These blocks are cached and do not need to be translated for every executed instruction. Several optimisations can be applied like deferred flag computation and block chaining. This environment is completely isolated from the host system and allows to execute whole operating systems realistically. Current whole system emulators like QEMU are based on these concepts.

One of the advantages of these systems is their capacity to stealthily monitor the execution of the whole system at any granularity level. The dynamically translated blocks can be instrumented in order to monitor different events in the system like API calls, instruction-level or block-level

tracing without requiring the injection of drivers or hooking techniques in the guest system.

Nevertheless, these systems still present some inconveniences. Red-pills is the name used for tests performed by malware in order to determine if the execution environment in which it runs is a virtual/emulated environment or a real machine. These techniques are typically used to evade analysis. Whole system emulators typically present implementation errors during the translation of the guest architecture to the host architecture. Complex instruction sets like x86-64 make this translation a difficult process prone to errors. In fact, Paleari et al. [PMRB09] proposed a method to automatically discover and generate red-pills in system emulators. In particular, during this dissertation we found 2 implementation errors in the Dynamic Binary Translation engine of QEMU that affected all its versions and impeded the correct emulation of the Armadillo packer. In this context, several publications and projects [KPY07, DZX13, BCK+10] have reported the incapacity of emulation based systems to correctly execute the Armadillo packer.

As a result, different authors have proposed dynamic analysis platforms focusing on transparency. Cobra [VY06] is one of the first systems that introduced the concept of transparent malware analysis, based on the substitution of instructions that could be potentially used for detection by *stealth implants* [VY05].

Ether [DRSL08] is based on the application of hardware virtualization extensions (i.e. Intel VT) making possible to trace the execution of the system at instruction level and providing mechanisms to deal with the semantic gap and Virtual Machine Introspection. Since it resides outside the operating system, there are no detectable software components. Nevertheless, this system requires to single-step the execution of the system, making it too slow for real-time deployment [BCK+10].

Kang et al. [KYH+09] proposed another hybrid system to enable the emulation of emulation-resistant malware by first tracing the execution of a reference system (Ether) and then detecting divergences in the execution with respect to an emulated environment. Their approach allowed to dynamically modify the state at the divergence points to correctly emulate the execution of the binary.

Balzarotti et al. [BCK+10] proposed a similar approach for the detection of split personalities in malware (e.g., emulation detection). To this aim, instead of executing the sample in a fine-grained stealthy analysis environment such as Ether, they traced the execution of system-calls in a ref-

erence machine. Then, they used the recorded trace to replay the execution on the emulated machine (configured to be execution equivalent), and detected deviations in the execution. Their replay infrastructure makes the approach resilient to execution differences caused by other aspects distinct than split-personalities. Other authors, in contrast, have focused on the detection of sandbox evasion techniques [LKC11, KKK11, RKK07].

Later, Yan et al. [YJZY12] proposed V2E, a hybrid platform that leverages hardware virtualization and emulation in order to provide a transparent analysis platform with a high instrumentation capacity. To this aim, they recorded the execution under a virtualized and transparent environment using virtual memory management mechanisms for trapping the execution of the virtualized system, instead of trapping every single instruction (e.g., Ether). Then, they implemented a technique to replay this transparent execution over TEMU. The instructions considered to be safely emulated can be executed directly in TEMU, while the rest of instructions (that might be part of emulator detection routines) are replayed considering the first execution trace. The second system (TEMU), allows to trace the execution at different granularity levels and an easy implementation of analysis plugins.

The recording and replay of execution traces is useful for binary analysis and reverse-engineering [DGHH+14]. Finally, Spider [DZX13] is an alternative approach for binary program instrumentation that implements a stealthy breakpoint mechanism by leveraging memory management features in a virtualised environment.

2.6.2 Trigger-based behaviour and multi-path exploration

According to Balzarotti et al. [BCK+10] there are two main aspects that limit the coverage of dynamic analysis systems: (i) limited test-coverage and (ii) malware programs that detect and evade the analysis environment.

Transparent execution is focused on dealing with malware capable of detecting the analysis environment and modifying its execution to evade detection. Nevertheless, these techniques do not explore the different execution paths that a binary may have.

In order to deal with this limitation, Moser, Kruegel and Kirda [MKK07b] proposed a system able to explore different execution paths of a binary based on taint analysis and symbolic execution. This system is capable of tracing the execution of a binary tainting different inputs (e.g., internet connectivity, mutex, files, registry entries, current time) and recording

the propagation of these taints over the system. Then, they built symbolic expressions for the taint propagating instructions and queried a linear constraint solver to obtain the set of values to force at each conditional jump in order to consistently explore the different paths of the binary. They implemented an on-line multi-path exploration engine that allowed a depth first search exploration of the execution tree by saving and restoring process snapshots.

Almost in parallel, Song et al. [SBY+08] developed a platform to enable binary analysis with two main components: TEMU, a whole-system emulator with taint-analysis implemented over QEMU and Vine, a tool that allows to translate binaries or execution traces to a intermediate language (Vine IL) and allows to apply many control flow analysis and data flow analysis techniques. This platform was used for many different use-cases and problems such as crash analysis [MCJ+10], identification of trigger-based behaviour [BHL+08], reasoning about code paths in malware using mixed concrete and symbolic execution [BHK+07], or even triggering the unpacking routine of environment sensitive malware [JWL+12]. Nevertheless, this last approach does not actually differ from multi-path exploration of malware.

Similarly, S^2E [CKC11] is a platform that leverages the symbolic execution engine KLEE [CDE08] to allow multi-path exploration and introduces the concept of selective symbolic execution (application of symbolic execution to only certain memory regions) and execution consistency models. This system has been applied to different use cases such as automated testing of device drivers, reverse engineering of closed source drivers and multi-path exploration.

Schwartz et al. [SAB10] summarise several of the challenges involved for the implementation of a dynamic taint analysis system and a symbolic execution engine. According to Schwartz et al. there are practical challenges and limitations that affect these kind of systems. Taint policies and the sanitization of tainted values have a direct impact on over-tainting and under-tainting errors. Indirect memory accesses with symbolic addresses, jump tables, path selection heuristics, or the size of the constraint systems are aspects that have no clear solution, and for which many different approaches have been proposed [BE13, CPM+10]. These aspects, nevertheless, have a high impact on the efficiency and the feasibility of multi-path exploration. Also dynamic information flow analysis techniques (i.e., taint analysis) present limitations that can be used by malware authors to evade analysis. Cavallaro, Saxena and Sekar [CSS08] highlighted several of these

limitations and proposed several attacks for dynamic taint analysis systems and symbolic execution.

Finally, some authors have proposed alternative multi-path exploration approaches that do not depend on complex symbolic execution engines. X-Force [PDZ+14] is a system capable of forcing execution paths inconsistently and recovering from execution errors by dynamically allocating memory and updating related pointers. This approach cannot report the conditions that have to be met in order to explore the different paths, but can help in different tasks such as reconstructing the control flow graph of a binary, discovering hidden behaviour or type inference analysis.

In general, most of the approaches proposed to date suffer from the well-known path explosion problem [CKNZ12]. This limitation makes necessary to develop heuristics to improve the feasibility of multi-path exploration.

2.7 Summary

This chapter summarises current literature on malware analysis, and more specifically on the unpacking problem.

First, we have introduced several concepts and definitions relevant for this dissertation. Then, we have described how run-time packers protect malware, as well as the main anti-analysis techniques employed and the different approaches adopted by the security industry.

Second, we have defined a classification for the unpacking techniques covered by this literature review, and then enumerated and described the most relevant contributions on packer detection and identification, unpacking, and other relevant aspects such as analysis platform transparency and trigger based behaviour.

Part I

Static analysis for packed binary classification

«Anyone who feels the mountain does not need an explanation and as long as there are walls, spires, and crests, there will be someone to climb them, who will enjoy what they do, even if they don't understand why.»

Josep Manuel Anglada (1933–)

Chapter 3

Portable Executable structural feature based classification of packed binaries

Packed software was originally intended to reduce the size of the original binary. Nevertheless, these primitive tools soon evolved into complex protection systems that employ different techniques in order to obfuscate the original code, data, and file structures. As a consequence, the analyst is no longer capable of determining the intention of the binary without unpacking the sample. As we have described in Chapter 2, software packing prevents static analysis techniques such as disassembly. Since the code and data are compressed or encrypted and cannot be directly analysed, malware detection mechanisms or static reverse engineering cannot determine the real functionality of the binary. Alternatively, dynamic analysis can uncover the actual functionality of the binary in the best-case scenario, assuming that the analysis platform is transparent and resilient to any anti-analysis technique that the binary may employ. The main disadvantage of dynamic analysis is the computational resources required, usually involving the execution of the sample during several minutes.

Packed malware has a high impact on different aspects related to malware analysis and classification:

- Static analysis is limited by packing techniques.
- Packed software is heavily obfuscated and prevents several dynamic analysis techniques such as debugging, binary instrumentation, memory dumping, or even API call tracing.
- Since a high amount of malware is protected using these techniques, it complicates automated high scale analysis of malware datasets. The presence of packers must be considered in order to apply different treatments to the samples.

Considering these aspects, an efficient filtering of the samples is a fundamental task in order to ensure the effectiveness of malware analysis techniques.

Traditionally, desktop malware has been focused on Microsoft Windows platforms. Although the security of Microsoft Windows systems has improved significantly in the last years, the majority of desktop malware is focused on this platform.

Consequently, several authors [PLL08a, STF09, Sun12, JCN+13] have addressed the problem of determining if a binary is packed or not, generally relying on different heuristics or statistical methods and focusing on Microsoft Windows binaries.

In this chapter, we focus on the static classification of packed binaries and evaluate the performance of several supervised machine-learning classifiers when different feature-sets are extracted. More concretely, we define a feature-set based on the structural properties of the Portable Executable (PE) file headers under the intuition that packer tools sometimes introduce modifications in the structure of the binary that differ from standard compilers. Moreover, we conduct an statistical evaluation of the performance of different classifiers over the feature-set in comparison to previously proposed heuristics. Also, we propose a complementary set of heuristics that can be extracted from PE files and evaluate the improvement introduced on packed binary classification systems.

Accordingly, the research questions we establish for this study are the following:

Research question 3.1 *Do PE structural features significantly improve the performance of supervised machine-learning classifiers for packed binary detection?*

Research question 3.2 *What are the best performing machine-learning algorithms when different feature-sets are employed for classification?*

In order to address these questions, we compare the performance of several feature-sets against the heuristics proposed by Perdisci et al. [PLL08a]. Other authors proposed classification systems based on the same heuristics [STF09], entropy [LH07], randomness [Sun12], or statistical byte frequency [PLL08b]. Nonetheless, we focus on the approach proposed by Perdisci et al. because it was the first work to use heuristics for packed binary detection.

Safiq et al. [STF09] proposed the use of structural features for malware classification. Nevertheless, they relied on heuristics in order to determine if binaries are packed or not. More recently, Jacob et al. [JCN+13] applied a reduced set of structural features for pre-filtering samples from a database in order to reduce the number of samples to process in a more time-consuming comparison stage. Nevertheless, the objective of their classification is different from our packed binary detection approach. They select a set of 16 structural features that are not likely modified by packers in order to compute a fast similarity measure between packed binaries, even it they are protected by different packers. In our study we include these features and we do not make any assumption about their capacity to correctly discriminate binaries. Therefore, this is, to the best of our knowledge, the first study that evaluates the capacity of structural features to enhance the discrimination of packed and non-packed binaries.

The rest of the chapter is structured as follows. Section 3.1 describes the feature-sets employed by previous approaches for the classification of packed binaries. Section 3.2 describes our proposed feature-set. Then, Section 3.3 enumerates the supervised machine-learning algorithms tested in this study. Section 3.4 describes the set of samples used for the evaluation, describing the labelling process and possible biases and risks. Section 3.5 shows the experiment configuration, Section 3.6 describes the evaluation methodology and statistical tests applied. Section 3.7 presents the results obtained from the study, and finally Section 3.8 extracts the conclusions and discusses the results obtained.

3.1 Previous approaches for packed binary detection

3.1.1 Heuristic-based detection

Windows executable binary files, also known as image files, follow the Portable Executable (PE) format specified by Microsoft.

These files can contain code, data, and different resources that are structured according to an specific format. The PE format defines different structures and values that are used by the operating system loader to load the file in memory and proceed to its execution, providing all the necessary resources.

The first approach based on supervised machine-learning techniques for the classification of packed and non-packed binaries using heuristics was proposed by Perdisci et al. [PLL08a]. Additionally, they later applied this technique [PLL08b] combined with n-gram analysis in order to filter executables and selectively apply a generic unpacker to those samples previously classified as packed as a previous step to the application of malware detection techniques.

Safiq et al. [STF09], similarly, applied the same heuristics for packed binary detection. Afterwards, instead of unpacking the samples, they applied machine-learning based techniques considering different feature-sets for packed and non-packed binaries.

The set of heuristics proposed is described bellow.

- **Entropy.** Byte-level Shannon entropy of the file. For a random variable X with n outcomes, $\{x_i : i = 1, \ldots, n\}$ the Shannon entropy $H(X)$ is calculated as

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

  where $p(x_i)$ is the probability mass function of outcome x_i and b is the number of different symbols of the "ideal alphabet" used to measure source alphabets. The source alphabet used for measuring entropy is the set of 256 possible values that can be represented in a byte. According to information theory, 2 symbols are necessary and sufficient to encode data. Therefore, the entropy of the source alphabet, which

represents the maximum possible randomness, is the number of symbols in the "ideal alphabet" needed to encode a symbol in the source alphabet: 8.

Many tools, (e.g., PEiD), measure the entropy of the whole file and the sections of the executable in a coarse-grained style.

This value represents the randomness of the bytes in the file. Due to the nature of compression and encryption algorithms, packed files generally present higher entropy values. For this reason, entropy is the most common heuristic for packed file detection.

Apart from measuring the entropy of the whole file, previous approaches have also considered the randomness of certain parts of the binaries, such as the header, code sections or data sections. Unfortunately, some packers do not always present clear boundaries for header, code, or data sections. For instance, the FSG packer inserts part of the unpacking stub in the header. In fact, FSG mixes the protected code and data in a single section. Other packers may scramble the corresponding section properties defined in the PE header that indicate the presence of code or data.

Anyhow, the entropy features considered in previous work [PLL08b, STF09] are the following:

  - *File entropy.*
  - *Header entropy.*
  - *Average entropy of code sections.*
  - *Average entropy of data sections.*

- **Standard sections.** Microsoft defines a set of standard section names and permissions. This specification is recommended but not compulsory, and while most common compilers follow the format specified, traditional packers do not follow it and define sections with alternative names or permissions. In spite of this specification, the standard use of sections is not enforced by the Windows loader, which does not apply almost any effective restriction.

  - *Number of standard and non-standard sections.*

Table 3.1: Section names and permissions according to the PECOFF specification. For each section name, there are different standard permissions or flags: (U) Contains uninitialized data, (I) contains initialized data, (R) read permission, (W) write permission, (X) execute permission, (L) LNK info, (D) memory discardable, (G) Global pointer relative and (C) contains code.

Name	U	I	R	W	X	L	D	G	C
.bss	✓		✓	✓					
.cormeta						✓			
.data		✓	✓	✓					
.debug$F		✓	✓				✓		
.debug$P		✓	✓				✓		
.debug$S		✓	✓				✓		
.debug$T		✓	✓				✓		
.drective						✓			
.edata		✓	✓						
.idata		✓	✓	✓					
.idlsym						✓			
.pdata		✓	✓						
.rdata		✓	✓						
.reloc		✓	✓				✓		
.rsrc		✓	✓						
.sbss	✓		✓	✓				✓	
.sdata		✓	✓	✓				✓	
.srdata		✓	✓					✓	
.sxdata						✓			
.text			✓		✓				✓
.tls		✓	✓	✓					
.tls$		✓	✓	✓					
.vsdata		✓	✓	✓					
.xdata		✓	✓						

- **Section permissions.** Section permissions are specified by the *read*, *write*, and *execute* flags (*Characteristics* field) of each section header in a PE file. These flags are used by the loader to set the appropriate memory access permissions to the different memory regions committed when the image is loaded in memory. Some packers typically modify the permissions of the binary, making section permissions an

interesting feature to consider. Nevertheless, section permissions in the image file do not always represent the actual permissions during execution: they can eventually be modified by a malicious program at runtime. In fact, an image file not declaring any executable section can successfully execute. An example of this kind of behaviour is the FSG packer, which takes advantage of the differences between the PECOFF specification and the actual implementation of the loader. In this case, the FSG packer declares a header size larger than the actual header, and inserts the unpacking code in the remaining part of the header, after the section table. The entrypoint declared by the PE header points to an address which is not located into any section: it points to the code added after the header. Considering that the section header of an image file is loaded into memory with read and execute permissions, this code can execute and unpack the original binary. As a result, previous approaches have considered the following features:

  - *Number of executable sections.*
  - *Number of readable, writable and executable sections.*

- **External libraries and functions imported.** When an image file is loaded, the operating system loader is responsible for loading all the libraries required by the image and providing an appropriate mechanism to call the functions present in the library. Portable Executable files contain an Import Table describing all the imported libraries and functions. In addition, image files also contain an Import Address Table, which maps every function to an address. This table is updated by the loader when the image is loaded into memory, after loading the corresponding Dynamic Link Libraries (DLLs). This process is completed before the first instruction in the image file is executed. Image files that do not present any protection mechanism declare all the functions that might be called during execution. A simple static analysis of the Import Table can reveal some clues about the functionality of the file (even though the execution order of these functions is unknown). Moreover, the operating system also allows image files to import additional functions at run-time. Malware writers take advantage of this feature of the operating system and avoid declaring the functions imported. Instead, these functions are bound at runtime using API functions such as `LoadLibrary`, `GetProcAddress`, or

even reading the Export Table of the DLL themselves. This technique, combined with the encryption of DLL and function names, allows an image to call dangerous system functions not declared in the Import Table. In contrast, non-packed files rarely use of this kind of techniques.

Given this situation, previous approaches have considered the number of imports as a useful heuristic to detected packed binaries.

3.2 Portable executable structural features

In this section we thoroughly describe the Portable Executable file format and discuss the adequacy of these features for packed binary detection.

The PECOFF specification defines the structure of Portable Executable files, as shown in Figure 3.1.

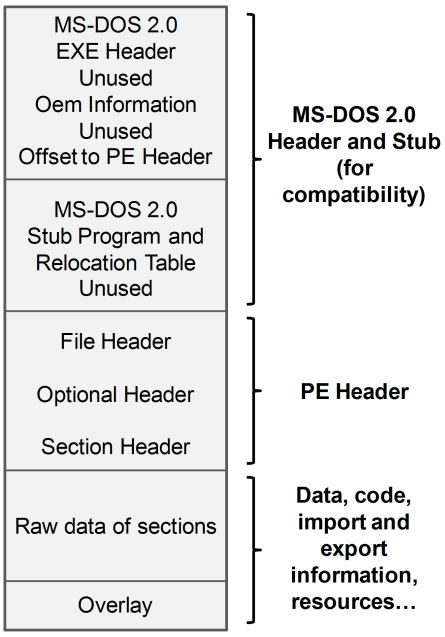


Figure 3.1: Structure of Portable Executable files.

The first structure present in an image file is the MS-DOS compatible section. This section starts with a magic number and contains several fields that allow the execution of the file into an MS-DOS platform. Normally, binaries contain a MS-DOS Stub after the DOS header that, in case the PE is run on a MS-DOS platform, a message is displayed indicating that the image cannot be run in MS-DOS. Nevertheless, this stub can be modified.

Next to the MS-DOS Header we can find the PE Header. The PE Header is divided into several structures: COFF File Header (present in both object and image files), Optional Header (present only in image files), and Section Header (present in both object and image files). The content of the file (i.e., data, code, various resources and information for the correct execution of the image) should be placed after these headers, structured into different sections defined in the section header. In addition, any PE file can contain overlay data not mapped into any section. Overlay data is not mapped into memory by the loader but can be read at runtime by the program.

3.2.1 DOS header

All Portable Executable files (PE) start with the DOS header, which is maintained for compatibility reasons. This header ensures that the binary can be executed in a DOS system. Also, it indicates the offset of the PE header (*e_lfanew*). Anyway, as some packers or compilers may use the rest of values for unknown purposes, we include them in our feature-set.

- **Magic** *(2 bytes)*. Identifier of all Portable Executable (PE) files. As the scope of this study is limited to Microsoft Windows Portable Executable files, this feature will always present the value 0x5A4D, which is the ASCII representation of the letters 'M' and 'Z', initials of Mark Zbikowski, architect of the MS-DOS operating system.
- **e_cblp** *(2 bytes)*. Number of bytes used in the last block of the program. If the entire block is used, the value for this feature should be 0.
- **e_cp** *(2 bytes)*. This feature represents the number of blocks in the file that are part of the file.
- **e_crlc** *(2 bytes)*. Number of relocations stored after the header.
- **e_cparhdr** *(2 bytes)*. This value represents the size of the header in paragraphs. The program's data begins after this header.
- **e_minalloc** *(2 bytes)*. Minimum number of extra paragraphs of memory that the program will need in order to execute.
- **e_maxalloc** *(2 bytes)*. Maximum number of extra paragraphs of memory. It is possible to limit the memory reserved for the program by setting this value.

- **e_ss** *(2 bytes)*. Initial relative address of the stack segment. The value is used to initialize the SS register.
- **e_sp** *(2 bytes)*. Initial relative address of the stack pointer of the program.
- **e_csum** *(2 bytes)*. Word-checksum, or 16-bit sum of all words in the file.
- **e_ip** *(2 bytes)*. Initial instruction pointer for the program.
- **e_cs** *(2 bytes)*. Initial relative address of the code segment.
- **e_lfarlc** *(2 bytes)*. Address of the relocation table into the file.
- **e_ovno** *(2 bytes)*. Overlay number. This value is normally set to 0.
- **Reserved space** *(8 bytes)*. In order to include this reserved space in the feature-set, we divided it into 4 fields of 2 bytes, considering that the rest of the header values are word-aligned.
- **e_oemid** *(2 bytes)*. OEM identifier.
- **e_oeminfo** *(2 bytes)*. OEM information.
- **Reserved space** *(20 bytes)*. In order to include this reserved space in the feature-set, we divided it into 10 fields of 2 bytes, considering that the rest of the header values are word-aligned.
- **e_lfanew** *(4 bytes)*. This field contains the offset of the PE Header.

3.2.2 COFF file header

The COFF File Header is located at the file offset indicated by the field e_lfanew and is preceded by a magic number of 4 bytes corresponding to the ASCII characters 'P' 'E' '\0' '\0'.

- **Machine** *(2 bytes)*. This value identifies the type of target machine.
- **Number of sections** *(2 bytes)*. This feature indicates the number of sections contained in the PE file.

- **Time stamp** *(4 bytes)*. The TimeDateStamp value represents the date and time when the file was created. This feature is discarded due to possible biases in the dataset.
- **Pointer to symbol table** *(4 bytes)*. Pointer to the COFF symbol table, or zero if the symbol table is not present. As COFF debugging information is deprecated, this value should be 0.
- **Number of symbols** *(4 bytes)*. Represents the number of entries in the symbol table. As in the previous case, the use of this value is deprecated and should be 0.
- **Size of Optional Header** *(2 bytes)*. This value represents the size of the optional header of the PE file, which is required for all executable files but not for object files.
- **Characteristics** *(2 bytes)*. This field of the DOS Header represents a set of flags.
  - *Relocations stripped (0x0001)*. For Windows CE, Windows NT and later versions, this flag indicates that the file does not contain base relocations and it must be loaded at its preferred address. Otherwise, the loader will report an error.
  - *Executable image (0x0002)*. This flag indicates whether the file is executable. If not set, it indicates a linker error.
  - *Line numbers stripped (0x0004)*. This flag is deprecated and should be 0, and indicates whether the line numbers were stripped.
  - *Symbols stripped (0x0008)*. This flag is deprecated and should be 0, and indicates whether the symbols of the image were stripped.
  - *Working set trim (0x0010)*. This flag is deprecated from Windows 2000 onwards, and should be 0. It forces to trim the working set.
  - *Large address aware (0x0020)*. Indicates whether the file can handle an address space larger than 2 GB.
  - *Reserved for future use (0x0040)*. Flag reserved for future use.
  - *Little endian (0x0080)*. This flag is deprecated and should be 0. Indicates that the image will be loaded in little endian order in memory.
  - *32 bit machine (0x0100)*. This flag indicates whether the machine is based on a 32-bit-word architecture.

- *Debugging information stripped (0x0200)*. This flag indicates that the debugging information has been stripped from the file.
- *Removable run from swap (0x0400)*. This flags states that, if the image is stored on a removable media, then it must be entirely copied to the swap file to load it.
- *Network run from swap (0x0800)*. This flags states that, if the image is stored on a network media, then it must be entirely copied to the swap file to load it.
- *System (0x1000)*. Indicates whether the image is a system file, not a user program.
- *DLL (0x2000)*. This flag indicates whether the image is a Dynamic Load Library.
- *Uniprocessor system only (0x4000)*. This flag states that the file should be run only on a uniprocessor machine.
- *Big endian (0x8000)*. This flag is deprecated and should be 0. Indicates that the image will be loaded in big endian order in memory.

3.2.3 Optional header

If the PE file is an image file, it will contain an optional header not required in object files. The size of this header is delimited by the corresponding field in the COFF File header. In addition, the *Number of RVA and sizes* field indicates the number of data directory entries contained. Another relevant field in this header is the magic number that determines if the image is a PE32 or PE32+ image. The number of fields and their size depend on this magic number.

3.2.3.1 Standard fields

The fields in this part of the optional header define values for all the implementations of COFF, including UNIX.

- **Magic number** *(2 bytes)*. This field adopts the value 0x010b if the optional header corresponds to a PE32 file, and 0x020b if the optional header corresponds to a PE32+ file.
- **Major linker version** *(1 byte)*. Major version number of the linker.

- **Minor linker version** *(1 byte)*. Minor version number of the linker.
- **Size of code***(4 bytes)* This field stores the size of the code (.text) section of the file, or the sum of all the code sections.
- **Size of initialized data** *(4 bytes)*. Size of the initialized data section, or the size of all the data sections containing initialized data.
- **Size of uninitialized data** *(4 bytes)*. Size of the uninitialized data section (.bss) or the size of all the data sections containing uninitialized data.
- **Address of entry point** *(4 bytes)*. Address of entry point of the file, relative to the image base field. For drivers, it contains the address of the initialization function. For files in which there is not an entry point defined, this value must be 0.
- **Base of code** *(4 bytes)*. This field contains the relative address corresponding to the beginning of the code section.
- **Base of data** *(4 bytes)*. This field contains the relative address corresponding to the beginning of the code section. PE32+ files lack this field. The additional fields in this part of the header contain Windows specific features needed by the linker and the loader. Some fields in this section have a variable size, requiring 4 bytes for PE32 files and 8 bytes for PE32+.
- **Image base** *(4/8 bytes)*. Preferred load address for the file.
- **Section alignment** *(4 bytes)*. Alignment of sections in memory, in bytes. This value must be greater or equal than the file alignment. By default, it is the page size of the architecture.
- **File alignment** *(4 bytes)*. Alignment for the raw data in the file. It should be a power of 2 between 512 and 64 K, and by default is 512.
- **Major operating system version** *(2 bytes)*. Major version of the operating system.
- **Minor operating system version** *(2 bytes)*. Minor version of the operating system.
- **Major image version** *(2 bytes)*. Major version number of the image.

- **Minor image version** *(2 bytes)*. Minor version number of the image.
- **Major subsystem version** *(2 bytes)*. Major version number of the subsystem.
- **Minor subsystem version** *(2 bytes)*. Minor version number of the subsystem.
- **Windows 32 version value** *(4 bytes)*. This field is reserved and should be 0.
- **Size of image** *(4 bytes)*. This field represents the size in bytes of the image, including headers, and must be multiple of section alignment.
- **Size of headers** *(4 bytes)*. This field must be rounded up to a multiple of the file alignment, and represents the sum of the sizes of the MS-DOS stub, the PE header, and the section headers
- **Checksum** *(4 bytes)*. This field contains the Image file checksum, which is checked for validation at load time in critical files such as drivers, DLLs loaded at boot time and DLLs for critical Windows processes.
- **Subsystem** *(2 bytes)*. These fields indicates the subsystem required to run the image.
- **Dll characteristics** *(2 bytes)*. If the file is a DLL, these fields define several flags to be considered by the loader.
  - *Reserved (0x0001)*. Reserved, must be 0.
  - *Reserved (0x0002)*. Reserved, must be 0.
  - *Reserved (0x0004)*. Reserved, must be 0.
  - *Reserved (0x0008)*. Reserved, must be 0.
  - *Not specified (0x0010)*. The function of this flag is not specified by the official documentation from Microsoft.
  - *Not specified (0x0020)*. The function of this flag is not specified by the official documentation from Microsoft.
  - *Dynamic base (0x0040)*. Indicates that the DLL can be relocated at load time.
  - *Force integrity (0x0080)*. Forces code integrity checks.

  - *NX compatible (0x0100)*. Indicates NX compatibility.
  - *No isolation (0x0200)*. Indicates that the image is isolation aware but must not be isolated.
  - *No SEH (0x0400)*. Indicates that the image does not use Structured Exception Handlers.
  - *No bind (0x0800)*. Do not bind the image.
  - *Reserved (0x1000)*. Reserved, must be 0.
  - *Wdm driver (0x2000)*. The image is a WDM driver.
  - *Not specified (0x4000)*. The function of this flag is not specified by the official documentation from Microsoft.
  - *Terminal server aware (0x8000)*. Indicates that the image is terminal server aware.

- **Size of stack reserve** *(4/8 bytes)*. This field indicates the size of the stack reserve, or the maximum stack size that can be made available to the process.
- **Size of stack commit** *(4/8 bytes)*. This field indicates the size of the stack to commit.
- **Size of heap reserve** *(4/8 bytes)*. This field represents the size of the heap reserve, or the maximum heap size that can be made available to the process.
- **Size of heap commit** *(4/8 bytes)*. This field indicates the size of the heap to commit.
- **Loader flags** *(4 bytes)*. This field is reserved and must be 0.
- **Number of RVA and sizes** *(4 bytes)*. Number of data directory entries in the optional header. This number should not exceed 16, which is the maximum number of data directory entries.

3.2.4 Data directories

The data directory entries are located after the optional header. These entries are pointers to tables in the payload of the file (usually a section of the PE), that are used by the operating system to load the PE and execute it. This section consists of a list of directory entries of 8 bytes (4 bytes for the

relative address and 4 bytes for the size). The number of directory entries is indicated by the Number of RVA and sizes field in the optional header, and should not exceed 16. In most cases, this field contains the value 16, but in some cases certain packers like SLV present unusual values, presumably to confuse analysis tools when parsing the header. Despite this trick, the loader will still correctly load the PE. In addition, the standard specification of Microsoft for the PE format indicates that certain directory tables should be located in specific PE sections, with concrete names and permissions. Although this specification is recommended by Microsoft and usually followed by common compilers, the loader does not enforce these standards and the data directories may be stored at any part of the PE file, regardless of the section division declared for the PE.

1. **Export Table**. The Export Table defines the symbols exported that other images can access through dynamic linking. In general, DLL files export symbols.

2. **Import Table**. The Import Table describes the DLLs imported by the image and the symbols imported from each DLL. The first structure it contains is the Directory Table which defines, for each DLL imported, the address of its corresponding Import Lookup Table and Import Address Table. Before loading the PE file, these tables are identical. Each table contains an array of 32/64 bit values, containing one record for each symbol imported. The Import Lookup Table, together with an extra table named Hint Name Table contain the necessary information to locate each imported symbol into the corresponding DLL. After loading the file, the Import Address Table is overwritten with the address of each symbol in the DLL bound to the image. In this way, the code of the image can directly refer to these symbols by indirect addressing jump instructions through the records in this table.

3. **Resource Table.** The Resource Table indexes the different resources that a PE file may contain, such as icons, images or information about its version. These resources are indexed by a multi-level binary-sorted tree structure, that, by convention, has three levels.

4. **Exception Table.** The Exception Table contains function table entries used for exception handling, and the structure of the table depends on the target architecture of the image.

5. **Certificate Table.** The Certificate Table contains information about image integrity hashes and security certificates that can be used to detect modifications of the file, such as those performed by viruses.

6. **Base Relocation Table.** This table contains information for base relocations. These relocations must be applied when an image cannot be loaded into its preferred load address. In these cases, the addresses hard-coded in the file must be relocated according to this offset.

7. **Debug.** The debug tables contain information about debug symbols generated by the compiler, if present.

8. **Architecture.** Reserved, must be 0.

9. **Global Pointer.** This entry contains the relative virtual address of the value to be stored in the global pointer register (GP). The size of this entry must be set to 0.

10. **TLS Table.** The TLS directory table provides support for Thread Local Storage, a feature in Windows operating systems that allows the threads of a process to store different values for the same global variable, instead of sharing all the memory.

11. **Load Config Table.** The load configuration structure contains additional data to load the image. For example, Microsoft Windows XP and later versions of Windows use this table to store reserved Structured Exception Handlers, a mechanism to provide safe SEH handling.

12. **Bound Import Table.** The Bound Import Table contains import information, like the Import Address Table, but instead of containing the relative virtual address of the symbol imported, it contains the actual absolute address. This mechanism optimises the process of binding the imported DLLs to the image in the cases in which these libraries are always loaded in the same address space. The loader must verify the DLLs load address and, in case of not matching the expected values, update all the references.

13. **Import Address Table.** This table, once the image is loaded, contains the addresses of the symbols imported by the image. The loader uses the Import Table to obtain the actual address of each imported symbol in the corresponding DLL, and stores the address in the IAT.

14. **Delay Import Descriptor.** This table contains information to support lazy loading of DLLs, which consists of not loading the DLLs imported by the image until they are needed (i.e., called for the first time).

15. **CLR Runtime Header.** This table contains Common Language Runtime (CLR) metadata, which means that the image contains managed code (code that runs over the Microsoft .NET platform).

16. **Reserved.** Must be 0.

3.2.5 Sections

Every PE file can have a variable number of sections. PE sections are used to organize code and data, resources, import and export information, and any other data necessary for the execution of the application. Microsoft defines several standard section names and section permissions, each aimed at containing specific data or code. Nevertheless, following this recommendation is not mandatory. We extract several features from section headers. On the one hand, we consider the section of entry point (i.e., the section containing the address that will be executed once the file is loaded into memory) and extract all the possible header values related to it. On the other hand, we define several features that summarise the number of sections that satisfy certain conditions in order to extract information from the rest of sections in the PE file.

- **Name** *8 bytes*. ASCII encoded null terminated section name.
- **Virtual Size** *4 bytes*. The virtual size of a section represents the space in memory reserved for the section, that can be greater than the actual raw data contained in the file. In this case, the section is zero-padded.
- **Virtual Address** *4 bytes*. The address of the first byte of the section, relative to the image base when loaded into memory.
- **Size of raw data** *4 bytes*. Size of the section in the file. It must be multiple of the file alignment specified in the optional header. If a section only contains uninitialized data, this value should be 0.
- **Pointer to raw data** *4 bytes*. Offset of the data in the file. For sections containing only uninitialized data, it should be 0. It must be a multiple of the file alignment specified in the optional header.

- **Pointer to relocations** *4 bytes*. File pointer to the relocation entries of the section. This value should be 0 for executable images.
- **Pointer to line numbers** *4 bytes*. File pointer to the line-number entries of the section. This value should be 0 as COFF line numbers are deprecated.
- **Number of relocations** *2 bytes*. Number of relocation entries for the section.
- **Number of line numbers** *2 bytes*. Number of line-number entries for the section. This value should be 0 as COFF line numbers are deprecated.
- **Characteristics** *4 bytes*. This field defines different flags.
  - *Reserved (0x00000001)*.
  - *Reserved (0x00000002)*.
  - *Reserved (0x00000004)*.
  - *No padding (0x00000008)*. This flag is obsolete and indicates whether this section should not be padded.
  - *Reserved (0x000000010)*.
  - *Contains executable code (0x00000020)*. Indicates whether the section contains executable code.
  - *Contains initialised data (0x00000040)*. Indicates whether the section contains initialised data.
  - *Contains uninitialised data (0x00000080)*. Indicates whether the section contains uninitialised data.
  - *Linker Other (0x00000100)*. Reserved.
  - *Contains comments (0x00000200)*. Indicates whether the section contains comments or other information.
  - *Reserved (0x00000400)*.
  - *Must remove from image file (0x00000800)*. This flag will be set for object files, indicating that the section will not become part of the image file.
  - *Contains COMDAT data (0x00001000)*. Flagged if the section contains COMDAT data.

- *Not specified (0x00002000)*.
- *Not specified (0x00004000)*.
- *Referenced by global pointer (0x00008000)*. The section contains data referenced by the Global Pointer.
- *Not specified (0x00010000)*. Reserved.
- *16 BIT (0x00020000)*. Reserved.
- *LOCKED (0x00040000)*. Reserved.
- *PRELOAD (0x00080000)*. Reserved.
- *Alignment (0x00100000 to 0x00E00000)*. Alignment of the data (1 byte to 8192 bytes). It is valid only for object files, and it is represented by a 4 bit field.
- *Contains extended relocations (0x01000000)*. Indicates whether the section contains extended relocations.
- *Discardable (0x02000000)*. The section can be discarded.
- *Not cached (0x04000000)*. The section cannot be cached.
- *Not paged (0x08000000)*. The section cannot be paged.
- *Shared (0x10000000)*. Indicates whether the section can be shared in memory.
- *Execute permission (0x20000000)*. Indicates whether the content of the section can be executed.
- *Read permission (0x40000000)*. Indicates whether the content of the section can be read.
- *Write permission (0x80000000)*. Indicates whether the content of the section can be written.

3.2.6 Complementary heuristics

In addition to the heuristics adopted in previous work [PLL08a, PLL08b], we have considered several complementary values based on common heuristics. These features are divided in 3 groups: general heuristics, heuristics related to the section of entry point, and heuristics related to entropy.

3.2.6.1 General heuristics

- *Maximum raw data per virtual size ratio.* Maximum ratio among all the sections.
- *Minimum raw data per virtual size ratio.* Minimum ratio among all the sections.
- *Ratio of sections with virtual size greater than raw data.*
- *Number of imported DLLs.*
- *Ratio of readable and executable sections.*
- *Ratio of readable and writeable sections.*

In addition, we modified 3 heuristics proposed by Perdisci et al. [PLL08a, PLL08b] in order to normalise the values considering the number of sections present in the executable file. Regarding the heuristic corresponding to the ratio of sections with execution permissions, we consider it also an structural feature. In order to avoid the repetition of redundant features, we include this feature in the structural feature group.

- *Ratio of standard sections.*
- *Ratio of non-standard sections.* The normalization process makes this feature redundant to the previous one.
- *Ratio of readable, writeable and executable sections.*

3.2.6.2 Heuristics related to the section of entry point

- *Entry point outside any section.* Indicates that the entry point of the PE file does not point to any section in particular. In these cases, all the features related to the section of entry point are considered missing values.
- *Section of entry point is standard.* Indicates whether the section of entry point is an standard section.
- *Section of entry point has an Import Address Table.* Indicates whether the section of entry point is an Import Address Table section.
- *Raw data per virtual size ratio of the section of entry point.* Ratio between raw data and virtual size for the section of entry point.

- *Entropy of the section of entry point.*

3.2.6.3 Heuristics related to file entropy

- *Minimum section entropy.*
- *Maximum section entropy.*
- *Average section entropy.*
- *Ratio of sections with entropy in a certain range*. More concretely, the entropy ranges considered were 0 to 0.5, 0.5 to 1, 1 to 1.5, 1.5 to 2, 2 to 2.5, 2.5 to 3, 3 to 3.5, 3.5 to 4, 4 to 4.5, 4.5 to 5, 5 to 5.5, 5.5 to 6, 6 to 6.5, 6.5 to 7, 7 to 7.5 and 7.5 to 8.

3.2.7 Considerations about the violation of the PECOFF format specification

During the implementation of our parser, we followed the Microsoft specification [MC10] as close as possible and tested it against our set of packed and non-packed binaries. As a result, we found several aspects that confused our parser. In order to ensure the correct parsing of all the files, we adapted our code to deal with these problems. In this way, the main problems encountered are listed below.

- **Uncommon values for *Size of raw data*.** This field indicates the raw size of a section (i.e., the size of the section in disk). More specifically, some Zeus samples presented a value greater that the actual size of the file itself. In these cases, we considered other values such as the virtual size in order to determine the actual size of the section.
- **Entry point outside any section.** The FSG packer generates binaries with an entry point at a memory address outside any declared section. More concretely, the entry point for these files is located after the PE header. In these cases, we did not consider any of the sections as the entry point section. Alternatively, we represented all the features related to the section of entry point as missing values.
- **Sparse Import Address Table.** According to the specification of Microsoft, the import information of a binary should be located at a section named *.idata*, and is represented as a 4-byte null terminated Directory Table, followed by several DLL Import Lookup Tables, also

null-terminated. Nevertheless, these conditions are not necessary to correctly load a binary, and this information can be stored at any point of the binary. Besides, Import Lookup Tables are pointed by each entry in the Directory Table. These pointers, instead of referring to the file offset of the table, point to its Relative Virtual Address (RVA). For this reason, it is necessary to infer the virtual address of every section of the binary in order to calculate the corresponding file offset of each Import Lookup Table.

- **Number of data directory entries specified.** PE files present a field that indicates the number of data directory entries present in the header (*Number of RVA and sizes* field). While the majority of files analysed declare 16 data directories (the maximum possible number of data directories), some binaries declare any random number of directories (higher than 16). This could eventually drive to unpredictable behaviour.

- **Reserved fields and flags.** Several fields in the PE file are not specified or are reserved, according to the Microsoft specification. Although these fields should not be used (and thus should be filled with 0's), we found that some binaries actually define different values for them. For this reason, we considered all the possible fields in the parsing process.

- **File header**. The boundaries of the header are not clear, considering that the first section in the image file can start at any point in the file. In fact, the FSG packer inserts the unpacking routine in the space between the end of the header structures and the start of the first section. Because of these situations, we configured our parser to assume that the file header comprises the address space from the first byte of the file to the byte preceding the first section of the binary. This size is employed, for example, to calculate the entropy of the header.

3.3 Supervised machine-learning algorithms

Machine learning is a field of Artificial Intelligence that deals with the design and development of algorithms that allow computers to reason and make decisions based on data [Bis06].

Supervised machine-learning algorithms employ a previously labelled dataset which is composed of data instances. For each data instance, a

set of features is measured which might be continuous, categorical or binary [Kot07].

In this study, we try to measure the effects of applying different feature-sets for the classification of packed and non-packed binaries using supervised machine-learning approaches. To this end, we test different algorithms commonly used for classification problems. The implementation of the algorithms tested is provided by the WEKA [G+95] machine-learning tool.

3.3.1 Decision Trees

Decision trees are machine-learning classifiers that can be represented as a tree structure. In this way, a decision tree can be formalised as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a finite and not empty set of nodes and $\mathcal{E}$ is a set of edges (v, w). A path in the graph $\mathcal{G}$ is defined as sequence of edges $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$. Any pair of nodes in the tree is connected by exactly one simple path (i.e., a path with no repeated nodes). If (v, w) is an edge in the tree, then v is considered the parent node of w. The node with no parents is considered the root node. Besides, the internal nodes of the tree represent conditions for the features used for training, while the leafs of the tree represent the final classification of the classifier [Qui86].

Several learning algorithms have been proposed in order to generate the structure of the tree from a labelled dataset. In this study, we have tested J48, which is the WEKA implementation of the C4.5 algorithm [Qui93] and Random Forest, which is an ensemble (i.e., combination of several classifiers) of decision trees generated randomly.

- **C4.5 algorithm.** Quinlan [Qui93] proposed the C4.5 algorithm as an extension of the algorithm ID3 [Qui86]. This algorithm generates decision trees by selecting nodes based on the information entropy of each feature. This term is usually referred to as Shannon entropy [SW49] which measures the expected information value in a message (i.e., Information Gain (IG)). For each of the nodes of the tree, the algorithm selects the feature that provides the best division between the two classes. This feature is given by the highest IG value. The process is repeated recursively until no more divisions of the data can be performed.

- **Random forest.** The Random Forest algorithm [Bre01] is a classifier that is composed by several sub-trees. For the classification of

an instance, it generates an input vector for each of the trees in the ensemble. Each tree provides a classification and the global classifier provides the result with the highest number of votes. In order to generate the random decision trees that conform the random forest, a random subset of data is selected by an stochastic discrimination method [Kle96]. A subset of features is selected for each node and the best possible division of the data is calculated. The Random Forest classifier produces a precise classifier for many datasets, and can deal with incomplete data. Nevertheless, these classifiers are prone to overfitting to the dataset, specially with noisy data [SLTW04].

3.3.2 Rules

Rule induction is a technique that tries to extract rules from observed data. While other supervised methods do not provide a result that can be interpreted by a human, rule sets can be understood [FW98].

- **RIPPER.** Cohen [Coh95] proposed the RIPPER algorithm (Repeated Incremental Pruning to Produce Error Reduction), as a modification of the previous IREP algorithm. The algorithm is divided into different stages. First, it performs the *growing* phase, in which rules are made restrictive. Afterwards, it continues with the *pruning* of the rules in order to avoid overfitting.

- **PART.** The PART algorithm, proposed by Frank et al. [FW98], infers rules by generating partial decision trees, avoiding global optimisation. Afterwards, it extracts a rule from each partial decision tree. In fact, it employs a separate-and-conquer strategy in which it builds a rule, discards the instances affected by the rule, and continues with the process for the remaining instances.

3.3.3 Support Vector Machines

Support Vector Machine (SVM) classifiers define an hyper-plane in an n-dimensional representation of data (where n is usually the number of features in the dataset). The hyper-plane is adjusted to maximise the margin (m) between the two regions of classes in the feature space. This margin is defined as the highest distance between the instances in both classes, and is calculated from the nearest elements to the hyper-plane.

Formally, SVM requires a dataset $\mathscr{D}$,

$$\mathscr{D} = \{(x_i, c_i) x_i \in \mathbb{R}^p, c_i \in \{-1, 1\}\}_{i=1}^{n} \tag{3.1}$$

where each instance is represented as a vector x_i in a p-dimensional space, and c_i represents the class of the instance. The objective of the algorithm is to find the hyper-plane which maximises the margin between the points $c_i = 1$ and $c_i = -1$.

When the dataset can be linearly divided into two categories, the method selects the hyper-planes in such a way that there are no points between them, maximising the distance between them.

In some occasions, the space cannot be divided by a linear hyper-plane and kernel functions are employed in order to substitute the linear kernel function $k(x_i, x_j) = x_i \cdot x_j$.

- **Polynomial kernel function** substitutes the scalar product with

$$k(x_i, x_j) = (x_i \cdot x_j)^d \tag{3.2}$$

  where d represents the degree of the polynomial [AW99].

- **Normalised Polynomial kernel function** substitutes the scalar product with

$$K(x_i, x_j) = \frac{(x_i \cdot x_j)^d}{\sqrt{(x_i \cdot x_j)^d}} \tag{3.3}$$

  where d represents the degree of the polynomial [AW99].

- **Radial Bases Function kernel** (RBF) substitutes the scalar product with

$$k(x_i, x_j) = \exp(-\gamma \cdot \|x_i - x_j\|^2) \text{ para } \gamma > 0 \tag{3.4}$$

  where the value of the function only depends of the distance to the origin γ [AW99].

- **Pearson VII kernel function** substitutes the scalar product by the following equation:

$$k(x_i, x_j) = \frac{1}{\left[1 + \left(\frac{\sqrt{\|x_i - x_j\|^2} \cdot \sqrt{2^{\frac{1}{\omega}} - 1}}{\delta}\right)^2\right]^{\omega}} \tag{3.5}$$

3.3.4 K Nearest Neighbours (KNN)

The K Nearest Neighbours algorithm is a classification method based on the neighbourhood of the testing instance and the data used for training [FHJ52]. It represents the training instances ($\mathscr{S} = \{s_1, s_2, ..., s_{\ell-1}, s_\ell\}$ where ℓ is the number of features) in a training space. In order to classify a given instance, the method measures the distance from the sample to the nearest training instances in the feature-space. The most simple technique is to classify the instance as the most common class among the k nearest instances in the training set. In this study we have applied the Euclidean distance in order to measure the similarity between instances.

3.3.5 Bayesian Networks

The Bayesian networks are based on the Bayes' theorem [Bay63], that determines, based on observations, the validity of a given hypothesis. Besides, it can adjust probabilities with new observations.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{3.6}$$

According to its formulation (see Equation 3.6), given two events A and B, the conditional probability $P(A|B)$ that A occurs if B has occurred can be obtained if we know the probability $P(A)$, the probability $P(B)$, and the conditional probability B given A, $P(B|A)$.

If we extend this approach, Bayesian networks are probabilistic models for multivariate analysis that can be formally defined as acyclic graphs with an associated probability function [CGH96]. The nodes in the graph represent variables while the edges represent conditional dependencies between the variables.

In this way, a Bayesian network $\mathscr{B}$ can be formally defined as a tuple $\mathscr{B} = (\mathscr{D}, \mathscr{P})$ where $\mathscr{D}$ is an acyclic directed graph and $\mathscr{P}$ is a set of n conditional probability functions, one for each variable.

Several algorithms can be used to learn the structure of a Bayesian Network.

- **Hill Climbing** is a local search method that tries to maximise a heuristic function $f(x)$ [RN03]. It starts by a random function and iteratively applies changes to it. The algorithm stops training when it cannot enhance the results. In many occasions, the algorithm obtains local maximum and does not reach the best solution.

- **K2** searches the most probable network structure in a dataset. The algorithm adapts a hill climbing algorithm applying constraints to the order of the variables. The difference between this algorithm and Hill Climbing is that K2 does not consider the edges of the first naïve network, while Hill Climbing erases them afterwards.

- **Naïve Bayes classifier** assumes that there is not any kind of dependence between the nodes in the bayesian network [CGH96]. Each node in the network is linked to the conclusion.

- **Tree Augmented Naïve (TAN)** defines a network in which every node has at most two parent nodes, and one of them is the conclusion. This algorithm considers the conditional mutual information among the features. TAN generally presents better precision results while maintaining a considerably simple structure [FGG97].

3.3.6 Artificial Neural Networks

Artificial Neural Networks were originally an attempt to model the processing of biological neurons mathematically [Bis06].

The notion of perceptron can be described as a node with n inputs $x_1, x_2, \cdots, x_n$, n weight values $w_1, w_2, \cdots, w_n$, and one output. The perceptron computes $\sum_i x_i w_i$, and if it surpasses a certain threshold, the output is 1, otherwise it is 0. Single perceptrons are able to correctly classify instances when data is linearly separable. Otherwise, the instances will not be classified properly.

- **Multilayered Perceptron.** Multilayered perceptrons were proposed in order to solve the problem of single perceptrons regarding the linear separability of the instances. In this way, multi-layer neural networks are modelled as networks with a high number of nodes or units (neurons), in which we can distinguish 3 classes: input units, hidden units and output units [Kot07]. Among the algorithms proposed to train the network, the most common approach (used in this study) is Back Propagation. This algorithm presents the training instances to the neural network, calculates the error in the output (difference between the desired output and the actual value), and then updates the weights backwards (from output to input perceptrons) according to this error. This process is repeated until it reaches a good weight configuration [Kot07]. Several stopping rules can be applied to decide

when the training ends (e.g., after a number of epochs, when an error rate is achieved, when the error rate does not improve for a number of epochs).

3.3.7 Bootstrap Aggregating (Bagging)

Bootstrap Aggregating, also known as bagging, is a meta-classification algorithm, also known as ensemble [Bre96]. This approach tries to improve classification accuracy by generating bootstrap replicates of the training set and using them to generate multiple classifiers that are afterwards aggregated by averaging their results. Besides, the approach was also proposed in order to reduce the instability of learning algorithms.

3.4 Dataset description

In order to answer the research questions formulated, we configured a set of 4,000 binaries to evaluate the performance of several supervised machine-learning classifiers for packed binary detection over different feature-sets.

The possible biases and limitations of the dataset have been thoroughly studied and discussed. Nevertheless, the intrinsic nature of packers, the efforts of malware creators to evade detection, and the limitations of already existing tools make difficult to discriminate packed and non-packed files. All the methods and tools for the detection of packed binaries are based on heuristics or statistical tests. In Chapter 2 we have discussed the decidability of the detection of an unpack-execute process.

In practice, researchers and analysts resort to well-known existing tools and heuristics to achieve an approximate classification of the binaries. In this section, we study the possible risks to the validity of the experiment and define a methodology to reduce the noise in the dataset to the extent possible. First, we define the requirements for the dataset. Then, we consider the limitations presented by current tools and techniques. Finally, we define a criteria for the selection and labelling of samples.

3.4.1 Requirements of the dataset

- **Goodware and malware presence.** The dataset should contain both goodware and malware samples. Otherwise, the differences among the headers of goodware and malware would introduce biases in the dataset, specially in the case of non-packed samples. In addition, it is

important to include samples generated by different compilers, presenting different sizes and origin (e.g. system files, common tools...).

- **Presence of different types of packers.** Off-the-shelf packers use different techniques in order to protect software. The first packers were simple compressors to reduce the size of the binary. Later, encryption based packers introduced layers of protection to the code. Some packers include different obfuscation and anti-analysis techniques that make the task of unpacking software more difficult. Other packers, such as Armadillo, use advance techniques like partial code revelation, exception based control-flow or invalid operational codes to confuse disassemblers. Finally, virtualization based packers employ a completely different technique. Current malware implements its own unpacking routines to avoid the use of off-the-shelf packers that could eventually be unpacked by existing scripts. These packers are referred to as custom packers. Some of these packers employ a benign file as a carrier, making the detection more difficult. In order to ensure that all kind of packers are represented in the dataset, different kind of commercial packers and custom packers must be included.

- **Dataset balancing.** In order to ensure the correct performance of supervised machine-learning classifiers, the dataset should be balanced to include the same number of samples for each class: Packed and Non-packed.

- **Variance of the dataset.** Another important aspect to consider is the variance of the dataset. In this sense, it is relevant to conform a dataset with unique samples, avoiding repetition even in their packed and non-packed form.

- **Number of samples.** Supervised machine-learning algorithms need sufficiently large datasets in order to ensure the validity of the results obtained. For this reason, the size of the dataset will also determine the confidence of the results obtained.

- **Correct labelling of the samples.** Ideally, all the samples included in the dataset should be correctly labelled. This is usually the most difficult task when creating a dataset, and sometimes it is necessary to assume the existence of noise in it. In order to minimise possible errors in the labelling process, it is important to consider the actual limitations of the tools employed for the analysis.

3.4.2 Limitations of existing tools and techniques

- **Signature Scanning.** Signature scanning tools analyse binary files and identify byte patterns previously identified and stored in a signature database. More concretely, PEiD identifies off-the-shelf packer signatures by searching for common fingerprints in the headers and the unpacking stub of the packer (i.e., the parts of the binary that remain unchanged among all the protected samples). Nevertheless, malware samples employ different obfuscation techniques and modify the bytes that are identified by signature scanning tools in order to avoid detection. Some samples even insert byte patterns corresponding to other off-the-shelf packers in order to confuse the analysis tool. This limitation implies that there is a risk of incorrectly labelling as non-packed a binary file that is actually packed, introducing noise in the dataset.

- **Detection based on heuristics.** Packer detection tools sometimes rely on heuristics for binary classification, raising alarms if certain attributes of the file are suspicious. While traditional packers can be detected with these methods, custom packers or modified versions of existing protectors sometimes apply different techniques to evade heuristic detection. In fact, the majority of these heuristics are well known by the community, allowing malware writers to focus on avoiding this kind of filters.

- **Generic Unpackers.** Some generic unpackers try to detect an unpack-execute process in order to determine the correct moment to dump the process memory which contains already unpacked code. The majority of generic unpackers employ techniques that can be evaded. Some packers identify the execution of previously written memory addresses at different granularity levels [KPY07, Ste06b]. These unpackers can be confused by applying several layers of packing, unpacking based on several stages, or even self-modifying code in order to add complexity. Other unpackers compare memory snapshots at different points of the execution [RHD+06], or employ different heuristic or statistical methods to identify unpacked code [SYS+08]. In addition, traditional packers include anti-analysis techniques that detect the presence of debuggers, virtual machines or emulators, modifying the behaviour of the unpacking routine if one of them is detected.

- **Manual analysis.** Finally, in order to know with certainty if a sample is packed or not, it is necessary to manually analyse the sample with both static and dynamic analysis techniques. Static analysis allows to overview the behaviour of the unpacking stub. Nevertheless, static disassembly can be limited by anti-disassembly techniques. Dynamic analysis is also useful to reverse engineer software, but, again, can be limited by anti-analysis techniques. Any of the approaches require a considerable effort of a skilled analyst in order to correctly classify the sample.

3.4.3 Criteria for the selection of samples

First, we obtained binary files from different sources in order to conform the dataset.

- **Malware samples.** We first obtained a set of 13,173 malware samples from VxHeavens. 6,216 of these samples were reported by PEiD as packed with well-known packers, while the other 6,957 samples were not detected by PEiD. In addition, we obtained 1548 packed samples from the Zeus malware family provided by the Spanish security company S21Sec. These samples were divided into 8 different groups according to their versions, and were gathered between 2009 and 2011. The majority of the malware samples collected were 32-bit executable files. DLLs or 64-bit executables were barely present in the dataset.

- **Goodware samples.** Secondly, we obtained unique benign 32-bit executable files from a Microsoft Windows XP installation with several tools installed such as text editors, internet browsers, or office tools. In addition, in order to select these samples, we analysed them with PEiD to finally select a set of 1,645 samples not reported by PEiD as packed. In order to ensure that all the samples were unique, their MD5 hash was computed and compared.

Second, we selected a set of packers to manually pack a subset of the samples. More concretely, we selected 10 common packers representing different kinds of packing techniques. In this way, we included compressors, protectors, and a virtualization-based packer: *Armadillo*, *Asprotect*, *FSG*, *MEW*, *Packman*, *RLPack*, *UPX*, *Themida*, *TELock*, *SLV*. While some of these tools are distributed as commercial products to protect legitimate

software (e.g., Armadillo, Themida), all of them are commonly used by malware writers to protect their samples.

Finally, we selected the samples for each of the groups that conform the dataset.

- **Manually packed malware samples.** We randomly selected a set of 500 malware samples that were reported by PEiD as non-packed and manually packed them with 10 different off-the-shelf packers. Despite PEiD might not correctly classify all the samples raising false negatives because of the presence of custom packers or modified versions of existing ones, this process would apply a second layer of protection and thus we can safely label all the samples of this category as packed.

- **Custom packed malware samples.** In order to include in the dataset custom packed malware samples, we considered the Zeus family samples provided by S21Sec. This family has been reported to use custom packers for the protection of the binaries [Wyk11] as a first layer of protection. These binaries are sometimes protected by a second layer of protection with an off-the-shelf packer. Given this background, we first selected 1,000 Zeus samples for which no known packer was detected by PEiD.

- **Manually packed goodware samples.** Additionally, we manually protected 500 non-packed goodware samples with the same traditional packers used to protect malware samples in order to also include this kind of files in the packed set of files.

- **Not packed malware samples.** First, we randomly selected 1,000 unique malware samples that were reported by PEiD as non-packed, not included in the manually packed set of samples. In order to reduce the risk of including packed samples in the non-packed set, we performed an additional step in order to discard possibly packed files. There are several alternatives to decide if a sample is actually packed or not. On the one hand, tools like PEiD or StudPE, apart from providing a signature scanning functionality, allow to examine the structure of the file as well as certain heuristics that can aid the task of labelling a sample as packed. On the other hand, dynamic and static analysis can provide a deeper insight of how the sample actually behaves. While most of the sandboxes are focused on malware behaviour, independently of the unpacking functionality of the packer,

there are some tools available for dynamic generic unpacking. Regarding pure manual analysis, both static disassembly and debugging require an effort not assumable for a high number of samples. Another alternative that we could be tempted to use is to run the files in a generic unpacker. These approaches present several inconveniences. On the one hand, many packers implement techniques to detect debuggers, virtual machines or emulators, avoiding the unpacking of the original code. These techniques can be normally evaded but require reverse-engineering and in practice imply the manual unpacking of the sample. The execution of the sample in an generic unpacker does not ensure the detection of the packer nor the final unpacking of the sample. On the other hand, most of the unpackers provide as output a dumped binary but do not provide information about the sample. Moreover, the result of this unpacking step cannot not be used for our experiment, considering that these approaches normally dump the content in memory and reconstruct the header of the file. This approach would introduce a bias in the dataset, as we are trying to measure the ability of PE headers to discriminate packed and non-packed samples. For these reasons, we finally decided to automatically select suspicious files by employing well known heuristics, and manually analysed them making use of the heuristic analysis capabilities of PEiD, and the overview of the file structure provided by StudPE, in order to label them as packed or not. The heuristic tests employed in order to prefilter suspicious files are detailed below. All the samples that raised positives for at least 2 of the 4 possible tests were manually analysed.

- *File entropy* > 7. Lyda and Hamrock [LH07] proposed the use of entropy to discriminate packed and non-packed files, and defined entropy confidence intervals for packed files. In this way, they defined the interval 7.199-7.2267 for packed executables and 7.295-7.312 for encrypted executables with a 99.99% of confidence. Considering this background, we apply a more conservative threshold and consider suspicious all files with an entropy over 7.00.
- *Number of import address table entries* < 5. Many packers destroy the *Import Address Table* and only import certain functions such as `LoadLibraryA` and `GetProcAddress` to obtain the address for each imported function before executing the original code.

- *Number of imported DLLs <= 2*. Similarly, we set a threshold for the number of imported DLLs, following the same assumptions as in the previous case.
- *Ratio of standard sections <= 0.5*. Finally, many packers employ not standard section names or permissions. In this way, we set a threshold to select samples that either use alternative names for the sections, or that use not standard permissions for typical sections (*.text*, .data).

In this way, 183 of the 1,000 samples were considered suspicious and 74 were labelled as packed after manual analysis. The majority of these samples were modified versions of well-known packers such as UPX, NSpack or Mew, among others. The 74 packed files were discarded, and another set of 150 unique samples was randomly selected. The same process was followed, identifying 38 suspicious samples, from which 26 were actually packed. Finally, 74 samples were randomly selected from this last set in order to substitute the initially discarded samples. Finally, the samples were analysed with the tool *Protection ID*, an alternative to PEiD to ensure that no sample was labelled as packed. After this process, we must assume that there is still a risk of including packed malware files in the set of non-packed files, resulting into noise in the final dataset. Nevertheless, as our intention is to measure the performance differences of classifiers over the same dataset using different features, we consider that this noise does not affect the conclusions of the study.

- **Not packed goodware samples.** Finally, the remaining 1,145 goodware samples were analysed to form the packed goodware dataset. Similarly to non-packed malware samples, we proceeded to select the samples suspicious of being packed. In this case, 104 samples were suspicious of being packed. We manually checked the samples with PEiD and StudPE, but finally did not discard any of the samples. Moreover, some of the samples were installer applications which could eventually be considered as packed. Nevertheless, we maintained them in the dataset considering that these samples do not belong to the kind of runtime packers that we are seeking to discriminate, and thus provide a more realistic sample distribution.

3.5 Experiment configuration

3.5.1 Summary of feature-sets tested

In order to study the influence of different feature-sets, we have divided the described features in different groups, shown in Table 3.2.

Table 3.2: Features considered in the study.

Category	Group	# features
Baseline Heuristics	Entropy	4
	Section properties	4
	Others	1
Structural Features	DOS Header	30
	File Header	20
	Optional Header	76
	Section of Entry Point	37
	Section properties	36
Complementary Heuristics	Entropy	19
	Section of Entry Point	5
	Others	9
Total		**241**

First, we consider the 9 heuristic features proposed by Perdisci et al. More concretely, this category of features is composed of 4 entropy features, 5 section properties and an additional feature referring to the number of entries in the Import Address Table. Regarding structural features, we have first considered the DOS header fields of the PE. Despite these fields are rarely used (with the exception of the pointer to the file header), we have included them due to the possibility of hiding some values. The only field excluded from this header is the Magic value that holds the same bytes for all PE files and therefore will not provide any information for classification. Second, we have considered the file header and the optional header, that are both present in image files, and contain different fields that could be interesting for packer detection. In a similar way, we have excluded the PE magic header, the DLL flag and the optional header magic number. These fields do not contain any information: the first value is constant for all PE files, the second one discriminates executable files from DLLs, and the last one discriminates 32 bit and 64 bit executables. Addi-

tionally, we excluded the time stamp field to avoid introducing biases in the dataset. Manually packed files could present similar time stamp values due to the automated process used to produce the samples, that would result into similar time stamp values for all the files packed by a concrete packer. Regarding the PE sections that the binary may contain, we have considered two groups of features. First, we have included all the fields corresponding to the section of entry point under the assumption that these fields may contain information for the classification. Besides, a PE file can contain any number of sections. For this reason, we cannot include all the header fields for all the sections. Alternatively, we have included in our feature-set a group of 36 values that *summarise* all the sections in the binary. In this way, we have included 28 features for the 28 different flags that a section can have. Each feature represents the ratio of sections that have the corresponding flags. Additionally, we have included the total and average values among all the sections for the raw data size, the virtual data size, number of relocations, and number of line numbers. The rest of the section header fields are pointers or values that cannot be easily summarised for all sections and were thus discarded.

To address the research questions defined for this study, we consider different feature-sets for the experiment.

- **Baseline heuristics.** First, we test the performance of the baseline heuristics in order to compare it with the rest of feature-sets defined.

- **Structural features.** We compare the performance between the use of structural features and the use of heuristics for packed binary classification.

- **Baseline heuristics and structural features.** Then, we evaluate the performance of structural features and heuristics with respect to previous configurations

- **All heuristics.** We test the performance of our complementary set of heuristics.

- **All features.** Finally, we measure the performance for all the features proposed.

3.5.2 Summary of algorithms tested

In order to evaluate the effects of using the features described in Section 3.2 for supervised machine-learning based techniques, we have tested several machine learning algorithms over different sets of features. In this way, the algorithms tested are listed in Table 3.3. More specifically, we have evaluated the performance of Bayesian Networks using different learning algorithms (Naïve Bayes, K2, Hill Climber and TAN), Support Vector Machines with different kernels (polynomial, normalised polynomial, PUK and RBF), K Nearest neighbours with different k configurations (1, 3 and 5), the C4.5 algorithm, Random Forest with different number of random trees (10, 30 and 50), and rules based classifiers (JRip, and PART). Additionally, we have included another two classifiers in order to cover all the configurations reported by Perdisci et al. [PLL08a]: Multilayered Perceptron (MLP) algorithm with one hidden layer of 243 nodes, and Bagging with the 100% of the training set with the C4.5 algorithm.

Table 3.3: Algorithms tested in the study.

Algorithm	**Configuration**
Bayesian Network	Naïve Bayes K2 learning algorithm Hill Climber learning algorithm TAN learning algorithm
Support Vector Machines	Polynomial Kernel PUK Kernel Normalised Polynomial Kernel RBF Kernel
K Nearest Neighbours	k = 1 k = 3 k = 5
Decision Trees	C4.5 Random Forest with 10 Random Trees Random Forest with 30 Random Trees Random Forest with 50 Random Trees
Rules	JRip PART
Neural Networks	MLP with 1 hidden layer with 243 nodes
Bagging	Bagging C4.5

3.6 Evaluation methodology

In this section we describe the methodology followed to evaluate the classification algorithms listed in Table 3.3 with the feature-sets described in Table 3.2, over the dataset configured for this experiment.

When the number of instances available for training and testing is limited, K-fold cross validation is an approach commonly used to evaluate the performance of machine-learning classifiers. This approach consists of generating k random sub-sets of instances and running k experiments using 9 sub-sets for training and the remaining subset for testing, rotating the subset used for testing in each experiment round [Koh95].

Besides, from the point of view of the replicability of the experiment, k-fold cross validation can be repeated using different configurations. The most used experimental set-ups are 5X2CV and 10X10CV, in which 5 runs of k=2 cross validation and 10 runs of k=10 cross validation are evaluated.

In this experiment, we have applied the 10X10CV evaluation method.

3.6.1 Classification performance evaluation

Regarding the evaluation of the performance of the different classifiers, different measures can be considered depending on the specific classification problem. In our case, we have measured classification accuracy, true positive rate, false positive rate and area under the ROC curve, considering that we only have 2 classes of data, packed and non-packed, and the straightforward interpretation of these measures for our classification task.

In this way, classification accuracy is defined as the number of correctly classified instances among all the instances tested:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.7}$$

Besides, True Positive Rate (TPR) measures the number of packed files correctly classified as packed with respect to the actual number of packed files. According to this definition, TPR indicates the capacity of the classifier to identify packed binaries. A low TPR would imply missing packed binaries and incorrectly classifying them as non-packed.

$$TPR = \frac{TP}{TP + FN} \tag{3.8}$$

False Positive Rate (FPR) measures the number of non-packed files incorrectly classified as packed with respect to files not packed. Hence, FPR

informs about the number of non-packed binaries that will be flagged as packed (that will presumably be analysed by computationally expensive tools such as a generic unpacker).

$$FPR = \frac{FP}{FP + TN} \tag{3.9}$$

Finally, the Receiver Operating Characteristic (ROC) curve plots TPR and FPR for the different discrimination thresholds for a binary classifier. The area under this curve measures the capacity of a classifier to provide an accurate classification regardless of the threshold selected in the training phase.

3.6.2 **Statistical tests**

In this section, we describe the statistical tests employed to compare the different classifiers and feature-sets. In machine-learning, and specially when resampling methods like 10X10CV are employed, the direct comparison of the average performance for each of the possible configurations (algorithm, feature-set, dataset) does not provide statistical significance to the results. In fact, the observed differences might be caused by random factors such as the distribution of the samples in the different training and test sets. For this reason, we apply different statistical tests for the comparison of the different approaches.

On the one hand, we compare the results obtained for each classifier applied to each different feature-set. This type of comparison allows us to determine if, for a certain classifier, the observed difference between two different feature sets is statistically significant. On the other hand, we compare the overall performance of the tested classifiers when different feature-sets are applied.

First, we need to compare the same classification algorithm over two different feature-sets and a single dataset. We can consider that this test compares two different classification methods over a single dataset. Furthermore, the dataset is randomised in each run and divided into 10 folds following the same distribution for the execution of the two classification methods. Consequently, it requires a dependent-measures test, as we will compare the performance of each classifier for each of the 100 possible dataset arrangements tested.

Demšar [Dem06] enumerates different approaches that have been proposed for the comparison of two classifiers over a dataset. Different au-

thors agree that the traditional paired t-test should not be applied over k-fold cross validation due to its tendency to produce Type-I errors. Dietterich [Die98] recommends the use of 5x2CV t-test instead of paired t-test over k-fold cross validation, in order to overcome the problem of the underestimated variance due to the dependencies of the data used in each cross validation round. Besides, Nadeau and Bengio [NB03] propose the corrected resampled t-test to adjust the variance. Later, Bouckaert and Frank [BF04] evaluated the replicability of experiments and tested several approaches: 5x2CV t-test, 100 runs of random subsampling, and 10X10CV using with the corrected resampled paired t-test proposed by Nadeau and Bengio. They concluded that 10X10CV with the corrected t-test achieved the best results.

Following the recommendation of Bouckaert and Frank, we apply the corrected resampled t-test over 10X10CV at an $\alpha = 0.05$ significance level.

Additionally, in order to provide statistical support to the research question established for this study, we must test if, in general, one feature-set presents a better performance than another feature-set. Demšar [Dem06] describes several statistical tests for the comparison of multiple classifiers over several datasets. More concretely, Demšar does not recommend the use of ANOVA, as it relies on certain assumptions (e.g., normality of data) that are probably violated for this kind of analysis. In the study, the non-parametric Friedman test is recommended. Despite the objective of our test is to compare different features sets over different classifiers and a single dataset, we can assume that data does not follow a normal distribution like in the case presented by Demšar. Consequently, following the recommendation of Demšar, we apply an statistic based on the Friedman statistic (see Equation 3.10) proposed by Iman and Davenport [ID80], which is less conservative than the Friedman test (see Equation 3.11).

$$\chi_F^2 = \frac{12N}{k(k+1)}\Big[\sum_j R_j^2 - \frac{k+(k+1)^2}{4}\Big] \tag{3.10}$$

$$F_F = \frac{(N-1)\chi_F^2}{N(k-1)-\chi_F^2} \tag{3.11}$$

The statistic proposed is applied over the average results for the 10 runs and 10 folds for each configuration. Although there are variations of these kind of tests that consider multiple observations per cell, they require the independence of the observations, a property that we cannot assume when we apply k-fold cross validation.

Finally, we perform a post-hoc analysis based on the Holm's step down procedure in order to contrast the rank of each feature-set against its baseline, in order to test if it presents an statistically significant improvement in each of the performance measures obtained.

3.7 Results

In this section we describe the results obtained in the experiment. First, we show the results obtained for each of the feature-sets extracted from the dataset, and compare them against the baseline heuristics proposed by Perdisci et al. [PLL08a]. Afterwards, we describe the statistical tests performed in order to answer the research question established for this study.

Table 3.4: Baseline heuristics.

Classifier	Accuracy	TPR	FPR	AUC
Naive Bayes	80.45±1.55	0.9583±0.0129	0.3493±0.0310	0.9612±0.0077
Bayes K2	94.50±1.17	0.9556±0.0146	0.0658±0.0176	0.9834±0.0048
Bayes Hill Climber	94.50±1.17	0.9556±0.0146	0.0658±0.0176	0.9834±0.0048
Bayes TAN	95.72±0.99	0.9594±0.0159	0.0450±0.0148	0.9890±0.0044
SMO Polykernel	93.25±1.11	0.9180±0.0181	0.0531±0.0132	0.9325±0.0111
SMO Puk	96.60±0.89	0.9588±0.0141	0.0269±0.0106	0.9660±0.0089
SMO Normalised Polyk.	86.79±1.64	0.8885±0.0233	0.1528±0.0247	0.8679±0.0164
SMO RBF	89.20±1.49	0.8976±0.0233	0.1136±0.0224	0.8920±0.0149
KNN k=1	97.92±0.75	0.9768±0.0120	0.0184±0.0102	0.9792±0.0075
KNN k=3	97.62±0.84	0.9740±0.0129	0.0216±0.0098	0.9886±0.0054
KNN k=5	97.24±0.81	0.9716±0.0129	0.0269±0.0106	0.9915±0.0045
C4.5	96.60±0.92	0.9638±0.0137	0.0319±0.0131	0.9707±0.0114
RandomForest 10	97.58±0.76	0.9778±0.0103	0.0263±0.0114	0.9940±0.0039
RandomForest 30	97.84±0.71	0.9783±0.0106	0.0216±0.0105	0.9961±0.0029
RandomForest 50	97.87±0.73	0.9782±0.0105	0.0209±0.0103	0.9967±0.0022
JRip	96.52±0.94	0.9642±0.0142	0.0337±0.0125	0.9742±0.0089
PART	96.88±0.88	0.9664±0.0146	0.0288±0.0131	0.9788±0.0082
Multilayered Perceptron	95.84±0.90	0.9531±0.0181	0.0362±0.0135	0.9790±0.0073
Bagging C4.5	97.08±0.79	0.9709±0.0114	0.0294±0.0110	0.9926±0.0042

Table 3.4 shows the results obtained for the baseline heuristics. We can observe that the majority of classifiers achieve a considerably high accuracy (over 95%) and AUC (over 0.95), with the exception of Naive Bayes and SMO with Polynomial Kernel, Normalised Polynomial Kernel and RBF Kernel. Similarly, almost all the classifiers obtain sound true positive rates, while some classifiers tend to produce high false positive rates (e.g., Naive Bayes, SMO with Normalised Polynomial Kernel and RBF Kernel). Regarding the AUC, SMO with Normalised Polynomial Kernel and RBF Kernel show inferior results. The best performing algorithms for this feature-set are Random Forest and K Nearest Neighbours, with an accuracy over

97.50% and an AUC over 0.98, true positive rates over 0.97 and false positive rates below 0.03. Although Bayesian Networks with K2, Hill Climber and TAN learning algorithms do not achieve similar results in terms of accuracy, they present a sound performance in terms of AUC (over 0.98).

Table 3.5: Structural features compared to baseline heuristics.

Classifier	Accuracy	TPR	FPR	AUC
Naive Bayes	67.94±1.73 ▽	0.9862±0.0078 ▲	0.6274±0.0341 ▽	0.7151±0.0182 ▽
Bayes K2	94.94±1.09	0.9662±0.0126	0.0674±0.0174	0.9850±0.0046
Bayes Hill Climber	94.92±1.08	0.9658±0.0127	0.0674±0.0174	0.9850±0.0046
Bayes TAN	97.26±0.86 ▲	0.9781±0.0109 ▲	0.0329±0.0135	0.9953±0.0024 ▲
SMO Polykernel	95.35±1.06 ▲	0.9476±0.0167 ▲	0.0405±0.0138	0.9535±0.0106 ▲
SMO Puk	97.06±0.77	0.9683±0.0110	0.0271±0.0113	0.9706±0.0077
SMO Norm. Pol.	94.24±1.18 ▲	0.9417±0.0170 ▲	0.0569±0.0160 ▲	0.9424±0.0118 ▲
SMO RBF	92.30±1.30 ▲	0.9675±0.0119 ▲	0.1217±0.0219	0.9230±0.0130 ▲
KNN k=1	98.24±0.56	0.9799±0.0089	0.0150±0.0073	0.9823±0.0057
KNN k=3	97.90±0.67	0.9792±0.0099	0.0211±0.0087	0.9867±0.0049
KNN k=5	97.30±0.81	0.9723±0.0118	0.0263±0.0109	0.9887±0.0043
C4.5	97.90±0.76 ▲	0.9781±0.0113 ▲	0.0201±0.0109 ▲	0.9823±0.0077 ▲
RandomForest 10	98.61±0.56 ▲	0.9877±0.0073 ▲	0.0155±0.0082 ▲	0.9977±0.0023 ▲
RandomForest 30	98.73±0.52 ▲	0.9874±0.0074 ▲	0.0128±0.0069 ▲	0.9984±0.0018 ▲
RandomForest 50	98.75±0.47 ▲	0.9872±0.0076 ▲	0.0122±0.0066 ▲	0.9986±0.0016 ▲
JRip	97.40±0.81 ▲	0.9726±0.0123	0.0247±0.0110	0.9791±0.0076
PART	98.15±0.68 ▲	0.9808±0.0096 ▲	0.0179±0.0103 ▲	0.9845±0.0082
MLP	93.73±2.46 ▽	0.9388±0.0409	0.0642±0.0516	0.9680±0.0124 ▽
Bagging C4.5	98.29±0.70 ▲	0.9825±0.0096 ▲	0.0168±0.0104 ▲	0.9969±0.0026 ▲

Statistically significant improvement (▲) or degradation (▽)

In Table 3.5 we can observe the results obtained when only structural features were used for classification (excluding all possible heuristics). In this case, we include the results obtained for the corrected paired t-test that determines if there is an statistically significant ($\alpha = 0.05$) improvement (▲) or degradation (▽) with respect to the baseline heuristics described in Table 3.4. This test was performed for each of the classifiers and performance measures tested. We can observe that for many classifiers there is a significant improvement in the results obtained regarding accuracy, true positive rate, and AUC (e.g., decision tree based classifiers, Bayesian network with TAN learning algorithm, SMO with Polykernel, Normalised Polykernel and RBF). Besides, Naive Bayes and Multilayered Perceptron show a significant degradation on accuracy, false positive rate, and AUC. Regarding false positive rate, there is a lower number of algorithms that show a significant improvement. Nevertheless, only Naive Bayes shows a significant degradation.

K Nearest Neighbours and Random Forest are again the algorithms that provide the best results. In this case, both algorithms achieve an accuracy

over 98%, an AUC over 0.985 (over 0.99 for Random Forest), a TPR near 0.98 and false positive rates below 0.025. In this case, some configurations also show sound results: Bayesian Network with TAN learning algorithm, PART, C4.5 and Bagging C4.5 with results very close to Random Forest and K Nearest Neighbours. Again, Bayesian Networks provide sound AUC results (over 0.985).

Table 3.6: Baseline heuristics with structural features compared to baseline heuristics.

Classifier	Accuracy	TPR	FPR	AUC
Naive Bayes	71.27±1.65 ▽	0.9843±0.0085 ▲	0.5590±0.0333 ▽	0.7487±0.0187 ▽
Bayes K2	96.19±0.81 ▲	0.9778±0.0101 ▲	0.0540±0.0142	0.9897±0.0042 ▲
Bayes Hill Climber	96.19±0.81 ▲	0.9778±0.0100 ▲	0.0541±0.0142	0.9897±0.0042 ▲
Bayes TAN	98.36±0.67 ▲	0.9881±0.0072 ▲	0.0209±0.0107 ▲	0.9978±0.0016 ▲
SMO Polykernel	97.32±0.76 ▲	0.9665±0.0115 ▲	0.0202±0.0102 ▲	0.9732±0.0076 ▲
SMO Puk	98.29±0.61 ▲	0.9735±0.0104 ▲	0.0078±0.0058 ▲	0.9829±0.0061 ▲
SMO Normalised Polykernel	96.52±0.89 ▲	0.9608±0.0135 ▲	0.0304±0.0120 ▲	0.9652±0.0089 ▲
SMO RBF	95.06±0.98 ▲	0.9577±0.0119 ▲	0.0566±0.0152 ▲	0.9506±0.0098 ▲
KNN k=1	98.97±0.46 ▲	0.9878±0.0072 ▲	0.0086±0.0059 ▲	0.9897±0.0046 ▲
KNN k=3	98.58±0.51 ▲	0.9831±0.0086	0.0115±0.0064 ▲	0.9925±0.0040
KNN k=5	98.15±0.60 ▲	0.9781±0.0099	0.0150±0.0074 ▲	0.9933±0.0037
C4.5	97.90±0.80 ▲	0.9772±0.0115 ▲	0.0193±0.0111 ▲	0.9797±0.0099 ▲
RandomForest 10	98.96±0.48 ▲	0.9914±0.0063 ▲	0.0122±0.0073 ▲	0.9984±0.0019 ▲
RandomForest 30	99.10±0.42 ▲	0.9914±0.0061 ▲	0.0094±0.0057 ▲	0.9992±0.0012 ▲
RandomForest 50	99.15±0.43 ▲	0.9917±0.0060 ▲	0.0087±0.0057 ▲	0.9992±0.0013 ▲
JRip	97.84±0.75 ▲	0.9758±0.0125 ▲	0.0190±0.0102 ▲	0.9824±0.0075 ▲
PART	98.27±0.61 ▲	0.9811±0.0095 ▲	0.0158±0.0092 ▲	0.9853±0.0073 ▲
Multilayered Perceptron	95.90±2.14	0.9678±0.0243	0.0497±0.0463	0.9863±0.0068 ▲
Bagging C4.5	98.24±0.74 ▲	0.9801±0.0102 ▲	0.0153±0.0098 ▲	0.9955±0.0042 ▲

Statistically significant improvement (▲) or degradation (▽)

Table 3.6 lists the results for the same classifiers when structural features are added to baseline heuristics for classification. In this case, we do not exclude the baseline heuristics. We can observe that most classifiers present a significant improvement for accuracy, TPR, FPR and AUC. More concretely, all classifiers except Naive Bayes and Multilayered Perceptron present an statistically significant improvement of the accuracy. Besides, all classifiers except K Nearest Neighbours with $k = 3$ and $k = 5$ and Multilayered Perceptron show an improvement on TPR. Regarding FPR, again, most of the classifiers improve the results except Naive Bayes, Bayesian networks with K2 and Hill Climber learning algorithms and Multilayered Perceptron. Finally, regarding AUC, only Naive Bayes and K Nearest Neighbours with $k = 3$ and $k = 5$ do not outperform the baseline.

As opposed to previous cases in which structural features were compared against baseline heuristics, Naive Bayes and Multilayered Perceptron

are the classifiers that do not present better results in general. Specifically, Naive Bayes presents an statistically significant degradation, while Multilayered Perceptron does not show any significant difference except for the improvement of its AUC.

In this case, the best performing algorithms are Random Forest and K Nearest Neighbours, together with Bayesian Networks with TAN learning algorithm, SMO with PUK kernel, PART, and Bagging C4.5. These algorithms achieve accuracy rates over 98% (reaching 99% for Random Forest), AUC over 0.985, true positive rates over 0.975 and false positive rates below 0.02.

Table 3.7: Baseline and complementary heuristics compared to baseline heuristics.

Classifier	Accuracy	TPR	FPR	AUC
Naive Bayes	81.88±1.65 ▲	0.9553±0.0141	0.3177±0.0322 ▲	0.9349±0.0101 ▽
Bayes K2	94.26±0.92	0.9607±0.0135	0.0755±0.0154	0.9851±0.0043
Bayes Hill Climber	94.26±0.92	0.9607±0.0135	0.0755±0.0154	0.9851±0.0043
Bayes TAN	96.36±0.91	0.9808±0.0097 ▲	0.0537±0.0149	0.9914±0.0035
SMO Polykernel	94.05±1.09 ▲	0.9491±0.0137 ▲	0.0682±0.0172 ▽	0.9405±0.0109 ▲
SMO Puk	98.20±0.62 ▲	0.9837±0.0085 ▲	0.0196±0.0089	0.9820±0.0062 ▲
SMO Normalised Polykernel	95.03±0.98 ▲	0.9529±0.0137 ▲	0.0524±0.0134 ▲	0.9503±0.0098 ▲
SMO RBF	91.95±1.38 ▲	0.9215±0.0190 ▲	0.0826±0.0193 ▲	0.9194±0.0138 ▲
KNN k=1	97.59±0.76	0.9723±0.0105	0.0205±0.0099	0.9759±0.0076
KNN k=3	97.29±0.84	0.9709±0.0114	0.0251±0.0107	0.9886±0.0052
KNN k=5	96.99±0.85	0.9695±0.0115	0.0298±0.0108	0.9903±0.0048
C4.5	96.83±0.93	0.9702±0.0123	0.0337±0.0127	0.9671±0.0125
RandomForest 10	98.25±0.60 ▲	0.9862±0.0080 ▲	0.0213±0.0095	0.9965±0.0028 ▲
RandomForest 30	98.36±0.55 ▲	0.9862±0.0080 ▲	0.0189±0.0080	0.9979±0.0017 ▲
RandomForest 50	98.40±0.56 ▲	0.9856±0.0077 ▲	0.0176±0.0087	0.9980±0.0017 ▲
JRip	97.06±0.83	0.9707±0.0134	0.0295±0.0115	0.9769±0.0081
PART	97.15±0.85	0.9715±0.0126	0.0285±0.0119	0.9740±0.0101
Multilayered Perceptron	97.68±0.72 ▲	0.9759±0.0115 ▲	0.0223±0.0094 ▲	0.9914±0.0050 ▲
Bagging C4.5	97.23±0.82	0.9746±0.0114	0.0300±0.0115	0.9924±0.0048

Statistically significant improvement (▲) or degradation (▽)

In Table 3.7 we can observe the results obtained when complementary heuristics are added to baseline heuristics. In this case, we can notice that there is not an overall performance improvement like in the previous case. However, some algorithms benefit from the inclusion of this feature-set. Random Forest and SMO classifiers present significant performance improvements regarding accuracy, TPR and AUC. Regarding FPR, SMO with Polynomial kernel presents a significant degradation, SMO with Normalised Polynomial and RBF kernels shows an improvement, while the rest of configurations do not vary significantly. As opposed to previous cases, Naive Bayes and Multilayered Perceptron manifest a significant improve-

ment, except for the AUC of Naive Bayes. The rest of algorithms, in general, do not present variations with respect to the baseline heuristics.

In this case, the best performing algorithms are Random Forest, SMO with PUK kernel and Multilayered Perceptron, with accuracies over 97.5%, AUC over 0.98, true positive rates over 0.98 (except for Multilayered Perceptron), and false positive rates below 0.025. Naive Bayes, despite the improvement, still presents an unacceptable false positive rate.

Table 3.8: Baseline heuristics with structural features and complementary heuristics compared to baseline heuristics.

Classifier	Accuracy	TPR	FPR	AUC
Naive Bayes	78.98±1.71 ▽	0.9783±0.0105 ▲	0.3987±0.0342 ▽	0.8021±0.0161 ▽
Bayes K2	95.93±0.82 ▲	0.9789±0.0099 ▲	0.0604±0.0156	0.9899±0.0041 ▲
Bayes Hill Climber	95.92±0.82 ▲	0.9789±0.0099 ▲	0.0605±0.0156	0.9898±0.0041 ▲
Bayes TAN	98.14±0.66 ▲	0.9888±0.0070 ▲	0.0260±0.0113 ▲	0.9973±0.0018 ▲
SMO Polykernel	97.91±0.70 ▲	0.9791±0.0107 ▲	0.0210±0.0086 ▲	0.9791±0.0070 ▲
SMO Puk	98.20±0.59 ▲	0.9754±0.0100 ▲	0.0115±0.0069 ▲	0.9820±0.0059 ▲
SMO Normalised Polykernel	97.45±0.74 ▲	0.9832±0.0089 ▲	0.0342±0.0127 ▲	0.9745±0.0074 ▲
SMO RBF	96.13±0.91 ▲	0.9732±0.0113 ▲	0.0506±0.0153 ▲	0.9613±0.0091 ▲
KNN k=1	98.59±0.59 ▲	0.9838±0.0092	0.0122±0.0070	0.9858±0.0059 ▲
KNN k=3	98.54±0.59 ▲	0.9834±0.0094	0.0126±0.0069 ▲	0.9926±0.0042
KNN k=5	98.42±0.64 ▲	0.9830±0.0097 ▲	0.0147±0.0079 ▲	0.9945±0.0038
C4.5	97.39±0.77 ▲	0.9718±0.0125	0.0241±0.0106	0.9733±0.0111
RandomForest 10	98.95±0.47 ▲	0.9922±0.0060 ▲	0.0133±0.0073 ▲	0.9981±0.0021 ▲
RandomForest 30	99.01±0.48 ▲	0.9915±0.0061 ▲	0.0114±0.0068 ▲	0.9988±0.0015 ▲
RandomForest 50	99.09±0.46 ▲	0.9920±0.0062 ▲	0.0103±0.0063 ▲	0.9991±0.0012 ▲
JRip	97.91±0.75 ▲	0.9765±0.0123 ▲	0.0183±0.0108 ▲	0.9836±0.0077 ▲
PART	98.08±0.65 ▲	0.9790±0.0100 ▲	0.0174±0.0092 ▲	0.9826±0.0079
Multilayered Perceptron	96.40±2.16	0.9661±0.0465	0.0381±0.0241	0.9865±0.0100
Bagging C4.5	97.83±0.72 ▲	0.9763±0.0116	0.0197±0.0091 ▲	0.9949±0.0037

Statistically significant improvement (▲) or degradation (▽)

Finally, Table 3.8 gathers the results achieved when all the features were considered, including the baseline heuristics. We can notice that, similarly to Table 3.6, there is an overall performance improvement with respect to baseline heuristics. Again, Naive Bayes shows a significant degradation, while Multilayered Perceptron does not present significant variations. Nevertheless, if we compare the results obtained for these classifiers against Table 3.6, we can observe that the results are improved with the addition of the complementary heuristics. These classifiers also manifested a significant improvement when the inclusion of complementary heuristics was tested on Table 3.7. Besides, Bayesian Networks, SMO, decision tree based and rule based algorithms show a similar performance to the case in which complementary heuristics were not considered, obtaining, in general, statistically significant improvements if compared against baseline heuristics.

In this case, the best performing algorithms are Random Forest and K Nearest Neighbours, together with Bayesian Networks with TAN learning algorithm, SMO with PUK kernel, and PART. The accuracy of these algorithms is, in all cases, over 98%, reaching 99% for Random Forest, and the AUC is over 0.98. True positive rates over 0.975 are achieved, while false positive rates remain below 0.025.

Table 3.5, Table 3.6, Table 3.7 and Table 3.8 present the results obtained for each classifier and feature-set tested. Nevertheless, it is not possible to determine the overall performance variation among the different feature-sets considering all the algorithms separately.

For this reason, we conduct the Friedman test over the results obtained, which ranks, for each algorithm, the best-performing feature-sets. Table 3.9 shows the average ranks computed for the Friedman test, together with the corresponding F_F value, which follows an F distribution with $k-1$ and $(k-1)(N-1)$ degrees of freedom, where k is the number of feature-sets tested, and N the number of algorithms trained. In this way, the critical value for $F(4,72)$ is 2.4989. In Table 3.9, we can see that this critical value is surpassed for every performance measure tested, which means that there is an statistically significant difference among the feature-sets for each of the performance measures.

Table 3.9: Friedman test over accuracy, true positive rate, false positive rate and area under the curve. The ranks for each algorithm and feature set are omitted. The average ranks and the F_F score are shown.

	B	**S**	**BS**	**BC**	**BSC**	F_F
Acc.	4.5263	3.0526	1.6315	3.7894	2	25.1474
TPR	4.7368	2.8947	2	3.7368	1.6315	32.6074
FPR	4	3.4736	1.4736	4	2.0526	21.5736
AUC	4.3421	3.3684	1.7894	3.6842	1.8157	20.0779

Considering the result of the Friedman test, we perform a post-hoc test in order to rank the differences for each of the performance measures and determine which feature-sets show statistically significant differences. We calculate the $z-score$ for each feature-set according to the difference of its average rank with the average rank for the baseline feature-set. Then, we compare the corresponding p-value (according to normal distribution) with the corresponding corrected α in order to control the family-wise error.

First, we compare the accuracy of the feature-sets proposed with respect to the baseline heuristics (see Table 3.10). In this case, we can observe that

the best performing feature-set is the structural feature-set added to the baseline heuristics, immediately followed by the feature-set which includes all the proposed features. Additionally, we can observe that the structural characteristics also present an statistically significant improvement with respect to baseline heuristics. The conjunction of baseline heuristics with complementary heuristics, in contrast, does not present an statically significant improvement.

Table 3.10: Holm step-down procedure for the Bonferroni-Dunn test over the accuracy of the different classifiers for each feature set.

i	Feature set	$z = (R_0 - R_i)/SE$	p	$\alpha/(k-i)$
1	BS	5.6428	0.0000000167	0.0125
2	BSC	4.9246	0.0000008449	0.0167
3	S	2.8727	0.0040692965	0.0250
4	BC	1.4363	0.1508971723	0.05

(B) baseline heur., (S) structural features, (C) complementary heur.

Regarding true positive rate (see Table 3.11), we can notice that, as opposed to accuracy, the feature-set that includes all the heuristics and structural features outperforms the set with baseline heuristics and structural features. Similarly, structural features present an statistically significant improvement over baseline heuristics. Complementary heuristics, again, do not show a significant improvement when added to baseline heuristics.

Table 3.11: Holm step-down procedure for the Bonferroni-Dunn test over the TPR of the different classifiers for each feature set.

i	Feature set	$z = (R_0 - R_i)/SE$	p	$\alpha/(k-i)$
1	BSC	5.6428	0.0000000167	0.0125
2	BS	4.9246	0.0000008449	0.0167
3	S	3.1805	0.0014700445	0.0250
4	BC	1.5389	0.1238122238	0.05

(B) baseline heur., (S) structural features, (C) complementary heur.

In the case of false positive rate (see Table 3.12), we can observe some similarities with respect to the results obtained for the accuracy. Structural features added to baseline heuristics conform again the best performing feature-set, followed by the feature-set including all the features. Nevertheless, in this case, when structural features are evaluated separately, the

difference in performance is not statistically significant, similarly to the complementary heuristics.

Table 3.12: Holm step-down procedure for the Bonferroni-Dunn test over the FPR of the different classifiers for each feature set.

i	Feature set	$z = (R_0 - R_i)/SE$	p	$\alpha/(k-i)$
1	BS	5.9506	0.0000000027	0.0125
2	BSC	4.8220	0.0000014206	0.0167
3	S	2.0519	0.0401738703	0.0250
4	BC	1.0259	0.3049017882	0.05

(B) baseline heur., (S) structural features, (C) complementary heur.

Finally, Table 3.13 shows the results regarding AUC. We confirm that the best performing feature-set is the one that adds structural features to baseline heuristics, followed by the approach that includes all the heuristics. In this case, structural features present a significant improvement even when no heuristics are included. As in previous cases, complementary heuristics do not provide a significant improvement with respect to baseline heuristics.

Table 3.13: Holm step-down procedure for the Bonferroni-Dunn test over the AUC of the different classifiers for each feature set.

i	Feature set	$z = (R_0 - R_i)/SE$	p	$\alpha/(k-i)$
1	BS	5.3350	0.0000000955	0.0125
2	BSC	5.2837	0.0000001265	0.0167
3	S	2.2571	0.0239985552	0.0250
4	BC	1.6415	0.1006801107	0.05

(B) baseline heur., (S) structural features, (C) complementary heur.

As we have observed in the post-hoc test, there is not a clear difference between the feature-set composed of baseline heuristics and structural features with respect to the feature-set that includes also the additional features. For this reason, we have also conducted the Holm procedure to compare these feature-sets, considering as baseline, in each case, the one with the lowest rank. In this way, for the accuracy, false positive rate, and AUC, the results indicate an improvement of baseline heuristics with structural features over the complete feature-set with p values of 0.4726, 0.2591 and 0.9591 respectively. This improvement is not statistically significant in any

of the three cases at a suitable α level. In the case of true positive rate, the complete feature-set outperforms the baseline heuristics with structural features with a p value of 0.4726. Again, this difference is not statistically significant.

Apart from the tests shown, for each algorithm and feature-set tested, we also have evaluated the impact of applying a feature selection step before classification. To this aim, we calculate the Information Gain (IG) of each feature used for classification, discarding every feature with an IG value of 0. However, the results obtained do not show any significant improvement when this approach is applied, and thus the results have been omitted.

3.8 Conclusions and discussion

The results presented in Section 3.7 show the impact of including structural features as well as complementary heuristics as input features for supervised machine-learning classifiers. Considering these results, we can answer the research questions established for this study.

Do PE structural features significantly improve the performance of supervised machine-learning classifiers for packed binary detection?

First, we have conducted different statistical tests in order to answer the first research question. In all cases, the feature-sets including structural features present an statistically significant performance improvement with respect to the baseline heuristics.

In this way, these results validate our initial hypothesis: there is an statistically significant improvement when structural features are employed in conjunction with baseline heuristics for the detection of packed binaries.

Besides, we have proposed the inclusion of an additional set of heuristics. Results show that, although these heuristics improve the results of several classifiers when considered together with the baseline heuristics, there is not an statistically significant improvement for any of the performance measures tested, and thus we cannot affirm that these complementary heuristics improve the overall performance of supervised machine-learning classifiers when added to baseline heuristics.

When these complementary heuristics are added to the baseline heuris-

tics and the structural features, we can draw similar conclusions. Despite in the case of TPR the results obtained by the complete feature-set outperform the rest of combinations, for accuracy, FPR and AUC the best results are obtained when these heuristics are not included. None of the cases show statistically significant differences according to the Holm procedure applied after the Friedman test.

As a consequence, we cannot confirm that these complementary heuristics provide a significant improvement for supervised machine-learning classifiers.

What are the best performing machine-learning algorithms when different feature-sets are employed for classification?

We can affirm that, in all the cases, the algorithm that presents the best results is Random Forest. Nevertheless, this algorithm is known for its tendency to overfit the dataset. If we consider the complete feature-set, we can observe that K Nearest Neighbours also shows sound classification performance with an accuracy over 98.5% (except for $k = 5$), AUCs between 0.985 and 0.995, and nearly balanced true positive rate and false positive rate, with an slight tendency to produce false positives. Among the rest of algorithms, the best configurations for the different feature-sets in general are SMO with PUK kernel, Bayesian Networks (TAN), and PART.

Besides, we can also conclude that Naive Bayes does not present results as sound as the rest of algorithms. In addition, this algorithm shows a tendency to present weaker results when structural features are used for learning. Similarly, Multilayered Perceptron also shows this tendency when only structural features are considered, but maintains its performance when baseline heuristics are considered.

Despite the results confirm our initial hypothesis, we must consider the limitations of the experiment. On the one hand, we have defined a methodology in order to reduce the risk of an incorrect assignment of labels to the dataset. However, labelling binaries as packed and non-packed is sometimes a difficult task. It is strictly necessary to include non-packed malware in the dataset (the class with the highest probability of being incorrectly labelled), in order to isolate the factor of packed/non-packed classification from the goodware/malware classification.

For this reason, we must assume that this noise might influence the results obtained. Nonetheless, as we compare the results against a baseline feature-set which is evaluated with the same dataset, the conclusions of the

experiment should not be affected by this noise.

On the other hand, the experiment may be influenced by the packers included. Although different and representative packers have been employed, supervised machine-learning algorithms sometimes present overfitting problems. In situations in which one of the classes to be detected presents a high variability, or when it is easier to represent one of the classes (i.e., binaries produced by standard compilers) it is possible to apply other methods such as anomaly detection.

3.9 Summary

This chapter proposes the extraction of Portable Executable structural features for the classification of packed binaries, based on the application of supervised machine-learning classification techniques.

First, it thoroughly describes the feature-set tested and the methodology followed to configure the dataset used for the experiments. Second, it describes the machine-learning algorithms employed and the statistical tests used to compare the performance of the different algorithms and feature-sets tested. Finally, it presents the results obtained during the experiment.

In summary, the study shows that structural features present an statistically significant performance improvement with respect to classic heuristics.

«An alpinist is a man who drives his body where his eyes once looked. And who gets back.»

Gaston Rébuffat (1921–1985)

Chapter

4

Anomaly detection based classification of packed binaries

Compiled applications generally follow the standards and conventions defined by Microsoft. In contrast, many packers introduce modifications on the file structure. In fact, while many of these modifications violate the specifications, the Windows Loader is rather permissive and allows to execute files that present certain malformations. Intuitively, it should be easier to model the class of files which follows conventions. Any deviation to this model may be treated as an outlier (in our case, a packed binary). Also, it is relatively easy to modify existing packers to generate new versions of protectors. Current malware writers employ both well-known off-the-shelf packers, modified versions (also known as scrambled) versions of packers, and sometimes they even implement their own protection engines. Supervised machine-learning algorithms *learn* from training sets that contain both classes of binaries: packed and non-packed. The study presented in this chapter is motivated by the hypothesis that packer binaries can be detected by only considering their divergence from non-packed binaries, instead of also considering their similarity to other packed files.

Anomaly detection techniques model one class of data, (i.e., normality)

and try to differentiate that class from everything else [WSAR13]. These techniques are widely used for classification problems with different limitations:

- Little or no data is available for the anomalous class. When there is more data representing one of the classes, the dataset used for training supervised machine-learning algorithms is unbalanced. This kind of algorithms do not usually perform well under these conditions. One example of this situation is fault prediction, a problem in which we can record data about the correct functioning of a system but we do not have sufficient examples of malfunctioning.
- The available anomalies are not representative enough. Supervised machine-learning classifiers tend to over-fit to the types of anomalies present in the dataset, and thus do not perform correctly when new types of anomalies have to be detected.

Considering this background, we propose the application of anomaly detection for the classification of packed binaries, modelling non-packed files as normality and measuring the deviation to this model. More concretely, we define the problem of classifying packed binaries as a preceding step for different tasks such as computationally expensive generic unpacking, or static analysis of the binary.

While many approaches for anomaly detection provide a binary result, other approaches can measure the anomaly score [CBK09] of an instance. For this reason, we propose and evaluate the application of a configurable anomaly detection system that can adjust the anomaly score to control the number of binaries reported as anomalous (in our case, packed). Depending on the use-case, this threshold can be adjusted to meet different requirements. More specifically, we evaluate two different feature-sets: (i) the complete feature-set described in Chapter 3, and (ii) a feature-set based on operational code frequency.

Given this background, we define the following research questions for the study:

Research question 4.1 *Does our anomaly detection approach present sound results for the classification of packed and non-packed files?*

Research question 4.2 *What is the impact of the data-reduction approach over the results obtained?*

Research question 4.3 *What is the impact on the results of the different distance measures evaluated?*

Research question 4.4 *What is the impact on the results of the different distance selection rules?*

Research question 4.5 *Which is the feature-set that best discriminates packed from non-packed files?*

The rest of this chapter is structured as follows. Section 4.1 reviews some of the existing anomaly detection approaches. Then, Section 4.2 describes the method proposed. Section 4.3 describes the evaluation method and results obtained for two different feature sets in terms of effectiveness and efficiency. Finally, Section 4.4 concludes the study and discusses the results obtained.

4.1 Anomaly detection

Anomaly detection refers to the problem of finding patterns in data that do not conform to expected values [CBK09]. These elements are denoted anomalies, while the expected patterns are considered normality.

Anomaly detection has been widely applied to different security problems [WPS06, FHSL96, LEK+03]. One of the main reasons for the application of such techniques is the variability of the data over time (i.e., it is not possible collect data about every possible present or future attack).

Anomaly detection systems are based on different learning approaches. Some of the methods produce an anomaly score as a result. Other approaches, in contrast, require to tune different parameters in order to adjust the threshold used to discriminate anomalies from normality.

- **Classification based approaches.** These techniques employ two different approaches in order to discriminate anomalies from normality. On the one hand, multi-class approaches train classification models that discriminate the instances in each of the classes considered normal from the rest of the instances. If the instance is not classified as any of these classes, then it is considered anomalous. On the other hand, one-class classification approaches define a boundary around the normal instances and consider anomalous any test instance outside the defined limits [CBK09]. Several approaches [HSKS03, PGL06] have applied one-class Support Vector Machines (SVM) (originally

proposed by Schölkop et al. [SPST+01]) to anomaly detection building spheroidal hyperplanes enclosing the data with a minimum volume [WSAR13]. Besides, Support Vector machines are predominantly designed to divide the space into two groups [DK05]. Nevertheless, some parameters can be modified in order to produce over-fitting or under-fitting (see ν-Support Vector Classification [SSWB00]), and thus allow to configure the algorithm to adopt an anomaly score based approach.

- **Nearest Neighbours based approaches.** Nearest Neighbours based machine-learning algorithms measure the distance between two instances in a linear n-dimensional space. Some approaches are based on the computations of the distance from the testing instance to its K Nearest neighbours [CBK09]. Some approaches have extended this method by adopting different similarity measures [WQZ+03] or improving the efficiency (e.g., applying clustering techniques [TXZ06]). Other approaches have focused on measuring the density of data instances. Breunig et al. [BKNS00] proposed a density-based method for local outlier detection. They defined the local outlier factor as a degree of a given instance of being an outlier, and defined lower and upper bounds for the factor based on the density of the instances representing normality.

- **Statistical approaches.** Other approaches fit different statistical models to the data representing normality, and then apply statistical inference tests such as hypothesis tests or regression [CBK09]. In this way, it is possible to determine if an instance belongs to the model or, on the contrary, is an anomaly. Both parametric and non parametric models have been used for statistical anomaly detection. Additionally, the confidence interval provided by these models can be associated to the anomaly score in order to stablish thresholds. Sometimes, this threshold is related to the number of samples that are considered anomalous (see Grubb's test [Gru69]). When univariate approaches are used, the computations are fairly simple. On the contrary, multivariate approaches such as the computation of the Mahalanobis distance [LJK+00], which requires to compute the covariance matrix, are computationally expensive when the number N of dimensions increases.

- **Clustering based approaches.** Unsupervised techniques can be used

to group similar instances into clusters. In the particular case of anomaly detection, clustering has been used to determine if a test instance belongs to any of the clusters formed by training data, or to measure the deviation to the centroids of the clusters [CBK09]. A technique commonly used is Self-Organizing Maps (SOM) [Koh01], which is an artificial neural network trained with unsupervised techniques.

4.2 Method proposed

In this section, we describe the method proposed to adopt anomaly detection for the classification of packed binaries. Considering the results obtained in the experiments described in Chapter 3, we select the feature-set comprising all the possible features. As we noted, the statistical tests for supervised machine-learning algorithms do not provide strong evidence in favour or against the inclusion of complementary heuristics together with baseline heuristics and structural features. For this reason, in order to avoid discarding features without enough evidence (that might contain valuable information), we include all the features in this study.

Regarding the results obtained for the different algorithms tested over the complete feature-set, the best performing approaches (considering all possible configurations) were Random Forest and K Nearest Neighbours. Both approaches obtained sound accuracy and AUC results. Nevertheless, Random Forest has been reported to present a tendency to overfitting [SLTW04]. Besides, the adoption of Random Forest for anomaly detection is not straightforward. On the contrary, the adoption of K Nearest Neighbours for anomaly detection is simple, the results of the algorithm are considered stable in contrast to other learning approaches [Bre96].

Distance based approaches present several advantages:

- **No assumptions about the distribution of the data.** Nearest neighbours approaches do not assume any particular distribution of the data. Instead, they just consider the distance from test instances to the available data.

- **Computational complexity of the distance measures.** K Nearest Neighbours allows the adoption of efficient distance measures. In this way, the computational complexity of the approach is $O(N^2)$, depending on the number N of samples that conform the instances representing normality.

- **Data-reduction capabilities.** In order to reduce the computational complexity, we can reduce the number N of instances for comparison applying clustering techniques.

- **Straightforward adoption of anomaly score.** The anomaly score is directly related to the distances measured from the test instance to the rest of instances. The process of adjusting thresholds for different use-cases in straightforward.

Given this background, we propose a distance-based anomaly detection method, combined with clustering techniques for the reduction of data in order to provide an efficient detection of packed samples. In addition, we propose 3 different neighbour selection rules and test them under different conditions.

4.2.1 Representation of normality

We model normality as a set of non-packed executables. In this way, any sample that deviates sufficiently from our representation of normality is classified as packed. As opposed to supervised learning approaches, this method does not need a model training phase requiring to label packed executables. Despite the evaluation process of this method requires the use of a previously labeled dataset, its deployment would only require non-packed samples, reducing the efforts needed to find and label a set of representative packed binaries.

In this way, we define our normality model as a set of i points $\mathcal{NP} = \{np_0, np_1, ..., np_i\}$ defined in an n-dimensional feature space such that $np_i = \{f_{i,0}, f_{i,1}, ..., f_{i,n}\}$.

Afterwards, we normalise the instances in our normality model by calculating the well-known standard score or z-score (see Equation 4.1), for each instance and feature.

$$z_{i,n} = \frac{x_{i,n} - \mu_n}{\sigma_n} \tag{4.1}$$

In this way, we obtain a representation of normality $\mathcal{N} = \{\mathcal{NP}', \mathcal{M}, \mathcal{D}\}$ where $\mathcal{NP}'$ is the set of normalised points $\mathcal{NP}' = \{np'_0, np'_1, ..., np'_i\}$, each point is defined as a set of normalised values such that $np'_i = \{z_{i,0}, z_{i,1}, ..., z_{i,n}\}$, $\mathcal{M}$ represents the means for each feature $\mathcal{M} = \{\mu_0, \mu_1, ..., \mu_n\}$ and $\mathcal{D}$ represents the corresponding standard deviations $\mathcal{D} = \{\sigma_0, \sigma_1, ..., \sigma_n\}$.

4.2.2 Distance between files

The model represents each executable as a point in the feature space. Our anomaly detection system compares points in the feature space and classifies executables based on their similarity. In this way, the classification of an executable consists of 3 different phases:

1. Extraction of the features from the executable file.
2. Normalization of the representation according to the reference model. In this step, we use the mean and standard deviation of each feature, and calculate the z-score for each field of the file.
3. Computation of the distance from the point representing the executable file to the points represented in the model (i.e., non-packed executables).

As a result, the distance measured is a score representing the deviation of the file with respect to non-packed executables. In this way, we can establish a threshold such that any point at a distance from normality higher than an established threshold is considered to be an anomaly and thus, a packed executable. In this study, we have considered 2 different distance measures:

- **Manhattan distance.** The distance between two points x and y is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes:

$$d(x,y) = \sum_{i=0}^{n} |x_i - y_i| \tag{4.2}$$

  where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.

- **Euclidean Distance.** The distance is the length of the line segment connecting two points. It is calculated as:

$$d(x,y) = \sum_{i=0}^{n} \sqrt{x_i^2 - y_i^2} \tag{4.3}$$

  where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.

Since we have to compute this measure with respect to a variable number of points representing non-packed executables, a combination metric is required in order to obtain a final distance value which considers every measure performed. Some approaches consider the K Nearest neighbours, while others form clusters in the dataset and compare each sample to each of the clusters. Our approach consists of applying 3 different distance selection rules. In this way, we can capture different aspects and measure how these aspects affect to the final classification of the samples.

- **Mean distance rule.** This rule averages the distances to all the samples in the dataset. It is robust to outlier points in the normality model, as the effects of such points are smoothed.

- **Maximum distance rule.** The highest distance value represents the distance to the normality instance which is less similar to the testing instance.

- **Minimum distance rule.** The lowest distance value represents the distance to the most similar normality instance.

In this way, when an executable is analysed, the final distance value calculated depends on the distance measure and the combination rule selected.

4.2.3 Data reduction

This approach makes necessary to compute as many distance values as executables in the non-packed set. We improve the efficiency of our system by designing a data reduction phase capable of boosting the detector's scalability. Figure 4.1 shows the architecture of our proposed system. The first objective of our method is to improve its efficiency by applying data reduction. The data reduction phase consists in the application of the Quality Threshold (QT) clustering algorithm to the original dataset to obtain a reduced version that maintains the original features of the dataset. In this way, the number of comparisons performed, and thus, the comparison time required for the analysis of each sample are much lower.

The second objective is to measure the precision of our system when the training set is incrementally reduced, in order to evaluate the trade-off between efficiency and accuracy. In addition, this data reduction approach

enables us to test the performance of the system when a unique representation of a *normal* executable is used, and to determine if it can correctly classify packed and non-packed executables.

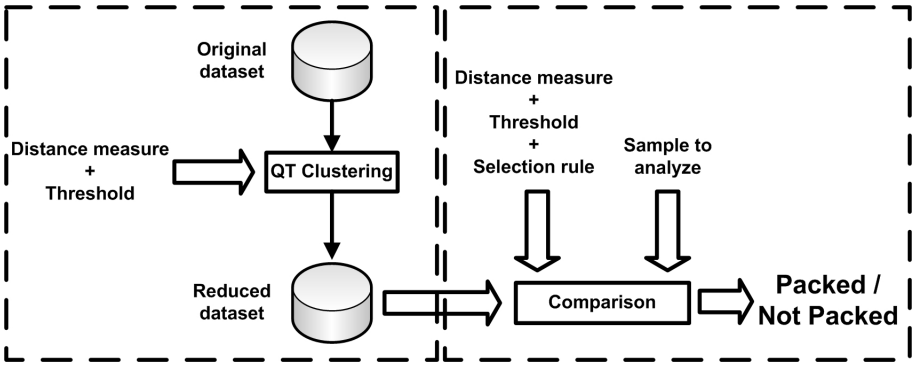


Figure 4.1: Architecture of the proposed system. The QT clustering algorithm transforms the original dataset into a new reduced synthetic dataset. It requires 2 parameters: the distance measure and the threshold. The approach compares samples against the reduced dataset obtained, applying a distance measure, a distance threshold, and a selection rule. Finally the system classifies the sample as packed or non-packed.

Dataset reduction is a step that has to be faced in very different problems involving large datasets. Now, we propose a data reduction algorithm based on partitional clustering. Cluster analysis divides data into meaningful groups [Kum00]. We can identify several types of clustering, but most common ones are hierarchical clustering and partitional clustering. The first approach generates clusters in a nested style, which means that the clusters generated from the dataset are related hierarchically. In contrast, partitional clustering techniques create a one-level (unnested) partitioning of the data points [Kum00]. We are interested in this last technique in order to divide a big set of executables that represent normality (i.e., non-packed executables) into a reduced set of representations.

The QT clustering algorithm was proposed by Heyer et al. [HKY99] to extract useful information from large amounts of gene expression data. K-means is a classic algorithm for partitional clustering, but it requires to specify the number of clusters desired. In contrast, QT clustering algorithm does not need this specification. It uses a similarity threshold value to determine the maximum radial distance of any cluster. In this way, it generates a variable number of clusters that meet a quality threshold. Its

main disadvantage is the high number of distance computations needed. Nevertheless, in our case, this computational overhead is admissible because we only have to reduce the dataset once.

Our algorithm, shown in Figure 1, is based on the concepts proposed by Heyer et al. [HKY99]. Let $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n\}$ be the set of potential clusters. For each point p_i in the dataset $\mathcal{D}$, there is potential cluster $\mathcal{A}_i \in \mathcal{A}$. A potential cluster $\mathcal{A}_i$ is composed of the set of points at a distance respect to p_i not higher than the *threshold* previously specified.

Once the potential clusters are calculated, we select the cluster with the highest number of vectors as a final cluster. Then, we calculate its centroid, defined as $c = x_1 + x_2 + \cdots + x_k/k$ where $x_1, x_2, \cdots, x_k$ are points in the feature space. The resultant centroid is added to the final reduced dataset. Afterwards, each point p_j present in the selected cluster $\mathcal{A}_i$ is removed from the original dataset $\mathcal{V}$ (as they will be represented by the previously calculated centroid). Moreover, the potential clusters $\mathcal{A}_j \in \mathcal{A}$ associated to each point p_j previously removed are also discarded. When there are not more clusters available with a number of points higher than the parameter min_points, the remaining points in $\mathcal{D}$ are added to the final reduced dataset and the algorithm finishes and returns the resulting reduced dataset. The final result is a dataset composed of one centroid representing each cluster and all the points that were not associated to any cluster by the QT clustering algorithm (i.e., outliers).

4.2.4 Selection of a threshold

The proposed anomaly detection method measures the distance from a given binary to the representation of normality. Nevertheless, the result of this classification process must be a binary attribute, making necessary to stablish a distance threshold to discriminate samples.

input : Dataset $\mathcal{D}$, quality *threshold* , minimum number of points in each cluster *min_points*, and the parameter *include_outliers*
output: The reduced dataset $\mathcal{R}$

```
// Calculate the distances among all the points.
foreach {p_i|p_i ∈ D} do
    foreach {p_j|p_j ∈ D} do
        if distance(p_i,p_j) ≥ threshold then
            A_i.add(p_j)
        end
    end
end
// In each loop, select the potential cluster with the
   highest number of points
while ∃A_i ∈ A : |A_i| ≥ min_points and ∀A_j ∈ A : |A_i| ≥ |A_j| and i ≠ j do
    R.add(centroid(A_i))
    foreach {p_j|p_j ∈ A_i} do
        A.remove(A_j)
        D.remove(p_j)
    end
    foreach {A_k|A_k ∈ A} do
        foreach {p_j|p_j ∈ A_k and p_j ∈ A_i} do
            A_k.remove(p_j)
        end
    end
end
if include_outliers then
    // Add the remaining points to the final dataset
    foreach {p_j|p_j ∈ V} do
        R.add(p_j)
    end
end
```

Algorithm 1: QT Clustering based dataset reduction algorithm.

Normally, supervised classification methods adjust thresholds considering the instances present in the training set, trying to maximise an score function such as the classification accuracy. Afterwards, the thresholds are applied for the classification of the test-set. In this case, the training-set is comprised only of non-packed binaries. Accordingly, we describe two different approaches for threshold selection.

- **Maximum tolerable false positive rate.** This approach tries to approximate the false positive rate of the testing set. As we mentioned previously, the false positive rate measures the number of non-packed samples incorrectly considered packed, a ratio that we can calculate regardless of the classification of packed binaries. In order to select this threshold, we extract from the training set a 10% of random binaries (i.e., validation set). In this way, we measure the distance from these instances to the rest of the training set, and select the thresholds that produce a fixed number of false positives. These thresholds, in an ideally distributed dataset, will approximate the false positive rate for the testing set. Besides, we will test the associated false negative rate and accuracy in order to evaluate their reliability for the detection of packed binaries.

- **Number of desired positives.** Another possible approach is the selection of the most anomalous instances for inspection, considering that they might present a higher probability of being packed. In this way, depending on the processing capabilities of a hypothetical automatic unpacking system, it would be possible to apply a threshold according to the number of positives it produces. Nevertheless, in order to deploy such approach it is interesting to approximate the distribution of binaries to analyse. For instance, an antivirus company might want to process a malware database. In such scenario, the majority of samples analysed would be packed, and thus, it would not make sense to accept a low number of binaries as packed. Besides, an on-line binary analysis system (e.g., Anubis[1], VirusTotal[2]) can receive binaries from many different sources, and the distribution of packed/non-packed binaries might vary.

While the first approach can be easily evaluated, the second approach entirely depends on the deployment scenario. Thus, we adopt the *maximum tolerable false positive rate* as an evaluation criteria for our anomaly detection method.

[1] `http://anubis.iseclab.org/`
[2] `http://www.virustotal.com`

4.3 Evaluation of the method proposed

In this section, we evaluate the proposed anomaly detection method over the dataset described in Chapter 3.

4.3.1 Evaluation method

In order to evaluate the performance of our anomaly detection system, we applied a variation of k-fold cross validation, with $k = 10$. First, we divided the training set (comprised of 2000 non-packed samples) into 10 groups of 200 binaries. For each fold, 1800 samples were selected as training instances, while 200 were selected for testing. In our approach, packed binaries are not considered for the training phase and thus, for each fold, all the available samples are used for testing.

Afterwards, for each training-set selected, a 10% of the instances (180) were reserved as validation set, in order to adjust the possible thresholds. In this way, each fold was configured in the following way:

- 1620 non-packed samples as training set.
- 180 non-packed samples as validation set.
- 2000 packed samples for testing.
- 200 non-packed samples for testing.

We can observe that the testing set is highly unbalanced. For this reason, we evaluate the performance of the approach considering the false positive rate, false negative rate, and the AUC, avoiding the accuracy. In this way, the false positive rate measures the non-packed samples incorrectly classified as packed, the false negative rate measures the packed samples considered non-packed, and the AUC measures the overall performance of the approach regardless of the threshold selected.

Regarding the data reduction process applied to each configuration, in all cases, the minimum number of instances for each cluster was set to 2. The distance thresholds for each configuration are dependant on the distance measure and feature set employed. In order to correctly represent different reduction rates for each configuration, the thresholds were adjusted considering the number of instances resulting from the reduction process.

Considering that it is not reliable to represent all the results in terms of the TPR and the FPR obtained for all the different reduction rates, we plot the AUC together with the number of instances obtained as a result of the reduction process (see Figures 4.2 to 4.6). Additionally, we show and describe the results obtained for the most interesting reduction rates for each configuration (see Tables 4.3 to 4.6 and 4.9 to 4.12). In each case, the reduction threshold was selected based on the trade-off between the AUC and the reduction rate, selecting the combination which presented the highest possible reduction rate while still maintaining a sound AUC. The selection of the appropriate configuration should fall on the expert's sound judgement, as it depends on the specific deployment scenario and its requirements.

4.3.2 Evaluation of the method for PE based features

In this section, we detail the results obtained for our anomaly detection approach when applied to the complete feature-set described in Chapter 3.

4.3.2.1 Results without dataset reduction

First, we list the results obtained for Euclidean distance (see Table 4.1) and Manhattan distance (see Table 4.2), applying the different selection rules, when no reduction process is applied. In both cases, we measured separately the false negative rate for the subset of off-the-shelf packers (P) (manually packed), and the subset of custom packed binaries (CP). Similarly, we show the AUC when custom packers are not considered, and the overall AUC when all packed binaries are included in the study. As we can observe, there is, in both cases, a clear difference between the detection rate for off-the-shelf packers and custom packers.

More concretely, for Euclidean distance, we can observe that the best results are achieved when Mean and Min selection rules are applied. More concretely, for off-the-shelf packers, the AUC presents values of 0.9723 and 0.9824 respectively. If we consider all the packed binaries, the AUC degrades. Nevertheless, the degradation observed for the minimum distance is much lower when compared against the Mean selection rule.

Besides, if we observe the FPR, first we can notice that in all cases the FPR of the validation set is adjusted to meet the maximum tolerable value established. Also, the actual FPR obtained for the testing set is below the FPR obtained for the Mean and Max distance selection rules, while for the

Table 4.1: Results obtained for PE based features and Euclidean distance.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC(P)	AUC
Mean	0.0100	0.0056	0.0040	0.8013	0.9371	0.9723	0.8511
	0.0200	0.0167	0.0120	0.5650	0.9238		
	0.0500	0.0500	0.0290	0.3850	0.9133		
	0.1000	0.1000	0.0640	0.0586	0.8751		
	0.1500	0.1500	0.1160	0.0010	0.7921		
	0.2000	0.2000	0.1535	0.0010	0.6829		
	0.2500	0.2500	0.2135	0.0010	0.5920		
	0.3000	0.3000	0.2770	0.0010	0.4437		
Max	0.0100	0.0056	0.0040	0.7997	0.9371	0.9040	0.7991
	0.0200	0.0167	0.0125	0.6579	0.9233		
	0.0500	0.0500	0.0400	0.4081	0.9013		
	0.1000	0.1000	0.0720	0.1840	0.8671		
	0.1500	0.1500	0.1115	0.1165	0.8261		
	0.2000	0.2000	0.1590	0.0932	0.7415		
	0.2500	0.2500	0.2380	0.0780	0.5822		
	0.3000	0.3000	0.3040	0.0657	0.5373		
Min	0.0100	0.0056	0.0045	0.8030	0.9371	0.9824	0.9526
	0.0200	0.0167	0.0110	0.5764	0.9193		
	0.0500	0.0500	0.0505	0.0289	0.7584		
	0.1000	0.1000	0.0945	0.0010	0.1432		
	0.1500	0.1500	0.1400	0.0010	0.0528		
	0.2000	0.2000	0.2185	0.0010	0.0302		
	0.2500	0.2500	0.2660	0.0010	0.0266		
	0.3000	0.3000	0.3280	0.0010	0.0237		

Min selection rule, the FPRs are slightly higher for the testing set than for the validation set.

If we consider the FNR associated to each tolerable FPR, we can observe that, for mean and minimum distances and off-the-shelf packers, sound results are obtained for a 0.10 Max. FPR (0.0640 FPR and 0.0586 FNR) and a 0.05 Max. FPR (0.0505 FPR and 0.0289 FNR) respectively. Nevertheless, if we consider the FNR regarding custom packers, those configurations do not provide acceptable results (0.8751 and 0.7584 FNR respectively). Furthermore, in the case of Mean distance selection rule, none of the configurations achieves good results for the detection of custom packed binaries. Besides, the Min selection rule achieves acceptable results if a higher false positive rate is tolerated (for 0.1400 FPR, a 0.0010 FNR for off-the-shelf packers and a 0.0538 FNR for custom packers are achieved).

Finally, for the Max distance selection rule, we can observe that the results are inferior to the other configurations. Similarly to the case of mean distance, none of the configurations achieves sound results regarding custom packers, while the results for off-the-shelf packers are inferior to the results obtained for mean or minimum distances (0.9040 AUC, and a 0.1115

FPR for a 0.1165 FNR with the best configuration).

Table 4.2: Results obtained for PE based features and Manhattan distance.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC(P)	AUC
Mean	0.0100	0.0056	0.0040	0.8182	0.9373	0.9782	0.8147
	0.0200	0.0167	0.0110	0.4837	0.9357		
	0.0500	0.0500	0.0305	0.2810	0.9220		
	0.1000	0.1000	0.0655	0.0398	0.8911		
	0.1500	0.1500	0.0940	0.0031	0.8623		
	0.2000	0.2000	0.1445	0.0010	0.8335		
	0.2500	0.2500	0.2170	0.0010	0.7930		
	0.3000	0.3000	0.2770	0.0010	0.7522		
Max	0.0100	0.0056	0.0040	0.7881	0.9371	0.9335	0.7674
	0.0200	0.0167	0.0100	0.6710	0.9367		
	0.0500	0.0500	0.0395	0.3728	0.9186		
	0.1000	0.1000	0.0655	0.2795	0.9035		
	0.1500	0.1500	0.1035	0.2024	0.8785		
	0.2000	0.2000	0.1540	0.1310	0.8580		
	0.2500	0.2500	0.2130	0.0910	0.8385		
	0.3000	0.3000	0.2830	0.0674	0.7930		
Min	0.0100	0.0056	0.0045	0.8008	0.9373	0.9846	0.9596
	0.0200	0.0167	0.0100	0.5351	0.9225		
	0.0500	0.0500	0.0550	0.0158	0.4081		
	0.1000	0.1000	0.0935	0.0010	0.1019		
	0.1500	0.1500	0.1400	0.0010	0.0397		
	0.2000	0.2000	0.2085	0.0010	0.0276		
	0.2500	0.2500	0.2690	0.0010	0.0238		
	0.3000	0.3000	0.3530	0.0010	0.0164		

When Manhattan distance is employed, we can observe that the results follow the same tendency as Euclidean distance. In general, the results are slightly better for this distance measure in all cases, except for Max distance selection rule when custom packers are considered. If we observe the AUC, the best results are obtained for mean distance and minimum distance (0.9782 and 0.9846 for off-the-shelf packers, and 0.8147 and 0.9596 when custom packers are included). As in the case of Euclidean distance, the only configuration that provides sound results with custom packers is the Min distance selection rule.

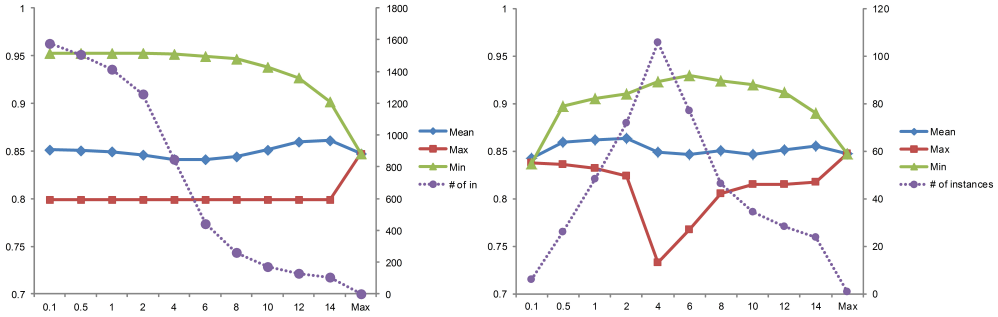
Regarding the FPR obtained for the testing set, we can observe that, for all the thresholds established for mean and max selection rules, the obtained values are below the tolerable FPR. Besides, for minimum distance, the FPRs are in some cases slightly higher than the tolerable limits.

If we observe the results and analyse the tradeoff between FPR and FNR, we can observe that the best configurations when only off-the-shelf packers are considered are slightly sounder than their Euclidean-based equivalents (0.0655 FPR with a 0.0398 FNR for mean distance, and 0.0550 FPR with

a 0.0158 FNR for minimum distance). Considering custom packers, again, the only configuration with sound results is Min distance selection rule. In this case, if a 0.0935 FPR is accepted, 0.0010 and 0.1019 FNRs are obtained. If a higher FPR is tolerated (0.14), lower FNRs are achieved: 0.0010 for off-the-shelf packers and 0.0397 for custom packed files.

4.3.2.2 **Results with dataset reduction**

Figures 4.2 and 4.3 plot the number of instances resultant of the reduction process together with the AUC obtained for each configuration.



(a) Outlier points not discarded. **(b)** Outlier points discarded.

Figure 4.2: AUC obtained for PE based features, Euclidean distance and the 3 selection rules, when different data reduction thresholds are applied.

Figure 4.2 shows the AUC results obtained when different reduction thresholds are applied to Euclidean distance, for the 3 distance selection rules.

Regarding Euclidean distance we can observe in Figure 4.2a the evolution of the performance when outlier points are not discarded. We can notice that when the maximum distance is measured, the results remain stable due to the presence of outliers until one unique centroid is calculated for the whole training set. Besides, the mean distance shows a slight degradation when higher thresholds are employed until the number of instances is below 250, that presents an improvement. Finally, regarding the best-performing distance selection rule (minimum distance), we can observe that the results degrade when the data is reduced. Nevertheless, we can select an ideal configuration evaluating the tradeoff between the degradation of the results and the number of instances used for comparison. In this way, for the appropriate threshold value (4.00), we obtained the results detailed in Table 4.3.

Table 4.3: Results obtained for PE based features, minimum distance selection rule and Euclidean distance, with the training set reduced with 4.00 threshold and outliers not discarded.

Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
0.0100	0.0056	0.0045	0.8030	0.9371	0.9517
0.0200	0.0167	0.0110	0.5764	0.9193	
0.0500	0.0500	0.0510	0.0289	0.7504	
0.1000	0.1000	0.0945	0.0010	0.1444	
0.1500	0.1500	0.1285	0.0010	0.0680	
0.2000	0.2000	0.2100	0.0010	0.0332	
0.2500	0.2500	0.2645	0.0010	0.0258	
0.3000	0.3000	0.3210	0.0010	0.0219	

For this configuration, the AUC obtained is 0.9517. If we observe the trade-off between the tolerable FPR and its associated FNR, the results are very similar to those obtained when no reduction was applied. In fact, if we include custom packers, we must assume a 0.1285 FPR to obtain an acceptable packed binary detection rate: 0.0010 FNR for off-the-shelf packers and a 0.0680 FNR for custom packers.

Besides, Figure 4.2b shows the evolution of the AUC when outlier points are discarded during the clustering process. In this case, we can observe that the number of instances for comparison grows when a higher threshold is applied (i.e., a higher number of clusters are formed). Nevertheless, when the threshold surpasses certain value (in this case, 4.00), the number of instances decreases, (a lower number of clusters which include a higher number of instances are formed). The results obtained for the different distance selection rules also vary with the reduction threshold applied. In this way, we can notice that the maximum distance presents poor results when the number of instances used for comparison is higher. This behaviour might be caused by the influence of outliers on this distance selection rule. When mean distance selection rule is applied, the results are considerably stable, achieving the best results for low reduction thresholds (2.00). As in the previous case, the minimum distance presents the best results. In this case, the results improve when the number of instances used for comparison is higher. Nevertheless, the best result is achieved for a 6.00 threshold, when the number of instances starts to decrease.

Table 4.4 shows the results for the best configuration: minimum distance and a 6.00 reduction threshold.

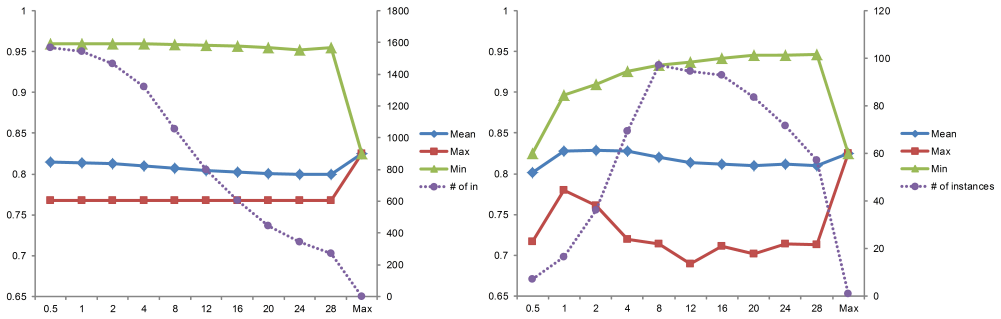
In this case, we can appreciate that the AUC obtained degrades with respect to the best configuration obtained when outliers are not discarded.

Table 4.4: Results obtained for PE based features, minimum distance selection rule and Euclidean distance, with the training set reduced with 6.00 threshold and outliers discarded.

Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
0.0100	0.0056	0.0040	0.8012	0.9371	0.9298
0.0200	0.0167	0.0110	0.5663	0.9240	
0.0500	0.0500	0.0340	0.3684	0.9073	
0.1000	0.1000	0.0670	0.0115	0.8435	
0.1500	0.1500	0.1090	0.0010	0.5117	
0.2000	0.2000	0.1765	0.0010	0.0514	
0.2500	0.2500	0.2355	0.0010	0.0320	
0.3000	0.3000	0.3225	0.0010	0.0211	

Regarding the trade-off between FPR and FNR, the best configuration needs a 0.1765 FPR to achieve an acceptable FNR for custom packers (0.0514), while for off-the-shelf packers, sound results are obtained (0.0010 FNR).

We can appreciate that the results present a slight improvement for higher thresholds, obtaining higher AUCs for an equivalent number of samples employed to build the model. This tendency might be produced by the noise reduction capabilities of the data reduction approach employed. As the outlier samples not included in any cluster are discarded, the possible negative effects of these samples are smoothed. For higher thresholds, some of the instances considered outliers for lower thresholds are included in the clusters. First, the effects of possible noise are smoothed. Second, the number of instances discarded is reduced, ensuring that a higher number of samples are represented in the final model.



(a) Outlier points not discarded. **(b)** Outlier points discarded.

Figure 4.3: AUC obtained for PE based features, Manhattan distance and the 3 selection rules, when different data reduction thresholds are applied.

Figure 4.3 shows the evolution for Manhattan distance, when different

reduction thresholds are applied. More concretely, Figure 4.3a shows the evolution when outlier points are not discarded. In this case, when the reduction threshold increases and the number of instances for comparison decreases, the 3 configurations show stable results. We can notice that minimum distance and mean distance show a slight performance degradation when the number of instances decreases. When maximum distance is considered, the results remain stable. The best configuration is minimum distance with 8.00 threshold.

Table 4.5: Results obtained for PE based features, minimum distance selection rule and Manhattan distance, with the training set reduced with 8.00 threshold and outliers not discarded.

Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
0.0100	0.0056	0.0040	0.8010	0.9373	0.9591
0.0200	0.0167	0.0100	0.5351	0.9225	
0.0500	0.0500	0.0550	0.0158	0.4038	
0.1000	0.1000	0.0940	0.0010	0.1017	
0.1500	0.1500	0.1390	0.0010	0.0404	
0.2000	0.2000	0.2110	0.0010	0.0277	
0.2500	0.2500	0.2810	0.0010	0.0234	
0.3000	0.3000	0.3375	0.0010	0.0199	

Table 4.5 shows the best configuration for the Min distance selection rule. In this case, the AUC achieved is 0.9591. Regarding the tradeoff between the FPR and the FNR, we can observe that if we tolerate a 0.0940 FPR, we obtain a 0.0010 FNR for off-the-shelf packers, and a 0.1017 FNR for custom packers. Nevertheless, if a higher FPR is tolerated (0.1390), a 0.0010 and a 0.0404 FNR are obtained.

In this case, for environments in which efficiency is an important aspect to consider, the expert might select a higher threshold, given that the results are not notably affected even when the number of instances is decreased to 273 (0.955).

In Figure 4.3b we can observe that, when the outliers are discarded, the results for the mean distance are considerably stable. For the maximum distance the results degrade for a higher number of instances, and for the minimum distance the results are better for higher thresholds. In this case we can see a tendency similar to the effect observed for Euclidean distance. When higher thresholds are applied, the results slightly improve. Again, this tendency might be caused by the noise reduction capabilities of the algorithm.

Table 4.6 shows the results for the minimum distance and a 28.00 thresh-

Table 4.6: Results obtained for PE based features, minimum distance selection rule and Manhattan distance, with the training set reduced with 28.00 threshold and outliers discarded.

Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
0.0100	0.0056	0.0040	0.8033	0.9373	0.9468
0.0200	0.0167	0.0130	0.4268	0.9297	
0.0500	0.0500	0.0395	0.0355	0.8590	
0.1000	0.1000	0.0685	0.0010	0.7334	
0.1500	0.1500	0.1205	0.0010	0.1126	
0.2000	0.2000	0.1845	0.0010	0.0309	
0.2500	0.2500	0.2525	0.0010	0.0229	
0.3000	0.3000	0.3105	0.0010	0.0189	

old. This configuration produces the highest AUC with 57 instances during comparison, a number considerably below the maximum number of instances tested (97). In this case, the AUC is inferior to the configuration with no outliers discarded: 0.9468. As a result, the best threshold configuration must assume a 0.1205 FPR to produce an acceptable FNR regarding custom packers (0.1126). If we would desire to provide a sound custom packer detection rate (0.0309), a 0.1845 FPR would have to be tolerated. In both cases, a sound FNR is obtained for off-the-shelf packers (0.0010).

4.3.3 Evaluation for operational code frequency

Given the results described in Section 4.3.2, we can observe that (i) the results are very sensitive to outliers, and (ii), the only distance selection rule that produces sound results when custom packers are considered is the minimum distance (i.e., distance to the nearest instance).

In order to test the appropriateness of anomaly detection for the classification of packed binaries, and to give a stronger support to the approach proposed, we have evaluated our method with a different feature-set based on operational code frequency, adapting a previous n-gram based approach [PLL08b] to anomaly detection based classification.

4.3.3.1 Description of the feature-set

Several approaches address malware detection using n-gram frequency and machine-learning methods [SPDB09, SEZS01, KM04].

In this way, we define an n-gram as a continuous sequence of elements for a given stream. In the case of a byte stream, an n-gram model will be

represented as all the possible byte sequences of length n in that stream. For instance, given the following byte stream:

```
8B DE AD AD 50 AD 97 B2 80 A4 B6 80 FF 13 73 F9 33 C9 FF 13 73 16 33 C0
```

The set of n-grams for $n = 1$ would be $\mathcal{N}_{n=1} = \{8B, DE, AD, \ldots\}$, for $n = 2$ it would be $\mathcal{N}_{n=2} = \{8B\ DE, DE\ AD, AD\ AD, \ldots\}$, for $n = 3$ it would be $\mathcal{N}_{n=3} = \{8B\ DE\ AD, DE\ AD\ AD, AD\ AD\ 50, \ldots\}$, and so on.

Perdisci et al. [PLL08b] employed this technique not only for the task of discriminating malware from goodware, but also to discriminate between packed and non-packed binaries. In fact, when compression or encryption algorithms are applied, it is common to observe high entropy values and flat byte histograms [Sun12]. Not packed files, in contrast, present a different byte frequency. One of the reasons for this differences is the prevalence of certain operational codes. Following this idea, Perdisci et al. proposed an approach based on measuring the presence of certain n-grams for the classification of packed and non-packed binaries. In order to select the most relevant n-grams or byte combinations, they employed Information Gain (IG) over a set of packed and non-packed instances. In this way, they selected the byte sequences that best discriminate both classes.

Unfortunately, IG cannot be calculated if only one of the classes is taken into consideration. Our anomaly detection method avoids taking any assumption about the packed executable class in order to construct a packer-agnostic classification system.

For this reason, we adapt the approaches proposed in previous work for anomaly detection. To this aim, we elaborate a list of all possible operational codes according to the Intel specification for the x86 architecture [Int13], and measure the frequency for each possible byte sequence.

The operational codes defined by Intel have a variable length. In fact, the instruction size for this architecture varies from 1 byte to 15 bytes.

For this study, some fields have been excluded given that they do not adapt correctly to a simple n-gram model, or because they introduce too much complexity deriving an exponential number of possible different byte sequences. For this reason, floating point instructions have not been included. Besides, some instructions use bits 3-5 of the ModR/M byte to encode the instructions. In this way, there are byte sequences that represent a group of instructions that are discriminated by these bits. We have omitted the ModR/M byte assuming that some byte sequences represent, in fact, a group of similar instructions. Accordingly, we define a set of 823 byte sequences formed by the 2 first fields in the instruction format shown

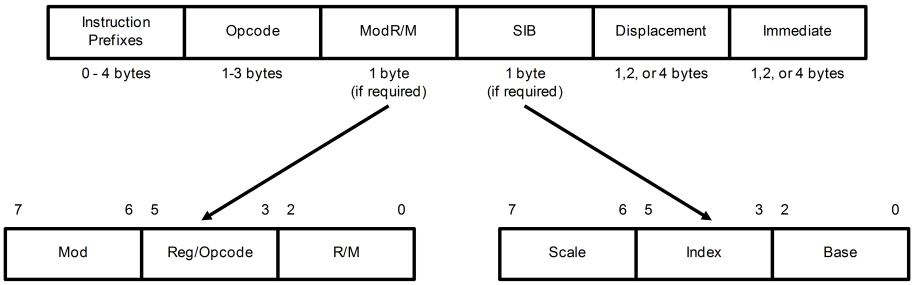


Figure 4.4: Intel 64 and IA 32 instruction format, adapted from the Intel 64 and IA 32 Architectures Software Developer's Manual.

in Figure 4.4, with a length varying from 1 to 5 bytes. This list of operational codes was gathered from Tables A-2 to A-5 of the Appendix A, Volume 2C of the Intel 64 and IA-32 Architectures Software Developer's Manual [Int13]. This approach, in contrast to a basic n-gram model, only considers the subset of n-grams for $n = 1$, $n = 2$, $n = 3$, $n = 4$ and $n = 5$ that represent an Intel instruction. Consequently, the set of possible values is reduced from 1,103,823,438,080 to 823.

Given this background, we define our feature-set as a set of tuples such that, for a given binary b, $SF_b = \{(s_{1,b}, f_{1,b}), (s_{2,b}, f_{2,b}), (s_{3,b}, f_{3,b}), \dots (s_{n,b}, f_{n,b})\}$, where $s_{i,b}$ represents every possible sequence, and $f_{i,b}$ is the frequency of that sequence in the file. More concretely, $f_{i,b}$ is the total number of occurrences of the sequence divided by the total number of sequences found for that binary.

4.3.3.2 **Results without dataset reduction**

First, we can observe the results obtained when no reduction is applied. Table 4.7 shows the results for Euclidean distance. In this case, both the Mean and Min distance selection rules show sound results regarding AUC. Max distance, on the contrary, presents an unacceptable AUC and thus we discard this distance selection rule. Regarding the tradeoff between FPR and FNR, we can notice that both configurations must assume a considerably high FPR to achieve a sound FNR. More specifically, when mean distance is compared, a 0.1635 FPR must be tolerated to achieve a 0.0402 and a 0.2117 FNR for off-the-shelf packers and custom packed files. If a higher number of FPR is tolerated (0.2210), then the results obtained for packed files are 0.0263 and 0.1105 FNRs.

In the case of minimum distance, the results show the same tendency. For a 0.1315 FPR, 0.0671 and 0.2627 FNR are obtained. Besides, if we

Table 4.7: Results obtained for operational code frequency and Euclidean distance.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0220	0.7890	0.8010	0.9136
	0.0200	0.0167	0.0300	0.6945	0.7311	
	0.0500	0.0500	0.0580	0.5245	0.6014	
	0.1000	0.1000	0.1010	0.2010	0.4296	
	0.1500	0.1500	0.1400	0.0557	0.2685	
	0.2000	0.2000	0.1635	0.0402	0.2117	
	0.2500	0.2500	0.2210	0.0263	0.1105	
	0.3000	0.3000	0.2935	0.0152	0.0148	
Max	0.0100	0.0056	0.0220	0.8008	0.8045	0.5318
	0.0200	0.0167	0.0295	0.7196	0.7435	
	0.0500	0.0500	0.0555	0.6128	0.6352	
	0.1000	0.1000	0.0980	0.5290	0.5589	
	0.1500	0.1500	0.1555	0.4956	0.5354	
	0.2000	0.2000	0.2215	0.4835	0.5226	
	0.2500	0.2500	0.2650	0.4789	0.5150	
	0.3000	0.3000	0.3115	0.4716	0.5071	
Min	0.0100	0.0056	0.0215	0.8099	0.8085	0.9206
	0.0200	0.0167	0.0295	0.6943	0.7273	
	0.0500	0.0500	0.0525	0.5318	0.6075	
	0.1000	0.1000	0.0945	0.1716	0.4007	
	0.1500	0.1500	0.1315	0.0671	0.2627	
	0.2000	0.2000	0.1890	0.0228	0.1028	
	0.2500	0.2500	0.2675	0.0142	0.0118	
	0.3000	0.3000	0.3505	0.0110	0.0071	

tolerate a higher FPR (0.1890), the results obtained are 0.0228 and 0.1028 for off-the-shelf packers an custom packed binaries.

Additionally, we can observe that, for mean and minimum distances, the FNR for custom packed files is higher than for off-the-shelf packers, and thus these kind of packers are more difficult to detect. For the maximum distance, however, this effect is not present.

Regarding Manhattan distance (see Table 4.8), the results obtained are sounder than the Euclidean distance based configurations. In this case, when mean and minimum distances are applied, the AUC obtained reaches 0.9522 and 0.9574 respectively. For the maximum distance the results are not acceptable (0.2788 AUC), and thus we must discard the configuration.

If we analyse the tradeoff between FPR and FNR, we can observe that, for mean distance, sound results are obtained when a 0.0885 FPR is assumed, obtaining 0.0487 and 0.0518 FNRs for off-the-shelf packers and custom packed binaries. If a higher number of false positives is tolerated (0.1535), then the results obtained are 0.0237 and 0.0110 respectively. Besides, if the Min distance selection rule is applied, a 0.1165 FPR achieves

Table 4.8: Results obtained for operational code frequency and Manhattan distance.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0145	0.8542	0.8801	0.9522
	0.0200	0.0167	0.0285	0.6299	0.7975	
	0.0500	0.0500	0.0530	0.1237	0.3735	
	0.1000	0.1000	0.0885	0.0487	0.0518	
	0.1500	0.1500	0.1535	0.0237	0.0110	
	0.2000	0.2000	0.1975	0.0200	0.0108	
	0.2500	0.2500	0.2190	0.0162	0.0097	
	0.3000	0.3000	0.2810	0.0105	0.0073	
Max	0.0100	0.0056	0.0065	0.9547	0.9279	0.2788
	0.0200	0.0167	0.0180	0.9171	0.8672	
	0.0500	0.0500	0.0405	0.8749	0.7916	
	0.1000	0.1000	0.1185	0.8415	0.7519	
	0.1500	0.1500	0.1665	0.8310	0.7348	
	0.2000	0.2000	0.2115	0.8219	0.7201	
	0.2500	0.2500	0.2665	0.8104	0.7104	
	0.3000	0.3000	0.3275	0.8047	0.7037	
Min	0.0100	0.0056	0.0120	0.8677	0.8305	0.9574
	0.0200	0.0167	0.0250	0.6231	0.6759	
	0.0500	0.0500	0.0480	0.2076	0.3710	
	0.1000	0.1000	0.1165	0.0247	0.0099	
	0.1500	0.1500	0.1790	0.0151	0.0063	
	0.2000	0.2000	0.2410	0.0117	0.0056	
	0.2500	0.2500	0.2950	0.0090	0.0048	
	0.3000	0.3000	0.3530	0.0075	0.0006	

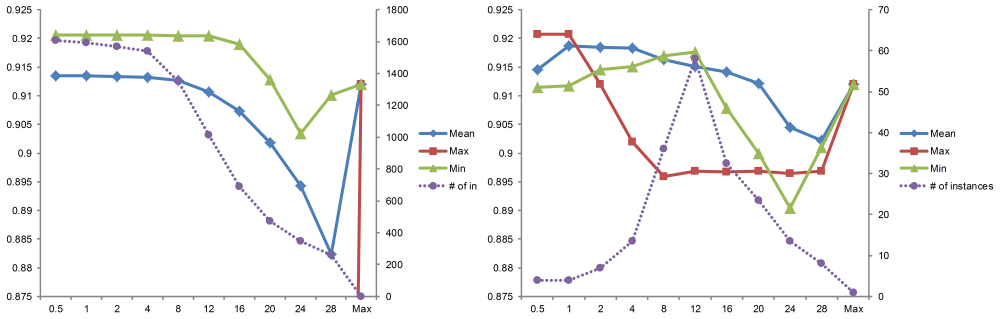
0.0247 and 0.0099 FNRs.

In both cases we can observe that the configurations based on the maximum distance are notably affected by this feature set, probably because of its instability in the presence of outliers. In contrast, the results do not present a notable difference regarding the FNR achieved for off-the-shelf and custom packed binaries, like in the case of the PE based feature set.

4.3.3.3 **Results with dataset reduction**

Figure 4.5 and Figure 4.6 plot the number of instances resultant from the reduction process together with the obtained AUC values for each configuration.

Figure 4.5a shows the evolution of the results when different reduction thresholds are applied with Euclidean distance, when outliers are not discarded. In this case, we can observe that the maximum distance presents results below the acceptable limits. Both mean and minimum distances produce results that deteriorate when the number of instances decreases.



(a) Outlier points not discarded. **(b)** Outlier points discarded.

Figure 4.5: AUC obtained for operational code frequency, Euclidean distance and the 3 selection rules, when different data reduction thresholds are applied.

In the case of mean distance, sound AUC values are achieved for reduction thresholds below 8.00. For the minimum distance, we can apply a reduction threshold of 12.00 without observing a degradation in the results. In both cases, we can observe that the results tend to deteriorate when the threshold applied is increased, showing a tendency similar to the PE structure based feature set.

Accordingly, we show in Table 4.9 a detailed version of these configurations. We can observe that, for mean distance, we must tolerate a high FPR to obtain sound FNR results (0.2215 FPR for a 0.0264 and 0.1139 FNR for manually packed and custom packed files respectively). Regarding minimum distance, the results slightly improve requiring a 0.1895 FPR to obtain a 0.0241 and 0.1078 false negative rates.

If we apply reduction to the Euclidean distance and discard the outliers during the process, we obtain the results shown in Figure 4.5b. In this case, the results for mean distance present a soft degradation as the reduction threshold increases. Besides, the minimum distance presents an improvement until the maximum number of instances for comparison is achieved for the 12.00 threshold. For higher thresholds, it presents a considerable degradation. The maximum distance, despite of the unacceptable results when outliers are not discarded (see Figure 4.5a), presents sound results for the lowest thresholds, and a degradation when a higher number of clusters is generated (showing a considerable sensitiveness to the instances used for comparison).

Table 4.10 shows the detailed results for the selected configurations. A

Table 4.9: Results obtained for operational code frequency and Euclidean distance, outliers not discarded, with the training set reduced with a 4.00 threshold for Mean distance selection rule, and a 12.00 threshold for Min distance selection rule.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0220	0.7890	0.8010	0.9132
	0.0200	0.0167	0.0300	0.6945	0.7311	
	0.0500	0.0500	0.0580	0.5246	0.6014	
	0.1000	0.1000	0.1010	0.2027	0.4298	
	0.1500	0.1500	0.1395	0.0563	0.2702	
	0.2000	0.2000	0.1635	0.0401	0.2129	
	0.2500	0.2500	0.2215	0.0264	0.1139	
	0.3000	0.3000	0.2935	0.0152	0.0151	
Min	0.0100	0.0056	0.0215	0.8099	0.8085	0.9205
	0.0200	0.0167	0.0295	0.6948	0.7274	
	0.0500	0.0500	0.0525	0.5318	0.6074	
	0.1000	0.1000	0.0955	0.1714	0.4011	
	0.1500	0.1500	0.1330	0.0667	0.2628	
	0.2000	0.2000	0.1895	0.0241	0.1078	
	0.2500	0.2500	0.2735	0.0139	0.0122	
	0.3000	0.3000	0.3450	0.0110	0.0076	

4.00 threshold for Mean distance selection rule obtains a 0.9184 AUC. In the case of Max distance, only the lowest thresholds (e.g. 1.00) that discard most of the samples produce sound results. Finally, in the case of minimum distance, we can appreciate that the results depend on the number of instances resulting of the clustering process. The configuration producing the highest number of samples obtained the best AUC (12.00 threshold). All the distance selection rules require high FPRs to achieve sound FNR results for custom packers. More concretely, for mean distance, a 0.2190 FPR is required to obtain a 0.0243 FNR for off-the-self packers and a 0.0508 FNR for custom packed files. Similarly, the maximum distance requires a 0.2270 FPR to achieve 0.0239 and 0.0219 FNRs. For the minimum distance, a 0.2550 FPR is necessary to obtain FNRs below 0.02 for both manually and custom packed files.

Figure 4.6a shows the results obtained for different reduction thresholds with Manhattan distance, when outlier points are not discarded in the clustering process. First, we must discard the Max distance selection rule for its unsound results. When mean and minimum distances are applied, the results are considerably stable, specially for the minimum distance, which outperforms mean distance for all the possible thresholds. These 2 distances, in addition, present a considerable degradation for high thresholds, when the number of instances for comparison are significantly

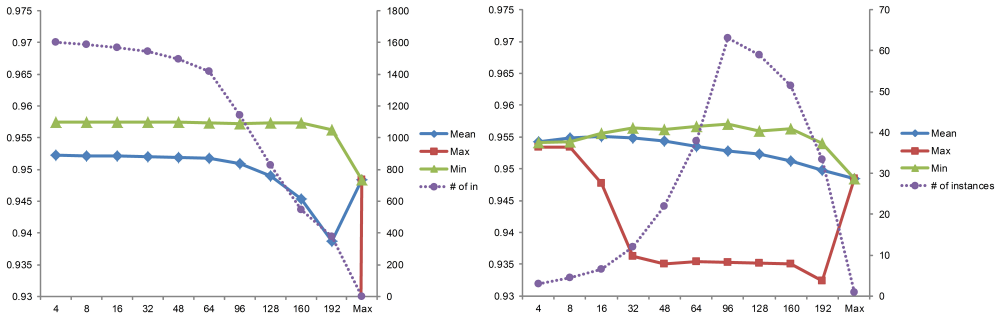
Table 4.10: Results obtained for operational code frequency and Euclidean distance, outliers discarded, with the training set reduced with a 4.00 threshold for Mean distance selection rule, a 1.00 threshold for Max distance selection rule, and a 12.00 threshold for Min distance selection rule.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0220	0.7882	0.8008	0.9184
	0.0200	0.0167	0.0305	0.6914	0.7293	
	0.0500	0.0500	0.0570	0.5209	0.6017	
	0.1000	0.1000	0.1040	0.1385	0.3944	
	0.1500	0.1500	0.1400	0.0487	0.2398	
	0.2000	0.2000	0.1625	0.0374	0.1988	
	0.2500	0.2500	0.2190	0.0243	0.0508	
	0.3000	0.3000	0.2845	0.0156	0.0123	
Max	0.0100	0.0056	0.0220	0.7877	0.8006	0.9208
	0.0200	0.0167	0.0300	0.6909	0.7290	
	0.0500	0.0500	0.0570	0.5183	0.6028	
	0.1000	0.1000	0.1000	0.1332	0.3924	
	0.1500	0.1500	0.1420	0.0447	0.2171	
	0.2000	0.2000	0.1605	0.0369	0.1867	
	0.2500	0.2500	0.2270	0.0230	0.0219	
	0.3000	0.3000	0.2965	0.0140	0.0108	
Min	0.0100	0.0056	0.0220	0.7872	0.8003	0.9176
	0.0200	0.0167	0.0315	0.6862	0.7257	
	0.0500	0.0500	0.0585	0.5285	0.6039	
	0.1000	0.1000	0.1040	0.1582	0.4038	
	0.1500	0.1500	0.1375	0.0523	0.2560	
	0.2000	0.2000	0.1710	0.0353	0.1919	
	0.2500	0.2500	0.2550	0.0181	0.0164	
	0.3000	0.3000	0.3180	0.0134	0.0099	

reduced. Considering the tendency of the results, we select the highest possible threshold (maximum reduction ratio) that does not degrade considerably the AUC: 96.00 (1140 instances) for mean distance and 160.00 (540 instances) for minimum distance. The results for the best threshold configurations are detailed in Table 4.11.

For mean distance, the 96.00 threshold produces a 0.9509 AUC. In this case, a balanced tradeoff between FPR and FNR can be obtained. For a 0.0870 FPR, a 0.0517 FNR for manually packed files, and a 0.0858 FNR for custom packed files are achieved. For a sound detection of packed files (0.0251 and 0.0110 FNRs respectively), a 0.1495 FPR must be tolerated. In the case of minimum distance, the AUC obtained is slightly higher (0.9574). Regarding the tradeoff between FPR and FNR, a lower FPR (0.1025) achieves sound packer detection results (0.0314 and 0.0131 FNRs respectively).

Finally, Figure 4.6b shows the evolution of the results for Manhattan



(a) Outlier points not discarded. **(b)** Outlier points discarded.

Figure 4.6: AUC obtained for operational code frequency, Manhattan distance and the 3 selection rules, when different data reduction thresholds are applied.

Table 4.11: Results obtained for operational code frequency and Manhattan distance, outliers not discarded, with the training set reduced with a 96.00 threshold for Mean distance selection rule, and a 160.00 threshold for Min distance selection rule.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0145	0.8535	0.8770	0.9509
	0.0200	0.0167	0.0270	0.6708	0.8084	
	0.0500	0.0500	0.0535	0.1313	0.3934	
	0.1000	0.1000	0.0870	0.0517	0.0858	
	0.1500	0.1500	0.1495	0.0251	0.0110	
	0.2000	0.2000	0.1950	0.0210	0.0108	
	0.2500	0.2500	0.2200	0.0186	0.0106	
	0.3000	0.3000	0.2860	0.0113	0.0075	
Min	0.0100	0.0056	0.0115	0.8803	0.8427	0.9574
	0.0200	0.0167	0.0245	0.6533	0.6866	
	0.0500	0.0500	0.0450	0.2282	0.3997	
	0.1000	0.1000	0.1025	0.0314	0.0131	
	0.1500	0.1500	0.1775	0.0158	0.0076	
	0.2000	0.2000	0.2455	0.0100	0.0063	
	0.2500	0.2500	0.3125	0.0077	0.0044	
	0.3000	0.3000	0.3575	0.0058	0.0020	

distance when outliers are discarded. On the one hand, when the minimum distance is considered, the results show a slight performance improvement as the number of generated clusters increases with higher thresholds, and then deteriorates when the highest thresholds are applied, reaching the best results for a 96.00 threshold. In the case of the mean distance, the results show an stable deterioration when the applied threshold increases, obtaining sound results for a 16.00 threshold. Maximum distance, like in

the case of Euclidean distance, shows results sensitive to the variability of the instances generated after the clustering process, but shows interesting results for the lowest reduction thresholds (8.00).

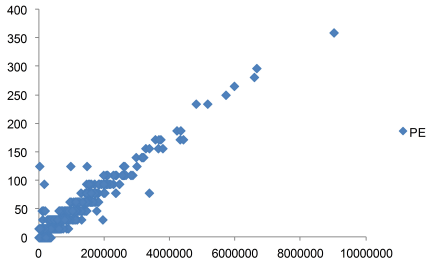
Table 4.12: Results obtained for operational code frequency and Manhattan distance, outliers discarded, with the training set reduced with a 16.00 threshold for Mean distance selection rule, a 8.00 threshold for Max distance selection rule, and a 96.00 threshold for Min distance selection rule.

Sel. rule	Max FPR	Tr. FPR	FPR	FNR(P)	FNR(CP)	AUC
Mean	0.0100	0.0056	0.0135	0.8563	0.8874	0.9550
	0.0200	0.0167	0.0325	0.4802	0.7422	
	0.0500	0.0500	0.0570	0.0985	0.3021	
	0.1000	0.1000	0.0945	0.0405	0.0124	
	0.1500	0.1500	0.1530	0.0228	0.0109	
	0.2000	0.2000	0.1990	0.0173	0.0091	
	0.2500	0.2500	0.2300	0.0128	0.0075	
	0.3000	0.3000	0.2770	0.0097	0.0060	
Max	0.0100	0.0056	0.0135	0.8579	0.8917	0.9534
	0.0200	0.0167	0.0305	0.5343	0.7787	
	0.0500	0.0500	0.0560	0.1090	0.3323	
	0.1000	0.1000	0.0895	0.0498	0.0350	
	0.1500	0.1500	0.1540	0.0236	0.0110	
	0.2000	0.2000	0.1940	0.0221	0.0097	
	0.2500	0.2500	0.2395	0.0145	0.0072	
	0.3000	0.3000	0.2765	0.0113	0.0070	
Min	0.0100	0.0056	0.0140	0.8550	0.8802	0.9570
	0.0200	0.0167	0.0325	0.4936	0.7374	
	0.0500	0.0500	0.0575	0.0856	0.2373	
	0.1000	0.1000	0.1180	0.0267	0.0110	
	0.1500	0.1500	0.1630	0.0155	0.0084	
	0.2000	0.2000	0.2160	0.0115	0.0076	
	0.2500	0.2500	0.2575	0.0070	0.0060	
	0.3000	0.3000	0.3035	0.0042	0.0060	

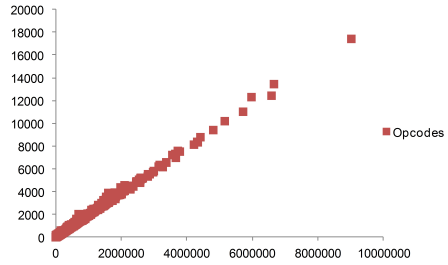
Table 4.12 shows results for the selected threshold configurations. When Mean distance selection rule is applied, a 0.9550 AUC is achieved. For this configuration, a 0.0945 FPR has to be tolerated to obtain a 0.0405 FNR for off-the-shelf packers and a 0.0124 FNR for custom packed files. For the maximum distance a 0.9534 AUC is achieved. For this configuration, we can also find a sound tradeoff between FPR and FNR (0.0895 FPR for a 0.0498 FNR and a 0.0350 FNR respectively). Finally, for the minimum distance, a 0.9570 AUC is achieved. In this case, a 0.1180 FPR achieves a 0.0267 FNR for manually packed files and a 0.0110 FNR for custom packed files.

4.3.4 Evaluation of the efficiency

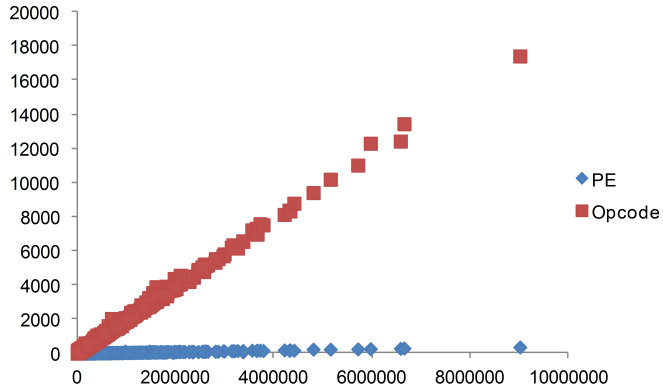
The feature extraction was performed in an isolated virtual machine using VMWare. The host machine was an Intel Core i5 650 clocked at 3.20 GHz and 16 GB of RAM memory, while the guest machine was configured with 2 processors, 4 GB of RAM memory and Windows XP SP3 as operating system.



(a) PE based features.



(b) Operational code frequency.



(c) Both feature-sets.

Figure 4.7: Feature extraction times for the different feature-sets. The X axis represents the file size, while the Y axis represents the time required to extract the features, expressed in milliseconds.

Figure 4.7 summarises the feature extraction process. We can observe that, on the one hand, the feature extraction time is directly proportional to the file size in both feature-sets. In the case of the feature-set based on the Portable Executable structure, this effect is caused by the values that depend on the full content of the file, like entropy. On the other hand, we can observe in Figure 4.7c that the extraction of operational code frequency is more time-consuming than the extraction of PE based features. In this way, the average comparison time is 0.0576 ms/KB for PE based features

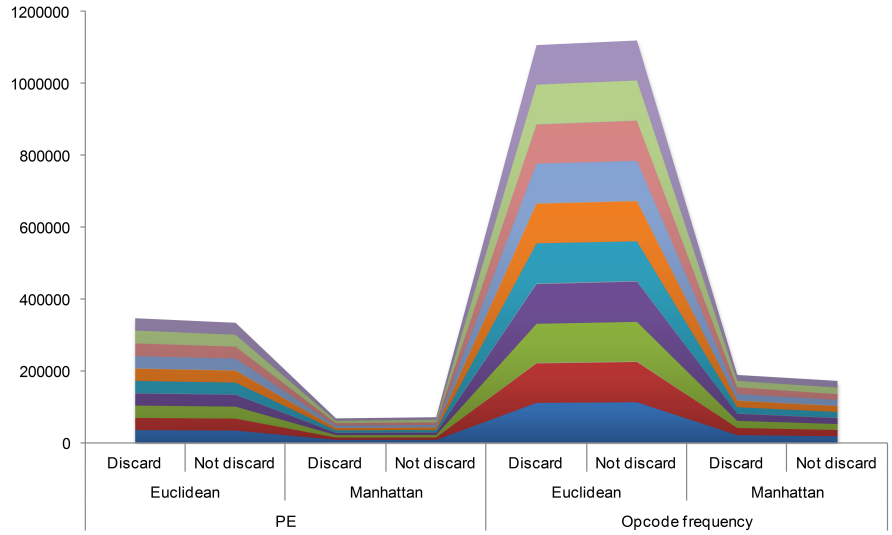
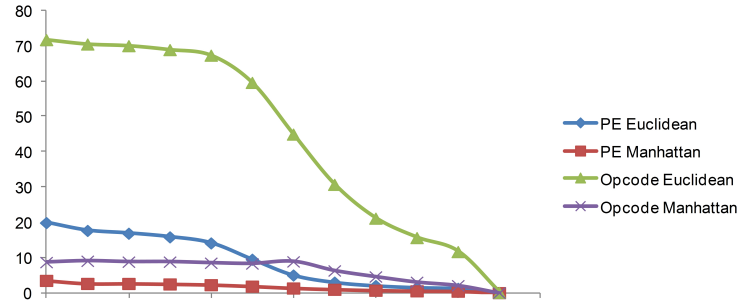


Figure 4.8: Reduction times for the different configurations. The X axis represents the different experiment configurations, and the Y axis represents the reduction time, expressed in milliseconds. Each of the 10 folds evaluated in the experiment is represented in a different color.

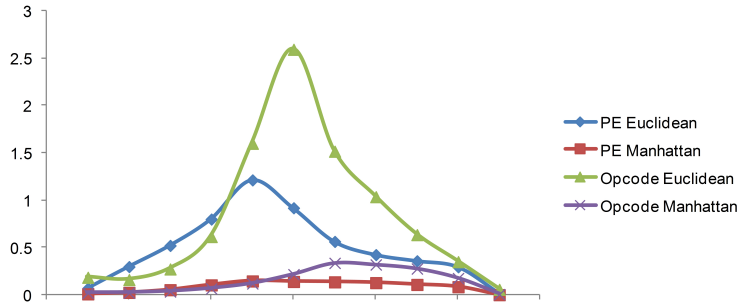
and 2.0188 ms/KB for operational code frequency. Although the extraction time in both cases depends on the implementation of our feature extraction algorithm, we can safely conclude that the computation of operational code frequency requires more computational resources than PE based features.

Once the features have been extracted, the instances are randomised and divided into 10 folds following the distribution described in Section 4.3.1. Then, the normalisation process is applied to the 1620 instances that form the training set. The μ and σ parameters used to calculate the z-score for each feature are then used to normalise the validation-set and the test-set. The normalisation process of each fold consumes on average 148.4375 milliseconds with an standard deviation of 62.1736 for the PE based feature-set, and 735.9375 with an standard deviation of 121.1315 for operational code frequency.

Figure 4.8 shows the time required by the dataset reduction algorithm for each configuration. First, we can notice that Euclidean distance, in both cases, presents higher reduction times. This difference is caused by the complexity of each distance measure. Besides, PE based feature-set requires, again, a lower processing time. This is caused by the number of possible features present in each feature-set. While the PE based feature-



(a) Outliers not discarded.



(b) Outliers discarded.

Figure 4.9: Distance computation times averaged for each possible configuration. The X axis represents the different thresholds applied. The threshold applied for reduction varies for each configuration, and thus, are not labelled on the axis. Nevertheless, the thresholds are plotted, in each case, in ascending order. The Y axis shows the average time (in milliseconds) to measure the distance from a testing instance to the model.

set is composed of 241 features, the feature-set composed of operational codes is represented by 823.

Finally, Figure 4.9 shows the average time required to measure the distance from a testing instance to the non-packed model. This time is directly proportional to the number of instances used for comparison in each case. In Figure 4.9a we can observe the results when outliers are not discarded. First, we can observe that PE based features require lower processing times than operational code frequency. Besides, Manhattan distance is more efficient than Euclidean distance. Figure 4.9b shows the average distance computation time when outliers are discarded. In this case, as the number of instances that represent the model is lower, the time required for compar-

ison decreases with respect to the previous approach. In these cases, the number of instances representing the model reaches its maximum value for the central thresholds. Once the threshold surpasses certain value, the clusters generated contain a higher number of instances and the number of clusters decreases. This effect has a direct effect on comparison time, that can be observed in Figure 4.9b.

4.4 Conclusions and discussion

The results presented in Section 4.3.2 and Section 4.3.3 show the performance of the anomaly detection method proposed for the classification of packed and non-packed binaries.

We have established several research questions for this study. First, we can notice that the results for PE based features and operational code frequency present differences. Therefore, we first address the last research question established:

Which is the feature-set that best discriminates packed from non-packed files?

The first observation is that the only distance selection rule that produces sound results for PE based features is the minimum distance, while in the case of operational code frequency, both mean and minimum distance are valid approaches. More concretely, we can observe that in the case of PE based features, the mean distance is not capable of correctly classifying custom packers.

Secondly, we can also observe that the FNR for custom packers is generally higher than the FNR for off-the-shelf packers when PE based features are compared. However, in the case of operational code frequency, this effect is not present. In fact, the results presented show that for most configurations, the operational code frequency present favourable results for both custom packers and manually packed files.

The third observation regarding the two feature-sets is that, while both present unreliable results for the Max distance selection rule, the results for operational code frequency are specially poor. We can notice that this feature-set is extremely sensitive to outlier points in the normality model.

From these observations we can conclude that our PE feature based anomaly detection method presents limitations to capture the difference

between non-packed binaries and custom packed files. Nevertheless, when the minimum distance is selected, our approach can provide sound custom packed file detection rates if it assumes a sufficient number of false positives. Besides, operational code frequency can correctly discriminate these files. This effect might be caused by the tendency of malware writers to implement custom made packers that try to evade heuristic detection as well as simple manual inspection using PE based characteristics of non-packed binaries.

Therefore, we can answer the research question established:

Despite the acceptable results obtained with PE based features for minimum distance, we can affirm that our anomaly detection method presents a better classification performance when operational code frequency is compared instead of PE based features.

If we examine the time required to extract, normalise, reduce data, and compute the distance between samples, we can observe that operational code frequency is not as efficient as PE based features. Nevertheless, depending on the efficiency requirements of the target system, it could be interesting to consider operational code frequency due to the results obtained.

What is the impact of the data-reduction approach over the results obtained?

In section 4.3.2 we show the performance of our anomaly detection system for different reduction thresholds. More concretely, we evaluate the AUC obtained for each configuration, considering that the false positive rate and the false negative rates depend on the distance threshold selected.

First, we can observe that in the case of PE based features, the results when the minimum distance is selected vary with the number of instances used for comparison. In both reduction approaches (outliers discarded and not discarded), the results present a degradation when the number of instances is lower.

Besides, when operational code frequency is compared, the Min distance selection rule shows a similar evolution. Nevertheless, when the mean distance is selected, the results tend to degrade as the reduction threshold increases, regardless of the number of instances used for comparison, except when a unique centroid is employed for comparison. This difference is mainly noticeable when the outliers are discarded from the comparison.

If we compare the two approaches for dataset reduction, we can observe that for PE based features, the results are affected by the method employed. In this case, the approach that does not discard the outliers presents sounder results, specially for Euclidean distance. In the case of operational code frequency, there is not a noticeable difference among both approaches.

In addition, if we observe the efficiency of both approaches, while there is no significant difference while reducing the dataset (there is no difference in the complexity of the calculations), we can observe that there is a significant difference when measuring the distance from a testing instance to the model, due to the number of instances used for comparison.

To answer the research question established, although all configurations are not affected equally by the data reduction process, we can conclude that, in all cases, it is possible to find a trade-off between efficiency and effectiveness, trying to minimise the processing time and the results obtained for the different reduction thresholds.

What is the impact on the results of the different distance measures evaluated?

In this case, we can observe that for PE based features, the Manhattan distance does not produce better results than Euclidean distance. More concretely, for Mean and Max distance selection rules the results are improved but when the minimum distance is selected (the only distance measure with acceptable performance), the results are nearly equal. On the contrary, for operational code frequency, the results observed for Manhattan distance actually differ, presenting superior results for AUC, and better tradeoffs between FPR and FNR.

What is the impact on the results of the different distance selection rules?

Regarding the distance selection rules, we can observe that, with the exception of certain configurations (operational code frequency with dataset reduction and low reduction thresholds), the Max distance selection rule does not present a sound performance, probably because of its sensitiveness to outliers.

Besides, in the case of PE based features, the only distance measure with a good performance is the Min selection rule.

When operational code frequency is compared, both Mean and Min distance selection rules achieve good results.

This difference might be caused by the impact of the results for the classification of custom packers. In fact, the results show that the distance between a non-packed instance and a custom packed instance is low. This fact affects the final value when the average distance is calculated. For this reason, the mean distance is not a valid approach when PE based features are used.

Does our anomaly detection approach present sound results for the classification of packed and non- packed files?

With the exception of certain configurations, the anomaly detection method proposed is capable of classifying packed and non-packed binaries efficiently.

While the results obtained are not as favourable as those obtained for supervised approaches, our model only models non-packed binaries.

The trade-offs between FPR and FNR selected for each configuration tend to produce high false positives rates (near 0.10 FPR). Nevertheless, these thresholds have been selected to maximise the packer detection rate at an assumable false positive rate. The simplicity of the distance-based approach allows to adjust the threshold according to the requirements of the deployment scenario.

- **Adjustable tradeoff between FPR and FNR.** Sometimes, it is important to consider the tradeoff between FPR and FNR. In some cases, it might be interesting to achieve nearly 0 FPR assuming the risk of misclassifying positive instances. For instance, a common user might be reluctant to install an antivirus solution that continuously throws false positives. In contrast, an antivirus company might desire to configure a different threshold when processing a malware dataset. In this particular case, misclassifying a packed binary as unpacked might imply not obtaining the original code of the sample.

- **Adjustable number of positives for different use cases.** The anomaly score can be used to prioritise the analysis of samples that show a high anomaly degree. Nevertheless, a high score does not necessarily mean that the sample will be more interesting than samples presenting a lower value. Also, this approach is highly dependant on the distribution of the samples analysed. The proportion of packed instances

can be different for a set of binaries already labelled as malicious than for a set of binaries submitted for analysis in an on-line submission system, in which any user could submit both packed, non-packed, benign, or malicious samples.

Future work in anomaly detection can be oriented in different ways. On the one hand, other anomaly detection approaches could be tested, like probabilistic models or one-class support vector machines. These more complex approaches might eventually capture differences among samples that cannot be represented in a linear system like this.

On the other hand, it would also be interesting to study possible feature selection techniques considering only one of the classes, given that the majority of such approaches make use of information from both classes in order to measure the importance of each attribute. In addition to feature selection, it also would be possible to rescale the feature-space measuring the capacity of each feature to characterise the set of non-packed samples.

In addition, it also would be interesting to combine the 2 feature-sets proposed in this chapter, or even to consider subsets of features.

During Chapter 3 and 4 we have relied on the extraction of certain features for the classification of packed binaries. Nonetheless, malware writers' efforts are focused on creating stealthy malicious software to bypass the filters implemented by both the security industry and the academic community. For this reason, it would be interesting to systematically study the different attacks malware writers may implement to bypass this kind of filters.

Although generic detection of packed files is useful to identify binaries protected by unknown packers, the correct identification of binaries packed by off-the-shelf packers is equally important. These binaries can be unpacked by specific unpacking routines. In addition, signature-based detection systems throw false negatives due to the capacity of malware writers to modify the binaries. For this reason, robust packer identification systems would avoid the generic unpacking of certain samples (applying custom made unpacking routines). Malware writers typically scramble well-known versions of packers in order to avoid detection. When it is not possible to determine the precise packer used to protect the binary, it is interesting to quickly analyse the unpacking routine in order to understand its structure.

Finally, it is important to mention the limitation presented by the dataset regarding the labelling of non- packed malware. Due to the possible noise

in the dataset employed (see Chapter 3), the results might be affected negatively. For a real-world deployment of such system, it would be interesting to sanitise the non-packed dataset using other manual methods to ensure its correct labelling. Nevertheless, in Chapter 3 we identified many limitations of the tools used to label the dataset employed for the experiments. In this sense, we consider that, while there are many sandbox systems focused on malware analysis, researchers lack appropriate tools for the analysis of packers.

4.5 Summary

In this chapter we propose and evaluate the application of anomaly detection techniques to the packer classification problem.

First, we describe our anomaly detection method and the data reduction process employed to improve the efficiency of the system. Second, we evaluate the approach over the complete feature-set proposed in Chapter 3. Finally, we propose an alternative feature-set based on operational code frequencies and evaluate the performance of our anomaly detection approach over this feature-set.

As a summary, the method proposed does not achieve the results presented by supervised machine-learning approaches, but it allows us to build a model that only depends on the information provided by a set of non-packed binaries.

Part II

Dynamic analysis for run-time packer analysis and unpacking

«Consider what you want to do in relation to what you are capable of doing. Climbing is, above all, a matter of integrity.»

Gaston Rébuffat (1921–1985)

Chapter 5

Longitudinal study of run-time packers structural complexity

Run-time packers, originally designed to reduce the size of executables, rapidly became one of the most common obfuscation techniques adopted by malware authors. They are now used by the vast majority of malicious samples to protect and encrypt their data and code sections – which are then restored at run-time by a dedicated unpacking routine. Nevertheless, while the first simple packers were aimed at reducing the size of the binary and presented a very low degree of sophistication, modern packers employ a wide variety of techniques in order to prevent the analysis of the sample (e.g., anti-debugging, anti-emulation, anti-disassembly techniques, obscuring the original entry point of the binary or employing several layers of unpacking code).

Run-time packers have been thoroughly studied in the literature, and several authors have focused their efforts on generic unpacking [KPY07, MCJ07, CX10, SYS+08, RHD+06]. Most of these solutions are based on the dynamic execution of the sample (e.g., by an emulator or a debugger) and rely on different heuristics to detect the end of the unpacking routine (i.e., the correct moment to dump the content of the process memory). Nearly all antivirus software adopted this type of solutions in order to provide some

form of generic unpacking before applying signature or heuristic based detection.

Given the early success of these efforts, the research community quickly focused their efforts on other forms of code protection. For instance, several recent studies have focused on virtualization-based protectors [Rol09, SLGL09], which involve completely new challenges, and stand as a completely separate and still unsolved problem. Nevertheless, these packers are hard to implement.

i Virtualization based packers are extremely complex pieces of software. These tools are generally commercial software employed to protect legitimate applications.

ii Applications protected by this kind of packers can sometimes introduce a high computational overhead in the system, since most of them are based on the translation of the original code to a different target architecture. This overhead may be acceptable to protect certain highly sensitive parts of legitimate applications. Once again, introducing such overhead in a malware sample that is continuously running on the system could raise suspicions.

iii This kind of protectors do not work properly on any kind of software. Generally, these tools allow to select the parts of the code to virtualize, and encourage the developer to properly test the application after the protection. Some of these tools even apply binary analysis techniques to disassemble the instructions of the compiled binary (e.g. Themida).

As a consequence, malware authors keep protecting their samples using the traditional run-time unpacking approach. The problem of how to perform runtime unpacking of their code still represents a challenge. In fact, traditional solutions rely on a number of assumptions that are not necessarily met by common run-time packers:

i If a sample contains multiple layers of packing, these are unpacked in sequence and the original application code is the one decoded in the last layer.

ii There is a moment in time in which the entire original code is unpacked in memory.

iii The execution of the packer and the original application are not mangled together (i.e., there is a precise point in time in which the packer transfers the control to the original entry point).

iv The unpacking code and the original code run in the same process with no inter-process communication.

These simplifications make previous approaches unsuitable to handle the real challenges encountered in complex run-time packers. Moreover, while there are several tools and on-line services available for malware analysis, there are no equivalent tools for the analysis of run-time packers. Available generic unpackers rely on heuristics that can be easily evaded, and, as an alternative, analysts develop unpackers that are specifically designed to work for a specific packer family and version.

Given the limitations of previous approaches, we establish the following research questions to be addressed on this chapter.

First, we are interested in understanding the level of complexity of the existing run-time packers that are used to protect malware.

Research question 5.1 *What is the maximum level of sophistication of run-time packers?*

To answer this research question we present a new fine-grained dynamic analysis system designed to collect a large amount of information from the execution of packed binaries. The collected data is then analysed and used to compute the packer structure and a number of indicators that summarise the features and internal characteristics of the packer. This is, to the best of our knowledge, the first approach that measures the complexity of run-time packers from different perspectives. Moreover, we used this tool to understand the level of sophistication of over 580 different packer configurations. The experiments conducted lead to a second open research question. It is well known that the malware writer often relies on off-the-shelf packers to protect and obfuscate their code. Tools like UPX or Armadillo are well known for both malware writers and malware analysts. However, malware writers often decide to avoid existing tools and implement instead their own custom packing routines. Whereas the unpacking of common protectors has been widely studied in research publications, there is no study that measures the complexity of custom packers. Accordingly, we establish a second research question:

Research question 5.2 *How widespread are custom packers in collected malware, and, how sophisticated are they compared to off-the-shelf packers?*

Also, it is interesting to know how the techniques employed by malware authors have evolved over the years in order to understand their motivations.

Research question 5.3 *How has the packer complexity evolved over the years?*

In order to answer these questions, we performed the first longitudinal study of run-time packer complexity using a dataset composed of packed malware collected during a period of 7 years, from mid-2007 to mid-2014.

Finally, it is important to understand the impact of packer complexity.

Research question 5.4 *How does this complexity affect the common assumptions of generic unpackers?*

In this chapter we first present a taxonomy capable of measuring the complexity of a packer from different points of view, providing as a result a single and incremental classification of packed binaries. This taxonomy is described in Section 5.1. Then, Section 5.2 describes the design and implementation of a framework based on dynamic analysis that allows us to precisely compute the complexity of a packer and to graphically represent its structure. Section 5.3 shows the results obtained for the complexity analysis of two different datasets (off-the-shelf and custom packers). Finally, Section 5.4 discusses and summarises the results obtained in these experiments.

5.1 Packer taxonomy

In this section we present the conceptual model required to capture and describe the characteristics and complexity of run-time packers.

The most simple form of run-time packer consists of a small routine executed at the beginning of the program that overwrites a certain memory range with the decompressed, deobfuscated, or decrypted code of the original application. After the unpacking routine has terminated, the execution is redirected to the original entry point (OEP) located in the unpacked region (an operation often called "*tail jump*").

These packers represent the most simple form of self-modifying code. However, run-time packers often involve several layers of unpacking routines, in which the first routine unpacks a second one, which in turn unpacks another routine, until the original code is reconstructed at the end. To complicate the situation, these layers are not necessarily completely independent and the execution may jump back and forth between them.

Previous approaches have proposed different models in order to capture this kind of behaviour, generally focusing on determining where the original entry point of the binary is located, or which are the code regions that were unpacked.

This research is motivated by the intuition that, in many cases, a simple model cannot capture every possible packer structure. In this way, we propose a packer taxonomy capable of representing several properties that measure the structural complexity of a run-time packer. The taxonomy, together with the visual representation of the structure of the packer can assist the analyst in the difficult and time-consuming reverse-engineering task.

In this section, we propose a model to represent the inner structure of run-time packers that can be employed to measure their complexity. This representation also allows to visually depict this structure and properties, helping the analyst to find interesting regions of code.

5.1.1 **Regions and layers**

First, we propose the use of two concepts, executed regions and layers in order to provide a way to represent the structure of a packer.

A decryption layer is, intuitively, a set of memory addresses that are written by code in the previous layer and afterwards executed. In this way, when the binary starts its execution, the instructions loaded from its image file belong to the layer $\mathcal{L}_0$. Accordingly, if any address written by any of those instructions is executed, it will be part of the next layer, in this case layer $\mathcal{L}_1$.

During our research, we have found packers that employ several processes (and thus, several address spaces) during the unpacking process, packers that interleave the execution of unpacking code with the original code, and even packers that incrementally unpack and re-pack the code on-demand before and after its execution. In our taxonomy, we capture these and several other aspects that allow us to measure the actual structural complexity of runtime packers from different points of view.

Tracing granularity and implications on the model

A basic block is a set of instructions with a single entry point and exit point. Therefore, a basic block starts at a memory address which is the target of any control flow instruction such as `jmp`, `call`, or `ret`, and ends in a control flow instruction. In contrast, our tracing engine monitors execution blocks. An execution block is a set of instructions that are executed atomically in our system. More concretely, the tracing engine first translates instructions sequentially until a control flow instruction is found, and then executes them atomically. Also, it is important to consider that, if an exception occurs in the middle of a block, then the execution block ends and a new block is translated for its execution. Execution blocks and basic blocks present some semantic differences.

i While there is no instruction in the program that jumps to an instruction in the middle of a basic block, in the case of execution blocks, this property can be violated. When a block is executed and the control flow jumps to an instruction in the middle of the block (e.g., something typical in loops), the next execution block will start in the middle of the previous block and will finish in the control flow instruction. In this situation, we would have 2 basic blocks. The first one would finish with the instruction before the target address of the control flow instruction (i.e., beginning of the loop). The second basic block would start at the beginning of the loop, and end in the control flow instruction. In contrast, while the number of execution blocks is also 2, the first block would consist of all the memory addresses executed, and the second one would only cover the loop. Also, the two blocks overlap.

ii Two basic blocks starting at the same address will finish at the same point. In the case of execution blocks, there are situations in which this assumption is not preserved. While a basic block is always terminated by a control flow instruction, an execution block can be finished by an exception or an event during execution. In this way, since an execution block can raise exceptions depending on the data it manipulates, it is possible to trace execution blocks that start at the same address and have a different size.

Formalization

Considering these aspects, we propose a fine-grained model to represent the concept of execution layers. We define an execution layer $\mathcal{L}_i$ as a

tuple $(\mathcal{X}_i, \mathcal{W}_i)$, where $\mathcal{X}_i$ is the set of execution blocks traced at that layer, and $\mathcal{W}_i$ represents the memory addresses modified by the instructions executed at that layer. In order to provide a comprehensive model that also considers inter-process communication, we define $\mathcal{W}_i$ as a map such that: $\mathcal{W}_i : P \rightarrow (\mathcal{M}, \mathcal{F})$ where P is the monitored process, $\mathcal{M}$ is the set of memory addresses in the address space of P modified at layer i, and $\mathcal{F}$ is a map such that $\mathcal{F} : Filename \rightarrow \mathcal{F}_{fn}$, where $Filename$ is the file name of the file modified, and $\mathcal{F}_{fn}$ represents the offsets of the file modified by process $\mathcal{P}$ at layer i.

During the execution, we maintain a set $\mathfrak{L} = \{\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_{maxl}\}$ where *maxl* is the innermost execution layer (i.e., deepest unpacked layer) of the binary.

i When the binary is loaded into memory, there is one single execution layer in $\mathfrak{L}$: $\mathcal{L}_0(\emptyset, \emptyset)$.

ii When the binary starts its execution on its entry point, we trace every execution block $\mathcal{EB}$. For each execution block, we compute the layer $\mathcal{L}_i$ it belongs to following Algorithm 2. In this way, if the memory space occupied by the execution block has not been modified by any other block, it is assigned to layer L_0.

Data: Set of execution layers $\mathfrak{L}$, set of memory addresses where the execution block is located $\mathcal{M}(\mathcal{EB})$.
Result: $\mathcal{L}_i$ of the execution block.
$i = 0, t = 0$
while $t < max_l$ **do**
  foreach $address \in \mathcal{M}(\mathcal{EB})$ **do**
    if $address \in \mathcal{W}_t$ **then**
      $i \leftarrow t$
    end
  end
  $t \leftarrow t + 1$
end
$i \leftarrow i + 1$

Algorithm 2: Layer selection.

Then, the corresponding layer $\mathcal{L}_i$ is updated: $\mathcal{L}_i = (\mathcal{EB} \cup \mathcal{X}_i, \mathcal{W}_i)$.

If the code is not self-modifying, it would be impossible to have two overlapping execution blocks. However, packers commonly use highly obfuscated routines that can eventually derive into a situation in which

two different execution blocks overlap for one layer. This situation might occur if the following conditions are met:

1. A memory address range is modified by some code at layer $\mathcal{L}_i$.
2. Eventually, an execution block at that memory address range is executed and $\mathcal{L}_{i+1}$ is updated with such execution block.
3. The same memory address range is modified again by some code at layer $\mathcal{L}_i$, setting different values for it.
4. A different and overlapping execution block is executed and added to $\mathcal{L}_{i+1}$.

In order to support this situation, our model allows to store, at the same layer, different execution blocks in the same range of addresses.

iii Finally, layer $\mathcal{L}_i$ is updated in order to keep track of the memory addresses modified by the executed block: $\mathcal{L}_i = (\mathcal{X}_i, \mathcal{M}_w(\mathcal{EB}) \cup \mathcal{W}_i)$, where $\mathcal{M}_w((\mathcal{EB}))$ is the set of memory addresses modified by the execution block.

Implications

This model has several implications:

- A layer only defines how deep a sequence of instructions is into the unpacking process, but it does not identify a semantically coherent piece of code. For instance, it is often the case that the original application code is located at the same layer with some other routines that are part of the packer (e.g., monitoring or obfuscating the execution, or interposing library calls).
- It is possible for a certain memory location to belong to different layers – if it is overwritten multiple times.
- There could be a case in which the original code is present in more than one layer. For example, two regions of the original code might be unpacked by different parts of the unpacking engine (located at different layers).
- If an execution block is located at a memory address space modified by execution blocks in different layers, its layer will be determined by the highest layer that modified its memory. Consequently, once a

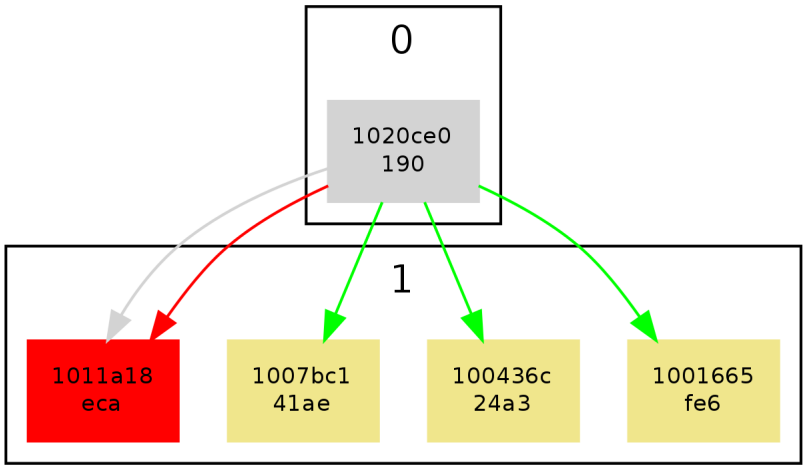


Figure 5.1: Graph generated for the Windows calculator packed with UPX.

memory address is modified by an execution block at layer $\mathcal{L}_j$, any write to the same address originated at an execution block at a layer L_k such that $k < j$ will have no effect on the future assignment of such execution block to a layer. Despite this aspect might seem counter-intuitive (one may think that a memory address should belong to the layer of the last instruction that modified it), there are cases in which different parts of the unpacking routine (located at different layers) are in charge of unpacking the same memory address space. For example, some code in layer 2 may first initialize some memory address range. Then, the execution may jump to layer 4, in charge of performing a first step of the decryption, and finally back to layer 3, in charge of finishing the last operations of the decryption algorithm. In such a case, layer 2, 3 and 4 would have participated in the unpacking of the original code, being layer 3 the last one to write over that address space. In these cases we consider that the original code should be placed at layer 5, considering that the 3 layers have participated in the unpacking process.

Memory regions and general overview

The execution of the binary results into a set of layers with their corresponding sets of execution blocks and memory addresses modified by those blocks. With this information, we compute blocks of contiguous memory addresses executed in order to delimit the regions of executed code.

In order to do this, there are some aspects that must be considered.

First, the code generated by a compiler is not always placed at contiguous addresses. For example, there might be paddings between functions containing no valid instructions. Second, since we are tracing dynamically the execution of the binary, only certain code paths are covered by the analysis. Consequently, we compute the set of executed memory regions $\mathcal{R}_i$ for each layer by merging the blocks of contiguous addresses executed. In this way, each region is represented by the lowest and the highest address of each set of executed addresses in which the distance from each execution block to the closest one in the set is not higher than a given threshold D_t.

Figure 5.1 shows the structure of the Windows calculator protected by UPX. The structure consists of 2 layers. In the first layer (0), there is only one region corresponding to the packer code (shown in grey). This region starts at address 0x01020ce0 and has a size of 0x190 bytes. Layer 1 has 4 different regions depicted in yellow and red. For every region, we display its base address and size. The different colours and edges represent properties defined later on in this chapter.

This model allows us to measure the number of layers of the packer and the size of each layer (in terms of bytes, instructions, execution blocks, and regions).

Also, this structure is the base to compute other properties by modelling aspects like write operations and execution transitions between regions and layers.

5.1.2 **Parallelism**

Many packers employ several processes to unpack the original code. Some packers take the form of droppers and create a file that is afterwards executed, while others create another process and then inject the unpacked code to it. Additionally, some packers make use of the debugging API of Microsoft Windows in order to (i) control the execution of a child process, not allowing it to start before the original code is injected to it and the thread context is set appropriately, (ii) coordinate the unpacking of different regions of code on demand and (iii) as an anti-debugging and anti memory dump trick, since it prevents other processes from attaching as a debugger to the process that actually contains the original code.

At this point, it is important to differentiate between processes involved in the unpacking process, and processes that are part of the payload (original code) of the protected binary. In fact, it is common to find malware binaries that expand several processes in order to execute their payload. In

this way, these processes lock a mutex and wait until it is released. The first process to lock the mutex is the one that actually executes the payload while the rest wait. If the process is killed, then the next process starts executing the payload. This approach is sometimes used as a mechanism to ensure the execution of at least one process in the system

In our model, we measure the number of processes created during the execution of a binary, but we only consider them to be part of the packer if they interact between them (they write to each others address space). Later in this chapter we explain how these interactions can be performed, and describe the different events we monitor.

In order to provide a comprehensive analysis of the packer, we trace all the processes created during execution by the original binary.

Regarding execution threads, we also record the number of threads created for every monitored process. As we will detail in the following sections, the parallel execution of threads has an impact over the defined taxonomy.

5.1.3 Transition model

Given the definition for layers and regions provided in Section 5.1.1, we define an execution transition as a succession of two blocks in the execution trace of a binary, such that the region in which the first block is located is different from the region of the second block.

Similarly, we define a write operation between regions as any memory write from an execution block located at a region to an execution block at a different region.

Given these definitions, the second characteristic measured in our model describes the type of execution transitions observed between the different layers of code. A transition between two layers occurs when an instruction at layer $\mathcal{L}_i$ is followed by and instruction at layer $\mathcal{L}_j$ such that $i \neq j$.

5.1.3.1 Linear vs. cyclic transition model

In the simplest case, there is only one transition from a layer (typically the end of the unpacking routine) to the next one (the beginning of the following unpacking routine or the original entry point of the application). In this case, if a packer has N execution layers, there are $N-1$ transitions, from a region at layer $\mathcal{L}_i$ to a region at layer $\mathcal{L}_{(i+1)}$. In our model we define this behaviour as *linear transition model*. Some packers do not satisfy the

aforementioned definition, and contain transitions from a layer to one of its predecessors. We define these cases as a *cyclic transition model*.

As we described in Section 5.1.1, the transition between execution blocks can occur for different reasons. One of the aspects that must be considered is process and thread scheduling. This mechanism implies that, at a certain point, the transition from an execution block to the next one can be triggered by a context switch. Also, in order to simplify the model and to reduce the amount of data collected, we leave out of the scope kernel mode execution and thus avoid tracing the kernel address space. Similarly, we also do not keep track of the basic blocks executed in the system library address space unless it is modified by the process.

As a consequence, if the monitored binary creates several processes, we might find an execution block at the address space of a process, followed by an execution block at the address space of another process. In this case, there is an execution transition between processes, but this transition does not imply that the code is designed to synchronize the execution of those blocks, which might be completely independent. Moreover, if the regions for those execution blocks are located at different layers, we consider there is an execution transition between layers.

Similarly, if a process creates several threads, the thread scheduling might also impact the transition model described in this section. If two threads execute code at different layers in parallel, there will be an execution transition whenever there is a thread context switch. For instance, a packer might create one thread for the original code and another for protection routines such as memory checksumming or anti-debugging. If both routines are located at different layers, we consider the transition model as a *cyclic* transition model.

In both cases, (several processes or several threads), we consider that labelling the sample as *cyclic* is a safe assumption because despite the execution context switch, both codes are interleaved in the execution (intentionally or not).

Finally, there is a risk of considering *cyclic* binaries that have their original code located at different layers, (something that could occur if different parts of the original code are modified by regions at different layers). In these cases, our model over-approximates the complexity of the transition model.

5.1.3.2 **Payload isolation**

Finally, the last property we measure is the interaction between the packer code (i.e., unpacking routines) and the payload (i.e., original code that was packed). Simple packers first execute all the packer code, and once the original code is recovered, the execution is redirected to it. For these cases, there is an execution transition such that all the previous execution transitions have their source and destiny in the packer code, and all the subsequent transitions have their source and destiny in the original code. We refer to this transition as tail-transition. Note that this transition does not necessarily take place from layer $\mathcal{L}_i$ to layer $\mathcal{L}_{i+1}$, since a packer might eventually unpack a region of packer code at the same layer as the original code, and the transition to the original entry point might take place between these two regions. If a packer does not meet this definition, we consider its execution model as *interleaved*. Some packers interleave the execution of certain parts of the unpacking routine with the original code. In some cases, this technique is implemented by hooking the Import Address Table to make the addresses in the table point to routines in the packer code. This approach can be used to obfuscate the use of API functions by redirecting them through the unpacking code. This approach can also be used to implement anti-debugging and anti-tampering techniques during the execution of the original code, in order to provide a better protection against reverse-engineering and memory patching techniques.

5.1.4 **Unpacking frames**

A simple packer consists of a decryption routine that first unpacks the code, and then the execution is redirected from the unpacking routine to the original code. In such a simple model, there is no interaction between the code of the packer and the protected code.

In Section 5.1.3 we have described possible alternative transition models in which the protected code and the unpacking routines interact during execution.

One of these forms of interaction can lead to a situation in which a part of the code (either unpacking routine or original binary) is written at different times. The Backpack packer [BLB11] is an example of this behaviour. This packer protects each function independently and instruments each function call and return. Every time a function is called, it is decrypted, executed, and then encrypted again when it returns. The execution of these functions is mangled together with the code of the packer. Additionally,

the packer encrypts and decrypts parts of the code each time. We consider that every time there is a decryption and execution iteration, there is a new frame in the unpacking process.

In this way, we can intuitively define unpacking frame as a succession of a memory write and a memory execution. In this way, the simplest run-time packer has at least one unpacking frame, since it first writes a memory area and executes it afterwards.

Let's consider a binary packed multiple times. In such a case, all the code would be fully unpacked in one layer before the next layers are unprotected: each layer of the unpacker routine is a fully independent piece of code.

According to the definition provided of unpacking frame, during the execution of this binary there would be as many unpacking frames as layers in the code, since for each layer there is a memory write and a memory execution.

Nevertheless, these layers are completely independent, and we should not consider that the contents of the binary are unpacked at different times.

In this way, we restrict the definition and consider an unpacking frame as a succession of a memory write, and a memory execution of a piece of code present in a certain memory address range.

This definition would be sufficient under the assumption that all the unpacking layers do not share the same address space, something that could eventually occur.

In order to cover such cases, we only consider that a binary is unprotected in several unpacking frames if these frames occur in the same execution layer.

Figure 5.2 shows the structure of the Kaiten malware protected by the Backpack packer. In this case, the figure displays for each region its base address, size, and number of frames. Grey edges represent execution transitions and are labelled with the number of transitions occurred between each pair of regions. Also, for each layer, it displays its layer number on the left, and the number of frames on the right.

If all the unpacked code present at each execution layer is unprotected at the same time and then executed, we can define the execution model as *single-frame*. On the contrary, if there is an execution transition from any region at a layer $\mathcal{L}_i$ to any region of layer $\mathcal{L}_j$, such that $i \neq j$ at time t_n, and a write operation from any region at layer $\mathcal{L}_k$ to any region of layer $\mathcal{L}_j$ (by definition, $k < j$) at time t_m such that and $m > n$ (followed by another execution transition), then we define the execution model as *multi-frame*.

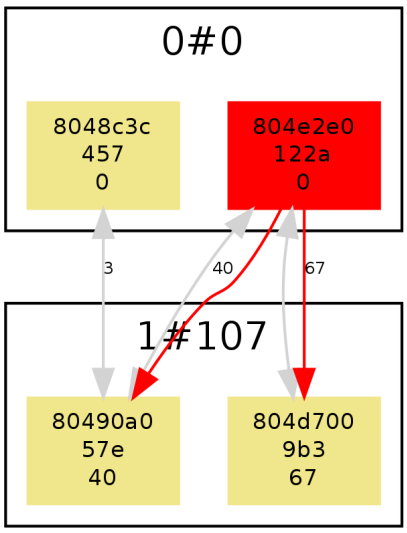


Figure 5.2: Graph generated for the Kaiten malware protected by Backpack.

5.1.4.1 Code visibility

Both the packer code and the original code can present several frames during the unpacking process. The vast majority of packers unpack the entire code and data of the original application before starting its execution. In these cases, the original code is isolated from the unpacking routines. The original code is not written once it is executed. However, more advanced examples selectively unpack only the portion of code that is actually executed (and therefore present several frames for the original code). This approach is used as a mechanism to avoid memory dumps of the whole contents of the binary. Based on the amount of code and on the way it is written in memory, we can distinguish three types of unpacking schemes:

- **Full-code unpacking.** These routines first unpack all the original code and data, and then redirect the execution to the original entry point. In this case, there is always a point in time in which the entire code of the malware can be scanned or retrieved from memory.

- **Incremental unpacking.** Incremental unpacking approaches reconstruct the original code on-demand, just before it gets executed. If the content of the memory is dumped in the moment the original entry point is reached, only the first frame of code will be available. Besides, if an interesting code path is not executed, the frames that cover that execution path will never be unprotected and will remain hidden.

- **Shifting decode frames.** A more complex version of incremental unpacking consists in re-packing each frame of code after its execution. Although this approach is less efficient, it forces a potential automated unpacker to extract several memory dumps and join the results in order to reconstruct a complete view of the original code.

There are several possibilities to implement incremental and shifting-decode-frames unpacking routines. All of them require a way to trigger the packer when a new code frame needs to be unpacked (or re-packed). The following are some of the most common approaches we observed in our study:

- **Exception-based redirection.** A simple approach to redirect the execution back to the packer is to force an exception. For instance, Armadillo and Beria take advantage of the memory protection mechanisms provided by the operating system. They mark memory pages as non-executable and then capture the exceptions produced when a protected memory page is reached. Then they overwrite the page with its unpacked content and then return to it.

- **Instruction-based redirection.** Another way to invoke the packer consists in injecting special instructions in the application code. For instance, ZipWorxSecureEXE replaces original instructions with an interrupt `INT 3` instruction. Whenever the execution reaches the protected address, an exception is generated and the control flow is redirected to the unpacking code, that substitutes the instructions with the original code. NoobyProtect, in contrast, substitutes areas of memory by `ret` instructions.

- **Instrumentation-based redirection.** In this case, the code to unpack each frame is inserted together with the original code. An example of this technique is used by Backpack (an advanced packer proposed by Lanzi et al. [BLB11]) which instruments the binary at compile time, using the LLVM framework. Backpack prepends a decryption routine and appends an encryption routine to every individually protected region of code. Themida is another example of this kind of instrumentation. It can be integrated directly into the development environment, allowing the developers to define macros where certain routines of the packer will be placed to protect specific regions of the code. In addition, if this approach cannot be used, Themida also

applies binary analysis techniques to discover and instrument functions in the code. Either way, the approach used by Themida is much more complex than any other run-time packer, since it completely transforms the code into a different representation, and scatters the instructions of the function at different points in memory. As a consequence, memory dumping based unpacking techniques are useless against this approach.

The mechanism used in order to redirect the execution has a high impact on the overhead of the process. While compile-time instrumentation only implies to execute an unpacking routine in the address space of the process (requires no context switch), exception based redirection requires a high overhead since the operating system must handle and treat the exception each time a new block is reached.

5.1.4.2 **Unpacking granularity**

In the cases in which the protected code is not completely revealed before its execution, the protection can be implemented at different granularity levels. Depending on the granularity, the protection level and also the efficiency of the packer will be affected. In particular, we distinguish three possible cases:

i *Page level*, in which the code is unpacked one page at a time.

ii *Function or functionality level*, in which each function is unpacked before it gets invoked.

iii *Basic Block* or *Instruction level* in which the unpacking is performed at a much lower level of granularity (either basic blocks or single instructions). These two cases are equivalent in practice.

While fine grained approaches are more difficult to analyse and unpack, they introduce a larger overhead during the execution. Also, depending on the packing granularity, different instrumentation approaches can be applied, introducing different overheads in the process execution.

5.1.5 **Packer complexity types**

The features described in previous sections allow us to define the complexity of a packer following a simple hierarchy that combines all the characteristics into a single classification. Figure 5.3 shows this hierarchy, where

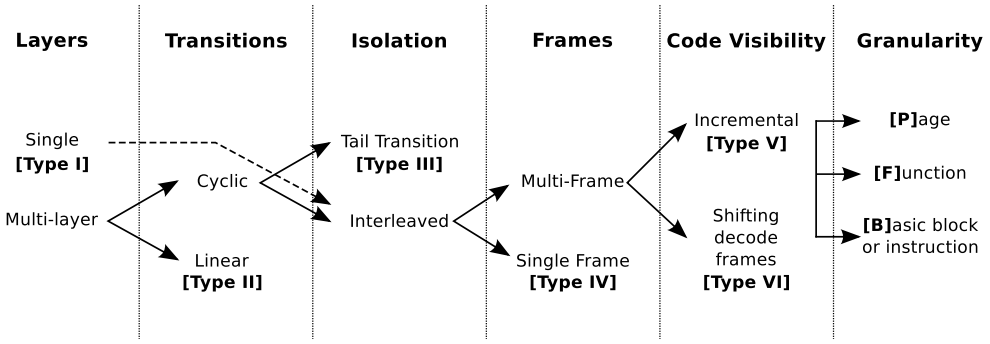


Figure 5.3: Packer features and complexity types.

Type I represents the simplest kind of packer, and Type IV represents the most complex structure.

[**Type I**] packers represent the most simple case in which the unpacking routine at layer 0 unpacks one layer corresponding to the original code. There is only one single transition from the unpacking routine to the original code. Simple packers such as UPX belong to this class of packers.

[**Type II**] packers present more than 2 layers. The shallowest layers represent a set of unpacking routines, each of them in charge of unpacking the following one. The last layer contains the original code, and there is one single transition from each layer to the following one (i.e., linear transition model). Binaries packed multiple times by Type-I packers belong to this category.

[**Type III**] packers are similar to Type-II packers. In this case, the layers are not executed in straight line, but forming cycles (i.e., there are execution transitions to shallower layers). As a consequence, the deepest layer may not necessarily contain the original code. During the experiments we have found a considerable number of packers that do not satisfy this simple assumption. Nevertheless, all these packers still have one single transition from the packer code to the protected code, and their execution is not mangled together.

[**Type IV**] packers can present any number of layers. These layers belong to the packer code and are interleaved during the execution of the original program. In some cases, the original code is instrumented to trigger certain routines of the packer in charge of controlling that

the program is not under analysis (i.e., implementing anti-debugging, anti-emulation, or anti-tampering techniques). Many packers typically calculate checksums of the code and periodically ensure that the program code has not been modified to either (i) bypass security of licence checks or (ii) introduce software breakpoints to enable reverse engineering the code. The consequence of this behaviour is that we find execution transitions between the packer code and the unpacking routines. However, there is always a point in time (i.e., tail transition) in which all the code is unpacked in memory. The packer routines interleaved with the original code do not perform any additional unpacking operation that affects the original code. One particularity of this type of packers is that it might be more difficult to locate the tail-transition because there is not one single point in which the code jumps from the unpacking code to the original program.

[**Type V**] packers also interleave the execution of the original program and the packer code. Nonetheless, in these cases the original code is revealed incrementally or on-demand. As a consequence, the unpacking behaviour is mangled together with the execution of the original program. In these cases the original code presents several frames because different parts of it are unpacked (i.e., written and then executed) at different points during the execution. Despite Type-V packers also have a tail-jump, at this point only one single frame of code is visible. A memory snapshot of the process at this moment would not reveal the content of the original binary. However, if a snapshot is taken at the end of the program, all the executed code will be available.

[**Type VI**] packers present the most complex structure in our taxonomy. These packers represent a special case of Type-V packers in which the code is protected again once executed. As a consequence, at any point of the execution of the original program only one single frame of code is visible. This frame of code can be as big as a memory page or as little as a single instruction. We denote the different possible granularities as **P** for memory page level granularity, **F** for function or functionality based granularity, and **B** for basic block or instruction granularity, which are equivalent from the point of view of the size of memory revealed during execution. In this way, both Type-V and Type-VI packers can have a **P**,**F** or **B** granularity.

Following this taxonomy, the complexity of the packer is computed with respect to the difficulty to dump the original program code.

It would be possible for a Type-III packer to present some obfuscations in the unpacking code (for instance, several frames in various layers or regions), and a complex execution topology in the unpacker code, but then a single transition to the original code. This case would be easier to unpack than a Type-IV packer with one single layer and one single frame of unpacker code that interleaves the execution of the original program making it difficult to recover a clean version of the program.

While the first case may present more complex features than the second one, from the analyst's point of view (analysing the original program), the latter case represents a more challenging protection mechanism.

5.2 Design of the analysis platform

In this section we describe the design and implementation details of our run-time packer analysis framework. The framework consists of over 6,000 C/C++ and 2,000 python lines of code.

5.2.1 Platform selection

There are multiple alternatives to perform dynamic fine-grained analysis of binaries.

Debugger based analysis takes advantage of the operating system process debugging capabilities to trace the execution of a process. Tools like PyDbg, allow to code scripts over the debugging API and allow to instrument the execution of a binary. Nevertheless, run-time packers are known to be heavily protected against debuggers. Although it is possible to bypass the majority of the known anti-debugging techniques, this approach can be easily detected.

Dynamic binary instrumentation techniques (e.g. Intel PIN) are a sound alternative for user-mode process instrumentation. This technique is stealthier than debugging, but it is still detectable and it does not provide a whole system view of the execution.

Virtual Machine Instrumentation stands as an alternative for whole system monitorization [DRSL08, DZX13]. Nevertheless, these techniques do not achieve the instrumentation capacity of an emulator.

Finally, whole system emulation has been extensively employed for binary analysis by several projects (e.g., Bitblaze [SBY+08], S2E [CKC11], or

Anubis [BKK06]). These approaches are stealthier than other methods such as debugging, provide a whole system view of the execution, and allow the direct manipulation of the execution context at any level, since they have a physical view of the machine.

The run-time packer analysis framework proposed is implemented over the two main components of the Bitblaze project [SBY+08]: TEMU and Vine. TEMU is a dynamic analysis tool based on QEMU, a whole system emulator. It provides an interface to monitor the execution of one or several processes. Additionally, it implements a dynamic taint analysis component. Vine is a framework to perform different binary analysis tasks over an intermediate representation.

This execution platform provides us a whole-system perspective of the execution, and allows the instrumentation of the execution of any process at the finest granularity-level. Nevertheless, like the rest of platforms, it presents several risks and limitations:

i **Semantic gap.** One of the challenges involved for this kind of platforms is dealing with the semantic gap between the physical view of the machine and the logical view of the operating system, an aspect that is, by itself, an open research problem [DGLHL13]. In order to deal with this problem, we complement the components already present in TEMU together with Virtual Machine Introspection techniques in order to extract the necessary semantics from the emulated operating system.

ii **Stealthiness.** Another important inconvenient about emulation based platforms is the ability of protected applications to detect the execution platform by implementing *red-pills*. *Red-pills* are assembly instructions whose semantic is not the same in a real processor and an emulator. Complex architectures with CISC instruction sets and variable size instructions make the implementation of an emulator a difficult task because of the complexity of the semantics of each operational code. As demonstrated by Paleari et al. [PMRB09], it is possible to automatically discover and generate red-pills for an emulator. To this end, we include in our experimentation platform the necessary mechanisms to detect the termination reason of the execution of a sample. In this way, even if a sample fails during its execution, we can detect the situation and treat the results appropriately.

   The Armadillo packer is an example of this limitation. Previous publications have tried to analyse this protector and reported execution er-

rors or their inability to correctly execute the samples [KPY07, BCK+10, DZX13] under a QEMU based emulator.

In order to analyse this complex packer, we studied it and found 2 instructions that impeded its correct emulation. After patching the Dynamic Binary Translation engine of the emulator, it was possible to fully analyse the sample. These patches were submitted to the QEMU mailing list[1].

Finally, we employed Microsoft Windows XP SP3 as a guest operating system. First, despite the number of malware samples targeted at Linux or OSX operating systems is growing, the majority of malware samples collected are still designed for Microsoft Windows systems. Besides, Microsoft does not provide official documentation for their operating system internals. Despite there are newer operating systems, the simplicity of Microsoft Windows XP makes possible to implement all the analysis infrastructure described above.

5.2.2 Execution tracing

In order to trace the execution of a given binary, we leverage the binary tracing capabilities present in TEMU, and implement our own techniques in order to deal with complex run-time packers that employ several processes. TEMU allows to implement plugins with callback functions that are triggered when certain events occur during the execution.

The main technique used by TEMU to this aim is the instrumentation of the Dynamic Binary Translation (DBT) engine provided by QEMU. In this way, TEMU inserts specific callbacks in the dynamically generated code at each translation block, and allows to monitor events such as the beginning and end of a translated block, the beginning and end of an instruction, or memory read and write operations at each instruction.

In this way, we monitor different events in the system that occur during the execution of a given binary in order to collect the necessary information.

- **Execution blocks.** Among other events, the TEMU analysis framework allows to instrument the execution of every block in the analysed binary. The DBT engine allows to execute code from different

[1] http://lists.nongnu.org/archive/html/qemu-devel/2014-02/msg01935.html

platforms into different target architectures by dynamically translating the assembly code in the guest architecture and generates code compatible with the host architecture similarly to Just In Time (JIT) compilers. Furthermore, it first fetches the instruction pointer and disassembles the instructions starting from that address until it finds an instruction that breaks the control flow. In these cases, it triggers the translation of a new block. Additionally, QEMU provides an optimization that allows to chain several blocks in a single translation block (i.e., block of instructions translated in a single burst). Nevertheless, in order to allow a sound instrumentation, TEMU disables this feature in the emulator. Also, QEMU leverages a mechanism to cache the already executed translation blocks in order to avoid the translation of already visited blocks. If any of the blocks is written at run-time, the translation blocks are invalidated and the content of the memory is translated again if necessary. In order to monitor the execution of every execution block, the framework inserts a call to a user-defined routine at the beginning and end of each translation block in QEMU, regardless of the process under execution in the system. In this way, the user-function is responsible for treating appropriately these events, depending on the process under execution. Although this process introduces a considerable overhead, it allows to perform a whole-system analysis monitoring several processes in both user and kernel mode. In order to correctly record the end of the execution blocks when exceptions are raised in the middle of a block, we patched the original TEMU implementation to trigger event callbacks for the execution block end when exceptions or interruptions are raised. In such cases, the CPU execution loop is broken, and the instrumented callback for the block end is not executed, since it is placed just after the code generated dynamically for the block. In order to cover these cases, we insert a callback when any exception is raised.

In order to identify every execution block, we use the address of its first instruction and a sequence number. Given the definition provided for execution blocks, two blocks starting at the same address should naturally finish at the same address, since the last address of the block should be an instruction that alters the control flow of the program. Nevertheless, there are two situations in which two different execution blocks can be different even if they have the same

starting address. First, if an exception is produced in the middle of a translated block, then the execution block recorded will have a lower size than the translated block. Besides, another different execution of the same block may produce no exception and be executed until its last instruction is reached. In this case, we would have two blocks starting at the same address and having different lengths. Whenever we find two blocks starting at the same address that have a different size or number of instructions, we create a new block with the same address, but increment its sequence number. The same problem occurs when the memory is modified during the execution. As we described in our model, if some code present at a given layer modifies and executes code several times in the same memory address, this code will be assigned to the next layer, and the blocks may start at the same address. In this cases, we also compare the blocks, and in case we find different blocks executed at the same address and layer, we increment the sequence number of the block.

- **Instruction count.** For each execution block we measure the number of instructions translated. In order to do this, we patched the execution engine present in QEMU in order to count the number of instructions.

- **Instruction size.** Similarly, during the binary translation step we compute the instruction size for each instruction.

- **Memory writes.** In addition to the instrumentation of blocks and instructions, TEMU allows to instrument memory writes as part of its taint analysis engine. Accordingly, we have leveraged this functionality in order to keep track of the written memory addresses for the monitored processes at each point. Also, in order to allow a fine-grained analysis of the packer, we can record the memory contents of each memory write if necessary.

- **Indirect function calls.** An interesting technique employed by most of the existing packers is the destruction of the original Import Table in order to hinder the reverse-engineering of the samples. On the one hand, it prevents the analyst from knowing which functions might be called by the binary by taking a simple look at the Import Table. On the other hand, this technique forces the analyst to reconstruct the table before starting the analysis, otherwise, when reverse-engineering

the samples, it is not possible to determine the destiny of the call instructions. One of the most common methods used to reconstruct the table is using the `LoadLibrary` and `GetProcAddress` functions. Although the implementation details vary among different compilers, in most cases, the original code still uses the same mechanism to call the API functions, which consists of making indirect calls to addresses stored into the Import Address Table, or calls to regions in memory that contain indirect jumps to addresses stored in such table. These instructions take the form of `CALL r/m` and `JMP r/m`, following the notation used in the Intel Instruction Reference Manual [Int13]. In order to detect potential Import Address Tables in the binary, we instrument the execution of such instructions in the dynamic binary translation routine in the emulator.

- **Process creation and termination.** The TEMU framework provides a driver for Windows XP based systems that allows to monitor events of special interest such as process creation and process termination. More concretely, this driver communicates these events to the emulator, providing for each process information about its identifier in the guest system (PID) and the value for the CR3 register (a register employed in the x86 architecture to provide process memory isolation). By using this register, the analysis framework in the host machine can monitor only the interesting processes. Besides, the driver also notifies the host system when a process terminates.

- **Modules loaded.** Additionally, the provided driver updates the host system whenever a new module is loaded in the address space of a process. Also, for each module, it lists the exported functions and their corresponding addresses.

- **API calls.** TEMU leverages the block tracing capability to hook important API calls in the guest system by resolving the memory address in which each module is loaded for every process under execution. The framework allows the user to trigger a certain function when an API function is called. Similarly, it is possible to hook the return address of the function in order to extract from the stack the interesting output parameters returned by the function.

Our framework makes use of API call tracing in order to listen for certain events of special interest in the analysis of runtime packers. Also, it

leverages Virtual Machine Introspection (VMI) techniques to retrieve information from kernel structures that are employed to manage processes, memory, and resources in the system.

The API offered by Microsoft Windows is exported by different libraries (e.g., `kernel32.dll`, `user32.dll`...). Internally, these libraries call to `ntdll.dll`, which contains the so called native API. This API is in charge of performing low level operations and finally system calls (i.e., interruptions used to request the kernel to perform operations.). Some of the *high level* functions aggregate several native API calls, simplifying the programming interface. Nevertheless, tracing this API is more difficult since the number of functions that can be used to perform similar operations is high. In this way, for certain operations in the `kernel32.dll` library, we know there are corresponding functions in the `ntdll.dll` (e.g. `WriteProcessMemory` and `NtWriteVirtualMemory` or `CreateProcessA` and `NtCreateProcess`).

For this reason, we trace function calls at the lowest possible level in user-mode execution: `ntdll.dll`.

Whenever an interesting API is called, we first read from the stack (starting at the memory address pointed by ESP) the appropriate number of bytes, retrieving the return address and the parameters for the function. These functions follow the same calling convention as the rest of standard Windows APIs (stdcall) (arguments are pushed into the stack in reverse order).

Also, some functions require output parameters that are passed to the function as pointers to a caller-reserved memory space. Since these parameters cannot be retrieved until the function returns, we trace the execution of the instruction located at the return address of the function call.

5.2.2.1 **Processes created by the packer**

We monitor process creation by hooking the `NtCreateProcessEx` function in the `ntdll.dll` system library.

When the function is called, an optional parameter `SectionHandle` is passed to the function containing a handle to a section object. This object is generally mapped to the file used to create the process (typically the executable PE file). This parameter is important since it allows us to know if the file has been modified by another monitored process. In such case, we consider that the initial memory allocated for the process has been indirectly modified by a process under execution.

Besides, when the function returns, an output parameter is updated

with the handle to the created process. In order to monitor a process, we retrieve the address of its Page Directory, which is written into the CR3 register on each context switch. This value is used in the emulator in order to know which process is under execution.

In order to retrieve the address of the Page Directory and PID for the created process, we traverse the psActiveProcess linked list located in the kernel memory address space, which is pointed by the psActiveProcessHead pointer. This pointer is present at a fixed address for Microsoft Windows XP SP3 (0x78 bytes offset starting from the address pointed by the kpcr pointer, located at address 0x0ffdff034).

The psActiveProcess contains one entry for each process (EPROCESS structure). We compare the PID of each process, stored in EPROCESS, until the entry for the calling process is found. Then, we retrieve the handle table for that process. A handle table is a multi-level table in which the innermost level contains one entry for each handle of the process, and the rest of levels contain offsets to the tables in the following levels.

For each handle, we read several interesting values:

- **Handle value.** The handle value is a 32 bit unsigned integer used to call API functions, and it represents the level and offset of the handle in the table.

- **Handle type.** The handle type is a constant value that determines the type of object referenced by the handle. The following handle types represent objects that we support in our framework: process handle (0x05), file handle (0x1c), and section handle (0x13).

- **Object pointer.** The object pointer points to the *payload* of the handle. In case of process handles, the object pointer points to the corresponding EPROCESS structure.

Once we find the handle table, we look for the handle corresponding to the new process. This object pointer points to the EPROCESS structure of the new process. Then, we can retrieve the PID and the CR3 of such process and start monitoring the newly created process in our tracing engine.

5.2.2.2 Interprocess communication: Remote memory writes

One of the methods provided by the Microsoft Windows XP to write the memory of a different process from any user-mode process in the system

(assuming it has the appropriate access token), is the `kernel32.dll` function `WriteProcessMemory`. This method is extensively used for process injection, both by malware and multi-process packers. This function internally calls to the `NtWriteVirtualMemory` function at `ntdll.dll`. Similarly, it is also possible to use the `ReadProcessMemory` function exported by `kernel32.dll`, or directly the `NtReadVirtualMemory` present at `ntdll.dll`.

In order to monitor this type of memory writes, we hook the `NtWriteVirtualMemory` and `NtReadVirtualMemory` functions.

These functions have several interesting input arguments. `ProcessHandle` is a handle of the remote process to read or write. `BaseAddress` is the address in the remote process to read or write. `Buffer` is the address that holds the content to write, or a buffer to store the content read from the remote process. Finally, `NumberOfBytesTo*` is the size of the memory read-/write.

Applying the process followed in Section 5.2.2.1, we can retrieve the PID and CR3 of the process that is accessed. In this way, we can trigger a remote memory write from the caller process to the remote process (in the case of a write), or from the remote process to the caller process in case of a read.

Besides, a process could eventually use these functions to read or write its own user-land memory using kernel code, and thus skip our tracing engine, since we do not trace the system in kernel mode for efficiency. Nevertheless, in case this function is called using the 0xFFFFFFFF handle (that represents the own process), we log a memory write to the caller process accordingly.

Finally, if we observe remote memory writes to processes not created by the analysed binary, then we consider that it is injecting memory to system processes, a behaviour typical in malware but no so common in packers.

5.2.2.3 **Interprocess communication: Files**

Another possible mechanism that can be used to write a process memory from another process is writing to files. Microsoft Windows provides several mechanisms to interact with the file system. In fact, a process can call to any of the numerous file read or write API functions. To keep track of these operations, we hook the `NtWriteFile` and `NtReadFile` functions at the `ntdll.dll` library, which are the lowest level functions that can be used to perform these operations in the user-mode execution environment.

As in the previous cases, we inspect the handle table for the calling

process and retrieve the object pointer associated to the corresponding file handle provided as argument to the function. This object pointer points to the file object, that contains several fields: the flag `IsOffsetMantained`, that indicates whether the file is configured to maintain the offset into the file, the current file offset of the given handle, and the file name. In this way, we keep track of every file write operation performed by any of the monitored processes. Whenever a file is read, we check if the read content was modified by some monitored process (by checking the offset of the read). If so, we log a memory write from the process that generated the file write to the process that reads the file to the appropriate address.

Given that these functions do not cover all the possible interactions with the file-system, we additionally consider the module load process during process creation (see Section 5.2.2.1) as a potential remote memory write.

In Section 5.2.2.1 we describe the `NtCreateProcessEx` function. One of the parameters provided as argument to the function is a section handle. We retrieve the object pointer for the section handle and ensure that it corresponds to a file mapped section by reading the corresponding flags from the section object. In such a case, we read the `SEGMENT` structure pointed by the section object, and then read the `CONTROL AREA` structure pointed by one of the fields in the `SEGMENT` structure. The `CONTROL AREA` structure contains a pointer to the file object linked to the section. Finally, we read the file object (`FILE` structure) and retrieve the file name to which it corresponds, and the file offset to which it is mapped. With this information, we check if the file mapped by the process was previously written by any monitored process. If it was written, we trigger a remote memory write (see Section 5.2 for a detailed description of remote memory writes). In order to calculate the offset for the write, we consider the offset at which the file was written, the file offset to which the section was mapped, and the base address where the section will be mapped in the process. Finally, we trigger a remote write from the process that modified the file at the appropriate addresses in the destiny process.

5.2.2.4 **Interprocess communication: Shared memory sections**

Finally, another interprocess communication mechanism available in Microsoft Windows is shared memory. A process can create a memory section that can be eventually mapped by another process. If this section is created with write permissions, a process can write to the address space of another process. Also, a section can be a Copy On Write (COW) section. In such a

case, a copy of the section will be created if the process writes any memory address in it, so that it does not affect to other processes that map the same section.

Typically, a programmer would use functions such as `CreateFileMapping` or `MapViewOfFile` at the `kernel32.dll` library to create named shared memory regions. Nevertheless, as in previous cases these functions will eventually call to the function `NtMapViewOfSection`, present at the `ntdll.dll` library. This function allows to map an existing section to the address space of a given process.

Similarly to previous cases, we identify the process to which the memory section will be mapped by looking for the corresponding PID and CR3 for the `ProcessHandle` provided to the function. If the handle specified is 0xFFFFFFFF, it will be mapped to the caller process. Also, we retrieve the object pointer for section handle passed as argument (`SectionHandle`) to the function.

When the function returns from its execution we read the `BaseAddress` parameter that contains the address where the section has been mapped, `SectionOffset`, that contains the offset of the section mapped to the base address returned, and `ViewSize`, that contains the size of the mapped view of the section.

This information, together with the Copy On Write bit, is stored in a list that is used to keep track of every mapped section for every process.

In the moment a memory section is mapped into the address space of a process, we check if the section was previously mapped to another process, and in such a case, we trigger a remote memory write from the process that mapped the section for the address space mapped. From then on, whenever there is at least one memory write to the memory mapped in any of the processes, we trigger a remote memory write to all the processes involved, only if the section is not a Copy On Write section.

We also check if the mapped section is associated to a file object. If so, any write to the memory section will be recorded as a file write. Since a file mapping section loads into memory the contents of the file, we log a memory write for each memory address corresponding to a position in the file that was written before by another process.

5.2.2.5 **Interprocess communication: Memory unmapping and deallocation**

Finally, we also trace the execution of two relevant API functions that can eventually be used by a packer in order to protect its memory contents from being read by a memory dumping tool. These functions are (in their lowest level versions) `NtUnmapViewOfSection` and `NtFreeVirtualMemory` at the `ntdll.dll` library.

First, tracing the `NtUnmapViewOfSection` allows us to keep updated the list of sections mapped to each monitored process. In this way, whenever a memory mapped section is unmapped, further writes to that address space will not be incorrectly triggered as remote writes. Since the two parameters provided to the function are input parameters, we do not need to trace the return address of the function. First, we retrieve the PID and CR3 given the process handle provided as the first parameter, and the we find the corresponding entry in the list of sections, comparing their base addresses, and remove the section.

Additionally, we consider that, if a memory region has been written by any of the methods described so far, then a section un-map implies a write to that address space, since the content previously available will not be accessible any more. For instance, a packer might apply page-level protection to its code, mapping a memory page whenever it is needed, and un-mapping it afterwards. The second step in this approach would imply a re-packing of the memory page.

Another low-level function that can be used to this end is `NtFreeVirtualMemory`, available in `ntdll.dll`.

Like in the previous case, we hook the execution of this function. In this case, we hook the return address in order to read the base address and size of the memory region freed (which does not necessarily have to match the base address and size provided as parameters). Again, we trigger a memory write for the corresponding process.

5.2.2.6 **Interprocess communication: Remote writes and the execution model**

In previous sections we have described the different system events monitored in order to record some of the most common inter-process communication mechanisms provided by the operating system. In this section we detail how these system events affect the execution model proposed in Section 5.1.

Accordingly, we define a remote memory write as any interprocess communication event in which the memory of a process is overwritten by means of any of the events described in previous sections. Note that a write operation performed by a process to its own memory by means of these functions is also considered a remote write. In order to record remote writes appropriately, we consider the following parameters:

- **Trigger.** Event that triggers the remote write.
- **Condition.** Conditions that must be met in order to trigger the remote write.
- **Destiny process.** Process modified in the event.
- **Destiny address.** Represents the address overwritten in the remote process.
- **Size.** Size of memory written, starting from the address specified.
- **Source address.** If the memory is copied from a buffer to the destiny address, the source address will be the address of the buffer copied.
- **Layer.** Layer of the model affected by the memory write.
- **Caller execution block and process.** If the operation is triggered by a function call, it represents the address of the basic block that triggered the call. Since the function calls recorded are located in `ntdll.dll`, we consider the caller block to the last block traced for the process before the API function was called.
- **Modifier process.** It is the process that modifies the memory in the destiny process. In some occasions, the modifier process is equal to the caller process, while in other occasions the modifier process is different from the one that triggers the action.

Table 5.1 to Table 5.9 specify the parameters defined for every possible action observed in the system. These parameters define how the different system events are represented in our model.

Table 5.1: Remote memory write in the execution model.

Action	**Remote memory write**
Trigger	NtWriteVirtualMemory
Condition	—
Destiny process	Indicated by the handle passed as argument
Destiny address	Indicated by function argument.
Size	Indicated by function argument.
Source address	Indicated by function argument.
Caller	Last recorded execution block for the caller process.
Layer	Highest between the caller and the layers containing modifications for the address.
Modifier	Caller process.

Table 5.2: Remote memory read in the execution model.

Action	**Remote memory read**
Trigger	NtReadVirtualMemory
Condition	—
Destiny process	Caller process.
Destiny address	Indicated by function argument.
Size	Indicated by function argument.
Source address	Indicated by function argument.
Caller	Last recorded execution block for the caller process.
Layer	Highest between the caller and the layers containing modifications for the address.
Modifier	Indicated by the handle passed as argument.

Table 5.3: File write in the execution model.

Action	**File write**
Trigger	NtWriteFile
Condition	—
Destiny process	—
Destiny address	File name and file offset written.
Siz:	Indicated by function argument.
Source address	Indicated by function argument.
Caller	Last recorded execution block for the caller process.
Layer	Layer of the caller execution block.
Modifier	Caller process.

Table 5.4: File read in the execution model.

Action	**File read**
Trigger	NtReadFile
Condition	The file read has been modified by a monitored process.
Destiny process	Caller process.
Destiny address	Indicated by function argument.
Size	Indicated by function argument.
Source address	File name and file offset read.
Caller	Last recorded execution block for the caller process.
Layer	Highest layer at which the file was modified.
Modifier	Process that modified the file region being read.

Table 5.5: Shared memory mapping in the execution model.

Action	**Shared memory section view map**
Trigger	NtMapViewOfSection
Condition	COW disabled, section mapped to process and modified.
Destiny process	Indicated by the handle passed as argument.
Destiny address	Indicated by function argument (base address).
Size	Indicated by function argument.
Source address	Computed from section offset to which the view is mapped and the base address in the remote process.
Caller	Last recorded execution block for the caller process.
Layer	Highest layer that modified the view in the remote process.
Modifier	Remote process that maps the section.

Table 5.6: File mapped section mapping in the execution model.

Action	**File mapped section**
Trigger	NtMapViewOfSection, NtCreateProcess
Condition	Section is file mapped, and the file has been modified.
Destiny process	Handle passed as argument (NtMapViewOfSection), or returned (NtCreateProcess).
Destiny address	Base address of the view.
Size	Indicated by function argument (NtMapViewOfSection), or the module loaded (NtCreateProcess)
Source address	—
Caller	Caller to the function call.
Layer	Highest layer at which the file was modified.
Modifier	Process that modified the file region being read.

Table 5.7: Write to shared memory section map in the execution model.

Action	**Write to shared memory section map: remote write**
Trigger	Memory write
Condition	COW disabled, the section is mapped to another process different from the caller process.
Destiny process	Process mapping the same section involved in the write.
Destiny address	Address of the write respect to the base address and section offset of the views of both processes involved.
Size	Size of the memory write.
Source address	—
Caller	—
Layer	Layer of the corresponding execution block.
Modifier	Process that triggers the memory write.

Table 5.8: File write from shared memory section map in the execution model.

Action	**Write to shared memory section map: file write**
Trigger	Memory write
Condition	The section is file mapped.
Destiny process	—
Destiny address	Section offset, base address, and file offset.
Size	Size of the memory write.
Source address	—
Caller	—
Layer	Layer of the corresponding execution block.
Modifier	Process that triggers the memory write.

Table 5.9: Memory deallocation and unmapping in the execution model.

Action	**Memory deallocation and unmapping**
Trigger	NtUnmapViewOfSection, FreeVirtualMemory
Condition	The memory affected has been modified.
Destiny process	Indicated by the process handle passed as argument.
Destiny address	Base address passed as argument.
Size	Size of the mapped view (NtUnmapViewOfSection), or allocated size (FreeVirtualMemory).
Source address	—
Caller	—
Layer	Layer of the caller execution block.
Modifier	Process that triggers the deallocation or unmapping.

5.2.2.7 **Memory type**

Another aspect analysed during the execution is the memory type associated to certain regions of the process. In this way, we distinguish among *Module*, *Heap* or *Stack*.

First, whenever a process is loaded in the system, the TEMU framework notifies every loaded module. With this information, we keep track of all the loaded modules for a given binary. Additionally, we monitor the execution of the functions `ZwAllocateVirtualMemory`, `RtlAllocateHeap` and `RtlReAllocateHeap` in the `ntdll.dll` system library. Whenever such a call is triggered, we retrieve the Process Enviroment Block (PEB) for the calling process, and the Thread Environment Block (TEB) for each thread executing for the process, by navigating the structures present in the kernel for the `EPROCESS` structure corresponding to the process. In the PEB and TEB, we can find the boundaries of the different heaps reserved by the process, and the boundaries of the stack for each thread. By keeping track of this information during the execution of the binary we know, for each execution block executed, the type of memory where it is located. This information is useful for the analysis of the binary since generally common compiled programs do not execute code from regions in the heap or stack, while some packers may use these regions of memory to place unpacking or deobfuscation routines.

5.2.2.8 **Analysis automatization**

The automatization of dynamic malware analysis is not a trivial problem. In some occasions, malware writers take advantage of these difficulties in order to avoid detection. Generally, malware sandboxes run the sample for a minimum amount of time before stopping the analysis. One method commonly used by malware writers in order to avoid analysis is to delay the execution of the malicious payload.

The automatization of the analysis of complex packers presents some differences with malware. We have implemented several heuristics in order to maximise the coverage of the analysis while keeping the analysis time reasonable for large scale experiments.

Traditional dynamic unpackers wait for the tail jump in order to dump the memory contents of the process, considering the unpacking finished. In general, this point in the execution is determined by using different heuristics or statistical methods, (e.g., variation of the entropy of the sections of the binary).

Nevertheless, for complex packers it is not easy to determine the appropriate moment in which the packer has revealed all the code. Given the observations during the initial experiments, there are cases in which the execution of the unpacker code is mangled together with the original code. In some cases, the original code is not unpacked at once, and the unpacking code that is interleaved with the original code is in charge of extracting the following frame of code.

Also, the great variety of techniques employed by packers in order to transfer the execution from the unpacking code to the original code make very difficult the task of determining where is the original entry point.

Finally, in order to perform a complete assessment of the structural complexity of the packer, it is necessary to continue the analysis even after the original entry point is reached, because, as mentioned before, the unpacker code might be interleaved with the original code.

We considered several aspects in order to achieve an effective dynamic analysis automatization.

- **Execution timing.** In order to monitor the activity of the monitored processes, we inspect every 2 minutes the user-time consumed by each of the processes in the monitored system by retrieving their corresponding EPROCESS structure from the kernel. More concretely, the value corresponding to the user-time consumed by a process is located at an offset of 0x3c bytes from the beginning of its EPROCESS structure. Since this structure is located at the same address during the execution of the process, it is sufficient to calculate the offset of this structure in the physical memory of the guest system. Furthermore, when QEMU is configured to perform whole-system emulation, it emulates the whole physical memory of the guest system. By calculating the offset of this variable in the emulated memory, we can seamlessly retrieve the user-time consumed by the process in the emulated system without introducing any computational effort.

- **Exceptions.** In order to detect exceptions, TEMU allows to monitor the execution of the `KiUserExceptionDispatcher` function at the `ntdll.dll` system library, which is called by the operating system under the context of the process that produced the exception. Whenever such function is called, we start a countdown. If after this countdown the process has not executed any address in the address space below 0x40000000 (under Microsoft Windows XP, the address space for process modules), we consider that the execution of the process is stuck

and will not recover from the exception, although the process is still present in the process list of the guest operating system.

Given these monitoring features, we first set a minimum analysis time-out of 5 minutes in order to avoid possible evasion techniques employed by malware.

Some samples, unfortunately, do not start their execution due to incompatibilities with the execution platform selected. In some cases the process is created but due to crashes in the process the execution does not reach the entry-point of the binary. In these cases, we set a time-out of 60 seconds. If after this time-out the execution does not reach the entry point of the binary, the execution is stopped.

The conditions that determine the appropriate moment to stop the analysis of the sample are summarised below.

- The sample does not reach its entry point after 60 seconds from the beginning of the analysis.
- All the processes analysed stop their execution.
- An exception is produced, and the process does not recover from it in a 2 minute time-out.
- The process reaches an stability point in which it does not consume user time in the emulated system.
- A maximum time-out of 30 minutes was reached.

5.2.3 Collected information

As a result of the sample analysis process, we collect information about different events that is useful for the analysis of the packer. This information is then processed, and allows us to compute the properties described in Section 5.1.

5.2.3.1 Execution trace

During the execution of a binary we generate an execution trace of the monitored processes. This trace is saved as a *zlib* compressed binary file in order to minimise the disk space needed during the analysis.

Each execution block is written to the trace file once its execution is finished. The memory writes are logged in the moment they are produced.

In this way, every memory write event is related to the following execution block in the trace. Remote memory writes are also recorded as special events. Another interesting event in the execution of the monitored processes is API calls. In fact, one of the most common heuristics employed to detect the original entry point of a binary is to wait for initilization function calls like `GetCommandLine`, `GetVersion` or `GetModuleHandle`. In our approach, we record every API function called by the binary. In this way, we can log certain important events such as calls to the aforementioned API functions and also compute the number of different API calls produced by each execution block or executed memory region, or the total number of API calls for each of them.

5.2.3.2 **Indirect jumps and calls**

During execution, we also monitor indirect jump and call instructions. While these instructions are generally not used to direct the control flow of a binary, they are commonly used in calls to functions imported through the Import Address Table. This log is useful to identify potential Import Address Tables in the binary during the post-processing step.

5.2.3.3 **Execution blocks, executed regions, and modified memory regions**

Once the execution of the sample is considered to be finished, we generate a log containing information about the executed blocks and the modified memory for each layer. This structure allows us to reconstruct and visualise a graph containing useful information about the sample, and also to compute certain properties of the binary in the post-processing step. More specifically this log summarises the following information:

- **Execution blocks.** For each different execution block, we record the size of the block and its number of instructions.
- **Modified memory regions.** For each process and layer we compute the set of modified memory regions. Two memory writes are in the same region if the distance between the addresses involved in the operations is not higher than the size of memory 1 page (4 Kbytes).
- **Executed memory regions.** We compute the executed memory regions and assign to every region the memory type of the majority of its basic blocks.

5.2.3.4 System events and general information

As an interesting source of information, we also log the different events detected, which can help the analyst to understand how the different processes interact and communicate. Some of the most relevant events logged by our system are summarised below:

- Exceptions raised during execution.
- Process and thread creation, process termination.
- Remote process interaction events and their parameters (see Section 5.2.2).

Besides, we also log other information related to the execution of the sample:

- Number of instructions decoded.
- Tracing of the entry point of the binary.
- Modules loaded for each process.
- Processing time.

5.2.3.5 Analysis automatization log

Finally, once the execution of the binary is finished, we record the termination reason of the analysis and the time required by the emulator to analyse the sample.

5.2.4 Post-processing the execution trace

The monitored execution of the sample is followed by a post-processing of the trace. In this phase, we compute several properties of the packer. We also provide a visualization engine that allows to represent graphically the structure of the packer together with different useful information. This graphical information is intended to assist the analyst in understanding the complexity of the packer without requiring its reverse engineering.

Considering the information collected, we can compute the execution transitions between regions and layers in the binary (see Section 5.2.4.1), the number and size of unpacking frames (see Section 5.2.4.2), special properties for each region (see Section 5.2.4.3), potential Import Address Tables (see Section 5.2.4.4), and, finally, a graphical representation of the packer (see Section 5.2.4.5).

5.2.4.1 **Execution transitions**

Following the definition of execution transition provided in Section 5.1.3, we create a log that lists every transition produced during execution by traversing the execution trace. For each transition, we record the following values:

- Transition number.
- Process, layer, region and address of origin.
- Destiny process, layer region and address.

This transition log will be afterwards employed to compute the transition model of the binary.

5.2.4.2 **Unpacked and repacked frames**

In order to compute the unpacking frames of a binary, its memory footprint, and its granularity, we first compute the execution frames for each layer and region.

For each layer computed during the execution of the binary we define a shadow memory that covers the page aligned address space from the lowest address to the highest address occupied by the regions in the layer. Similarly, we define an equivalent shadow memory for each region. These shadow memories allow us to compute the number of unpacking frames and their size, and also allow us to detect if the code involved in the frames is repacked.

The shadow memory can be modelled as a finite state machine for every byte which is updated on every operation recorded in the execution trace (see Figure 5.4).

For every byte covered by an execution block at layer $\mathcal{L}_j$, there is an execution transition (x) in the finite state machine associated to $\mathcal{L}_j$. Besides, for every byte written in the address space of layer $\mathcal{L}_j$, there is a write transition (w) in the finite state machine associated to $\mathcal{L}_j$.

Accordingly, each byte can be in any of the following states:

- **Unknown** (O) Indicates the initial state of the memory.
- **Executed** (X) Indicates that the memory has been executed, without being previously written (this can only be true for the first layer of the packer).

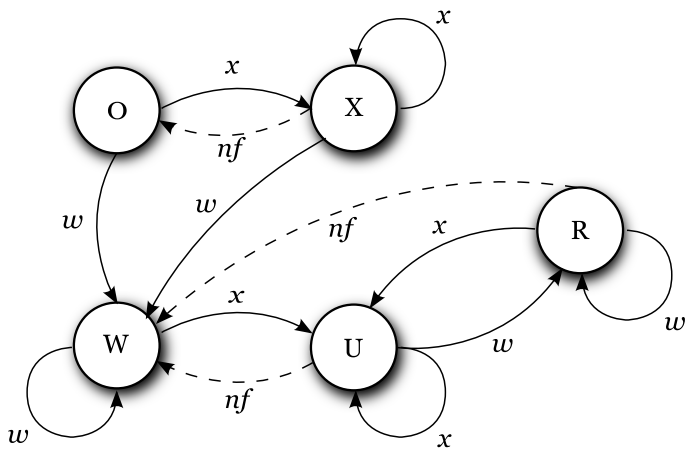


Figure 5.4: Finite state machine representing the memory state for each byte.

- **Written** (W) Indicates that the memory has been written but not yet executed.
- **Unpacked** (U) Indicates that the memory has been executed after being in the Written state.
- **Repacked** (R) Indicates that the memory has been overwritten after being in the Unpacked state. This may sound incorrect, since not all overwritten code is necessarily repacked. However, we discuss later how we distinguish between repacked code and new overlapping code.

The update operations described are defined according to the following intuitions:

i A memory address written at some point of the program can contain original code, intermediate data, or junk (over-written) code.

ii If a memory address is executed, then, at that point of the program, the address is in an executed state.

iii If a written address is executed, then that address has been unpacked, and it is in a written and executed state.

iv If an executed address is written, then we do not consider that the address, at its current state, is in an executed state because its content has been modified.

v If a unpacked memory address is overwritten, the we consider it is repacked (and therefore, is not in an executed state any more).

In order to correctly distinguish the memory writes recorded in the last execution frame from the memory writes recorded before, we keep an extra bit that indicates whether the corresponding byte has been modified in the last frame or not: the frame bit W_f.

Every time there is a transition to the U state for any byte in the layer or region with the frame bit set to 1, we consider that an unpacking frame has occurred and compute all the information corresponding to that frame. Also, the frame bit is cleared for all the addresses in the shadow memory. If the execution continues and further written blocks are executed, we do not compute a new frame because all the bytes in the memory have the frame bit set to 0. The next frame will not be computed until new writes for a region or layer are followed by an execution of those memory addresses.

We also add a bit to every execution block that indicates whether it has been unpacked in the current frame or not: U_f. This bit allows us to correctly compute the number of execution blocks and instructions unpacked in each frame.

During trace post-processing we maintain a map of executed blocks.

$$\mathcal{EB} : (\text{Address}, \text{Sequence}) \rightarrow (\text{Size}, \text{Number of instructions}, U_f)$$

Finally, we define the following rules in order to update the shadow memory depending on the memory state and operation recorded in the trace. Every memory write triggers a write operation $\mathcal{W}_a$ for each address a updated.

$$\mathcal{W}_a : \left| \begin{array}{lcll} x[a] & \leftarrow & 0 & \\ w[a] & \leftarrow & 1 & \\ w_f[a] & \leftarrow & 1 & \\ r[a] & \leftarrow & 1 & \text{if } x[a] = 1 \text{ and } w[a] = 1 \end{array} \right.$$

Every execution block recorded $\mathcal{EB}(\text{Address}, \text{Sequence})$ triggers an execution operation $\mathcal{X}_a$ at every address a covered by its instructions.

$$\mathcal{X}_a : \left| \begin{array}{lcll} x[a] & \leftarrow & 1 & \\ r[a] & \leftarrow & 0 & \\ \textit{Log new frame} & & & \text{if } w_f[a] = 1 \end{array} \right.$$

These rules summarise all the transitions in the finite state machine presented in Figure 5.4. Additionally, whenever an execution block is found in the trace, it is updated accordingly.

$$u[a] \leftarrow 1$$

Also, whenever a new frame is detected during the post-processing of the trace, we perform several computations that allow us to measure the size of the frame in terms of unpacked execution blocks, repacked execution blocks, and written memory size. This computation requires several steps:

1. Compute unpacked memory blocks. First, we compute the different blocks (sets of contiguous memory addresses) that are either in the U or the R state.

2. Considering this information, we count the number of unpacked execution blocks, their size in bytes, and the number of instructions involved. These execution blocks overlap the unpacked memory blocks and have their U_f bit set to 1.

3. Compute the repacked memory blocks. In a similar way, we compute the memory blocks that have been repacked in the current frame (bytes at the R state).

4. Compute the repacked execution blocks. We measure the number of execution blocks repacked, their size and number of instructions, considering that they overlap the repacked memory blocks and have their U_f bit set to 1.

5. Compute the written blocks in the current frame. In order to do this, we consider the W_f bit for every memory position.

6. Compute the size written in the current frame, which is the sum of the size of the blocks computed in the previous step.

7. Clear the frame information: bit W_f for all the positions in the shadow memory, and U_f for all unpacked execution blocks.

8. Clear the x and r bits of the shadow memory. Consequently, all the positions in the shadow memory in the U and R state, will be updated to the W state. The positions in the X state are updated to the 0 state.

Whenever a new frame is detected, we clear the x bit of all memory positions in the shadow memory. Consequently, the memory positions are not in an unpacked state any more. During the next frame, new blocks of memory will be written, and also some memory blocks will be executed. All the executed blocks, regardless of the moment in which they have been written will be considered unpacked blocks of the frame. This implies that, if during the execution of a frame part of the blocks executed were modified in previous frames, we also consider those blocks to be part of that frame.

This approach has a second implication. Since the unpacked bytes (U state) are transformed into written bytes (W state), any further write to these bytes will not transform them into repacked bytes. In contrast, these bytes will be considered unpacked memory of the next frame. While this approach is restrictive, we consider only to be repacked the memory which is overwritten before starting the next frame of execution.

Finally, by clearing the R bit in each frame event, the corresponding memory positions are updated from state R to state W. After the frame, if those bytes are written, their state will not be modified, and in case of execution, they will be updated to the U state, but will not be considered repacked bits any more.

Following the method described, once the execution is finished, we know the number of execution frames for each region and layer. Layer 0 and the regions contained in it will have 0 unpacking frames. The simplest possible packer, with 2 layers, will have 1 frame in layer 1.

5.2.4.3 **Regions**

For each executed memory region computed during execution, we compute some additional data useful in order to measure the complexity of the packer.

Each time an system API is called during the execution of the binary we create an entry in the trace for such call, including information about the execution block that triggered the call. In this way, when we post-process the trace we can compute, for each region, the total number of API calls and the number of different functions called.

Also, there are several interesting API calls commonly used as heuristics to detect the original entry point. For each region, we compute a flag indicating the execution of each of these API functions (`GetVersion`, `GetCommandLine` and `GetModuleHandle`).

Finally, when a memory write operation is processed in the trace, we

consider the execution block that originated the call and find the region in which it is included. In this way, we consider as a *leaf* every region that does not write to another region that is executed. These regions are susceptible of containing the protected code (but not necessarily, since some routines of the packer might contain other type of code such as anti-analysis tricks).

5.2.4.4 **Potential Import Address Tables**

We collect information about indirect calls in order to compute potential import address tables in the binary. This technique also allows to detect situations in which the execution of the original code is interleaved with the unpacker code if IAT hooking is employed. In these situations, the IAT employed by the original code points to a routine in the unpacker code which will execute its own code (such as integrity checks, or anti-debugging and anti-analysis techniques) before it redirects the execution to the actual API function.

Once the execution of the binary is finished, we locate sets of memory addresses used in indirect calls and jumps not distant more than one memory page (4 Kbytes). These sets of addresses might contain potential IATs.

5.2.4.5 **Visualization**

In addition to the all the information collected during binary tracing and post-processing, we represent graphically part of the information in order to provide the analyst a quick overview of the structure of the packer. Although data visualization techniques can help the analysis task, few approaches are focused on the analysis of run-time packers. Vera [QL09] allows to represent graphically the execution trace of a binary with an instruction granularity. Nevertheless, we believe that packer analysis can benefit from a coarse-granularity approach, specially when the complexity of the packer is high, (e.g., with multiple unpacking layers and interaction between several processes). In this way, we propose the use of fine-grained monitoring and coarse-grained visualization of the execution of the binary in order to provide the analyst a precise but interpretable source of information for the reverse engineering task.

More concretely, we visualize the execution trace of the binary as a graph structure of processes, layers, and regions. In order to render the graphs, we use the *graphviz* tool, but other more powerful tools could be used to represent the data in an interactive fashion.

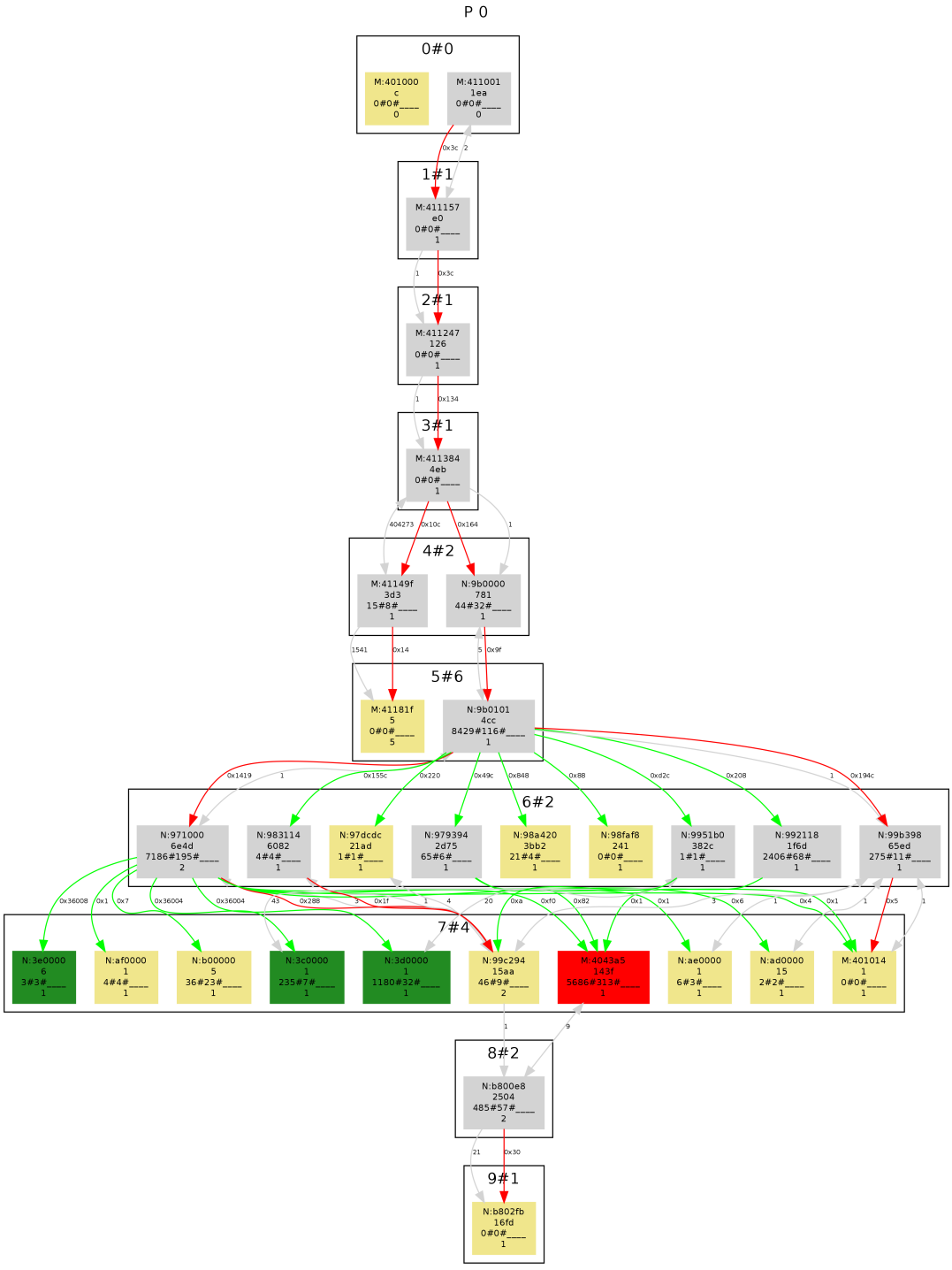


Figure 5.5: Graph generated for the ASProtect 2.1 packer.

The graphs are divided into processes, layers, and regions. Figure 5.5 shows an example of a complex packer. In this case, the original code is located at layer 7. There are still 2 deeper layers with code belonging to the unpacker. Also, layer 7 contains code of the packer together with the original code.

For each process, we show the process number. For each layer, we show its layer number and the number of unpacking frames, and finally, for each region, we show the following information:

- Memory type.
- Start address and size.
- Total number of API calls, number of different API calls, and flags for special API calls: `GetVersion`, `GetCommandLine` and `GetModuleHandle`.
- Number of frames.

We draw edges that connect the different regions in the graph, showing their relationship in terms of write and execution transitions.

The regions follow a color scheme in order to quickly recognise several important parts of the binary.

- Gray nodes represent memory regions involving execution blocks that modified other executed memory regions. The code contained in these regions might probably be part of the unpacking routine of the binary.
- Yellow nodes represent memory regions that do not contain any execution block that modified another executed memory region. The code contained in these regions can be part of the unpacking routine, or part of the original code of the binary.
- Green edges represent memory writes from one region to another executed region. Every time there is a memory write, the code executed in the region is associated to the next execution layer. Therefore, there are no write edges to regions in the same layer or previous layers. In order to make feasible rendering and visualising the graph, we only represent memory writes to the following layer. In the cases in which there is an execution transition from a memory region to another region in the following layer, the edge is displayed in red. Also, the number of bytes written is specified for each edge, indicating the number of bytes into the region written by the previous layer.

- Gray edges represent execution transitions between executed memory regions. In this case, we only represent transitions to the following and the preceding layer, since the representation of every transition in the graph would make infeasible its representation and visualisation. Also, execution transitions derived from the context switch between processes are not visualised for simplicity. More concretely, we adopt a heuristic to discard such transitions. We only visualise the execution transitions that follow an inter-process memory write, since in these cases we assume that the execution of both processes is synchronised.

Finally, in some cases we paint the regions in different colors in order to highlight special properties. In this way, regions that contain memory written remotely are painted in green, while regions that contain the last instructions executed are displayed in red.

5.2.5 Computation of packer complexity

The last step involved in the analysis of a binary is to finally measure its structural complexity. In order to do this, we collect all the information recorded during the execution tracing and post-processing in order to asses the level of sophistication of the structure of the packer.

The first set of properties we compute is the number of processes and threads of the packer, and the inter-process communication mechanisms employed for unpacking.

- **Number of processes.** First, we consider the number of processes created as a consequence of the execution of the binary. We monitor the execution of each of those processes. Second, we measure the number of processes that interact between them. We consider that two processes interact if there are write operations to the address space of each other. Finally, we compute the number of existing processes to which the binary injects some memory. Although this behaviour is not typical in off-the-shelf packers, it is quite common for malware.

- **Number of threads.** For each binary, we count the number of threads created during its execution.

- **Usage of `WriteProcessMemory` and `ReadProcessMemory`.** During the execution of the sample, we record all the calls to these functions.

Nevertheless, not every call to these functions is an indicator of inter-process communication. Whenever a process is created, a series of calls to the `NtWriteVirtualMemory` function follow the `ZwCreateProcess`, targeting the addresses 0x10000, 0x20000, and 0x7ff00000 (with the bit-mask 0xFFF00000). Apart from filtering these calls, in all the inter-process communication events, we only consider those function calls that affect to executed memory regions.

- **Other inter-process communication events.** Shared memory section mapping, memory writes to shared memory mapped sections, remote memory writes by reading files, and memory deallocation of unpacked memory.

Besides, we also compute another set of properties that quantitatively measure the size of the execution graph computed.

- Number of instructions decoded.
- Number of layers.
- Number of regions.
- Minimum, average, and maximum number of execution blocks per layer.
- Minimum, average, and maximum number of instructions per layer.
- Minimum, average, and maximum size of execution blocks per layer.
- Minimum, average, and maximum number of regions per layer.

Additionally, we compute several other indicators that measure the complexity of the structure.

- Number of transitions to deeper and shallower layers.
- Number of layers with more than one unpacking frame.
- Number of exceptions.

Finally, we categorise the sample according to the taxonomy defined in Section 5.1.

We consider that a run-time packer will exhibit at least 2 layers of code. Therefore, if the execution of the sample is successful and only one layer is executed, we consider the sample as non-packed, because it does not show any unpacking behaviour.

If a sample is packed, we categorise the binary in one of the 6 types of packers defined in our taxonomy.

The information extracted by our system allows us to correctly distinguish, in an automated fashion, between packers of Type-I, Type-II, and Type-III and between Type-V and Type-VI. However, in order to correctly distinguish Type-III from Type-IV, and Type-IV from Type-V and Type-VI, we need to clearly identify the original code of the binary and separate it from the code of the packer. Unfortunately, this task does not always have a clear and sound solution.

It might seem reasonable to think that the original code always resides in the deepest layer of the structure. Nevertheless, our experiments show that this is not necessarily true. Also, there are cases in which part of the packer code is unpacked at the same layer as the original code, making the problem even more complicated. If these pieces of code are mangled together or located at the same address space it is practically impossible to differentiate between the two types of code.

For this reason, we implemented a set of heuristics to distinguish the original code from the rest of routines based on the assumption that both types of code do not reside in the same address space or are, at least, separated by a significant distance in memory. Both codes may reside in the same layer, but not in contiguous addresses in memory. We empirically set this distance to 10 pages during our experiments.

Also, we base our heuristic on the concept that, if a certain part of the code modifies some other code that is later executed, this region and all the contiguous code is part of the packer. Also, we assume that the original code will be located at the module address space of the binary, and not in the heap, stack, or memory dynamically allocated by other means.

In this way, we can initially label all the executed memory regions that do not write to other parts of the code as potential candidates to hold the original code of the program.

Finally, it would be possible to find a packer that places the unpacking routines near the original code of the application. In this case, our heuristic would incorrectly flag the original code as packer code. Considering these

possible cases, we adopted a conservative approach to distinguish the different packer types in our taxonomy.

5.2.5.1 **Differentiating Type-III from Type-IV**

If we can discriminate the original application code from the unpacking routines, differentiating between cyclic transition models and interleaved packers is straightforward.

1. If our heuristic does not label any region of code as potentially original code, then, for the lack of evidence, we classify the sample as Type-III (i.e., cyclic transition model).

2. If the last instructions executed in the trace are part of the packer while previous instructions were part of the original code, then we consider the sample as interleaved (Type VI).

3. If the last instructions executed in the trace are part of the original code, we iterate the trace in reverse order until we find some code classified as packer code. If we find potentially original code executed before this point, then we consider the sample as interleaved (Type-VI). Otherwise, we consider it cyclic (Type-III).

5.2.5.2 **Differentiating Type-IV from Type-V and Type-VI**

In order to differentiate between Type-IV and Type-V/Type-VI packers, we first compute all the memory regions that present several frames and potentially belong to the original code. In order to discard simple obfuscations, we only consider memory regions with a size higher than 32 bytes.

If the majority of the regions potentially belonging to the original program present multiple frames, we consider that the packer presents an on-demand unpacking algorithm (Type-V or Type-VI).

To property distinguish Type-V from Type-VI, we analyse the unpacked and repacked regions of memory recorded for each frame detected during trace post-processing that affects the unpacked regions considered original code. If there are repacked blocks for these regions, we consider the sample as shifting-decode-frames. Otherwise, we consider the sample as incremental.

Finally, we consider the size of the memory written on each of the unpacking frames. If the majority of the frames present an unpacked size

multiple of 4K (the size of a page), we consider the packer to have a page-granularity. If instead the average number of basic blocks per frame is one, then we assign the granularity of basic-block. In any other case (e.g., when the size of the unpacked frame is always different) we label the sample as function granularity. This category includes the packers that protect each function separately, but also comprehends other types of granularity that protect pieces of related code in any other way.

5.3 Measuring run-time packer complexity

Our approach allows us to measure the complexity of run-time packers based on the information collected in our analysis platform. This section presents the results obtained by using our system to study two different datasets: (i) a set of off-the-shelf packers, and (ii) a set of custom packers.

5.3.1 Datasets

The dataset composed by off-the-shelf packers contains 762 samples which cover 389 unique packers. Some popular packers are present multiple times to cover different versions or different packing behaviours obtained by using different parameters or configuration options.

In order to measure the prevalence of these techniques in custom packers, we selected a set of samples from Anubis, a public sandbox that has been running on-line since 2007. More concretely, we first retrieved a list of unique PE files that were not labelled as packed by Sigbuster, the signature based packer detection tool employed by the sandbox to label the binaries.

More concretely, we obtained over 1000 samples per year between mid 2007 and mid 2014. The samples are equally distributed for each month of the year, based on their submission time. Moreover, the dataset was balanced to contain only one sample of the same malware family per month, in order to avoid biasing the dataset towards very common polymorphic families such as the Zeus botnet.

In order to conform this dataset, we queried the VirusTotal public service in order to retrieve complementary information about the samples. More concretely, we retrieved the samples present in the database for each day of the year during the period from 2007 to 2014, and then queried VirusTotal discarding the samples that did not met the following conditions:

Table 5.10: Sample distribution of collected custom packers.

	Packed	Not packed	% Packed
2007	213	265	80.38%
2008	903	1148	78.66%
2009	923	1167	79.09%
2010	816	1008	80.95%
2011	868	1127	77.02%
2012	1005	1259	79.83%
2013	908	1168	77.74%
2014	452	587	77.00%

- At least one of the sections with an entropy higher than 7.
- Not considered malicious by at least 3 antivirus solutions.
- Not labelled as packed by any of the three tools employed by VirusTotal to detect known packers: PEiD, F-Prot, and Command.
- Not belonging to a malware family previously selected for the same month.

The process followed to select the samples was the following: for each day of the year, we retrieved one sample and queried VirusTotal. If the sample met the aforementioned restrictions, we added it to the dataset, if not, we selected another sample for the same day. When there were not enough samples for one day, we selected the rest from the next day.

In order to determine the family of the sample, we elaborated a list with the 1000 most common families reported by the antivirus solutions for the queried samples. Then, we labelled each sample with the family reported by the majority of the antivirus products included in the report from VirusTotal.

Table 5.10 shows the distribution over the years of 7729 working malware binaries.

5.3.2 Analysis infrastructure

We analysed every sample using our framework. More concretely, we ran the experiments on a distributed set-up of 22 virtual machines with 2 processors and 4 gb of RAM, each running one instance of the experiment framework.

The analysis of the samples was automated and each sample was run for a minimum of 5 minutes until one of the following conditions was satisfied:

- All the processes under analysis terminated their execution.
- An exception was produced, and it was not recovered in a 2 minute time-out. In order to detect exceptions, we monitored the execution of the `KiUserExceptionDispatcher` function at `ntdll.dll` system library.
- The process reached a stability point. A process reaches its stability if it does not consume user-time in a 2 minute period. In order to monitor the activity of the monitored processes, we inspected every 2 minutes the user-time consumed by each of the processes in the monitored system by retrieving their corresponding EPROCESS structure from the kernel.
- A maximum time-out of 30 minutes was reached.

In order to maintain this experimentation framework, we developed our own set of scripts to manage a task queue, a task dispatcher, and a set of 22 workers that processed each sample and submitted the results to a central file server.

5.3.3 **Analysis of off-the-shelf packers**

The first interesting result we observed in the group of off-the-shelf packers is the fact that 559 of them (81.6%) only employ one process for unpacking, 121 (17.6%) use two concurrent processes, and five adopted more than two. The packer complexity is summarised in Table 5.11 for both off-the-shelf and custom packers. The results show that the most prevalent type of packer is Type III (i.e., more than 2 layers and cyclic transition model), followed by the most simple class, Type I, which represents one quarter of the samples analysed. We can appreciate that the number of cases with more than 2 layers and a linear transition model (i.e., no cycles between the unpacking layers) is below 10%. Therefore, we can affirm that the packers with more than 2 layers tend to present cycles between them. Finally, we can appreciate that there is a significant number of packers with complex structures that either interleave the original application code with the packer code (12.6%) or perform some kind of on-demand unpacking (2.7%).

Table 5.11: Summary of the packer complexity of the studied samples.

Type	Off-the-shelf	Custom packers
I	173 (25.3%)	443 (7.3%)
II	56 (8.2%)	752 (12.4%)
III	352 (51.4%)	3993 (65.6%)
IV	86 (12.6%)	843 (13.8%)
V	6 (0.9%)	46 (0.8%)
VI	12 (1.8%)	11 (0.2%)

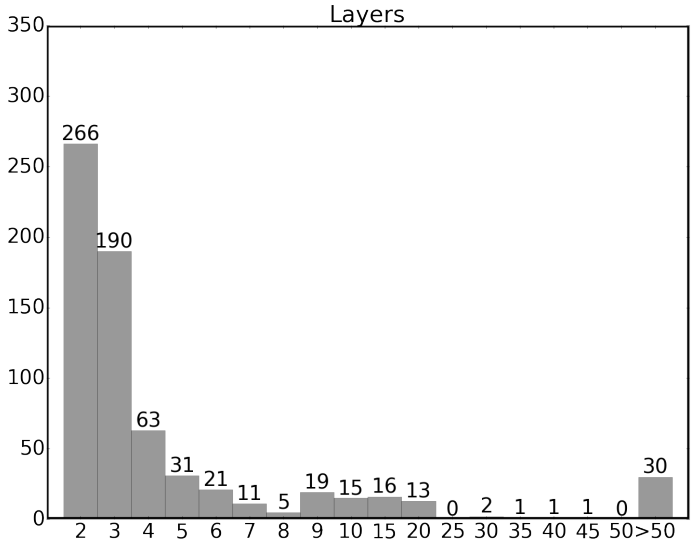


Figure 5.6: Number of layers of the packer.

Figure 5.6 summarises the number of layers observed in the dataset. While the majority of packers adopt less than four layers, there is a significant number of packers (4.4%) that use more than 50 layers. Even more interesting, almost 10% of the packers (65) in this dataset did not have the protected code in the last unpacked layer.

Regarding the number of transitions to shallower layers, one third of the packers did not present any cycle (and thus, have a linear transition model). Besides, 15 packers presented more than 1 million cycles (transitions to shallower layers).

Finally, Figure 5.7 shows the prevalence of the different interprocess communication techniques monitored for this set of packers.

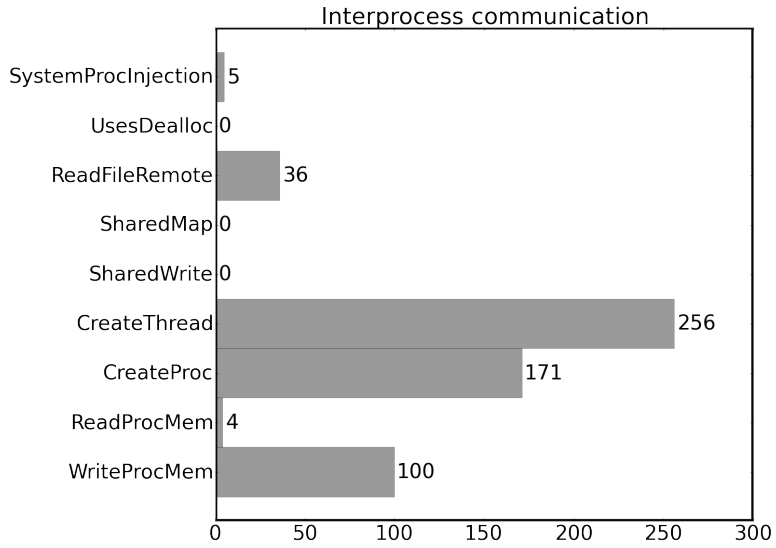


Figure 5.7: Interprocess communication techniques observed.

In this dataset, we did not find any sample using shared memory mapping, writing to shared memory regions, or deallocating memory of unpacked regions. The most common interprocess interaction techniques were the creation of files (that are afterwards executed), and the use of the Windows API `ReadProcessMemory` and `WriteProcessMemory` for process interaction. Five samples were reported to use injection techniques to processes not directly created by the process itself. Also, we can observe that a significant number of samples (171) create several processes. Thread creation is an even more common technique with 256 samples in the dataset.

5.3.4 Off-the-shelf packer distribution

In the previous section we have analysed the complexity of different off-the-shelf packers. In order to understand the relevance of these results, we obtained access to the Anubis malware sandbox database (containing over 60 Million unique samples) to measure the number of malware samples protected with each packer, labelled by the signature based packer detection tool present in the sandbox.

Table 5.12 shows the distribution of the most common packers over the years. Since the tool is capable of identifying the polymorphic protection engines adopted by famous worms, we indicate in the first row the percent-

Table 5.12: Distribution of known packers over the years.

	2007	2008	2009	2010	2011	2012	2013
Pol. worms	15.0%	17.3%	9.5%	14.6%	17.9%	15.3%	1.3%
Packers	11.7%	4.6%	4.0%	2.6%	0.8%	0.7%	0.1%
UPX	30.0%	49.1%	56.8%	41.4%	52.7%	55.3%	55.4%
PECompact	6.8%	5.5%	3.0%	4.7%	8.8%	6.3%	3.7%
Aspack	3.8%	4.6%	5.7%	4.6%	5.7%	5.0%	8.7%
FSG	11.4%	9.5%	0.9%	0.8%	0.9%	0.6%	0.8%
Asprotect	4.7%	1.5%	1.6%	3.2%	2.2%	2.3%	2.2%
NSPack	3.9%	2.1%	1.2%	1.3%	1.1%	1.0%	1.4%
Themida	6.0%	2.4%	0.8%	0.9%	0.6%	0.6%	0.6%
Upack	2.9%	2.3%	1.4%	0.3%	0.3%	0.4%	0.9%
Xtreme	0.5%	0.6%	0.3%	1.3%	0.5%	0.3%	0.2%
Others	19.4%	8.5%	5.2%	4.9%	3.8%	2.9%	3.7%
Unknown	2.7%	4.6%	5.3%	3.9%	2.4%	2.0%	2.3%
Installer	7.8%	9.2%	17.9%	32.6%	20.9%	23.4%	20.2%

age of those samples. The second row of the table shows the percentage of samples which are packed with an off-the-shelf packer. This last set of samples is divided into different categories in the lower part of the table. We can see a list for the 9 most prevalent packer tools in the database, the percentage of well-known packers not listed in the table (Others), the percentage of packers labelled by the tool as *Unknown* packers, and finally the percentage of tools labelled as Windows installer tools.

We can observe that the number of samples detected as packed with well-known packer tools has significantly decreased over the years. This trend might be caused by the fact that the packer signature database is not up-to-date with recent packer families, or that malware writers now prefer to employ their own custom unpackers or to rely on simple packers (such as UPX) that do not raise suspicion on their programs. Also, we can notice that the UPX packer, a very simple Type-I packer dominates the off-the-shelf packer landscape.

5.3.5 Analysis of custom packers

The packers analysed in the previous section are well-known tools that can be recognised by signature-based detection tools such as PEiD or Sigbuster. Nevertheless, a significant number of malware samples are protected by

custom protection engines or modified versions of existing packers in an effort to complicate reverse engineering.

To study the complexity and the characteristics of these packers we run our analysis tool on 7,729 malware binaries uniformly distributed over the past seven years. Despite all the samples had a section with entropy higher than 7, 6,088 samples presented an unpacking behaviour during our analysis.

Table 5.13: Custom packer complexity over the years.

Type	2007	2008	2009	2010	2011	2012	2013	2014
I	5.2%	8.1%	6.1%	8.1%	9.2%	4.6%	8.3%	8.0%
II	18.3%	15.6%	10.2%	15.4%	10.5%	8.9%	11.5%	15.0%
III	63.8%	64.6%	71.2%	62.5%	64.4%	69.0%	63.7%	61.3%
IV	11.3%	11.4%	12.1%	13.7%	15.7%	15.1%	15.0%	15.0%
V	-	0.2%	0.1%	-	-	2.3%	1.7%	0.7%
VI	1.4%	0.1%	0.3%	0.2%	0.2%	-	-	-

Table 5.11 shows the packer complexity classes in both datasets, off-the-shelf and custom packers. Custom packers show a higher prevalence for Type-II and Type-III, while very simple packers (Type-I) and very complex ones (Type-VI) are more common in the off-the-shelf dataset. Also, we can observe that the number of Type-IV packers (i.e., interleaved execution of original application code and unpacking engine) is almost equivalent for both datasets. Table 5.13 shows the evolution of custom packer complexity over the years, and Figure 5.8 summarises it by plotting the average packer complexity for each quarter of the years covered by the study. The data shows no clear trend, with all the complexity classes remaining constant over the past seven years.

Figure 5.9 and Figure 5.10 plot the overall number of layers and inter-process communication methods found in the dataset of custom packers. Like in the case of off-the-shelf packers, the majority of custom packers use few layers. Nevertheless, in contrast to off-the-shelf packers, there is a significant number of samples that present between 3 and 6 layers. From the total of 6088 samples, 5226 present the original code in the last level. Regarding inter-process communication, we can observe an increment in the use of system process injection and unpacking by using external files, caused by the presence of malware that injects the code to other processes or drops files that are afterwards executed. In this case, there is a marginal number of samples reported to use shared memory or memory deallocation

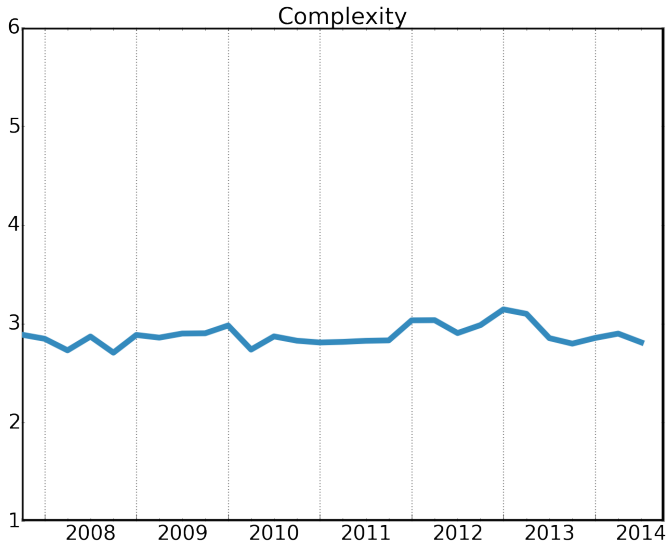


Figure 5.8: Average complexity used by custom packers over time.

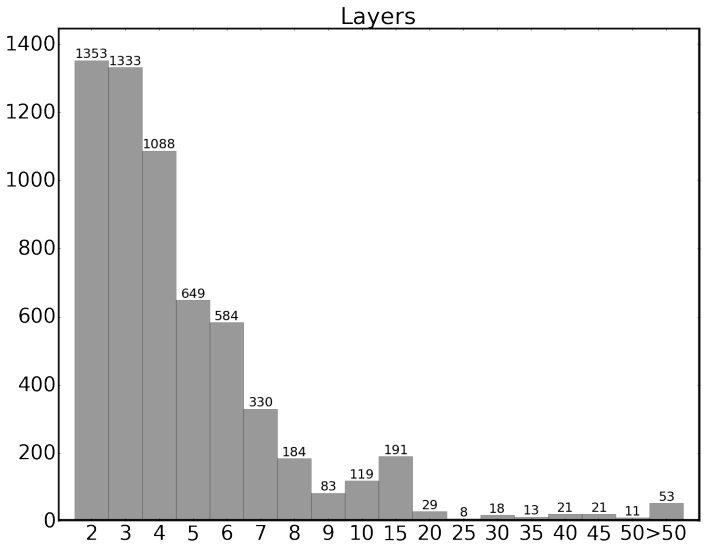


Figure 5.9: Number of layers used by custom packers.

as part of the unpacking process. 3260 samples created processes during their execution, and 4509 created one or more threads. Nevertheless, this behaviour might not be, in all the cases, part of the unpacking process.

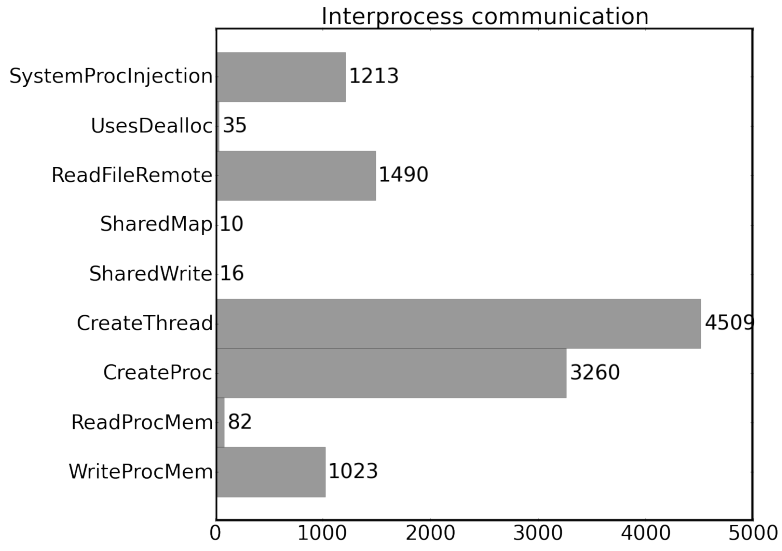


Figure 5.10: Interprocess communication techniques found in custom packers.

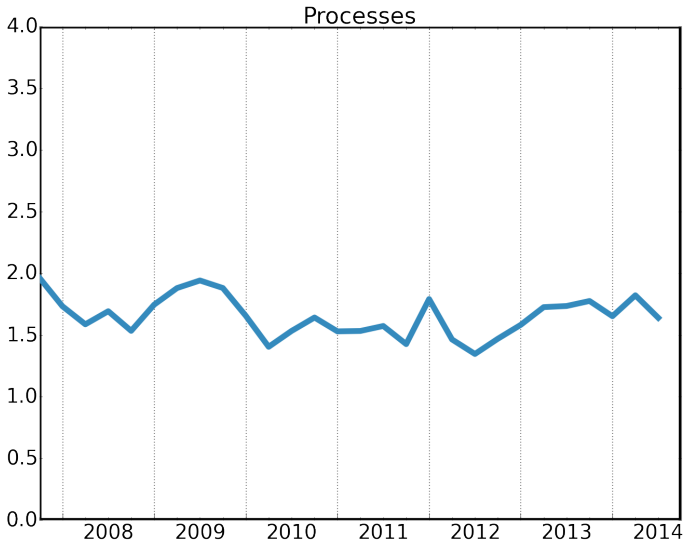


Figure 5.11: Average number of processes used by custom packers over time.

Figure 5.11 shows the evolution of the number of processes considered to be part of the unpacking process (i.e., processes with some sort of un-

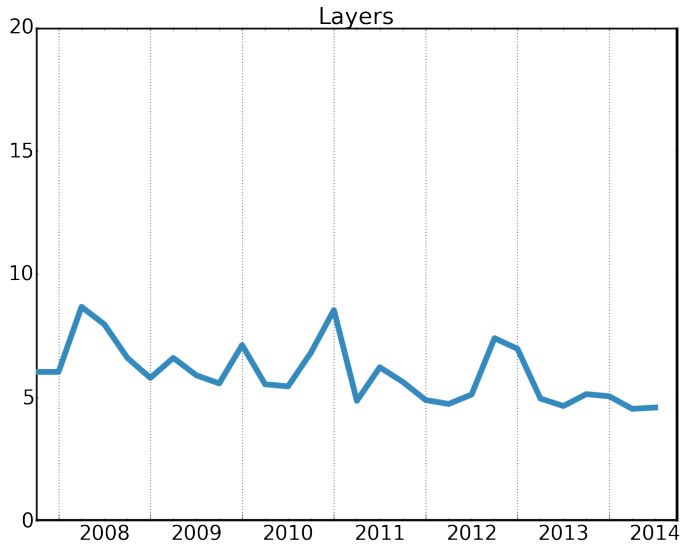


Figure 5.12: Number of layers used by custom packers over time.

packing related interaction). Figure 5.12 shows the evolution the number of layers over time. In both cases we do not observe any significant variation in our experiments. Combining this information, we can conclude that the *average* custom packer in our dataset presents a multi-layer unpacking routine with a cyclic transition model.

5.4 Conclusions and discussion

In order to answer the research questions established for this study, in Section 5.3 we have measured the complexity of off-the-self packers and custom packers.

What is the maximum level of sophistication of run-time packers?

Based on our results, we can conclude that the average packer has a Type-III complexity. Also, we can observe that the majority of samples that present more than 2 layers and no interleaving of the original application code and the unpacking routine (i.e., Type-II and Type-III) present cyclic transition models. This fact reveals that the majority of these cases do not correspond to binaries packed multiple times by Type I packers, but to ob-

fuscated unpacking routines located at different layers, or binaries packed multiple times in which at least one of the layers corresponds to a Type-IV packer. Finally, for both datasets there is a significant number of samples that interleave the execution of the original code, and a few samples that unpack the code on demand, showing some kind of partial code revelation.

How widespread are custom packers in collected malware, and, how sophisticated are they compared to off-the-shelf packers?

The results show that, by simply looking for samples with a high entropy in one of their sections not labelled as packed by any signature-based tool, the number of samples that actually showed an unpacking behaviour is relatively high (around 80%). While Type-I packers are very common for off-the-shelf packers, in the case of custom packers there is a relatively low number of samples in this category - showing that malware writers look for more complex protection schemes. Also, the number of Type-V and Type-VI packers is slightly higher for the set of off-the-self packers. Finally, we can conclude that custom-packed binaries rarely employ these complex protection techniques.

How has the packer complexity evolved over the years?

We did not observe a clear evolution in the packer characteristics and complexity over time. In fact, the complexity of custom packers has remained pretty stable during the past 7 years, and apparently malware writers are not developing more complex packer structures. Despite it is relatively easy to develop more complicated packing techniques, apparently malware authors are not adopting them for the protection of their binaries. This fact might be a sign that current automated scanners can be evaded by employing simple Type-III packers.

How does this complexity affect the common assumptions of generic unpackers?

Generic unpackers rely on a number of assumptions about the packer structure that, as we have seen in our study, are not always true for the samples observed in the wild. First of all, there is a significant number of packers with more than 2 layers. This implies that every unpacked code

is not necessarily part of the original code. Despite some approaches like Renovo [KPY07] consider cases with several layers of protection, not many studies have focused on determining in which layer resides the original code, and it is not clear how to proceed in these cases. In this case, things are complicated by the fact that around 10% of the off-the-shelf packers and 14% of the custom packed malware did not have the original code in the last layer.

Many unpackers try to identify the *tail-jump* to dump the protected code at the appropriate moment. Type-IV, Type-V and Type-VI packers complicate this operation. Even though Type-V and Type-VI packers are not very common in the wild, we have found several examples in our dataset. These packers require a more complex development and also impose a run-time overhead that may not be desired by malware writers. It is important to highlight that none of the generic unpackers proposed to date has dealt with this kind of packers.

5.5 Related work

Now that we have described and evaluated our run-time packer complexity analysis framework, we can compare it to previous work.

Generic unpackers have generally focused on determining the correct moment to dump the original code from different perspectives.

i **Discovery of hidden code.** PolyUnpack [RHD+06] is proposed as a tool based on the static analysis of the packed binary, and the dynamic execution of the sample in order to discover executed instructions that were not originally present in the binary. By applying a disassembly process starting from the executed instructions, the model can be used to find memory areas containing hidden code. In order to handle programs protected multiple times, it proposes to re-execute the resulting binary again with the tool. Nevertheless, this approach does not address the problem of distinguishing where the unpacking code and the original code reside, or where the original entry point of the binary is. Also, it assumes that the different unpacking layers are isolated and independent, which is something that might not be true for complex packers.

ii **Memory writes followed by memory execution.** Renovo [KPY07] is a generic unpacker that applies fine-grained granularity tracing. When

the binary is under execution, it maintains a shadow memory with the memory addresses that have been modified. Whenever a modified address is executed, it dumps the memory contents of the process and cleans the shadow memory to continue with the execution of the sample. Using this approach, it can dump an image of the process each time some written code is executed. Nevertheless, it does not help in the task of identifying the original entry point, since it assumes that the packer has a number of N independent layers of unpacking code, and every time the execution hits a written memory address, a new layer is discovered. Besides, it cannot distinguish the unpacking code and unpacked code.

iii **Statistical properties of the code.** Eureka [SYS+08] proposes the use of bi-gram frequency based statistical analysis in order to determine if certain regions of code are packed or not during the unpacking process of the sample. Other approaches [CX10] measure the change in the entropy of the binary in order to determine the appropriate moment to dump the contents of the binary. This kind of approaches assume that there is a significant entropy change in the code when it is unpacked, which is something that can be easily evaded by using certain encodings for the data.

iv **System-call based heuristics.** Similarly, Eureka [SYS+08] assumes that the code is unpacked in the moment the program reaches the `NtTerminateProcess` API call. While for simple packers this assumption may hold, other packers apply repacking to certain regions of code. Omniunpack [MCJ07] keeps track of modified and executed memory pages, and calls an antivirus over those pages whenever a dangerous system call is executed. Unfortunately, this approach does not measure the complexity of the packer. Universal PE Unpacker considers the execution of the `GetProcAddress` function call as a heuristic to determine the end of the unpacking routine. Nevertheless, some packers do not call these functions to recover the Import Address Table, but instead only need the base address where the DLL is loaded in order to parse its export table and update the import table by themselves. As we see, system-call based heuristics are generally focused on the identification of the original entry point, but they are very dependent on the implementation of the packer.

While these models are effective for simple packers, they are not aimed

at measuring and presenting the structural complexity of the packer.

The concept of layer was addressed by some of the generic unpackers proposed in the past (e.g., Renovo [KPY07]), but it was not formalised until Debray and Patel [DP10] proposed an approach to extract the unpacking code of a binary based on dynamic slicing techniques that considered the fact that a binary might be protected by many layers of unpacking code. While this approach considers a model based on execution layers, it is focused on the extraction of the unpacking code. Nevertheless, the model is not focused on measuring the complexity of packers.

Besides, Marion and Raynaud [MR13] propose a a formal definition of the concept of execution layers as *code waves*. They define a formal language to represent the self-modifying code in a binary, and describe a set of first-order logic rules that can be applied to their model in order to test certain properties, like self-modifying code. Nevertheless, in their model they do not represent and address the technical challenges involved in the analysis of complex packers, that may not follow the classic assumptions.

Although both approaches defined a concept similar to our layers (under different names like phases or waves), these definitions differ from ours in the following way.

Debray et al. [DCT08, DP10] proposed a formalization of the semantics of self-unpacking code, and modeled the concept of execution phases. In their model, a phase involves all the executed instructions written by any of the previous phases. This concept is related to our definition of execution frames, with the difference that, in our model, execution frames only occur in the context of a single unpacking layer. Guizani et al. [GMRP09] proposed the concept of waves. This first definition of waves is equivalent to our concept of execution layers. Later, Marion et al. [MR13] proposed a different formalization in which they modified the semantics of waves, proposing a model similar to the phases proposed by Debray et al. Some of these publications have measured the number of layers present in malware samples. Nevertheless, they do not cover other complexity aspects such as the transition model, execution frames, or code visibility. Moreover, our model differentiates between the concept of unpacking layers and unpacking frames, allowing us to compute different properties that can be combined to provide a complexity score based on the class of the packer.

5.6 Summary

In this chapter we have presented a packer taxonomy to measure the complexity of run-time packers. We also developed an analysis framework that we evaluated on two datasets: off-the-shelf and custom packers.

The results show that, while many run-time packers present simple structures, there is a significant number of samples that present more complex topologies. We believe that this study can help security researchers to understand the complexity and structure of run-time protectors and to develop effective heuristics to generically unpack binaries.

«A man does not climb a mountain without bringing some of it away with him and leaving something of himself upon it.»

Martin Conway (1856–1937)

Chapter

6

Generic unpacking of samples with partial code revelation

Generic unpacking techniques have been widely studied in the literature. Some of the approaches proposed adopt static analysis techniques but the most successful approaches are based on dynamic execution. More specifically, existing approaches have focused their efforts in (i) making the platform resilient to anti-analysis techniques [DRSL08], (ii) tracing the execution of the binary at different granularity levels [RHD+06, KPY07] (iii) adopting different heuristics to detect the original entry point of the binary [CX10], or to dump the code at the appropriate moment [SYS+08] and (iv) improving the efficiency of the process [MCJ07].

Nevertheless, one of the main limitations of dynamic analysis is that it can only explore the execution trace given a concrete input, exploring one execution path at a time. Malware writers can take advantage of this limitation in several ways in order to evade dynamic unpackers [BLB11].

Many malware samples conditionally execute certain parts of the code depending on the inputs provided by the system. In some occasions, these triggers are explicit anti-sandbox implementations in which, if certain conditions are observed (e.g., monitoring tools, virtual machine presence) the malicious payload is not executed.

In other cases, the malware sample under analysis is designed to communicate with external entities (e.g., a Command and Control Server). If the sample is executed in a network contained environment, the sample might not have connectivity to the internet. Although there are network simulation tools, the correct simulation of proprietary network protocols generally requires a hard reverse engineering effort. In these cases, certain parts of the code will never be executed under a single-path dynamic execution engine.

Both aspects affect to the unpacking of samples. First, a malware sample might eventually trigger the decryption routine only under certain conditions. In these cases the original code will not be present in memory (not even a single region) unless the adequate input is provided to the program.

Second, during the study conducted in Chapter 5, we describe the technique named shifting-decode-frames (Type-VI in our taxonomy) technique and identify several packers that apply it in order to protect the code. A sample protected by this technique will only reveal one region at a time (in case it applies repacking) and also, it will only uncover the regions covered by the execution path triggered by the concrete input provided by the system.

The first problem described has been previously addressed in several publications [MKK07a, BHL+08, BHK+07, JWL+12, PDZ+14] that design and evaluate multi-path exploration techniques in order to discover code only executed under certain conditions.

While these publications address the problem of multipath exploration, all of them have focused on malware analysis and software testing. Nevertheless, no previous approach has addressed the adoption of this technique for the generic unpacking of samples protected by the shifting-decode-frames technique.

Packers heavily rely on self-modifying code and obfuscated control flow, making very hard to automatically explore different execution paths. At the same time, we do not need to execute all possible paths, but only to guide the execution in a way to maximise the recovered code.

For this reason, in this chapter we address the technical challenges of these approaches when applied to this specific problem. Moreover, one of the major limitations of multi-path exploration techniques is their efficiency, making the approach almost infeasible for large-scale malware analysis. In order to deal with this limitation, we propose several enhancements and a heuristic that allows to drastically reduce the number of paths to explore in order to discover as much code regions as possible.

Accordingly, we define the following research questions for this study:

Research question 6.1 *Is it possible to direct the execution by applying heuristics in order to efficiently uncover the protected code?*

Research question 6.2 *What are the technical challenges and limitations of multi-path exploration techniques when applied to the unpacking of samples protected with shifting-decode-frames?*

In order to answer these research questions, we leverage the Bitblaze platform and implement a series of modifications and heuristics that allow us to generically unpack samples protected by this technique, creating a memory dump of each region and exploring the paths that trigger the unpacking of previously uncovered memory regions.

In this chapter we propose a set of domain specific optimizations over a baseline implementation of a multi-path exploration engine. The design and implementation aspects of this system are detailed in Section 6.1. Section 6.2 describes the domain specific optimizations proposed. Section 6.3 proposes a heuristic based on the concept that it is not necessary to fully explore the binary, but only to drive the execution to the most interesting points in the binary (i.e., those that might trigger the execution of new regions of code). Section 6.4 describes our evaluation of the system, while it outlines other limitations that should be addressed in order to deal with complex obfuscations present in off-the-shelf packers. Finally, Section 6.5 concludes the chapter.

6.1 Implementation of the multi-path exploration engine

Moser et al. [MKK07b] proposed for the first time the application of multi-path exploration for the analysis of environment-sensitive malware. In their approach, they leveraged taint analysis and symbolic execution in order to query a linear constraint solver that allowed to compute the values to force at each moment. Their approach explored the paths in depth-first order, and in order to achieve this, they implemented a user-level process snapshot mechanism in order to restore the execution of the binary.

In order to evaluate our system, we implemented a baseline framework based on the concepts proposed by Moser et al. [MKK07b] over the Bitblaze platform. Also, we propose a set of domain specific optimizations of this approach that allows us to conditionally explore multiple-paths over certain *interesting* regions of code: the code of the original program protected by the packer. This modification is described in Section 6.2.2.

In this section we describe the technical aspects of our multi-path exploration engine implemented over the Bitblaze project.

The Bitblaze project offers 2 main components, TEMU, a binary tracing platform based in the whole-system emulator QEMU and Vine, a tool capable of analysing binaries or execution traces. More concretely, Vine translates the code in the x86 architecture to an intermediate representation (Vine IR). This intermediate representation allows to compute different control-flow and data-flow analysis algorithms over a simple language avoiding the inherent complexity of x86 assembly language (CISC architecture). Every instruction is translated as a set of atomic instructions. The side effects of assembly instructions are modelled as separate instructions.

TEMU also provides a plugin (*Tracecap*) that outputs execution traces that are interpretable by Vine. These instruction traces contain one entry per assembly instruction, and contain not only the instruction bytes but also the values and taint propagation data associated to their parameters.

Finally, TEMU also allows to implement callbacks on every conditional jump instruction executed that is affected by a tainted flag (*cjmp*). When a *cjmp* is executed, it means that the conditional branch depends on the tainted variables (input values provided to the program). In these cases, we can process the execution trace and compute a set of values for the input variables that would trigger the execution of both sides of the branch.

On top of the framework described in Chapter 5, we implemented our multi-path exploration engine for the experiments consisting of 3500 C/C++ and 800 OCaml lines of code.

6.1.1 Execution tree, execution paths, and execution trace

First, in order to record the execution trace and snapshots we represent the multi-path exploration state as an execution tree. Considering that the branch instructions in x86 assembly involve only two different paths, we define our execution tree as a binary tree in which every node represents a tainted conditional jump instruction found during execution. In this way, every node contains the following elements:

- **Instruction trace.** The instruction trace for a node comprises the previous conditional jump, and the instructions executed from that point until the next tainted branch in the trace.
- **Address of the cjmp.** Address where the conditional jump was found.
- **System snapshot.** When we find a tainted branch, we store a system snapshot that allows us to restart the execution at that point.
- **Execution paths.** Every node has one or two execution paths, each path containing a new node. We define a *natural* execution path as a path that does not need any modification in the program state in order to be explored. On the contrary, a *forced* execution path requires to update the system state in order to trigger the alternative path of the branch. When we query the constraint solver and it provides a feasible set of values to force the execution of both sides of the branch, we store these values in order to force the path whenever it is necessary.

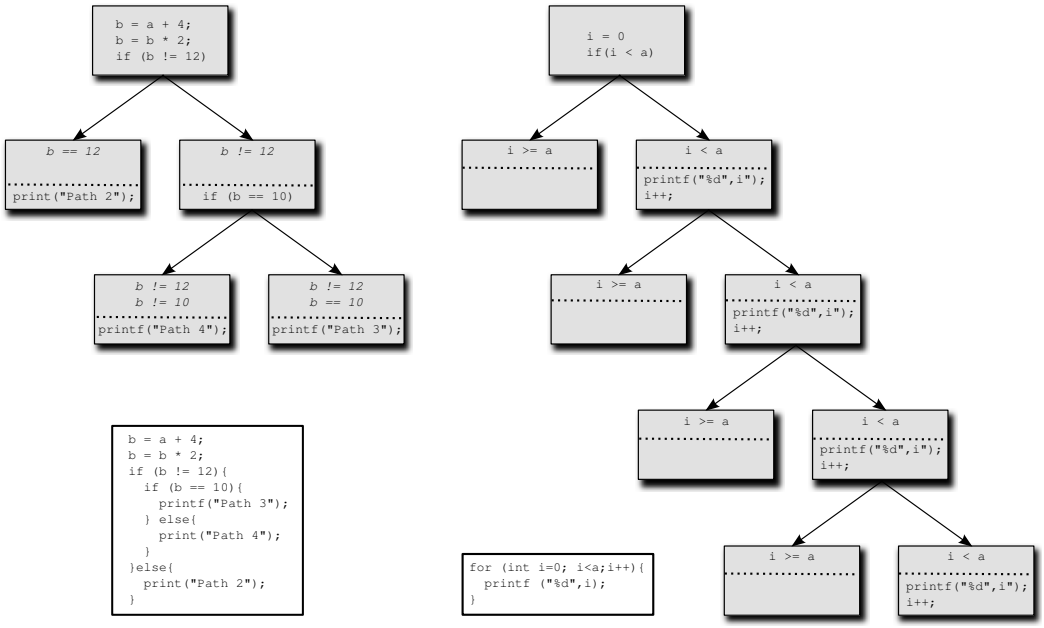


Figure 6.1: Execution trees, nodes, and paths for different sample programs.

Figure 6.1 shows this tree structure for a simple program in C language. Loops in which the loop condition is tainted are represented as a series of conditional jumps in which one of the branches exits the loop, and the other branch performs another iteration in the loop.

When the execution starts, the execution tree contains an execution node N_0 with no execution paths or snapshots, and an empty execution trace. We define the current execution node N_i as the node that is updated at a given instant by the executed instructions. Therefore, when the execution starts, $N_i = N_0$. Then, we trace the execution of the binary, and whenever a tainted instruction is executed, we store it in the execution trace associated to the current execution node. When a conditional jump is executed, we update the execution tree in the following way:

1. Update N_i and set the address of the *cjmp* that expands the node.
2. Add a *natural* execution path to N_i, create a new node for this path, and update the current node N_i to point this node.
3. Add the conditional jump instruction (already evaluated) to the current node N_i (new node).
4. Query the constraint solver providing the trace of tainted instructions to the Vine-based component. If the constraint solver provides a set of feasible values to force the alternative of the branch, create a new path and a new node for the alternative branch, and update the previous node accordingly.
5. If both paths of the branch can be explored ensuring the program state consistency, store a system-snapshot for the previous node.

Whenever a conditional branch is executed, or if the execution of the program is finished (process is terminated), we can decide the next path to explore. Moser et al. [MKK07b] proposed a Depth First Search algorithm to explore the execution tree. Alternatively, we propose the application of heuristics in order to determine the exploration order to maximise the code coverage efficiently.

When an snapshot is loaded, we recover the execution state previous to the execution of the conditional jump (Section 6.1.3 explains the implementation details). At this point of the execution, the conditional jump has not been yet evaluated. In order to force an alternative path, we adjust the program state with the values provided by the constraint solver. Then, the execution continues until the next conditional jump is found. Nevertheless, the *cjmp* that triggered the branch must be logged to the trace indicating the values of the flags necessary to take that branch. This *cjmp* will be interpreted as a path constraint the next time the constraint solver is queried, ensuring the program state consistency is maintained.

6.1.2 Symbolic execution

Symbolic execution allows to evaluate a program over a set of symbolic inputs instead of concrete values. In this way, for the code snippet shown in Listing 6.1, a concrete evaluation would require a concrete input value for variable a. A concrete evaluation of the program for $a = 1$ would have a program state at (1) in which $b = 10$, and the branch 1.1 would be executed printing the text *Path 1.1*. On the contrary, symbolic execution allows to reason about the values that the variables can hold at different points of the program.

Listing 6.1: C code snippet to ilustrate symbolic execution.

```
b = a + 4;
b = b * 2;
if (b != 12){
        printf("Path 1");
        if (b == 10){
                printf("Path 1.1"); (1)
        }
        else{
                print("Path 1.2");
        }
}
else{
        print("Path 2");
}
```

In this way, when line 1 is evaluated, b adopts the symbolic expression $b := a + 4$. Then, line 2 is evaluated and c is updated accordingly: $b := (a + 4) * 2$. Finally, when the branch at line 3 is evaluated, there are two possible paths to follow: $(a + 4) * 2$!= 12 and $(a + 4) * 2$ == 12. At this point, a constraint solver can evaluate the expression that determines the path constraint that must be met to follow each of the paths, providing an appropriate set of values for each input variable: a in each case.

One of the limitations of this approach is that the size of the expressions grows for every instruction executed. An option to reduce the size of the formulas is to split the expression into sets of sub-formulas. Also, an expression queried to the constraint solver is an equation that cannot express the program state and program execution order present in imperative languages such as assembly code. In order to achieve this, the program must be translated to *single static assignment* form. In this way, the code would

be represented as $b_0 := a_0 + 4; b_1 := b_0 * 2;$, and the path condition would be represented as $b_1 ! = 12$.

Some symbolic execution engines [CDE08, CKC11] simplify these expressions to reduce their size and enhance the efficiency of the computations of the constraint solver.

Alternatively, several publications [BWJS07, FS01, Lei05] propose the use of weakest preconditions [Dij76], a method that keeps the computational complexity and size of the formulas $O(n^2)$ [SAB10].

Vine provides a tool (*Appreplay*) that allows to compute the weakest precondition of an execution trace and to generate a query to the STP constraint solver. As a result of this process, the path restrictions imposed by the conditional branches are represented as a postcondition that is queried to the solver.

Listing 6.2: Query necessary to reproduce an execution trace in CAML based pseudo-code.

```
let a0 = INPUT in
let b0 = a + 4 in
let b1 = b0 * 2 in
let post0 = (b1 != 12) in
let post1 = post0 and (b1 == 10) in
post1
```

Appreplay allows to query the STP constraint solver for a set of input values to reproduce the execution trace (See Listing 6.2). This query is performed by asking for a counterexample (set of values for input variables) that makes a given expression to be evaluated as false. Accordingly, in order to reproduce the trace it is necessary to query the constraint solver to provide a counterexample for $\neg post1$, or a set of values that make $\neg post1$ false, or $post1$ true.

We leveraged this tool in order to query the constraint solver to provide a counterexample that negates the last component of the postcondition (last branch taken in the trace) while maintaining the path constraint imposed by previous branches. Note that this set of values cannot be obtained by querying a counterexample for $post1$. More specifically, a counterexample for $post1$ would be any arrangement of values that makes $post1$ false, or $\neg post1$ true. Following boolean logic rules, $\neg post1 = \neg post0 \vee \neg(b1 == 10)$. Accordingly, both negating the path constraint or the branch condition makes $post1$ to be false.

Alternatively, we modify the postcondition in the following way and query the STP constraint solver in order to obtain a set of values that drives the execution to the alternative path (see Listing 6.3).

Listing 6.3: Query to force an alternative path in CAML based pseudo-code.

```
let a0 = INPUT in
let b0 = a + 4 in
let b1 = b0 * 2 in
let post0 = (b1 != 12) in
let post1 = post0 and not(b1 == 10) in
post1
```

Finally, provided the set of input values provided by the STP constraint solver, we evaluate the expressions associated to each memory address and register at the appropriate execution instant. This computation is made over the intermediate language implemented in Vine. Afterwards, the necessary values are returned to TEMU in order to force the path.

Indirect memory accesses (i.e., memory access instructions in which the address used is tainted and depends on program input) are a recurrent problem in symbolic execution. While a concrete evaluation of the program will have at each point of the execution a concrete value for the address used to access the memory, when the same program is evaluated symbolically, the address can present any value in the address space constrained by the corresponding expression. This limitation is specially problematic for the symbolic execution of jump tables, a mechanism widely used by compilers to implement *switch* statements.

In taint-analysis systems there are different possible approaches, depending on the tainting policy specified. The taint might be propagated only if the memory accessed during the execution is tainted (may result in under-tainting), or also (and optionally) if the expression that determines the memory address to access is tainted (might result in over-tainting).

In the case of symbolic execution this situation is more problematic. The symbolic expression referred by the operation depends on the possible values that can be adopted by the expression used as a memory index. Some approaches let the constraint solver reason about the possible values, while other approaches perform alias analysis in order to determine the possible memory ranges pointed by the index [SAB10].

In our approach, we let Vine adopt the concrete value observed during execution for every tainted memory index avoiding any possible symbolic

processing of these values. Although this unsound assumption implies that some paths will never be executed, it simplifies the reasoning process involved in multi-path exploration.

Also, considering that some shifting-decode-frames based packers protect the original code with a page or function granularity, this limitation might be overcome by the fact that several paths in the program might trigger the execution of a page or function. Successfully exploring one of those paths is enough to trigger the unpacking of such function or memory page.

6.1.3 System-level snapshots

In order to save the execution state at a given point (before a *cjmp* is evaluated), we leverage the functionality present in TEMU to make system-snapshots. Previous approaches have proposed the use of process snapshots, a technique more efficient in terms of computational overhead and disk space. Nevertheless, making snapshots of the process state (memory and registers) involves many technical problems that are not easy to address. Processes running on the system generally use resources provided by the operating system like files, sockets, or the registry. Besides, the kernel of the operating systems maintains innumerable structures with information regarding memory assignment, heaps, stacks, threads or handles. While ensuring the consistency of the snapshot and saving and recovering the memory and register state is not difficult to implement, it is difficult to maintain the system consistency when the state of a process is restored. Moser et al. [MKK07b] proposed several methods to ensure that the process can continue running even if it is restored to a previous state (e.g., avoiding closing handles).

Since this aspect is beyond the scope of this study and stands as a research problem itself, we adopt a system-snapshot approach that, in spite of sacrificing system efficiency, allows us to securely restart the execution of a program at any point ensuring the consistency of the whole system.

There are some technical aspects that need to be addressed in order to reliably save and restore snapshots during the execution of the emulated guest system.

In TEMU, taint propagation is performed as part of the instrumentation code inserted in the translated blocks executed by the dynamic binary translation engine. In the case of *cjmps*, the event that triggers the callback function for their detection is performed in the moment the in-

struction is evaluated and the taint is propagated. Besides, among other aspects, TEMU optimizes flag computation by performing a lazy evaluation of the flags considering the fact that some instructions do not access them. For this reason, making a snapshot in the moment before the execution of a conditional jump is not reliable since the flag state has still not being computed. Nevertheless, we know that in order to process the *cjmp* it is necessary to evaluate the flag state. Also, this instruction represents the end of the translation block. In this way, when a *cjmp* callback is executed and the execution trace is evaluated by Vine, we defer the snapshot saving operation to the end of the block. At this point, the flag state is in a consistent state and we can safely record the execution state since the emulator has exited the translated routine in order to determine the next translation block to execute. Finally, when a system-snapshot is recovered, we need to re-evaluate the conditional jump over the forced set of values. In order to do this, we modify the instruction pointer recorded in the snapshot to point the address where the conditional jump is located. Since *cjmps* do not affect the program state (except the instruction pointer), we do not need to modify any other value in the snapshot.

It is also important to determine the appropriate moment to load an snapshot. If the snapshot is loaded in the middle of the CPU execution loop, the execution loop will continue over the recovered snapshot producing an undetermined behaviour. For this reason, we defer the snapshot loading to the moment before the system enters the execution loop, which is exited every time there is an interruption in the system (e.g., a context switch or a system call). In this way, when we recover the snapshot after an interrupt we disable the pending interrupts and clean the translation block cache and the Translation Look-aside Buffer. At this point, the emulator will continue with the execution of the process in the instant before the conditional jump.

6.1.4 Taint sources

TEMU provides several ways to introduce taints in the system. Moreover, it allows to taint the network interface, files, memory address ranges, and keystrokes.

Nevertheless, since many of the API calls requiring I/O operations block the execution until some input is read, we alternatively taint the output of interesting API calls (Section 6.2.5 describes how to deal with blocking system calls). In this way, we taint network operations such as `connect`,

recv, gethostbyname or gethostbyaddr, file operations such as ReadFile or CreateFile, API functions for retrieving command line arguments such as __wgetmainargs or ReadConsoleInput, and other functions typically used to query the system state like GetSystemTime or Process32First / Process32Next.

6.2 Domain specific optimizations

6.2.1 Inconsistent multi-path exploration

In some occasions, traditional symbolic execution approaches impede the execution of certain paths that, despite of being feasible, cannot be solved by a constraint solver. In our system, one example of this limitation is tainted key-strokes. TEMU supports tainting key-strokes both for Linux and Windows based guest systems. When the monitored program calls to the scanf function with the "%d" format string, the system reads from standard input a series of characters and then translates them to a decimal number. The translation from an ASCII representation to a decimal number is done by using a conversion table. As a consequence, the memory index to access this table will be tainted (because it is based on each of the characters provided as input). In our case this access is made concrete in the symbolic engine, and the solver cannot provide a feasible set of values to force multiple paths.

In these cases, we take an unsound assumption and query the constraint solver ignoring the path restrictions imposed by the trace. This approach lets us explore the path by forcing a set of values inconsistent with the path restrictions.

In our specific domain, maintaining the consistency of the system is only important in order to avoid system crashes and to maximise the execution time of the analysis. Nevertheless, from the perspective of unpacking the code, the objective is to trigger the execution of as many regions as possible, regardless of the conditions necessary to do it. While other domains may suffer from this unsound implementation (e.g., malware analysis may require to know under which circumstances a certain path is triggered), in our case this information is not relevant.

6.2.2 Partial symbolic execution

Multi-path exploration is a computationally expensive approach. Reducing the amount of code explored in this fashion allows to drastically reduce the complexity of the problem. Besides, in our problem domain, it is desirable to restrict this approach to the execution of the original code, avoiding code belonging to the packer or system libraries. First, this code will not trigger the unpacking of new regions of code. Second, the unpacker code is generally highly obfuscated and does not follow standard calling conventions, making more difficult to correctly trace and symbolically process the code. For this reason, we restrict multi-path exploration to the regions that contain the code of the original application.

Intuitively, it might seem that the packer code will hardly ever execute tainted conditional jumps because the execution paths inside this code will not depend on the program input. This might render unnecessary to restrict the areas explored. Nevertheless, many packers perform several tasks apart from unpacking the original code aimed at protecting the binary (e.g., licensing). For instance, Armadillo fetches the system date using the `GetSystemTime` API function in `kernel32.dll`, affecting to conditional instructions.

For this reason we modified the basic approach described in order to restrict the areas explored. In this way, whenever a conditional jump is executed outside the regions susceptible of multi-path exploration, we do not expand the possible execution paths and continue normal execution.

6.2.3 Local and global consistency

Another aspect to consider is the consistency of the symbolic execution engine. For example, the S2E project [CKC11] allows to run programs at different consistency levels. In our case, global consistency is maintained when all the instructions that propagate a taint are recorded. When conditional jumps are found in regions of code that are not interesting to explore, these instructions will be recorded in the trace imposing path constraints. An example of this situation is, again, the tainting of keystrokes. If a program performs a `scanf("%d",val)` operation to read a number from standard input, and we taint one keystroke provided as input, when the explored code is reached the variable containing the tainted value will be affected by all the path restrictions from that taint origin. Among other aspects, these restrictions will limit the possible values for the number to be

between 0 and 9 (numbers that can be represented by one single ASCII character). Nevertheless, in some occasions these restrictions introduce a high computational overhead and sometimes involve conversions that make use of tables accessed by indirect memory accesses.

In order to minimise the computational overhead of such operations we implement a locally consistent multi-path exploration approach of the regions that may contain the original code of the binary. This locally consistent approach ignores global consistency allowing any possible value when a tainted variable is accessed in the explored code. In this way, when the value read in the `scanf` operation is used for the first time in the explored code, we create a completely new symbolic variable not dependant on the previous instructions. A first approximation to this model could be achieved by avoiding to trace any instruction propagating taints if it is executed outside the explored regions. In this way, when the execution trace is interpreted in the symbolic engine, only the instructions in the explored regions impose restrictions over the symbolic variables.

In order to ensure that local consistency is correctly preserved, we modify the tainting propagation policy in TEMU. Whenever we taint a value in TEMU, we introduce a taint of one byte for each memory address or register byte. We use one bit of this byte in order to distinguish values tainted outside the explored code. In this way, when a byte tainted outside the explored code is read in the explored code, instead of propagating the taint we create a new taint source by updating the corresponding bit. This tainted variable will be considered as a free symbolic variable by Vine because it represents a new taint source. Also, in the aforementioned case, when the keystroke is tainted only one byte is affected. When the explored code first reads the value provided by the `scanf` function, this byte has been manipulated by the system library and at that point the 4 bytes of the integer value have the same taint. If at this point we create a completely new taint and assign the 4 bytes the same taint source, Vine will interpret them as identical copies of the same value. In order to avoid this, we define a new taint source for each byte and allow the integer to adopt any possible value. The instructions traced in the explored code will restrict the possible values that the variable may contain ensuring the local consistency of the code explored, but allowing a global inconsistency if the values tested in the explored code are values that can never be returned by system libraries.

Finally, whenever the program calls to a function outside the region delimited, if the arguments of the call are tainted then the result of the call is consequently tainted. As we do not trace the execution of such code, the

taint propagation chain will be broken and Vine will be unable to provide a solution.

Executing symbolically all the code present in API functions or not interesting regions is not computationally feasible. In fact, one of the main limitations of multi-path exploration is its inability to scale to large programs.

For this reason, we avoid tracing the execution of code outside the boundaries of our regions of interest. In order to allow Vine to process these traces with broken taint propagation, we create a new independent symbolic variable whenever necessary.

In this case, again, we loose program consistency. Nevertheless, as we describe in Section 6.2.1, this inconsistency does not affect our approach but on the contrary, lets us explore as many paths as possible (triggering the execution and thus the unpacking of new regions of code).

6.2.4 Size of the traces

One of the limitations that make multipath exploration infeasible to analyse large programs is the well-known state explosion problem [CKNZ12]: when the number of state variables increases, the number of states grows exponentially. In our case, this growth is represented by the number of *cjmps* in the program trace that are successfully expanded. Furthermore, some cases present infinite state spaces, for example when unbounded loops are implemented in the explored code.

Unfortunately, the Vine tool and constraint solvers in general cannot compute too long execution traces. In our case, we configured our multipath exploration engine to discard execution paths with a trace longer than a given threshold. This parametrization allows us to keep the analysis as simple as possible and computationally feasible.

6.2.5 Blocking API calls

Also, we have implemented a mechanism that allows to bypass blocking API function calls. In some cases, the program gets blocked waiting for user input or certain events in the system. For this reason, when certain APIs such as the `read` or `recv` functions are called, instead of letting the program run, we restore the instruction pointer to the return address in the moment of the call. Also, we fill the output buffers and output values with junk data, and taint those buffers. As a side effect, the state of the dynamic

binary translation engine in TEMU is affected. In order to correctly restore the execution we also manually restore the stack pointer of the translated routine before returning to the translation engine.

6.2.6 String comparison optimization

The last optimization implemented deals with string comparison, an operation commonly performed by malware that parses commands (e.g., IRC bots or bots managed from a C&C). These string comparisons are generally implemented by means of system API calls such as `strcmp`, or `strlen`. In some occasions these libraries are statically linked into the compiled binary, while in other cases, the libraries are linked dynamically. In any of the two cases we are not interested in exploring the code in these libraries. Nevertheless, the execution of this code imposes path restrictions that affect the output of the function call. Also, taint propagation presents a limitation in these cases. Different paths in this code may overwrite the return value of the function with different values. However, this value is not directly tainted. Since we do not explore these paths, we restrict the return parameter to only one possible value, limiting the subsequent multi-path exploration for branches that depend on this output.

In order to deal with this limitation, we hooked up to 25 different string comparison functions in several DLLs in order to taint the output of the function whenever a tainted value is provided as input to them.

Finally, these libraries might be statically linked or even in-lined in the compiled code. In the first case, it would be possible to identify this code by using common signatures (following a process similar to IDA Pro). In the second case, since the code is inlined, it would be explored as part of the interesting code of the binary.

6.3 Heuristic to guide multipath exploration

One way to reduce the state space and thus the complexity of multipath exploration is to apply heuristics in order to determine which paths should be expanded first. We propose a heuristic based on the intuition that, for a packer protected using the shifting-decode-frames technique, multiple instructions in the program (and therefore, multiple paths) can trigger the execution of a region (e.g., function or memory page). In these cases, it is not necessary to explore the whole state space in order to fully unpack all the content of a binary. Even in the case in which the protection of the code

is performed at basic block or instruction level, multi-path exploration can benefit from this approach, since many conditional branches might lead to the same blocks, making unnecessary to expand all of them.

The proposed heuristic is built over the packer analysis framework described in Chapter 5. First, we extract all the executed code and unpacked memory areas from a single-path execution trace in order to recover as much code as possible. Then, we analyse this code and determine the instructions that reference locations in the program that have still not been unpacked. We construct the call graph and control flow graph of the trace and find the paths that reach these interesting instructions, and finally we provide this information as input to our multi-path exploration engine in order to heuristically prioritize the execution of certain paths that trigger the unpacking of new regions of code. The next sections detail how this process is performed.

6.3.1 **Dumping unpacked memory frames**

First, we use the packer analysis framework described in Chapter 5 in order to create a complete dump of the unpacked memory regions of the packer. This dump is performed during the post-processing phase. More specifically, whenever an unpacking frame is detected at any execution layer, we dump every modified memory address in the page-aligned address space covered by the layers observed during the execution of the sample. This process is performed in an off-line fashion. We keep a mirror memory and update it on every memory operation regardless of the execution layer involved. For each memory address affected by a write operation, we also update a shadow memory that records the memory address ranges modified in the previous frames.

When a new frame is detected for any of the layers, we first compute the sets of contiguous addresses modified (i.e., unpacked memory regions). Then, we compute the intersection between these blocks and the memory blocks dumped in previous frames. The blocks that have been already dumped in previous frames are discarded. Finally, we fetch the list of executed basic blocks that overlap these memory space, and compare their contents. We dump each block if the content of the majority of the overlapping instructions is equivalent.

The memory writes recorded in an execution trace might eventually correspond to code repacking. Nevertheless, the method followed to calculate the intersections of the already dumped blocks allows to detect this

situation. Figure 6.2 shows the packing-repacking process of a shifting-decode-frames based packer. In step 1, R1 is unpacked (written, and then executed). In step 2, the execution jumps from R1 to R2. At this point, R1 is repacked, and R2 unpacked. In the moment we detect the new frame (the first instruction of R2 is executed), the written memory covers R1 and R2. Nevertheless, R1 has been already unpacked. R1 $\cap$ (R1 $\cup$ R2) is R2, so only R2 is compared against the corresponding basic blocks and dumped accordingly. The same applies when the execution jumps from R2 to R3.

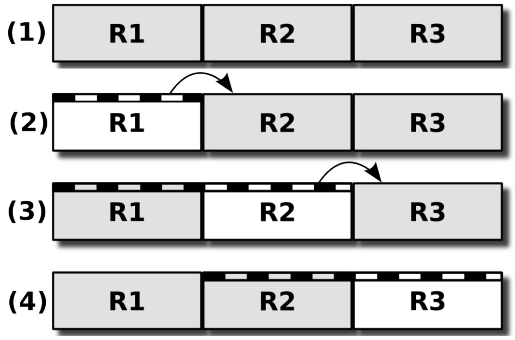


Figure 6.2: Unpacking process with shifting-decode-frames.

Once we have a complete memory dump, we filter it in order to keep only the regions to be explored in a multi-path fashion. In order to do this, we first indicate which regions we want to explore, and then we generate a filtered memory dump containing (i) the memory blocks dumped that overlap those regions, and (ii) all the execution blocks traced for those regions.

6.3.2 Disassembly and translation to intermediate language

In order to analyse the memory dumped by our tool, we developed a component in C/C++ and OCaml over the Vine set of tools and libraries.

In order to analyse the memory dumped by our tool, we implemented our custom disassembly engine to process the unpacked frames of code. This engine is based on the binutils disassembly interface and the libdisasm library.

First, for each execution block recorded during the analysis of the packer, we start a linear sweep disassembly process over the execution blocks. As defined in Chapter 5, execution blocks do not contain any instruction that affects the control flow of the program except the last instruction. For this reason, a linear-sweep algorithm will always successfully extract the code for these blocks.

Second, we find all the conditional jumps in the disassembled instructions. We record both the destiny and the fall-back addresses of the jump. Each time the block is executed, it is followed by any of these two addresses. In some cases both targets of the jump may have been executed, but in other cases, only one of the two possible targets will have been uncovered.

At this point we start the second phase of the disassembly. We disassemble the corresponding instruction for each address recorded if it is located in the memory address ranges already dumped. In this case, we follow a recursive-traversal algorithm in order to disassemble as many instructions as possible from the non-executed parts of the unpacked frames. When a jump, conditional jump, or call instruction is found, we determine the target address if it uses direct, absolute or relative addressing and continue the disassembly process until no more instructions can be fetched.

We avoid applying speculative disassembly techniques or trying to guess the target address for indirect jump instructions, since these techniques might result into the incorrect disassembly of the dumped memory.

First, we translate the disassembled result into Vine IL, an intermediate representation. This language has no side-effects and simplifies the complexity inherent in x86 architecture. Over this language, the Vine tool-set allows to apply different data flow analysis and control flow analysis techniques, as well as compiler optimizations such as alias analysis.

6.3.3 **Interesting memory addresses**

Once the content of the memory dump is disassembled and translated, we build the Control Flow Graph of the disassembled code and process the result in order to find execution paths not yet explored that might drive the execution to unpack new code regions.

Control Flow Graphs (CFG) are generally used to represent independent functions in the code. In many cases binaries export their symbols, including the addresses at which the different functions declared are located. In other cases, these symbols are stripped from the binary and functions can be located by analysing the `call` instructions present in the code. Call Graphs are generally used in order to represent the relationship between functions, while Control Flow Graphs display the inner structure of each function.

Our approach does not employ any symbol information because the majority of packers strip the symbols from the binary in order to difficult reverse engineering. Also, the code extracted from the memory dump covers

program	::=	*decl* stmt**
decl	::=	`var` *var* `;`
stmt	::=	*lval* `=` *exp* `;` | `jmp(`*exp*`);` | `cjmp(`*exp*`,` *exp*`,` *exp*`);` | `halt(`*exp*`);` | `assert(`*exp*`);` | `label` *label*`:` | `special` *string*`;` | `{` *decl* stmt** `}`
label	::=	*identifier*
lval	::=	*var* | *var*`[`*exp*`]`
exp	::=	`(` *exp* `)` | *lval* | `name(`*label*`)` | *exp* $\diamond_b$ *exp* | $\diamond_u$ *exp* | *const* | `let` *lval* `=` *exp* `in` *exp* | `cast(`*exp*`)`*cast_kind*`:`τ_{reg}
cast_kind	::=	`Unsigned` | `U` | `Signed` | `S` | `High` | `H` | `Low` | `L`
var	::=	*identifier*`:`τ
$\diamond_b$	::=	`+` | `-` | `*` | `/` | `/$` \| `%` \| `%$` | `«` | `»` | `@»` | `&` | `^` | `|` | `==` | `<>` | `<` | `<=` | `>` | `>=` | `<$` \| `<=$` | `>$` \| `>=$`
$\diamond_u$	::=	`-` | `!`
const	::=	*integer*:τ_{reg}
τ	::=	τ_{reg} | τ_{mem}
τ_{reg}	::=	`reg1_t` | `reg8_t` | `reg16_t` | `reg32_t` | `reg64_t`
τ_{mem}	::=	`mem32l_t` | `mem64l_t` | τ_{reg}`[`*const*`]`

Table 6.1: Grammar of the Vine Intermediate Language (IL), obtained from the project documentation [SBY+08].

all the executed instructions and a significant part of the unpacked code, but does not distinguish the functions present in the code.

Using the Vine tool-set, we compute the Control Flow Graph of all the code ignoring the fact that there might be several functions contained in it. Therefore, as a result, we will find in the CFG several independent unconnected sub-graphs.

6.3.3.1 Obtaining interesting pointers

Once the CFG is computed, we extract, for each basic block, pointers that potentially lead to the unpacking of new regions of code.

- **Control flow instructions.** Control flow instructions (`jmp`,`call`, and `cjmp`) alter the execution flow of the binary and therefore are susceptible of triggering the unpacking of new frames of code. First, if non-conditional control flow instructions are present in the trace, their target addresses will also be executed. In the case of `cjmp` instructions, it is possible to find cases in which only one of the branches is executed. Nevertheless, considering that we also disassemble instructions not executed and extracted from the unpacked memory frames, we can find `jump` and `call` instructions that lead to regions of code not previously observed.

Assembly `jmp` and `call` instructions take the form of `jmp(`*exp*`)` IL statements. Memory addresses referenced as targets of these instructions are declared in vine as `name(`*label*`)` expressions, where *label* refers to the actual target address.

Conditional `jmp` instructions take the form of `cjmp(`*exp*`,` *exp*`,` *exp*`)` statements. In this case, we evaluate the potential pointers contained in the evaluated expression (first parameter), and consider the target address (second parameter) and the fallback address (third parameter) as interesting pointers.

- **Direct memory addressing.** When we find an instruction that accesses a memory address in the module address space of the binary, we keep the pointer referencing the memory.

  In Vine IL, direct memory accesses take the form of *var*`[`*exp*`]` expressions, in which *exp* is an integer constant representing the memory address accessed.

  Nonetheless, if the block was executed, then the memory address was actually accessed and therefore unpacked. In these cases, we keep a record with all the executed memory accesses that will help us to filter pointers to regions already unpacked.

- **Constants.** Finally, we also analyse the constant values provided as immediate values in the code. Constants are represented by *const* IL expressions. In this case, we discard every constant not referencing to a memory address contained in the module address space. This approach allows us to consider potential register-indirect or memory-indirect addressing operations.

- **Indirect function calls.** Indirect function calls also constitute a problem in multi-path exploration. When the register containing the call address is tainted, we need to reason about all the possible values that it can adopt. In our case, we have simplified this problem by concretely evaluating the call address regardless of its taint value. In order to allow the exploration of multiple values for these calls, we consider them interesting points in the program. Therefore, the different paths that drive the execution to this point may write different values over the register or memory address used in the indirect call, allowing our system to execute several paths.

6.3.4 Finding interesting paths

Once we have found the pointers which may potentially lead to new regions, we can compute in the CFG the paths that drive to each basic block. When the pointer observed corresponds to a direct memory addressing operation, a constant, or an unconditional jump, if the basic block containing the pointer is executed, then the execution of such instruction will be triggered. In the case of conditional jumps, we store two possible pointers. In these cases, reaching the basic block containing the conditional jump does not mean that both paths will be executed. Instead of associating the pointers to the basic block containing the conditional jump, we associate them to each of the successor basic blocks.

In this way, we can compute the paths that drive to each basic block containing interesting pointers by recursively fetching its ancestor basic blocks. Whenever a loop in the CFG is detected, we consider 2 possible paths: one that enters the loop, and another one that does not meet the loop condition. We keep iterating the ancestor basic blocks until we reach the function entry point.

When a node with no ancestors is located, it is treated as a potential function entry point. For each of these nodes, we retrieve the first instruction of the block.

6.3.4.1 Identifying interesting functions

After obtaining all the pointers contained in the basic blocks of the CFG, we discard any pointer that references instructions already executed, instructions already written, and memory addresses accessed in any operation involving a direct memory access.

Nevertheless, there might be functions that, despite executed, still contain paths that were not executed but were successfully disassembled. In these cases, the function entry-points are still pointers of interest and should not be discarded. Furthermore, if there is another function containing one interesting `call` to any of these functions but not containing any other interesting pointer, we should not discard the path that reaches this instruction although it calls to a function already executed.

In order to ensure that all the necessary functions are reached, we consider as an interesting pointer to any call to a function that, even if already executed, contains unexplored interesting pointers.

In order to compute the complete set of pointers of interest, we follow Algorithm 3. The $get\_functions$ function computes the list of function

entry points for the provided pointers.

Data: List of executed memory addresses, X.
List of written memory addresses, W.
List of accessed memory addresses, A.
List of interesting pointers, P.
Result: Filtered list of interesting pointers, P'.
$F \leftarrow X \cup W \cup A$
$P' \leftarrow P - F$
$functions \leftarrow get\_functions(P')$
$N = 0$
while $N < |functions|$ **do**
  $N \leftarrow |functions|$
  $P' \leftarrow P' \cup (P \cap functions)$
  $functions \leftarrow get\_functions(P')$
end

Algorithm 3: Computation of the list of interesting pointers.

6.3.4.2 Computing the paths

Finally, once we obtain the list of pointers that should be reached during execution, we compute the set of paths that reach each pointer as a succession of *(cjmp,address)* pairs. For each conditional jump, we indicate the address that should be executed next in order to reach the pointer.

Note that these paths only cover the code of the function containing the pointer. Since we also consider function calls as interesting pointers, we include any possible path that reaches a `call` instruction to any of these functions.

Eventually, for a given pointer, there might be several different paths reaching its container basic block. Instead of simplifying the list, we keep all the possible paths because they might introduce different path restrictions during execution. In fact, many of the paths computed will not be feasible (i.e., there is no possible assignment for the variables in order to force the path). This feasibility will be tested by the constraint solver during multi-path exploration.

The output of our system is a complete list of the interesting pointers that can be reached for each of the two possible branches of each *cjmp*. This list is provided as input to the multi-path exploration engine to guide the execution to the interesting parts of the code.

6.3.5 Execution path selection algorithm

Given the output provided by the implemented Vine module, our TEMU module keeps a list of every pointer not previously visited with the conditional jumps associated to each pointer.

Whenever a tainted conditional jump is reached during execution, an event is triggered, and we check the list of interesting conditional jumps. If the *cjmp* is present in the list, we inspect the number of interesting pointers for each of its paths. Also, we check which is the path that will be executed if we do not force the tainted flag. If it has 2 paths with interesting pointers, or only one path but it is not the default path, we query the constraint solver. If there is a feasible set of values that can be forced in order to follow the alternative path, we create a snapshot and decide the next path to execute. If the solver cannot provide a feasible solution, we query the solver again ignoring the path restrictions imposed by the execution trace. In any of the two cases, if the solver provides a solution, we store the values to force and create the snapshot. If the path was solved inconsistently, we will prioritise any other consistent path before forcing it.

The search algorithm used in order to find the next snapshot to load is Breath First Search. This algorithm allows us to incrementally expand all the paths in the execution tree. Besides, we maintain a list with the number of times each path is expanded, and select, among the available paths, the one with the minimum number of expansions. This approach allows us to avoid exploring recursively loops which drive to interesting pointers that can be reached more efficiently by other paths requiring a lower number of snapshots and queries to the SMT solver. When several paths have the same number of expansions, we check if one of them was created inconsistently. If this is the case, we give priority to the consistent path over the inconsistent one.

Finally, whenever a memory region is written in the execution level in which the original code resides, we check if it overlaps any of the interesting pointers. In this cases, we assume that the interesting pointers have been unpacked and remove them from the list of interesting pointers to explore. If there are no remaining pointers for a conditional jump, we also remove the conditional jump from the list.

As a result, if a certain memory region can be reached from different execution paths, even if the constraint solver is not capable of providing a feasible set of values, we will reach the region if there is at least one satisfiable path.

Also, in cases like page-granularity protection, we only need to trigger a subset of the paths in order to reach all the code pages, avoiding to explore the rest of paths and thus reducing the state space.

6.4 Evaluation

In order to test our approach, we protected a real malware sample using *backpack*, a packer proposed by Bilge et al. [BLB11] that protects the binary with function granularity. In order to test the packer, we downloaded the source code of the Kaiten IRC bot, recently reported to be distributed using the *shellsock* bash vulnerability[1]. This sample connects to an IRC channel on different servers, and receives commands to perform actions such as remote command execution or network flooding. Backpack is designed to protect the binary at compile time. It is implemented as an LLVM plugin and it can protect C programs. In order to successfully compile Kaiten using the LLVM plugin provided by Bilge et al., we had to modify the command dispatching routines to substitute the function calls that used function pointers instead of direct calls. Given the functionality of the malware, we configured our system to taint network input considering the `recv`, `connect`, `read`, `write` and `inetaddr` system API functions.

Table 6.2 shows the results obtained for the Kaiten malware, consisting of 31 protected functions. The unpacking is performed iteratively. In the first iteration multi-path exploration was not applied, revealing only 5 of the 31 functions. The heuristic reported 48 interesting pointers and 36 conditional jumps for the code revealed. In the first multi-path iteration, 6 new functions were unpacked requiring a total of 44 snapshots. After this, our heuristic was capable of finding 97 interesing pointers and 107 conditional jumps. Finally, in the last iteration 27 functions were triggered requiring 1219 snapshots. These results show that a single concrete execution only reveals a little portion of the real contents of the binary. Also, the heuristic allows to discover all the functions in the binary exploring a relatively low number of paths.

Table 6.2 also shows the number of consistent and inconsistent queries performed in each of the 2 iterations. The results show that the number of inconsistent queries is very low in both cases. Our local-consistency based

[1] Shellshock Vulnerability Downloads KAITEN Source Code. `http://blog.trendmicro.com/trendlabs-security-intelligence/shellshock-vulnerability-downloads-kaiten-source-code/`. Accessed: 2014-11-12

Table 6.2: Results obtained for the Kaiten malware packed with backpack.

	It. #0	**It. #1**	**It. #2**
Interesting pointers	-	48	97
Cjmps	-	36	107
Snapshots	-	44	1238
Consistent queries	-	42	1219
Inconsistent queries	-	2	30
Long traces discarded	-	3	158
Pointers deleted	-	34	83
Cjmps deleted	-	19	43
Functions unpacked	5/31	11/31	27/31

exploration algorithm and the rest of domain-specific optimizations allow us to assume certain inconsistencies with the rest of the system that improve the capacity of the approach to force locally consistent paths. Nevertheless, there are still a few cases in which inconsistent assumptions allow to explore alternative paths that otherwise would be infeasible to explore.

Also, we can observe that during the execution almost all the pointers listed by our heuristic were correctly unpacked (and therefore discarded) during execution. However, there is a number of pointers that were never unpacked. These pointers correspond to functions or code regions that could not be explored given the taint inputs provided to the system.

6.5 Conclusions and discussion

Previous sections have described the domain-specific optimizations and heuristics that can be implemented over multi-path exploration to deal with the unpacking problem. More concretely, we propose a set of unsound assumptions that let us enhance the efficiency of multi-path exploration approaches and a heuristic based on the idea that many different execution paths might reach the same protected memory regions. We have evaluated our approach over Kaiten, an IRC flooder malware we packed with Backpack, a packer that protects each function independently.

In this way, we can answer the research questions established in the beginning of this chapter.

Is it possible to direct the execution by applying domain specific optimizations or heuristics in order to efficiently uncover the protected code?

In order to answer this question, we have implemented a heuristic in order to drive the execution to the points in the binary that most probably trigger the unpacking of new regions of code. The results show that it is possible to reduce the state space by keeping a list of the interesting pointers that can be reached by every possible conditional jump in the code.

What are the technical challenges and limitations of multi-path exploration techniques when applied to the unpacking of samples protected with shifting-decode-frames?

We have proposed a set of domain specific optimizations in order to deal with some of the limitations of multi-path exploration approaches. Although these optimizations generally require to take unsound assumptions, we have demonstrated that they can be applied to the unpacking problem without affecting to the main objective of the analysis: recovering the original code of the binary.

Nevertheless, multi-path exploration still presents many other limitations that should be addressed in order to deal with highly obfuscated software such as packed binaries. The following list describes several of the methods employed by packers that should be considered for the implementation of a comprehensive multi-path exploration engine.

- **Calling convention violation.** Malware samples typically violate calling conventions in order to confuse the analyst and obfuscate the code. In some cases these techniques are extended to API function calls, thus affecting API call tracing mechanisms.
- **Alternative methods to redirect control-flow.** In order to evade multi-path exploration approaches, malware samples can potentially use alternative methods to redirect the control flow of a binary. One way to implement such obfuscation is to use alternative combinations of instructions such as `push` + `ret`, indirect calls, `call` + `pop` + `push` + `jmp`, SEH or VEH based redirection, opaque predicates in branch instructions, or even obfuscating the computation of triggers [SLGL08].
- **API call interception.** Some packers intercept API function calls in order to instrument the execution of the protected binary.

- **Nanomites.** This technique consists in replacing conditional branch instructions by software interrupts (e.g. `INT 3`) that cause the execution to break. A parent process intercepts the exception and then overwrites the conditional jump. A more complicated example involves redirecting the execution of the child by evaluating its context (state of the `EFLAGS` register) and redirecting its execution to the appropriate address, without even replacing the interrupt instruction with the original instruction.

- **Use of unsupported instructions.** Symbolic execution engines do not always support every possible instruction in the Intel architecture. Floating point operations, as well as MMX and other special instruction sets are typically not covered by these engines. A malware may eventually use these operations in order to break taint propagation, or to limit the capacity of symbolic engines to reason about the different possible execution paths.

6.6 Summary

In this chapter we have studied the different domain-specific optimizations that can be implemented in order to apply multi-path exploration to the unpacking of Type-V and Type-VI packers, a class of packers that, although not very common, violate the assumptions made by existing generic unpackers. We can conclude that, although multi-path exploration presents many limitations to scale to large programs, it is possible to apply different heuristics to reduce the complexity of the approach.

« It is not the mountain we conquer but ourselves.»

Edmund Hillary (1919–2008)

Chapter 7

Conclusions

During this dissertation we have presented different methods based on static and dynamic analysis in order to deal with different problems related to run-time packer analysis. This chapter summarises the result of this research and measures the achievement of the objectives proposed for this dissertation. The rest of the chapter is organised as follows. Section 7.1 enumerates the contributions of this dissertation. Section 7.2 discusses the limitations of the proposed approaches. Section 7.3 describes future lines of research that derive from this work. Finally, Section 7.4 outlines the final remarks of this dissertation.

7.1 Main contributions

In previous chapters we have presented the different approaches proposed in order to solve several problems related to run-time packer analysis. Now, we revisit the initial hypothesis established in the first chapter of this document.

> «*It is possible to classify and measure the complexity of run-time packers using static and dynamic analysis techniques and to recover the code of binaries protected with partial code revelation.*»

Next, we summarise the main contributions presented in this dissertation from Chapter 3 to Chapter 6.

1. **An evaluation of supervised machine-learning classifiers over structural feature based and heuristic based feature-sets for the classification of packed binaries.**

   We have proposed a model based on the extraction of structural features to discriminate packed from non-packed executables. Moreover, we have compared the performance of our approach against the most typical heuristics used for packer detection. To this aim, we have tested the most popular supervised machine-learning classifiers and concluded that our representation improves the performance of these approaches when considered together with traditional heuristics.

2. **An anomaly detection based approach for packed binary classification based on two different feature-sets.**

   We have proposed and evaluated an anomaly detection based approach capable of differentiating packed from non-packed binaries modelling only one of the classes: non-packed binaries. This model is based on the assumption that common compilers generally follow standard conventions. The resulting binaries are easier to model than packed binaries that may present any possible scrambled structure. Also, adjusting the anomaly threshold in this model is an straightforward process. This allows to adapt the model to the use-case and specific requirements of the deployment scenario.

3. **A comprehensive framework for run-time packer analysis based on dynamic tracing of binaries inside an emulator.**

   In the second part of the dissertation, we have focused on dynamic analysis. More specifically, we have developed a whole framework over an existing platform for binary tracing focused on the analysis of run-time packers. This framework allows us to record several events that can later be used to compute metrics and information useful for the analyst. Also, it allows to compute and visualise the structure of the packer dividing the code into layers and regions. It is, to the best of our knowledge, the first framework for run-time packer analysis that combines fine-grained granularity tracing with coarse-grained visualisation.

4. **A taxonomy capable of measuring run-time packer structural complexity from different perspectives.**

   Based on the information collected by our framework, we have proposed a set of properties that, combined together in a single taxonomy, allow to compute the structural complexity of a packer. This taxonomy is the result of the analysis and understanding of hundreds of different packers.

5. **The first longitudinal study of run-time packer complexity over off-the-shelf and custom packed binaries**.

   The implemented framework and the proposed taxonomy allowed us to conduct the first longitudinal study of packer complexity over off-the-shelf packers and custom packed binaries submitted to Anubis, a public malware analysis sandbox running since 2007. The results of this study document, for the first time, the structural complexity of run-time packers and their temporal evolution.

6. **A set of domain-specific optimisations that improve the feasibility of multi-path exploration for unpacking samples with partial code revelation.**

   Finally we have focused our efforts on multi-path exploration. Although these techniques present severe limitations for large-scale analysis and do not show a sound performance for large programs, we have studied some of the limitations that affect to a potential generic unpacker for samples that apply partial code revelation. In this sense, we have proposed a set of domain-specific optimizations in order to improve the feasibility of these approaches for generic unpacking.

7. **A heuristic to select the most interesting execution paths for multi-path exploration in the domain of generic unpacking**.

   Another important limitation of multi-path exploration is the well-known path explosion problem. In order to reduce the number of paths to explore, we propose a heuristic based on the concept that, for coarse-grained partial code revelation, it is only necessary to trigger one of the execution paths that drives to the unpacking of each independently protected region. We have successfully evaluated this heuristic over a real malware protected with Backpack, a packer that implements function-based protection.

With the presentation of these contributions we have accomplished the specific objectives established in Chapter 1 for this dissertation.

- Improve packed software filtering systems providing alternative classification techniques.
- Provide a deep understanding of the structural complexity of runtime packers.
- Study the limitations and propose heuristics to enable the application of multiple path exploration techniques for the generic unpacking of samples that implement partial code revelation.

Additionally, we have fulfilled the operational objectives defined in order to conduct this research.

- Propose and evaluate new representation methods capable of differentiating packed and non-packed samples.
- Develop and evaluate classification methods to improve the limitations of current approaches.
- Develop a system capable of capturing all the system events associated to the unpacking of complex packers.
- Develop a model to analyse all the system events produced during the unpacking of a sample.
- Propose a taxonomy capable of measuring the structural complexity of a packer from different perspectives.
- Study the limitations of current multi-path exploration techniques based on symbolic execution in order to deal with the unpacking of samples.
- Build a multi-path exploration based generic unpacking system.
- Propose and evaluate a heuristic to guide multi-path exploration in the context of software unpacking in order to improve its feasibility.

With the accomplishment of these specific and operational objectives, we believe that we have successfully *improved the packer analysis process from different perspectives in order to enable malware analysis*, the general objective for this dissertation. Therefore, we consider that the present work validates the general hypothesis established.

7.2 Discussion of the main shortcomings

In each chapter we have discussed the limitations for each of the contributions of this dissertation. Nevertheless, in this section we summarise some of the main shortcomings.

The first part of our research deals with packed binary classification. In order to test the feature-sets and approaches proposed, we collected and labelled a dataset formed by packed and non-packed binaries covering both malware and goodware samples and including different types of packers. In Chapter 3 we described the methodology followed to select and label this dataset. Nevertheless, this method was influenced by two different facts: (i) the availability of samples in the moment we conducted the experimentation and (ii) the limitations of the tools available. Although we believe that the dataset employed to validate our approach is sufficiently representative, it would be interesting to perform a large-scale analysis in order to test the capacity of these methods to classify packers in the current malware landscape.

Moreover, during this research we noticed that there are no publicly available datasets that satisfy the requirements to perform such a large-scale analysis. The off-the-shelf packer dataset described in Chapter 5 is composed of unpack-me challenges and, in most cases, there is only one single binary per packer configuration. Also, the binary protected is in most cases the same original program. For this reason, this dataset lacks the variability necessary for these experiments. A few datasets or services provide information about the packer used to protected the sample (e.g. VirusTotal), but unfortunately these services rely on signature-based detection tools and present the same limitations: custom packed samples are not labelled, and signature based detection tools do not always provide a sound detection rate for scrambled versions of known packers.

This limitation leads us to the second shortcoming of the approaches proposed. It is well known that malware analysis represents a moving target. Every successful detection method is studied by malware writers and sooner or later evaded. Any of the existing static approaches for packed binary filtering can be evaded by carefully adjusting the binary to fit the model representing non-packed binaries. This kind of attacks have also been referred to in the literature as mimicry attacks [GJM06, BCL07, KKM+05]. Certain aspects such as the lax implementation of Windows loaders and its versatility allow the malware writers to tweak binaries in innumerable ways [JS12]. Unfortunately, our approaches are also vulnera-

ble to this kind of attacks.

The second part of this dissertation is focused on dynamic analysis techniques. We have successfully measured the structural complexity of two different datasets. This approach is based on whole-system emulation and fine-grained execution tracing. These approaches provide a rich source of information at a high computational cost. Considering the current number of malware samples in the wild (e.g., the Anubis database has received more than 60 million unique binaries since 2007) this fine-grained granularity might be too heavy for large scale analysis. In order to operate at this scale, it would be interesting to study possible optimisations of our model.

Finally, although we have proposed several domain specific optimisations for multi-path exploration, this kind of approaches are far from being adequate for their deployment in real scenarios. These complex implementations suffer from many limitations when dealing with current malware that implements uncountable obfuscations in order to hinder reverser engineering.

7.3 Future lines of research

Below we describe some possible solutions to the main shortcomings identified in the previous section. Also, we outline some lines of future research based on the conclusions extracted.

1. **Large scale evaluation of static filtering methods.** We have successfully evaluated our static binary classification methods over a specific dataset. Nevertheless, the efficiency of this kind of static analysis approaches allows to apply them in a large-scale fashion. Such studies would benefit the community and help researchers understand the current malware landscape.

2. **Aiding binary labelling task with our proposed run-time packer analysis framework.** One of the main limitations found during the labelling process of our dataset was the availability of packer analysis tools beyond signature-based detection and classic heuristics such as entropy. We believe that this labelling efforts would have been reduced by employing a tool like the one developed in the second phase of this dissertation. Also, this kind of tools can also help to reduce both the false-positives and false-negatives produced by signature based packer detection tools.

3. **Study of new static filtering techniques.** Current on-line services do not provide much information about whether the sample is packed or not. For instance, VirusTotal scans the sample with 3 different signature based tools: PEiD, TrID, and F-Prot. There also private signature databases for packed binaries such as Sigbuster. Nevertheless, malware writers typically employ scrambled or modified versions of these packers, or implement their custom packers in order to evade detection. Given this background, we believe that these services require new and more effective methods to classify binaries besides the classic heuristics that are now well-known by malware writers.

4. **Study of the resilience of different static analysis techniques to obfuscations and mimicry attacks.** Another possible future line of research would be to systematically study the resilience to mimicry attacks of the different approaches proposed to date for packed binary classification. It is well-known that binaries are modified in many different ways by malware writers. To which extent is it possible to modify a binary in order to evade detection? How much can it resemble to a non-packed binary?

5. **Graph similarity analysis to identify scrambled versions of packers.** Finding scrambled versions of well-known packers in malware databases can be beneficial in many different ways. First, many tools incorporate unpacking routines for well-known packers. These tools are generally more efficient than generic unpackers. Identifying these scrambled versions may allow to apply these scripts to the modified versions of the packers. Also, this would help in understanding how malware writers perform these modifications in order to develop more effective detection methods. We believe that our run-time packer analysis framework could be leveraged to study and identify these scrambled versions of packers.

6. **Evaluating the impact of the proposed visualisation technique on the reverse engineering task.** As part of our run-time packer analysis framework, we have developed a packer structure visualisation tool to quickly overview the structure of a packer. It is well-known that data visualisation can aid reverse engineering. In order to evaluate the usefulness of our visualisation approach, it would be interesting to conduct experiments with experts and non-experts in the field.

7. **Deployment of a scalable on-line run-time packer analysis sandbox.** While there are many on-line malware analysis sandboxes, there are not available services for run-time packer analysis. Security researchers and computer emergency response teams may benefit from this tool. Also, it would provide a very rich and diverse source of information for future experiments on this field.

8. **Study of new multi-path exploration techniques to deal with different obfuscations presented by complex packers.** Complex packers, as well as highly obfuscated malware, still represent a challenge for multi-path exploration. In fact, dealing with trigger based behaviour in dynamic analysis platforms is still an open research problem. As we have affirmed in our study, not all the scenarios require the consistent execution of every possible path. Which other optimisations could be developed in order to enhance the feasibility of these approaches?

7.4 Final remarks

The majority of current malware is still protected by run-time packers. Nevertheless, the absence of tools for comprehensive packer analysis, the lack of information about packers in on-line malware analysis sandboxes and the fact that there are no clear studies about packer prevalence in the current malware landscape make us think that the research community has shifted the efforts towards other problems leaving the run-time packer problem partially unsolved. It is true that generic unpacking approaches proposed to date can deal with simple packers. Nevertheless, recent studies [SM14] highlight that it is relatively simple for malware writers to implement evasion techniques in order to avoid detection. Different academic publications [BLB11, SLGL08, SRL12] have tried to attract the attention on still unsolved problems in the domain of run-time packers. Real examples like the Gauss virus [Lis13] reveal that it is possible to completely thwart reverse engineering employing relatively simple (but clever) protection techniques.

Unfortunately, malware research is a cat and mouse game in which the malware writer generally plays one step ahead.

Bibliography

[AMDB07] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In "Information Hiding", pages 232–248. Springer (2007).

[AW99] Shun-ichi Amari and Si Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks* **12**(6), 783–789 (1999).

[Bay63] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society* **53**, 370–418 (1763).

[Bay09] Ulrich Bayer. "Large-Scale Dynamic Malware Analysis". PhD thesis, (2009).

[BCK+10] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In "Network and Distributed System Security Symposium (NDSS)" (2010).

[BCL07] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. An efficient technique for preventing mimicry and impossible paths execution attacks. In "Performance, Computing, and Communications Conference (IPCCC)", pages 418–425. IEEE (2007).

[BCSB13] Denis Bueno, Kevin J Compton, Karem A Sakallah, and Michael Bailey. Detecting traditional packers, decisively. In "Research in Attacks, Intrusions, and Defenses", pages 184–203. Springer (2013).

[BE13] Suhabe Bugrara and Dawson R Engler. Redundant state detection for dynamic symbolic execution. In "USENIX Annual Technical Conference", pages 199–211 (2013).

[BEKLL13] Munkhbayar Bat-Erdene, Taebeom Kim, Hongzhe Li, and Heejo Lee. Dynamic classification of packing algorithms for inspecting executables using entropy analysis. In "8^{th} International Conference on Malicious and Unwanted Software (MALWARE)", pages 19–26. IEEE (2013).

[BF04] Remco R Bouckaert and Eibe Frank. Evaluating the replicability of significance tests for comparing learning algorithms. In "Advances in Knowledge Discovery and Data Mining", pages 3–12. Springer (2004).

[BHK+07] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope: Automatically dissecting malicious binaries. *School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-07-133* (2007).

[BHL+08] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In "Botnet Detection", pages 65–88. Springer (2008).

[BIG+13] Tao Ban, Ryoichi Isawa, Shanqing Guo, Daisuke Inoue, and Koji Nakao. Efficient malware packer identification using support vector machines with spectrum kernel. In "8^{th} Asia Joint Conference on Information Security (Asia JCIS)", pages 69–76. IEEE (2013).

[Bis06] Christopher M Bishop. "Pattern recognition and machine learning". Springer New York. (2006).

[BK09] Komal Babar and Faiza Khalid. Generic unpacking techniques. In "2^{nd} International Conference on Computer, Control and Communication (IC4), 2009", pages 1–6. IEEE (2009).

[BKK06] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. "TTAnalyze: A tool for analyzing malware". (2006).

[BKNS00] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In "ACM Sigmod Record", volume 29, pages 93–104. ACM (2000).

[BLB11] Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Thwarting real-time dynamic unpacking. In "Proceedings of the Fourth European Workshop on System Security", page 5. ACM (2011).

[BM06] Tom Brosch and Maik Morgenstern. Runtime packers: The hidden problem. *Black Hat USA* (2006).

[Böh08] Lutz Böhne. Pandora's bochs: Automatic unpacking of malware. *University of Mannheim* (2008).

[Bre96] Leo Breiman. Bagging predictors. *Machine learning* **24**(2), 123–140 (1996).

[Bre01] Leo Breiman. Random forests. *Machine learning* **45**(1), 5–32 (2001).

[BWJS07] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In "20^{th} IEEE Computer Security Foundations Symposium (CSF)", pages 311–325. IEEE (2007).

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)* **41**(3), 15 (2009).

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In "Proceedings of the 8^{th} USENIX conference on Operating Systems Design and Implementation (OSDI)", volume 8, pages 209–224 (2008).

[CDKT09] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic static unpacking of malware binaries. In "16^{th} Working Conference on Reverse Engineering (WCRE)", pages 167–176. IEEE (2009).

[CGH96] Enrique Castillo, José M. Gutiérrez, and Ali S. Hadi. "Expert Systems and Probabilistic Network Models". Springer, New York, NY, USA (1996).

[CJMS09] Juan Caballero, Noah Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In "Proceedings of the 17^{th} Annual Network and Distributed System Security Symposium", pages 391–408. ISOC (2009).

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News* **39**(1), 265–278 (2011).

[CKJ+05] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware normalization. Technical Report, University of Wisconsin (2005).

[CKNZ12] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In "Tools for Practical Software Verification", pages 1–30. Springer (2012).

[CKOR08] Yang-seo Choi, Ik-kyun Kim, Jin-tae Oh, and Jae-cheol Ryou. Pe file header analysis-based packed pe file detection technique (phad). In "International Symposium on Computer Science and its Applications (CSA)", pages 28–31. IEEE (2008).

[CL06] Mohamed R Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In "Proceedings of the 4^{th} ACM workshop on Recurring Malcode", pages 73–78. ACM (2006).

[CLD11] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In "Proceedings of the 18^{th} ACM Conference on Computer and Communications Security", pages 275–284. ACM (2011).

[CO07] Boja Catalin and A Vi Oiu. Optimization of antivirus software. *Informatica* **11**(2007), 99–102 (2007).

[Coh86] Fred Cohen. "Computer viruses". PhD thesis, (1986).

[Coh95] William W Cohen. Fast effective rule induction. In "Twelfth International Conference on Machine Learning (ICML)", volume 95, pages 115–123 (1995).

[CPM+10] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babic, and Dawn Song. Input generation via decomposition and re-stitching: Finding bugs in malware. In "Proceedings of the 17^{th} ACM Conference on Computer and Communications Security", pages 413–425. ACM (2010).

[CSS08] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In "Detection of Intrusions and Malware, and Vulnerability Assessment", pages 143–163. Springer (2008).

[CVERL02] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical Report, (2002).

[CX10] Silvio Cesare and Yang Xiang. Classification of malware using structured control flow. In "Proceedings of the 8^th Australasian Symposium on Parallel and Distributed Computing-Volume 107", pages 61–70. Australian Computer Society, Inc. (2010).

[DCKB07] Richard Dawkins, Alan Clements, Deborah Kidd, and Russell Barnes. "The Root of All Evil?" IWC Media (2007).

[DCT08] Saumya Debray, Kevin Coogan, and Gregg Townsend. On the semantics of self-unpacking malware code. Technical Report, Dept. of Computer Science, University of Arizona, Tucson (2008).

[Dem06] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* **7**, 1–30 (2006).

[Der03] Michael Derntl. Basics of research paper writing and publishing. *Faculty of Computer Sciencie, University of Vienna, Austria* (2003).

[DGHH+14] Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering for the greater good with panda. Technical Report, (2014).

[DGLHL13] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In "Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security", pages 839–850. ACM (2013).

[Die98] Thomas G Dietterich. Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation* **10**(7), 1895–1923 (1998).

[Dij76] Edsger Wybe Dijkstra. "A discipline of programming", volume 4. Prentice-hall Englewood Cliffs (1976).

[DK05] Kai-Bo Duan and S Sathiya Keerthi. Which is the best multiclass svm method? an empirical study. In "Multiple Classifier Systems", pages 278–285. Springer (2005).

[DP10] Saumya Debray and Jay Patel. Reverse engineering self-modifying code: Unpacker extraction. In "17^{th} Working Conference on Reverse Engineering (WCRE)", pages 131–140. IEEE (2010).

[DRSL08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In "Proceedings of the 15^{th} ACM conference on Computer and Communications security", pages 51–62. ACM (2008).

[DZX13] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: stealthy binary program instrumentation and debugging via hardware virtualization. In "Proceedings of the 29th Annual Computer Security Applications Conference", pages 289–298. ACM (2013).

[Fer07] Peter Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[Fer08] Peter Ferrie. Anti-unpacker tricks. In "Amsterdam: CARO Workshop" (2008).

[FGG97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine learning* **29**(2-3), 131–163 (1997).

[FHJ52] Evelyn Fix and Joseph L Hodges Jr. Discriminatory analysis: Nonparametric discrimination: Small sample performance. *Technical Report Project 21-49-004, Report Number 11* (1952).

[FHSL96] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In "Proceedings of IEEE Symposium on Security and Privacy", pages 120–128. IEEE (1996).

[FS01] Cormac Flanagan and James B Saxe. Avoiding exponential explosion: Generating compact verification conditions. *ACM SIGPLAN Notices* **36**(3), 193–205 (2001).

[FW98] Eibe Frank and Ian H Witten. Generating accurate rule sets without global optimization. In "Proceedings of the 15^th International Conference on Machine Learning", pages 144–151. Morgan Kaufmann Publishers Inc. (1998).

[G+95] Stephen R Garner et al.. Weka: The Waikato environment for knowledge analysis. In "Proceedings of the New Zealand Computer Science Research Students Conference", pages 57–64 (1995).

[GFC08] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In "Proceedings of the 2008 Conference on Recent Advances in Intrusion Detection (RAID)", pages 98–115 (2008).

[GHD12] Sudeep Ghosh, Jason Hiser, and Jack W Davidson. Replacement attacks against vm-protected applications. In "ACM SIGPLAN Notices", volume 47, pages 203–214. ACM (2012).

[GJM06] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Automated discovery of mimicry attacks. In "Recent Advances in Intrusion Detection", pages 41–60. Springer (2006).

[GKRW10] Isabelle Gnaedig, Matthieu Kaczmarek, Daniel Reynaud, and Stéphane Wloka. Unconditional self-modifying code elimination with dynamic compiler optimizations. In "5^{th} International Conference on Malicious and Unwanted Software (MALWARE)", pages 47–54. IEEE (2010).

[GMRP09] Wadie Guizani, J-Y Marion, and Daniel Reynaud-Plantey. Server-side dynamic code analysis. In "4^{th} International Conference on Malicious and Unwanted Software (MALWARE)", pages 55–62. IEEE (2009).

[Gri01] Roger Grimes. "Malicious mobile code: Virus protection for Windows". O'Reilly Media (2001).

[Gru69] Frank E Grubbs. Procedures for detecting outlying observations in samples. *Technometrics* **11**(1), 1–21 (1969).

[HKY99] Laurie J Heyer, Semyon Kruglyak, and Shibu Yooseph. Exploring expression data: identification and analysis of coexpressed genes. *Genome research* **9**(11), 1106–1115 (1999).

[HSKS03] Katherine Heller, Krysta Svore, Angelos D Keromytis, and Salvatore Stolfo. One class support vector machines for detecting anomalous windows registry accesses. In "Workshop on Data Mining for Computer Security (DMSEC), Melbourne, FL, November 19, 2003", pages 2–9 (2003).

[ID80] Ronald L Iman and James M Davenport. Approximations of the critical region of the friedman statistic. *Communications in Statistics-Theory and Methods* **9**(6), 571–595 (1980).

[Int13] Intel. Intel 64 and ia-32 architectures software developer's manual. (2013).

[JCL+10] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee. Generic unpacking using entropy analysis. In "5*th* International Conference on Malicious and Unwanted Software (MALWARE)", pages 98–105. IEEE (2010).

[JCN+13] Grégoire Jacob, Paolo Milani Comparetti, Matthias Neugschwandtner, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. In "Detection of Intrusions and Malware, and Vulnerability Assessment", pages 102–122. Springer (2013).

[JDF08] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology* **4**(3), 251–266 (2008).

[JS12] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In "IEEE Symposium on Security and Privacy", pages 80–94. IEEE (2012).

[JWL+12] Chunfu Jia, Zhi Wang, Kai Lu, Xinhai Liu, and Xin Liu. Directed hidden-code extractor for environment-sensitive malwares. *Physics Procedia* **24**, 1621–1627 (2012).

[KIE+09] Hyung Chan Kim, Daisuke Inoue, Masashi Eto, Yaichiro Takagi, and Koji Nakao. Toward generic unpacking techniques for malware analysis with quantification of code revelation. In "The 4^{th} Joint Workshop on Information Security" (2009).

[KII10] Yuhei Kawakoya, Makoto Iwamura, and Mitsutaka Itoh. Memory behavior-based automatic malware unpacking in stealth debugging environment. In "5th International Conference on Malicious and Unwanted Software (MALWARE)", pages 39–46. IEEE (2010).

[Kin12] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In "19^th Working Conference on Reverse Engineering (WCRE)", pages 61–70. IEEE (2012).

[KKK11] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In "Proceedings of the 18^th ACM Conference on Computer and Communications Security", pages 285–296. ACM (2011).

[KKM+05] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In "Proceedings of the 14^{th} conference on USENIX Security Symposium", pages 11–11. USENIX Association (2005).

[Kle96] EM Kleinberg. An overtraining-resistant stochastic modeling method for pattern recognition. *The annals of statistics* **24**(6), 2319–2349 (1996).

[KM04] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In "Proceedings of the 10^{th} ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)", pages 470–478 (2004).

[Koh95] Rons Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In "International Joint Conference on Artificial Intelligence", volume 14, pages 1137–1145 (1995).

[Koh01] Teuvo Kohonen. "Self-organizing maps", volume 30. Springer (2001).

[Kor14] Joxean Koret. Breaking antivirus software (2014). Available on-line: `http://mincore.c9x.org/breaking_av_software.pdf`.

[Kot07] Sotiris B Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. In "Proceeding of the Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies", pages 3–24 (2007).

[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In "Proceedings of the 2007 ACM Workshop on Recurring Malcode", pages 46–53 (2007).

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In "Proceedings of the 13^{th} conference on USENIX Secu-

rity Symposium", pages 18–18. USENIX Association Berkeley, CA, USA (2004).

[Kum00] Vipin Kumar. An introduction to cluster analysis for data mining. *Computer Science Department, University of Minnesota, USA* (2000).

[KYH+09] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In "Proceedings of the 1^{st} ACM workshop on Virtual machine security", pages 11–22. ACM (2009).

[Lab10] Reversing Labs. Titanmist: Your first step to reversing nirvana. In "Black Hat USA Conference Whitepaper" (2010).

[Lei05] K Rustan M Leino. Efficient weakest preconditions. *Information Processing Letters* **93**(6), 281–288 (2005).

[LEK+03] Aleksandar Lazarevic, Levent Ertoz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A comparative study of anomaly detection schemes in network intrusion detection. In "Proceedings of the 3^{th} SIAM International Conference on Data Mining" (2003).

[LH07] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy* **5**(2), 40–45 (2007).

[Lis13] Secure List. The mystery of the encrypted gauss payload (2013). Available on-line: `http://securelist.com/blog/incidents/33561/the-mystery-of-the-encrypted-gauss-payload-5/`.

[LJK+00] Jorma Laurikkala, Martti Juhola, Erna Kentala, N Lavrac, S Miksch, and B Kavsek. Informal identification of outliers in medical data. In "Proceedings of the 5^{th} International Workshop on Intelligent Data Analysis in Medicine and Pharmacology", pages 20–24. Citeseer (2000).

[LKC11] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In "Recent Advances in Intrusion Detection", pages 338–357. Springer (2011).

[Man13] Mandiant. Apt1: Exposing one of china's cyber espionage units (2013). Available on-line: http://intelreport.mandiant.com/Mandiant_APT1_Report.pdf.

[Mas05] Yury Mashevsky. Watershed in malicious code evolution (2005). Available on-line: http://securelist.com/analysis/36053/watershed-in-malicious-code-evolution/.

[MC10] Revision Microsoft Corporation. Microsoft portable executable and common object file format specification, revision 8.2 - september 21 (2010). Available on-line: http://msdn.microsoft.com/en-us/library/windows/hardware/gg463119.aspx.

[McA08] McAfee. The mcafee gateway anti-malware engine. (2008).

[McA09] McAfee. The good, the bad and the unknown (2009). Available on-line:http://www.mcafee.com/us/resources/white-papers/wp-good-bad-the-unknown.pdf.

[McA12a] McAfee. McAfee Virus Glossary (2012). Available on-line: http://home.mcafee.com/virusinfo/glossary?ctst=1#M.

[McA12b] McAfee Labs. Mcafee 2012 threats predictions (2012). Available on-line: http://www.mcafee.com/us/resources/reports/rp-threat-predictions-2012.pdf.

[McA13] McAfee Labs. Mcafee threats report: Fourth quarter 2013 (2013). Available on-line: http://www.mcafee.com/sg/resources/reports/rp-quarterly-threat-q4-2013.pdf.

[McA14] McAfee Labs. Mcafee threats report: Second quarter 2014 (2014). Available on-line: http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q2-2014.pdf.

[MCJ07] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In "Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)", pages 431–441 (2007).

[MCJ+10] Charlie Miller, Juan Caballero, Noah M Johnson, Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Crash analysis with bitblaze. *BlackHat USA 2010, Whitepaper* (2010). Available on-line: https://media.blackhat.com/bh-us-10/whitepapers/Miller/BlackHat-USA-2010-CMiller-Bitblaze-wp.pdf.

[MKK07a] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In "Proceedings of the 23^{rd} Annual Computer Security Applications Conference (ACSAC)", pages 421–430 (2007).

[MKK07b] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In "Proceedings of the 28^{th} IEEE Symposium on Security and Privacy" (2007).

[MMF10] Samir Mody, Igor Muttik, and Peter Ferrie. Standards and policies on packer use. In "Proceedings of the Virus Bulletin Conference", pages 272–280 (2010).

[MP10] Maik Morgenstern and Hendrik Pilz. Useful and useless statistics about viruses and anti-virus programs. In "Proceedings of the CARO Workshop" (2010).

[MR13] Jean-Yves Marion and Daniel Reynaud. Wave analysis of advanced self-modifying behaviors. *Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel* pages 137–152 (2013).

[Mun14] Ken Munro. Defeating corporate anti-virus (2014). Available on-line: `https://www.pentestpartners.com/blog/defeating-corporate-anti-virus/`.

[Nac98] Carey Nachenberg. Understanding heuristics: Symantec's bloodhound technology (1998).

[NB03] Claude Nadeau and Yoshua Bengio. Inference for the generalization error. *Machine Learning* **52**(3), 239–281 (2003).

[NLGV12] Smita Naval, Vijay Laxmi, M. S. Gaur, and P. Vinod. Spade: Signature based packer detection. In "Proceedings of the First International Conference on Security of Internet of Things (SecurIT)", pages 96–101, New York, NY, USA (2012). ACM.

[NR14] Gabriel Negreira and Rodrigo Rubira. Prevalent characteristics in modern malware (2014). Available on-line: https://www.blackhat.com/docs/us-14/materials/us-14.Branco-Prevalent-Characteristics-In-Modern-Malware.pdf.

[Pan12] Panda Antivirus. Cibercrimen (2012). Available on-line: http://www.pandasecurity.com/spain/homeusers/security-info/cybercrime/.

[PDZ+14] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In "Proceedings of the 2014 USENIX Security Symposium, San Diego, CA" (2014).

[PGL06] Roberto Perdisci, Guofei Gu, and Wenke Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In "6^{th} International Conference on Data Mining (ICDM)", pages 488–498. IEEE (2006).

[PLL08a] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters* **29**(14), 1941–1946 (2008).

[PLL08b] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In "Annual Computer Security Applications Conference", pages 301–310. IEEE (2008).

[PMRB09] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In "Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) Vol 41.", page 86 (2009).

[PRM11] Kil Jin Brandini Park, Rodrigo Ruiz, and Antonio Montes. Tool for recognition of packed executables. *International Journal of Forensic Computer Science* **6**(1), 44–58 (2011).

[QL09] Daniel A Quist and Lorie M Liebrock. Visualizing compiled executables for malware analysis. In "6^{th} International Workshop on Visualization for Cyber Security (VizSec)", pages 27–32. IEEE (2009).

[QS07] Daniel Quist and Val Smith. Covert debugging circumventing software armoring techniques. *Black Hat Briefings USA* (2007).

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine learning* **1**(1), 81–106 (1986).

[Qui93] J. Ross Quinlan. "C4. 5 programs for machine learning". Morgan Kaufmann Publishers (1993).

[Res07] Panda Research. Malware formation statistics (2007). Available on-line: `http://www.pandasecurity.com/mediacenter/malware/malwareformation-statistics/`.

[RHD+06] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In "22^{nd} Annual Computer Security Applications Conference (ACSAC)", pages 289–300. IEEE (2006).

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. *Information Security* pages 1–18 (2007).

[RM13] Kevin A Roundy and Barton P Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)* **46**(1), 4 (2013).

[RMI11] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Evolution of computer virus concealment and anti-virus techniques: A short survey. *International Journal of Computer Science Issues (IJCSI)* **8**(1) (2011).

[RN03] Stuart Russell and Peter Norvig. "Artificial Intelligence: A Modern Approach". Prentice Hall (2003).

[Rol09] Rolf Rolles. Unpacking virtualization obfuscators. In "3^{rd} USENIX Workshop on Offensive Technologies.(WOOT)" (2009).

[SAB10] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In "IEEE Symposium on Security and Privacy (SP)", pages 317–331. IEEE (2010).

[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. pages 1–25 (2008).

[SEZS01] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In "Proceedings of the 22^{nd} IEEE Symposium on Security and Privacy.", pages 38–49 (2001).

[SKH11] Orathai Sukwong, Hyong S Kim, and James C Hoe. Commercial antivirus software effectiveness: an empirical study. *Computer* **44**(3), 0063–70 (2011).

[Sku90] Fridrik Skulason. Virus encryption techniques. *Virus Bulletin* pages 13–16 (1990).

[SLGL08] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In "Network and Distributed System Security Symposium (NDSS)" (2008).

[SLGL09] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In "30^{th} IEEE Symposium on Security and Privacy", pages 94–109. IEEE (2009).

[SLTW04] Vladimir Svetnik, Andy Liaw, Christopher Tong, and Ting Wang. Application of Breiman's random forest to modeling structure-activity relationships of pharmaceutical molecules. *Multiple Classifier Systems* pages 334–343 (2004).

[SM14] Arne Swinnen and Alaeddine Mesbahi. One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques. *BlackHat USA 2014, Whitepaper* (2014). Available online: https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf.

[SPDB09] Igor Santos, Yoseba K Penya, Jaime Devesa, and Pablo Garcia Bringas. N-Grams-based file signatures for malware detection. In "Proceedings of the 11^{th} International Conference on Enterprise Information Systems (ICEIS)", pages 317–320 (2009).

[SPST+01] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural Computation* **13**(7), 1443–1471 (2001).

[SRL12] Chengyu Song, Paul Royal, and Wenke Lee. Impeding automated malware analysis with environment-sensitive malware. In "USENIX Summit on Hot Topics in Security" (2012).

[SSWB00] Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural Computation* **12**(5), 1207–1245 (2000).

[Ste05] Adrian E Stepan. Defeating polymorphism: Beyond emulation. In "Proceedings of the Virus Bulletin International Conference" (2005).

[Ste06a] Adrian E Stepan. Improving proactive detection of packed malware. *Proceedings of the International Virus Bulletin Conference* (2006).

[Ste06b] Joe Stewart. Ollybone: Semi-automatic unpacking on ia-32. In "Proceedings of the 14^{th} DEF CON Hacking Conference" (2006).

[STF09] M. Shafiq, S. Tabish, and M. Farooq. Pe-probe: leveraging packer detection and structural information to detect malicious portable executables. In "Proceedings of the Virus Bulletin Conference (VB)", pages 29–33 (2009).

[STMF09] M Zubair Shafiq, S Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: mining structural information to detect malicious executables in realtime. In "Recent Advances in Intrusion Detection", pages 121–141. Springer (2009).

[SU04] Thomas F Stafford and Andrew Urbaczewski. Spyware: The ghost in the machine. *Communications of the Association for Information Systems* **14**, 291–306 (2004).

[Sun12] Li Sun. "REFORM: A framework for malware packer analysis using information theory and statistical methods". PhD thesis, (2012).

[SW49] Claude E Shannon and Warren Weaver. The mathematical theory of communication (1949).

[Sym11a] Symantec Security Response. W32 duqu: The precursor to the next stuxnet (2011). Available on-line: `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next_stuxnet.pdf`.

[Sym11b] Symantec Security Response. W32.stuxnet dossier (2011). Available on-line `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`.

[SYS+08] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A Framework for Enabling Static Malware Analysis. In "Proceedings of the European Symposium on Research in Computer Security (ESORICS)", pages 481–500 (2008).

[Szo05] Peter Szor. "The art of computer virus research and defense". Addison-Wesley Professional (2005).

[TD05] William MK Trochim and JP Donnelly. "Research methods: The concise knowledge base". Atomic Dog Pub. (2005).

[TXZ06] Yufei Tao, Xiaokui Xiao, and Shuigeng Zhou. Mining distance-based outliers from large databases in any metric space. In "Proceedings of the 12^{th} ACM SIGKDD international conference on Knowledge Discovery and Data Mining", pages 394–403. ACM (2006).

[Van05] Pierre Vandevenne. Using the universal pe plug-in in ida pro 4.9 to unpack compressed executables (2005).

[Vir12] Viruslist. Viruslist.com - 1970's (2012). Available on-line: http://www.viruslist.com/sp/viruses/encyclopedia?chapter=153310937.

[VY05] Amit Vasudevan and Ramesh Yerraballi. Stealth breakpoints. In "21^{st} Annual Computer Security Applications Conference", pages 10–pp. IEEE (2005).

[VY06] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In "IEEE Symposium on Security and Privacy", pages 15–pp. IEEE (2006).

[WBR13] Christian Wressnegger, Frank Boldewin, and Konrad Rieck. Deobfuscating embedded malware using probable-plaintext attacks. In "Research in Attacks, Intrusions, and Defenses", pages 164–183. Springer (2013).

[WCZ09] Yanjun Wu, Tzi-cker Chiueh, and Chen Zhao. Efficient and automatic instrumentation for packed binaries. In "Advances in Information Security and Assurance", pages 307–316. Springer (2009).

[WPS06] Ke Wang, Janak J Parekh, and Salvatore J Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In "Recent Advances in Intrusion Detection", pages 226–248. Springer (2006).

[WQZ+03] Li Wei, Weining Qian, Aoying Zhou, Wen Jin, and X Yu Jeffrey. Hot: Hypergraph-based outlier test for categorical

data. In "Advances in Knowledge Discovery and Data Mining", pages 399–410. Springer (2003).

[WSAR13] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on n-grams in intrusion detection: Anomaly detection vs. classification. In "Proceedings of the 6^{th} ACM Workshop on Artificial Intelligence and Security", pages 67–76. ACM (2013).

[Wyk11] James Wyke. What is zeus?. (2011). Available on-line: http://www.sophos.com/es-es/why-sophos/our-people/technical-papers/what-is-zeus.aspx.

[YJZY12] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and software emulation for transparent and extensible malware analysis. In "8^{th} ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments". ACM (2012).

[Zha08] Qinghua Zhang. "Polymorphic and Metamorphic Malware Detection". PhD thesis, North Carolina State University (2008).

This dissertation was finished
in Bilbao, January 13, 2015.