Packing detection and classification relying on machine learning to stop malware propagation

Lamine Noureddine

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
Lamine NOUREDDINE

Packing detection and classification relying on machine learning to stop malware propagation

Thèse présentée et soutenue à Rennes, le 21 décembre 2021
Unité de recherche : Inria

Rapporteurs avant soutenance :

Frédéric ALEXANDRE	Directeur de Recherche, UMR IMN / Inria Bordeaux
Henri-Pierre CHARLES	Directeur de Recherche, CEA LIST Grenoble

Composition du Jury :

Président :	Jean-François LALANDE	Professeur, CentraleSupélec Rennes
Examinateurs :	Isabelle CHRISMENT	Professeur, Telecom Nancy / Université de Lorraine
	Annelie HEUSER	Chargé de Recherche, CNRS, IRISA Rennes
	Christelle URTADO	Maître de Conférences, IMT Mines Alès
Dir. de thèse :	Stéphane UBEDA	Professeur, INSA de Lyon, détaché à Inria Grenoble
Co-dir. de thèse :	Olivier ZENDRA	Chargé de Recherche, Inria Rennes

Remerciements

Voilà que ces quatre années de thèse s'achèvent par un succès. Je suis tout content et joyeux d'être devenu docteur en informatique. Dans cette lettre de remerciement, je souhaiterais exprimer ma reconnaissance et ma gratitude envers tous ceux qui ont participé, directement ou indirectement, à atteindre ce succès et à remplir cette joie.

Tout d'abord, je veux remercier Dieu, le tout miséricordieux, d'avoir exaucé mes prières et de m'avoir ainsi offert l'opportunité de poursuivre une thèse de doctorat en France, avec une thématique passionnante en sécurité informatique, comme je le souhaitais. Oui, cette thèse s'est déroulée à l'Inria Rennes – Bretagne Atlantique, qui non seulement est un des meilleurs endroits pour faire de la recherche en France, particulièrement en sécurité informatique, mais qui se trouve aussi à Rennes, en Bretagne, cette merveilleuse région de France. Je le remercie également de m'avoir accordé la sérénité, la patience et la persévérance tout au long de ce parcours, pour finalement l'achever par ce manuscrit et une très belle soutenance.

Je tiens ensuite à remercier mes directeurs de thèse et encadrants. Merci à mon directeur de thèse Stéphane Ubéda pour les échanges scientifiques fructueux qui m'ont parfois permis d'éviter de se diriger vers de fausses pistes, en particulier dans le cadre de ma deuxième contribution de thèse. Merci à mon co-directeur de thèse Olivier Zendra de m'avoir suivi et encadré pendant la majeure partie de cette thèse. Merci pour ses retours constructifs lors de nos discussions scientifiques hebdomadaires. Merci pour son soin particulier à répondre à mes préoccupations diverses. Merci pour ses relectures pour mes écrits et présentations. Grâce à ses recommandations, j'ai significativement amélioré ma clarté à l'oral, mon écrit, et nettement amélioré la qualité de mes présentations. Il m'a appris à mieux croire en moi, à rester toujours positif et à ne jamais dénigrer mon travail. Je le remercie pour toutes ses transmissions, pour ses encouragements et remotivations sans cesse jusqu'à la fin. Merci à mon encadrante Annelie Heuser de m'avoir aussi suivi et encadré avec Olivier. Merci pour sa grande sympathie et gentillesse. Merci pour sa disponibilité, malgré son emploi du temps très chargé. Aussi, comme pour Olivier, je la remercie pour ses retours constructifs, ses relectures pour mes écrits, son soutien et ses encouragements sans cesse jusqu'à la fin.

Merci à Axel Legay, mon premier directeur de thèse, qui m'a accueillis au sein de l'équipe TAMIS et qui m'a donné la possibilité de poursuivre cette thèse. Merci à Fabrizio Biondi, mon premier encadrant, de m'avoir suivi et encadré durant ma première année de thèse. Merci pour les précieuses discussions scientifiques qui ont contribué à me donner une base solide pour la suite de ma thèse et à me former en tant que jeune chercheur. L'article de recherche que nous avons publié ensemble en restera un souvenir parlant.

Je remercie tous les membres de mon jury de thèse, à commencer par les rapporteurs Henri-Pierre Charles et Frédéric Alexandre pour le temps qu'ils ont consacré à relire mon manuscrit de thèse. Les points qu'ils ont soulevé ont permis d'améliorer ce manuscrit de thèse dans sa version finale. Merci à Isabelle Chrisment et à Christelle Urtado pour l'intérêt qu'elles ont manifesté à l'égard de mes travaux de thèse. De même, merci à Aurélien Francillon et à Jean-Yves Marion, membres de mon comité de suivi individuel doctoral, pour l'intérêt porté à mes travaux de thèse et pour le suivi de mon avancement annuel.

Je me dois aussi de remercier le partenaire industriel CISCO, en particulier l'équipe de Steve Rich, pour avoir stimulé les discussions scientifiques et fourni un vaste jeu de données de malware réels.

Par ailleurs, j'exprime ma gratitude à Sylvain Guilley. C'est en grande partie grâce à lui que cette opportunité de thèse s'est présentée à moi.

J'adresse mes remerciements à tous les membres de l'ex-équipe de recherche TAMIS, dans laquelle j'ai passé la plus grande partie de ma thèse. Certains d'entre eux sont devenus mes amis au fil du temps. Merci pour tous les moments conviviaux, enrichissants et apaisants que nous avons passé ensemble. Je pense naturellement à Tristan, avec qui je partageait le bureau. Merci pour son accueil, ses échanges très diverses et agréables, les parties au jeu de Go et les séances d'escalade. Merci à l'adorable Cassius pour son aide technique inestimable dans le domaine des malware et de l'obfuscation, son aide organisationnelle pour le bon déroulement de ma soutenance, son humour déstressant, son soutien et ses encouragements jusqu'à la fin. Merci à Stefano pour son dévouement acharné à m'aider à débloquer des pistes de recherche, en particulier pour ma deuxième contribution de thèse. Merci d'avoir partagé avec moi de nombreuses astuces pour la structuration, l'écriture et la relecture d'un article scientifique. Merci pour le partage des retours d'expériences concernant les conférences et journaux scientifiques. Merci pour son soutien moral et ses encouragements sincères. Merci à Alex pour les nombreuses

discussions assez variées et passionnantes que nous avons eu ensemble, en particulier lorsqu'il s'agissait des malware et de l'obfuscation. Merci pour son soutien moral et les astuces en tout genre qu'il m'a fait partager pour que je puisse tenir bon, surtout en fin de thèse. Merci à Sébastien Josse qui m'a accueilli plusieurs fois chez lui pour partager avec moi un tant soit peu de son large savoir pointu et technique dans le domaine des malware et de l'obfuscation, dont les packers font partie. Merci à Cécile Bouton pour son accueil et son assistance. Je n'oublie pas bien sûr Christophe, Matthieu, Yoann, Céline, Phuc, Yulliwas, Aghate, Olivier Decourbe, Nisrine, Delphine, Tania, Ioana, Thomas Given-Wilson, Najah, Laurent ... merci à vous tous, j'ai énormément appris de vous. Je souhaite à chacun d'entre vous le meilleur, et qui dit, ne sachant pas ce que l'avenir nous réserve, peut être que nous nous reverrons à d'autres occasions.

Mes remerciements vont aussi à tous les membres de l'équipe de recherche DiverSE, dans laquelle s'est effectuée la dernière année de ma thèse. Je n'ai malheureusement pas eu la chance de les rencontrer et d'échanger avec eux à proprement dit, notamment à cause de la fin de mon contrat de thèse avec l'Inria et principalement à cause de la pandémie COVID-19 qui nous a tous obligés à recourir au télétravail. Néanmoins, merci pour leur accueil, merci d'avoir suivi ma soutenance et de m'avoir félicité pour la réussite. Merci à Sophie Maupile pour sa gentillesse et son assistance. Merci à Olivier Barais de m'avoir soutenu et d'avoir assuré le bon déroulement de ma dernière année de thèse au sein de son équipe DiverSE.

Je suis également reconnaissant aux membres du personnel de l'Inria qui, par leur sérieux et dévouement, assurent un environnement de travail très agréable. De même pour les membres du personnel de l'université de Rennes 1, en particulier ceux de l'école doctorale MathSTIC, pour leur accueil et leur disponibilité à répondre aux préoccupations des doctorants, dont je faisait partie.

Je remercie profondément mes proches pour leur soutien, leurs encouragements et leurs chaleureuses félicitations pour ma réussite au doctorat. Merci du fond du cœur à mes parents Abdelkader et Lynda qui n'ont cessé de prier pour mon succès, m'ont encouragé et m'ont soutenu moralement et financièrement du début à la fin de cette thèse. Mais plus important, je les remercie de nous (moi, mon frère et ma sœur) avoir inculqué une très bonne éducation morale, la culture du mérite, et à toujours se dépasser pour être parmi les meilleurs. Merci à vous deux, merci pour vos transmissions et vos investissements pour voir vos enfants réussir. Vos efforts couplés aux miens m'ont permis d'arriver à ce stade

et de devenir docteur en informatique. Merci à mon petit frère Anis et à ma petite sœur Amel qui me suivaient de très près, me soutenaient et m'encourageaient tout au long de ce parcours de thèse. Ils étaient très contents et fiers de voir leur grand frère aîné devenir docteur en informatique. Je serai là moi aussi à vous suivre, à vous soutenir, et à vous encourager dans vos études et parcours, c'est promis ! Merci à Bedra et à Brahim, et merci à leurs deux filles Rima et Sarah à qui je souhaite plein de réussite. Merci à Nedra et à la petite Narimen à qui je souhaite beaucoup de bonheur et de succès à l'avenir. Merci à Amina, à Yasmine, à Khalida, à mon oncle Ahmed, et à tous les membres de ma famille qui m'ont soutenu et félicité à la fin. Je pense aussi à mes grand-parents, particulièrement à ma grand-mère maternelle décédée qui aurait été si contente et si fière de voir son petit fils devenir docteur en informatique. Merci à Farouk pour son soutien moral apaisant et ses précieux conseils d'organisation et de rigueur dans le travail. Merci à mon ami Mustapha pour sa proximité, son humour unique, et ses anecdotes passionnantes.

Enfin, malgré ma volonté de citer et de remercier toutes les personnes ayant contribué à mon succès et à ma joie, il est possible que j'en ai oublié certaines. Mes sincères excuses pour cet oubli involontaire, et merci.

Cet épisode de thèse se termine pour que d'autres épisodes puissent commencer, avec de nouveaux horizons et de nouvelles aventures. Néanmoins, cette expérience de thèse à l'Inria Rennes – Bretagne Atlantique, en France, me restera marqué à vie.

Résumé

Ce résumé donne au lecteur un aperçu des travaux effectués au cours de cette thèse. Nous commençons par introduire le contexte dans le lequel intervient cette thèse, à savoir le principe global de l'empaquetage binaire, son déploiement par les logiciels malveillants et les problèmes qu'il crée au sein de la chaîne d'analyse de logiciels malveillants d'un antivirus. Puis, nous fixons les objectifs que cette thèse se veut d'atteindre. Enfin, nous présentons les contributions qu'apporte cette thèse à la littérature.

Contexte

L'empaquetage (*packing*) a été historiquement utilisé pour la réduction de la taille des données (compression) en raison d'un manque de ressources informatiques en termes de capacité de stockage et de débit de transmission. Cependant, ces contraintes sont rapidement devenues moins pertinentes au fil du temps en raison de l'évolution considérable de ces ressources informatiques.

Lorsque le contenu d'un exécutable est compressé, sa taille est naturellement réduite, et une fonction est généralement intégrée au sein du fichier compressé pour le décompresser au moment de l'exécution, c'est-à-dire retrouver le code et les données originaux en mémoire, afin que l'exécutable puisse être exécuté comme à l'état d'origine.

Le fait de compresser un exécutable rend son code original illisible. C'est précisément cet aspect qui a rapidement motivé les développeurs de logiciels malveillants (*malware*) à tirer profit des empaqueteurs (*packers*) en tant que moyen de camouflage, devenu quasi-omniprésent, dans leur course sans fin contre les logiciels antivirus.

Ainsi, le principe de l'empaquetage a été détourné pour répondre à des fins malveillantes. Ce principe englobe de nos jours une variété de techniques qui compressent et/ou cryptent le contenu du logiciel malveillant, produisant un nouveau fichier binaire syntaxiquement différent de celui d'origine, entravant ainsi l'analyse et la détection statique en dissimulant le code malveillant.

Détecter qu'un logiciel malveillant est empaqueté et classifier par suite l'empaqueteur utilisé sont donc deux étapes fondamentales pour pouvoir désempaqueter (*unpacking*) le logiciel malveillant et l'étudier, que cela soit manuellement ou automatiquement.

L'empaquetage cause de nombreux problèmes au niveau de la chaîne d'analyse de logiciels malveillants d'un antivirus. En effet, les variantes d'un même empaqueteur rendent ineffectifs les antivirus classiques basés sur des signatures reposant fortement sur des propriétés syntaxiques pour détecter et classifier les empaqueteurs. Ces signatures syntaxiques sont souvent en incapacité de capturer de petites différences entre variantes, donc en incapacité de détecter et de classifier les différentes variantes d'un même empaqueteur. L'empaquetage crée également un problème d'efficacité, car la détection et la classification de l'empaqueteur peuvent être très coûteuses, ce qui pourrait rendre une chaîne d'analyse de logiciels malveillants peu pratique. De plus, les empaqueteurs évoluent constamment et rapidement au fil du temps, apportant de nouvelles techniques d'empaquetage qui dégradent l'effectivité des stratégies antivirus pour détecter, classifier et désempaqueter les binaires malveillants.

Face à ces problèmes, la littérature existante sur la détection et la classification d'empaquetage s'est focalisée sur l'effectivité. Cependant, la robustesse et l'efficacité requises pour faire partie d'une chaîne pratique d'analyse de logiciels malveillants restent peu étudiées.

Objectifs

Cette thèse vise à proposer des solutions de détection et de classification d'empaqueteurs effectives, efficaces et robustes, constituant des parties pratiques de la chaîne d'analyse de logiciels malveillants d'un antivirus.

Plus précisément, l'effectivité signifie que nos solutions doivent avoir un taux élevé de vrais positifs et un taux faible de faux positifs pour la détection et la classification des échantillons empaquetés. L'efficacité signifie que le coût de calcul moyen nécessaire pour détecter et/ou classifier un seul échantillon doit être réduit, et ce afin que nos solutions puissent faire face en pratique à un grand nombre d'échantillons par jour. Enfin, la robustesse signifie que nos solutions doivent maintenir leur effectivité dans le temps, face à l'émergence de nouveaux échantillons comportant de nouvelles techniques d'empaquetage.

Contributions

Conformément aux objectifs fixés, nous apportons à la littérature deux contributions.

★ Dans notre première contribution, nous présentons une étude visant à mieux comprendre l'impact de la labellisation (*ground truth*), la sélection d'algorithme d'apprentissage automatique (*machine learning*) et la sélection de caractéristique (*feature*) sur l'effectivité, l'efficacité et la robustesse des systèmes de détection et de classification d'empaqueteurs basés sur l'apprentissage automatique supervisé, s'inspirant sur des travaux de tests empiriques de l'analyse des logiciels malveillants avec apprentissage automatique (par exemple [3]). Plus précisément :

- Nous étudions les moyens de produire des labellisations de différentes qualités et tailles. Puis, nous évaluons l'impact de ces différentes labellisations sur l'effectivité et la robustesse d'algorithmes d'apprentissage automatique à détecter et à classifier des empaqueteurs dans un vaste jeu de données de logiciels malveillants réels. Les algorithmes sont testés à la fois via la méthode de validation croisée à k blocs (*k-fold cross-validation*) sur le même jeu de données ayant servi à l'entraînement, ainsi que contre des échantillons réels de logiciels malveillants recueillis après la phase d'entraînement. Nos résultats montrent que la taille d'une labellisation est plus pertinente que sa qualité, et ce quand il s'agit d'assurer plus d'effectivité et de robustesse à des algorithmes d'apprentissage automatique supervisé destinés à détecter et à classifier des échantillons empaquetés provenant d'environnements réels. En particulier, nos tests de robustesse montrent que la méthode de validation croisée à k blocs n'est pas adaptée à des domaines tels que la détection et la classification de logiciels malveillants et d'empaqueteurs, où de nouveaux échantillons et de nouvelles techniques d'empaquetage émergent constamment et rapidement dans l'environnement réel. Cette constatation sur l'évolution rapide de l'écosystème des logiciels malveillants et des empaqueteurs contribue à expliquer les résultats de [3] sur les raisons pour lesquelles les algorithmes d'apprentissage automatique seraient effectives en laboratoire, puis deviendraient défectueux une fois testés contre des échantillons réels.
- De plus, pour construire des solutions suffisamment efficaces tout en préservant l'effectivité et la robustesse, nous procédons en deux étapes. Dans la première, nous extrayons un grand nombre de caractéristiques d'exécutables servant à la détection et à la classification d'empaqueteurs basé sur l'apprentissage automatique supervisé. Puis, nous effectuons une sélection minutieuse de ces caractéristiques se basant sur leurs contributions à la fois à l'effectivité des

algorithmes et au coût d'extraction à partir d'un échantillon. Dans la seconde étape, nous effectuons une optimisation à grande échelle des hyperparamètres de chaque algorithme d'apprentissage automatique, afin de réduire le temps de détection et de classification d'un échantillon ainsi que le coût nécessaire à son réentraînement. Grâce à cette sélection et à cette optimisation, nos résultats montrent qu'une diminution négligeable de l'effectivité permet de réduire le temps de détection et de classification par échantillon jusqu'à 44 fois de moins.

- Enfin, nous effectuons une analyse du coût de réentraînement, évaluant quelle combinaison d'algorithmes et de caractéristiques produirait le meilleur rapport entre la durée d'effectivité d'un algorithme et le coût nécessaire à son réentraînement. Nos résultats montrent que des algorithmes simples avec moins de caractéristiques peuvent être plus efficaces à utiliser en pratique que des algorithmes complexes avec plus de caractéristiques.

Dans cette première contribution, bien que les réentraînements que nous proposons pour nos modèles supervisés de détection et de classification d'empaqueteurs soient réguliers et efficaces, nous constatons que ces réentraînements restent assez restreints, particulièrement pour les modèles supervisés de classification de familles d'empaqueteurs. En effet, ces modèles supervisés sont en incapacité théorique d'identifier de nouvelles classes, donc ne peuvent pas identifier les nouvelles familles d'empaqueteurs qui émergent dans la période de temps survenant entre deux réentraînements. Par conséquent, l'objectif de robustesse, pour ces modèles en particulier, se restreint face à l'évolution rapide des empaqueteurs au fil du temps. C'est dans cette insuffisance particulière que notre seconde contribution se présente.

★ Dans notre seconde contribution, nous proposons, concevons et implémentons SE-PAC, un nouveau framework auto-évolutif de classification d'empaqueteurs (*Self-Evolving PAcker Classifier*) qui repose sur le regroupement (*clustering*) incrémental de façon semi-supervisée, afin de faire face à l'évolution rapide des empaqueteurs au fil du temps. Plus précisément :

- Notre technique auto-évolutive prédit les empaqueteurs entrants dans notre système en les assignant aux groupes (*clusters*) les plus similaires, et s'appuie sur ces prédictions pour mettre à jour automatiquement les groupes, les remodeler et/ou en créer de nouveaux. Par conséquent, SE-PAC apprend en continu à partir des empaqueteurs entrants. Il améliore, intègre, fait évoluer et adapte constamment son regroupement en fonction de l'évolution des

empaqueteurs dans le temps. Nos résultats montrent que SE-PAC atteint l'objectif de robustesse en identifiant correctement les familles d'empaqueteurs connues et nouvelles apparaissant au fil du temps, faisant ainsi face à l'évolution des empaqueteurs dans le temps.

- Nous montrons comment combiner différents types de caractéristiques d'empaqueteurs dans la construction d'une métrique de distance composée. Nos résultats montrent que notre distance composée surpasse les distances simples en termes d'effectivité.
- Nous dérivons une méthodologie de regroupement incrémental établissant un bon compromis entre effectivité et efficacité. Nos résultats montrent que notre méthodologie permet de réduire le temps de mise à jour par échantillon d'un facteur 44 en moyenne.
- Enfin, nous proposons une nouvelle stratégie de sélection post-regroupement qui extrait un sous-ensemble réduit d'échantillons pertinents de chaque groupe d'empaqueteurs trouvé, et ce afin d'optimiser le coût de traitement post-regroupement. Nos résultats montrent que notre stratégie de sélection post-regroupement réduit le nombre d'échantillons de 99% en moyenne.

Les expérimentations menées durant cette thèse s'appuient sur deux jeux de données. Le premier contient plus de 280 000 échantillons de logiciels malveillants réels. Le second contient plus de 18 000 échantillons d'exécutables manuellement empaquetés. Les résultats obtenus sont prometteurs en termes d'effectivité, d'efficacité et de robustesse pour la détection et la classification des empaqueteurs.

Par ailleurs, deux outils ont été développés : PE-PAC implémentant les solutions proposées dans notre première contribution et SE-PAC implémentant les solutions proposées dans notre seconde contribution.

Enfin, les deux contributions apportées dans cette thèse ont mené à la publication de deux articles de recherche. Le premier [1] décrit notre première contribution. Le second [2] décrit notre seconde contribution.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introduction to the thesis starts by presenting in Section 1.1 the context and motivations, framing it in the global context of computer security, before digging more specifically into the malware packing problematic. Section 1.2 then explains the challenges we faced and the objectives we set. Section 1.3 depicts our contributions. Section 1.4 lists our publications. Finally, Section 1.5 describes the structure of this thesis.

1.1 Context and Motivations

This section first puts our thesis in the global context of computer security, then zooms in the malware threat. Finally, it focuses on the malware packing problematic, which is in the heart of our thesis.

1.1.1 Security: More Than a Must for Digital Systems

The digital revolution, also called the "third industrial revolution", began in the latter half of the 20^{th} century, with the adoption and proliferation of digital computers and digital record-keeping, and continues to blossom to the present day [4]. The rise of home computers, invention of the Internet, invention of the World Wide Web, mainstreaming of the Internet, Web 1.0, Web 2.0, social media, and smartphones, have undoubtedly moved our world forward to a new awesome digital age, with unprecedented social and economic changes in societies. Everything becomes faster, everything becomes more sophisticated. Withal, these achievements have not finished impressing us yet, the upcoming decades would bring spectacular technological novelties that no one would have imagined would become basic things of our daily life.

This digital world can be illustrated as a worldwide circuit where data and software constantly travel with extraordinary speed. It has provided us an unimaginable comfort,

work efficiency, communication speed, and closeness. All areas of our daily life are today highly dependent to it, e.g., banks, e-business, e-learning platforms, electronic journalism, teleworking, etc.

However, this high dependence does not come without disadvantages. Indeed, despite the multiple benefits our worldwide information circuit has provided us, it has unfortunately also provided flexible ways for cybercriminals' bad intents to fly over the oceans and continents. Thus considerably increasing the risks that well-designed attacks would easily succeed to propagate at large scale and break the ice of our digital computing and communication.

As I write this thesis, the "coronavirus pandemic" [5] (also known as COVID-19 pandemic) has led the governments of many countries over the world to decide to put their populations in lockdown for many months repeatedly, and to make massive use of digital computing such as e-learning, e-business, and teleworking, in order to simply keep life going on, and particularly to hold their economy up.

Unfortunately, during this pandemic, many cybercriminals took advantage of the situation to launch a *cyber pandemic*, i.e., to multiply their bad intents in the form of cyberattacks. Indeed, the study [6] reports that as soon as the COVID-19 pandemic started, the number of cyberattacks in general thrived up to 350 in April 2020 in Switzerland, compared to the norm which goes from 200 to 250 cyberattacks. Similarly, according to [7], United Arab Emirates saw an increase of at least 250% in cyberattacks during the year 2020.

The cyberattacks reported during the COVID-19 pandemic are quite diverse [8, 9]: many of them affect user privacy, taking advantage of the poor security awareness of many users who had to work from home with their unsecured personal computers and networks; malware attacks exploiting the rushed online launch of certain insecure services used to keep organizations' operations going on; etc. Interestingly, [6] also reports that cyberattacks using previously unseen malware or methods increased by 15% compared to the norm, such that users fell into the trap in unexpected ways.

In particular, [6] reports that email and SMS phishing attacks increased drastically during the COVID-19 period. Attackers send different emails or SMS messages with false claims such as having a "cure" or encouraging donation. Furthermore, [8] reports that as soon as the "COVID-19's cure" race started, many laboratories and Biotech firms that were leading vaccine research were targeted by ransomware attacks. The goal of these ransomware was to strike the latest breakthrough in vaccine development from

these medical organizations by stealing thousands of patient records and threatening to publish their confidential contents. Many healthcare institutions have been targeted as well, with attackers exploiting the pandemic pressure and numerous vulnerabilities found in healthcare facility networks, often for business profit [9]. Moreover, on July 2, 2021, a colossal ransomware attack spread worldwide, affecting thousands of businesses, e.g., Swedish grocery store chain Coop to close all of its 800 stores because the whole paying system at their (self-service) checkouts stopped working [10].

These examples of cyberattacks show where the imagination, cunning, and pitiless of cybercriminals could go to satisfy their malevolent goals. Still, we are not done yet: at the time I write this thesis, the pandemic is still going on, and many new malicious strategies are developed to gain maximum benefits.

The concern is not specifically about COVID-19 pandemic since it is temporary and things should get back to the norm, but the concern is about observations reported during the pandemic, which obviously revealed the unavoidable correlation between the drastic increase of our dependence to digital systems and the drastic increase in emergence of cyberattacks. Unfortunately, this ascertainment is a bad new since we are projected in the future to be more and more dependent to digital systems across the world, even without pandemic diseases, which consequently would lead to face the emergence of more and more cyberattacks. Today and tomorrow, *security becomes more than a must for digital systems.*

1.1.2 The Malware Threat

Malware, standing for malicious software, represents software that is specifically designed to disrupt, damage, or gain unauthorized access to a computer system [11]. There are a wide variety of malware types, including computer viruses, worms, trojan horses, ransomware, spyware, adware, rogue software, scareware, etc. These malicious programs are generally conceived to exploit vulnerabilities of a targeted system. These vulnerabilities are often due to insecure design or user errors. The intent of these malware programs can include revealing sensible private information about political and public persons, hacking an electronic vote system, modifying the contents of medias, stealing crucial information of bank accounts, conducting a denial of service on critical business systems, hacking nuclear weapon systems, etc.

Malware are having their (r)evolution over years and are predicted to continue spreading with greater motivations. Overall, this (r)evolution is in terms of quantity and

complexity. Indeed, by the end of the year 2019, the number of detected malware exceeded the bar of 1 billion programs. Currently, this number reaches nearly 1.28 billion programs [12]. This (r)evolution concerns the complexity as well, since [13] reports that malware become more and more complex.

This phenomenal (r)evolution takes its energy and its potential mainly from the huge business profit that malware authors and some cyber-criminal organizations can gain from their targeted victims. Ransomware damage alone cost more than 5 billion USD in 2017. It was estimated to reach 20 billion USD by 2021 and to exceed 265 billion USD by 2031 [14]. This latter prediction constitutes an increment of 1325% compared to 2021 and 5300% compared to 2017. According to some studies [15], malware would be the most expensive attack type for organizations.

Another reason relies on the availability of malware development and deployment kits, available on the Internet. These kits made the malware weapon becomes no only available for "malware experts" but also for what it is called "script kiddies" [1]. Indeed, the latter are able with simple clicks to generate many instances (i.e., variants) of the same malware (family). Furthermore, many of these tools are even tunable and it is possible to choose different settings that will be applied when composing the malware. In addition, these engines ae bolstered by anti-analysis techniques such as polymorphism and metamorphism [16] which make it possible to generate complex instances of the same malware (family), with the goal to better defeat antiviruses by concealing their (syntactic) fingerprints and thus preventing static analysis. One highly effective technique used by malware kits to achieve this goal is to *pack* malware.

1.1.3 Packers

Packers were historically used for data size reduction (compression) because of limited resources like storage capacity and data transmission rate. Some of these constraints may still apply under some systems: at some point, we all have had to zip an (executable) file because its size was too large to be submitted in the (remote) system. However, these constraints quickly became less relevant over time because of the considerable evolution of storage resources and network throughput.

When an executable content is packed, its size is naturally reduced, and a function is typically integrated into the packed file to unpack the binary at runtime, i.e., to recreate

1. Non-expert individuals who use existing scripts or programs developed by other people to attack computer systems and networks.

(decompress) the original code and data back in memory, so that it can be executed as it was originally.

The packing process changes the structure of the original file and makes the original code unreadable until being unpacked. This packing artifact motivated malware authors to take advantage of packing facilities as a new camouflage strategies in their never-ending cat-and-mouse game against antiviruses.

Therefore, the principle of packing has been drifted to fulfill malicious goals, by encompassing a variety of techniques that compress and/or encrypt the content of the malware, producing a new binary that is syntactically different from the original one, hence hiding the malicious code and preventing static analysis of the malware. In addition, modern packers include a multitude of protection and anti-analysis techniques [17, 18] (e.g., anti-disassembly, anti-debug, multi-threading, etc.) that aim to hinder the detection, classification, and thus unpacking of the packed malware. For these reasons, packers became quickly a favorite weapon of malware authors.

From a security point of view, detecting, classifying, and unpacking a given packed binary is fundamental to being able to verify whether it is malicious or benign (i.e., makes use of packing for legitimate purposes, such as digital rights management or software integrity protection). Antiviruses have adapted to the reality of malware packing by developing countermeasures at least in terms of effectiveness, i.e., detecting, classifying, and unpacking the most common packing versions, but this is still largely not sufficient.

Indeed, different packer variants [2] make *ineffective* the classical signature-based antiviruses that rely heavily on syntactical properties to detect and classify packers. These syntactic signatures are often unable to capture small differences between variants, thus unable to detect and classify different variants of a same packer (family). Besides causing ineffectiveness, packing creates a problem of *efficiency* for antiviruses, because the binary sample must be either unpacked or analyzed dynamically to detect malicious behavior (see Figure 1.1). Dynamic analysis requires execution of the (packed) malware sample in a sandbox virtual environment. The computational cost of starting a sandbox being in the order of seconds or more before the malware analysis can even begin. Otherwise, analyzing a packed malware statically, by examining the malware code directly via disassembly or measuring syntactic properties, would lead to a failure in catching the malware because the malicious code is unreadable, as explained before.

2. In the context of packers, variants can represent different versions, configurations, or polymorphic instances of the same packer (family).

Finally, as it happened for malware race, packers did their (r)evolution over time as well. This last decade saw the emergence of custom packers, which are packing programs that are either developed from scratch or partially from well-known existing packers (e.g., Vanilla UPX [3]). Their usage has become so widespread that by 2015, Symantec detected their use in over 83% of all malware attacks [19]. Some research works have also followed this (r)evolution (e.g., [18]).

The simple fact that these "custom" packers are novel and not commonly known allows malware to stay below the radar of obsolete antiviruses which can neither detect them as being packed nor unpack them effectively, because the unpacking techniques are unknown. Furthermore, since they appear constantly and rapidly over time, this causes a lack of robustness for antiviruses, i.e., makes them *ineffective over time against the evolution* [4] *of packers*.

To summarize, packing causes problems of effectiveness, efficiency, and robustness for antiviruses in analyzing and detecting malware. Therefore, proposing solutions to solve these problems is an obvious necessity of any modern and practical antivirus. This is where our thesis comes in.

1.2 Challenges and Objectives

Challenges. Packing makes malware analysis significantly harder since the binary must be either unpacked or analyzed dynamically to detect malicious behavior (see Figure 1.1). A first challenge (packing detection) then is to detect whether or not a potential malware sample has been packed. If a sample has been determined to be packed, then a second challenge (packing classification) to detect which packer was used to pack the sample. If either of these challenges fails or provides incorrect results, then static analysis of malware fails; furthermore, dynamic analysis will be required (or applied erroneously). Thus, *effective* packing detection and packing classification are crucial to building *effective* malware analysis workflow.

However, both packing detection and packing classification must be *computationally much cheaper* than dynamic analysis, since in case of multiple layers of packing the detection-classification-unpacking loop may have to be executed multiple times to prepare the sample for static analysis (see Figure 1.1).

3. UPX stands for Ultimate Packer for eXecutables.
4. That is, the emergence of new classes or new variants of existing packers.

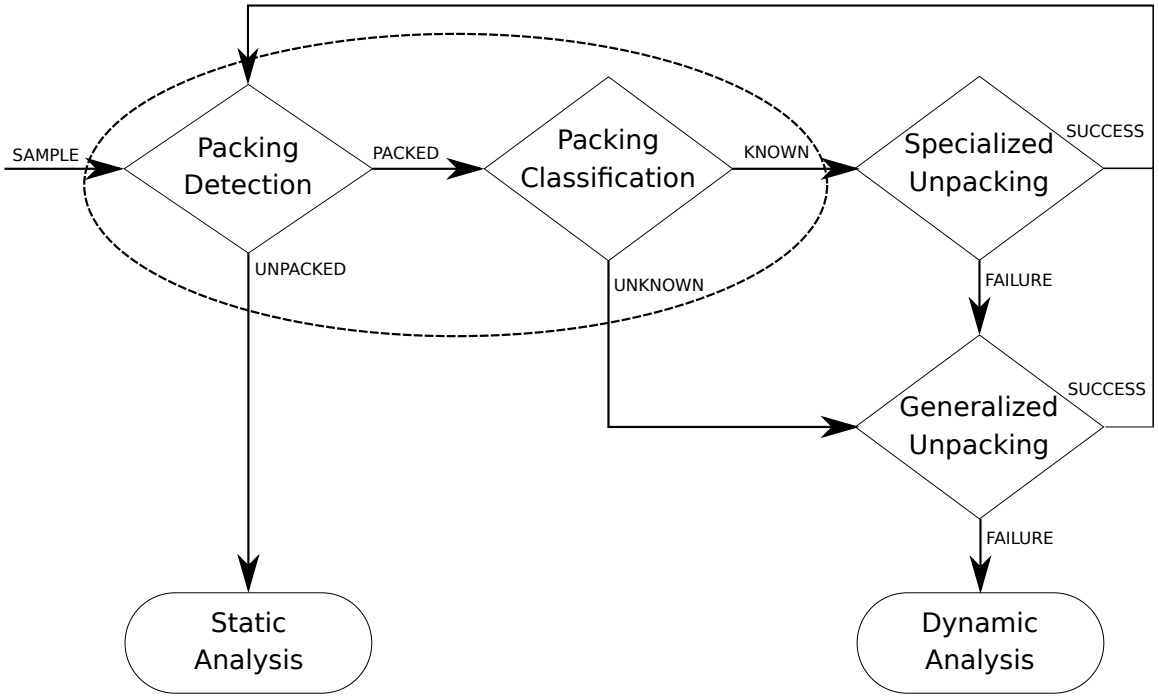


Figure 1.1 – The packing detection and packing classification fragment of a typical malware analysis workflow. The dotted circle highlights the parts explored in this thesis: packing detection and packing classification. An in-depth description of this workflow is given in the next Chapter, in Section 2.2.2.

Creating malware detection techniques that perform well on real malware (not just in laboratory tests) is known to be a hard problem [3]. One main reason for this is that the malware ecosystem evolves in the wild, changing the frequency and composition of malware, including the packing used. This forces malware analysis and detection tools to be *constantly updated* about the evolution of packers over time in order to keep the pace with the evolving malware ecosystem.

Objectives. This thesis focuses on proposing solutions for packing detection and packing classification to be practical parts of a malware analysis workflow. Our solutions must thus defeat the challenges sketched above by satisfying the following objectives:

- *Effectiveness.* The solutions have a high true positive and a low false positive rate. This is measured by testing the technique against packed and unpacked samples.
- *Efficiency.* The solutions have a low computational cost for packing detection and classification on a single sample, in order to scale to a large number of samples per day.

- *Robustness.* The solutions maintain their effectiveness when used on samples different from the ones they have been built with. This is measured by robustness assessment over time, i.e., testing the solutions on samples captured after the solutions have been built, representing the evolving ecosystem of packers used by malware in the wild.

1.3 Contributions

This thesis aims to propose solutions for effective, efficient, and robust packing detection and classification in order to be practical parts of a real malware analysis workflow (see Section 1.2 and Figure 1.1).

To this end, our thesis brings in a nutshell two contributions to the literature:

- We introduce a study which aims at understanding the impact of ground truth generation, machine learning algorithm selection, and feature selection on the effectiveness, efficiency, and robustness of supervised machine-learning-based packing detection and classification systems, following the example of works on empirical testing of machine learning malware analysis including [3].
- We propose, design, and implement SE-PAC, a new Self-Evolving PAcker Classifier framework that relies on incremental clustering in a semi-supervised fashion, in order to cope with the fast-paced evolution of packers.

In more details, *our first thesis contribution* studies ways to produce ground truth labels of different qualities (in terms of confidence) and sizes in order to train supervised Machine Learning (ML) based packing detection and classification algorithms on large-scale real packed malware database. Furthermore, it investigates how these qualities and sizes impact the effectiveness of our ML algorithms for packing detection and classification when assessed with k-fold cross-validation method, and when assessed with packed malware samples gathered from the wild after the training phase.

Our findings show that algorithms trained with higher-reliability and smaller-size ground truth seem to perform slightly better than lower-reliability and larger-size ground truth, when tested with k-fold cross-validation method. However, when tested against samples gathered from the wild after the training phase (representing the evolution of the packing ecosystem over time), higher-reliability and smaller-size ground truth perform worst with up to 30% loss in effectiveness, while lower-reliability and larger-size ground truth were outstandingly more resilient with a maximal loss of 3%.

Through these findings, we show on the one side that the first results above are just an artifact of k-fold cross-validation method. On the other side, we show how to establish a good trade-off between the quality of the packing ground truth and its size to be robust for detection and classification in in-the-wild scenarios.

Furthermore, to construct efficient solutions that are fast enough to be integrated in a practical malware analysis workflow while preserving their effectiveness and robustness, we do two things. One, we extract a large number of features for ML packing detection and classification, then perform a careful feature selection based on both the features contribution to the algorithms effectiveness and the cost of extracting features from a sample. Two, we perform a large scale hyperparameter optimization of ML algorithms, to reduce the time to classify a sample and cost of retraining an algorithm.

Through these two optimization, we show that a negligible decrease in effectiveness can reduce classification time per sample by up to 44 times.

Finally, we perform a retraining cost analysis evaluating which combination of ML algorithms and features yield the best ratio of uptime to retraining cost. Our findings show that simple algorithms with less features can be more efficient to use compared to complex algorithms with more features.

The results of robustness assessment show that packer classifiers loose their effectiveness over time due to the wide and rapid evolution of malware and thus packer ecosystems. This decrease in the effectiveness over time is in part because these packer classifiers rely on supervised ML systems which suffer from the theoretical inability to identify new classes, thus new packer families can not be classified correctly. While we offer our models regular and efficient retraining with new packer families (and versions), these retraining are still limited because supervised packer classifiers would still be unable to identify new packer families that appear constantly and rapidly in the wild, in the period of time occurring between each two retraining.

Our second thesis contribution specifically covers this theoretical limitation by proposing, designing and implementing SE-PAC, a new Self-Evolving PAcker Classifier framework which relies on incremental clustering in a *semi-supervised fashion*, aiming to provide an effective, incremental, and robust solution to cope with the fast-paced evolution of packers. Our self-evolving technique predicts incoming packers by assigning them to the most likely clusters, and relies on these predictions to automatically update clusters, reshaping them and/or creating new ones. Therefore, SE-PAC continuously learns from incoming packers, adapting its clustering to packers evolution over time.

In the context of SE-PAC, we show how to combine different types of packer features in the construction of a composite pairwise distance metric, we derive an incremental clustering methodology which establishes a good trade-off between effectiveness and update time performance, and we propose a new post-clustering selection strategy which extracts a reduced subset of relevant samples from each cluster found, in order to optimize the cost of post-clustering packer processesing.

The experimentation we conduct during this thesis relies on two datasets. The first is a real-world malware feed dataset which gathers more 280,000 samples. The second is a synthetic dataset which gathers more than 18,000 samples of manually crafted packed binaries. The results we obtain are promising in terms of effectiveness, efficiency, and robustness for packing detection and classification.

Finally, we developed two tools: PE-PAC [5] and SE-PAC[5]. The first tool implements the solutions we proposed in our first contribution. The second implements the solutions we proposed in our second contribution.

1.4 Publications

During this thesis, the two contributions explained in the previous section have been published in two papers, a journal and a conference:

- Fabrizio Biondi, Michael A Enescu, Thomas Given-Wilson, Axel Legay, **Lamine Noureddine**, and Vivek Verma. 2019. Effective, efficient, and robust packing detection and classification. Computers & Security 85 (2019), 436–451 [1].

- **Lamine Noureddine**, Annelie Heuser, Cassius Puodzius, and Olivier Zendra. 2021. SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution. In Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY 2021), April 26–28, 2021, Baltimore-Washington DC Area, MD, USA. (acceptance rate: 24.5%) [2].

5. Implementation of these tools are detailed in Sections 3.5 and 4.5 respectively. For access to these tools, please contact me on this email: laminho@live.fr.

1.5 Outline

This thesis is organized as follows. Chapter 2 gives the background material useful to understand this thesis. It both reports the related work approaches adopted to solve the packing detection and classification problems, and shows where our thesis takes its originality. Chapter 3 presents our first thesis contribution in which we introduce a study covering the impact of several ML parameters (ground truth generation, algorithm selection, etc.) on supervised ML-based packing detection and classification systems. Chapter 4 presents our second thesis contribution in which we present SE-PAC, a new Self-Evolving PAcker Classifier framework that copes with the fast-paced evolution of packers problem. Finally, Chapter 5 concludes this thesis, and shows possible future perspectives.

Chapter 2

Background and Related Work

This chapter provides the background material necessary to understand this thesis. It provides as well an analytic analysis of the related work approaches that have been adopted to solve the packing detection and classification problems, and where our thesis takes its originality.

This chapter is organized as follows. In Section 2.1, we start by giving a brief background on malware analysis and detection techniques. Then we recall in Section 2.2 background on packers and their usage in malware. Finally, we present in Section 2.3 the related work approaches adopted for detecting and classifying packers and we discuss for each approach how our thesis is different. This last section provides as well essential background for understanding these approaches and ours.

2.1 Malware Analysis and Detection

This section gives a brief background on malware analysis and detection techniques.

To analyse and/or detect malware, systems and analysts rely generally on a set of (observational) *features* that can be descriptive of the syntactic and/or semantic (called also behavioral) aspect of the malware. In practice, the values taken by (part of) these features are different from the ones taken by legitimate software, which makes possible the distinction. Furthermore, these features can be extracted principally relying on static, dynamic, or concolic analysis. We note that these analyses are sometimes combined (i.e. hybrid analysis).

Static analysis examines the malware without resorting to the execution of its code directly, i.e., via disassembly or measuring syntactic properties. As a result of a static analysis, syntactic features are (often) extracted from the malware such as strings, opcode mnemonics, byte n-grams, entropy, imported calls, header field values, etc., [20].

Dynamic analysis requires executing the malware sample, e.g., in a virtual sandbox environment. As a result of a dynamic analysis, semantic features are (often) extracted

witnessing the manifestation of malicious actions during the execution of the sample at the level of the API calls invoked, memory reads/writes/executes, the state of some registers, the files created/opened/modified/closed, the network activity, etc., [20].

The dynamic analysis makes possible the extraction of semantic features. The latter are often more resilient to semantic transformation which can easily bypass the syntactic feature which are often extracted through static analysis. However, the dynamic analysis is in general largely more expensive that static analysis due to the computational cost of starting a sandbox being in the order of seconds or more before the malware analysis can even begin.

The Concolic analysis (a portmanteau of CONCrete and symbOLIC) extracts the malware's behavior while covering as many of the binary's possible execution paths as possible [21], in contrast to the traditional dynamic analysis which is able to extract and analyze only one of the possible execution paths (which can be the one which skips revealing the malicious action) of the analyzed malware. However, since Concolic analysis integrates symbolic analysis, it is also well-know to be extremely costly and not scalabale in practice due to the path explosion problem.

Malware authors employ various techniques to defeat these analyses, like inserting many portions of dead-codes, re-ordering the functions of the malware, detecting and evading the sandbox virtual machine, executing some actions at specific timing, inserting opaque predicates, etc., [22]. In particular, packing the malware content is one of the most common obfuscation techniques used to hinder static analysis. On the one side, it increases the difficulty to reverse engineer the binaries, which entails higher analysis costs for malware analysts. On the other side, it makes possible escaping the detection becode the malicious code is hidden. Therefore, the malware's code needs to be unpacked before being analyzed and detected.

2.2 Packers

The first part of this section (2.2.1) explains the principle of binary packing, how it is employed to obfuscate malware from antiviruses, and what is the degree of its complexity. The second part (2.2.2) shows how a typical malware analysis workflow deals with the packing problem.

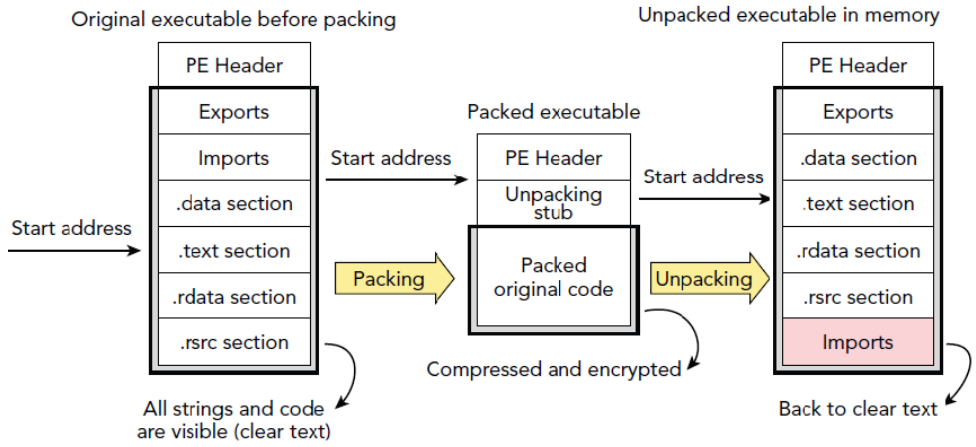


Figure 2.1 – Life cycle of a program being packed and loaded into memory [23]. The packer takes all the content of the Windows Portable Executable (PE) file [1] in the highlighted rectangle (not the header), then compresses and encrypts this into a new payload. An unpacking stub is added and placed at the updated start address of the PE file's execution. The PE header is updated accordingly. The unpacking is performed when the program is executed, by the unpacking stub decrypting and decompressing the original content and recreating the parts of the file shown in the black rectangle. In particular, the import table shown in the pink rectangle is restored so that the addresses of the program's functions are populated when running on the target machine, as every Windows system may be different. Finally, the unpacking stub returns execution to the reinstated start address and the program executes as normal.

2.2.1 Binary Packing and its Usage in Malware

Packers were designed to fulfill either or both goals of size reduction and thwart analysis or detection [23, 24]. Binary Packing consists in compressing and/or encrypting an executable binary to produce a new binary that is syntactically different from the original one. It works by taking as input the target binary (called payload) and generating a new one that embeds the original in a packed, "scrambled" form, together with an unpacking routine (called unpacking stub) that can, in memory, undo the packing process and execute the original binary. Therefore, packers are also referred as runtime packers, and sometimes as self-extracting executables. An overview of the life cycle of a program being packed and loaded into memory is shown in Figure 2.1.

In the context of malware detection, the main challenge of the analysis of an unknown packed sample is to determine whether the payload is malicious or benign. This analysis is only possible after partial or complete unpacking of the sample (see

1. A detailed description of the PE file format is given in the Appendix 1.

Section 2.2.2 below). Classical signature-based antiviruses that heavily rely on syntactical properties and do not have unpacking capabilities are thus ineffective against packing. So a special concern for packers is to alter the syntactic properties of the different instances (i.e., variants) generated from the same malware (family) in order to better defeat antiviruses. Polymorphic codes take different forms at each instance generation (e.g., using encryption). Metamorphic codes rewrite themselves at each execution [16]. The protection features (or called anti-reverse-engineering features) sought by malware developers include many different techniques for anti-debugging, anti-disassembly, obfuscation, anti-VM, and other anti-analysis techniques [24] which the goal is to make harder the unpacking process.

This difficulty to unpack a packed binary is refereed as the complexity of packers in [18]. Therefore, depending on the requirements entailed by the developer, various packers would have different complexities [18]. In this latter work, a taxonomy including six increasing complexity classes (I to VI) is proposed to evaluate packers, based on the increasing difficulty to unpack the binary, gauged relatively to the multiple anti-analysis techniques that are added to the unpacking stub routine. Type I packers are the simplest and work accordantly to the packing principle aforementioned, including a single unpacking routine that restores the original program before passing the execution and running it. From Type II to Type VI, multiple anti-analysis techniques are added, such as multi-layer packing, interleaved execution of the unpacking stub and original program, shifting decode frames, etc., [18].

Furthermore, authors added a separate class for *virtualization-based packers*, which also provide a way to convert binary files without changing semantics. Despite substantial differences in its inner working principle in comparison to classical packers, binary virtualization is seen as an *advanced packing technology* [25]. Here, the target code is seen as the bytecode (also termed p-code [26]) of an interpreter (or *virtual machine*) whose the language is randomly chosen at the time of protection [26]. The interpreter attached into the new binary (to execute the bytecoded program) is polymorphic by design with respect to the randomly chosen language.

Finally, some popular tools work on the similar principle of packing such as compressors (e.g., Zip, WinZip, WinRAR) and installers that change the syntactical properties of the original files. The difference, however, is that they aim to obtain file bundling and size reduction without any special concern with protection against reverse engineering.

2.2.2 In-depth Scanning

To determine whether a sample is malicious, the antivirus needs to analyze its characteristics. This analysis has to account for packing techniques used to obfuscate malware and hinder reverse engineering. Packing makes malware analysis significantly harder since the binary must be either unpacked or analyzed dynamically to detect malicious behavior.

Therefore, if the antivirus determines that the sample is packed, a payload extraction is attempted before proceeding with the malware analysis. This process is called in-depth scanning [27] (see Figure 1.1), and it is considered as being the packing detection and packing classification fragment of a typical malware analysis workflow. The analysis pipeline starts with very lightweight syntactic operations, and continues when necessary with more robust and expensive interventions.

If the sample is detected as unpacked, it is analyzed statically. Otherwise, if the sample is found to be packed, the packing technique is classified. In case of known classification, the payload is retrieved with a specialized unpacker and delivered back to packing detection (to account for multi-layered packing). If the classification is unknown or the specialized unpacking technique fails, generalized unpacking is alternatively attempted, with the payload delivered back to packing detection in case of successful extraction. If the generalized unpacking technique fails, the whole sample is analyzed dynamically otherwise. Observe that a failure of packing detection may lead to a failure of malware detection if that latter uses static analysis (that tends to fail when the binary is packed), or attempted unpacking of already unpacked samples likely to conclude in expensive dynamic analysis.

Specialized unpacking can concern the unpacking function itself, or may concern a specialized environment for unpacking a specific packed malware. There may be specialized unpacking functions for common packers; new functions can be developed for new packers. They are generally lightweight and fairly precise, but require a correct prior identification of the packer used (and sometimes the version). Generalized unpackers are intended to handle various packing algorithms. However, they are generally more costly for lower success rates, especially if the packer is very complex and requires rather a specialized unpacking.

Therefore, the correct classification of the packer (and version sometimes) is vital for a good, cost-effective in-depth scanning, by deciding whether a specialized unpacker can be tried before any attempt with a generalized unpacker that is more expensive and more

likely to end in costly dynamic analysis. Finally, the classification process must also be efficient enough not to add too much overhead in the analysis workflow.

2.3 Packers Detection and Classification Approaches

Most of the works present in literature use approaches that are based on syntactic signatures, entropy metrics, or Machine Learning (ML). In some works these categories are combined (e.g., ML relying on entropy metrics).

This section provides essential background for understanding each approach. Then, it provides the related work of each approach and discusses how our thesis differs from it.

We start in Section 2.3.1 by introducing a brief background on the syntactic signatures approaches, the popular tools that rely on these approaches, as well as their advantages and their limitations. Then in Section 2.3.2, we give a brief background on entropy, before presenting and discussing on the one side the works that rely strongly on the entropy metric to detect and classify packers, and on the other side the works that show the unreliability of this metric against modern sophisticated packers. Then we offer a mediate discussion to show how such a metric is used in our thesis. Finally, the Section 2.3.3 provides useful background on ML to both understanding the ML-based techniques used in literature and our ML-based solutions. It provides as well the related work that rely on ML to solve the problem of packing detection and classification, and how our thesis distinguish itself.

2.3.1 Syntactic Signatures

In the context of packers, syntactic signatures approaches rely on a set of patterns that identify a (version of) specific packer. These signatures have different complexities, they can represent simple sequences of raw bytes, text strings (e.g., PE section names), and/or more complex anomaly values found in the PE-header fields or PE-content (like high value of entropy in a specific section of the file) that characterize a (version of) specific packer.

In practice, there are many tools relying on syntactic signatures, but PEiD [28], Yara [29] and DIE [30] remain nowadays the most popular and actively maintained tools within the security community. The common principle of these tools is to statically parse a binary file and match it against a signatures database to see if it is packed or not and by which packer it has been packed.

PEiD is a well-known signature-based tool working essentially on Windows [28], used to detect and identify packers for PE files. To detect and identify packers, its signatures are mainly constituted of sequences of raw bytes that are matched against a sequence of raw bytes extracted from the PE binary file at different locations, but often at the entry point. The reason is that at this specific file location the raw bytes sequences represent often the unpacking stub, since it is assumed that at runtime the packed binary starts directly executing the unpacking code in order to unpack the malicious payload. The sequence of raw bytes composing this unpacking code is a strong characteristic for identifying in distinctive way different packers. An example is shown in Figure 2.2 and used by PEiD to recognize versions from 0.5 to 0.70 and 0.72 of UPX. We note that PEiD is not open source, but its signatures are made public [31].

```
[UPX 0.50 - 0.70]
signature = 60 E8 00 00 00 00 58 83 E8 3D
ep_only = true

[UPX 0.72]
signature = 60 E8 00 00 00 00 83 CD FF 31 DB 5E
ep_only = true
```

Figure 2.2 – A syntactic signature used by PEiD to identify files packed by UPX from version 0.50 to version 0.72. The `signature` field represents the sequence of bytes characterizing the packer and version, while the `ep_only` flag determines whether the sequence is found only at the entry point of the binary or anywhere in it.

Yara is another widely-used signature-based tool to detect and identify packers [29]. It is multi-platform, running on Windows, Linux and Mac OS X. Its packer signatures are publicly open [32], and are regularly maintained and updated by malware researchers. A Yara rule has a name and is composed principally by three fields: meta that describe textually the rule; String, which contains a list of raw bytes, text strings and/or regular expressions; Condition, which defines boolean expressions over the elements present in the strings session. For the sake of clarity, an example of Yara rules is shown in Figure 2.3 to show how Yara can recognize a specific packer.

Furthermore, these signatures are based on the scripting language Perl Compatible Regular Expressions (PCRE). The latter is a library written in C and implements a regular expression engine, inspired by the capabilities of the Perl programming language [33]. On the one side, this scripting language offers a great flexibility to generate very

```
rule PEXv099meta
{
    meta:
        description = "This is PEX packer version v099met"

    strings:
        $a = { 60 E8 01 [4] 83 C4 04 E8 01 [4] 5D 81 }

    condition:
        $a at pe.entry_point
}
```

Figure 2.3 – Rule used by Yara to identify files packed by the packer PEX version v099meta. The rule above tells Yara that any file containing the raw bytes contained in the string *a*, at the entry point location of the file under analysis, must be reported as PEXv099meta, which is described by the field meta as PEX packer version v099meta.

complex and powerful rules describing different packer families. On the other side, it makes Yara engine very lightweight and fast when processing files.

Finally, **DIE** (Detect It Easy) is a cross-platform (Windows, Linux and MacOSX) signature-based tool which has totally open architecture of signatures [34]. Its signatures (or called also algorithms of detects [30]) for packers can be created or modified through a scripting language very similar to JavaScript, hence offering a great ability to express complex and very fine-grained rules. Furthermore, its scripting engine is optimized, which offers a great scalability.

Besides detecting and identifying packers, we note that these tools can detect essentially non-packed malware, compilers, and some meta-information about the binaries e.g., OS, architecture, etc., which we do not discuss here as it is out of the this thesis scope.

Overall, the syntactic signatures approach has the advantage to be very fast in execution and precise towards well-known packers. However, such static syntactic techniques suffer in practice from several limitations. Indeed, their effectiveness is restricted to rules priorly generated, they are thus unable to detect new (variants of) packers which they do not possess a signature.

In particular, malware authors tend to hide their malicious binaries using custom packers. The latter represent packers that are developed either from scratch or partially from publicly available ones (e.g., Vanilla UPX). Furthermore, tools relying on these techniques often need to be manually updated by an analyst who writes signatures as new (variants of) packers are analyzed. This signature update method (manual reverse

engineering) is extremely costly in practice, given the tremendous number of new malware appearing every day, which increases the time new (variants of) packers remain undetected by these tools.

2.3.2 Entropy

Shannon entropy has been classically used as a proxy for packed or encrypted data in packer detection and classification [35, 36]. To compute byte entropy of n bytes, the frequency of occurrence $F(v)$ of each of the 256 byte values [2] within n bytes is computed and expressed as a probability distribution where $P(v) = \frac{F(v)}{n}$ is the probability that a byte has value v. Then the entropy of n bytes is computed as:

$$-\sum_{v=0}^{255} P(v) log_2 P(v)$$

The first published academic work to considers the problem of packing detection was [35]. Their approach was to use entropy as a proxy for packing, since packed files tend to have much higher entropy value due to the random distribution of bytes that compose the executable. Their approach was calculated to be able to achieve a false positive rate of 0.038 and false negative of 0.005, however this was not verified experimentally.

Multiple successive works focused on entropy metrics to detect a sample as packed and/or classify which packer was used for a sample [37–43].

For instance, [41] extracts fragments of fixed size from files and calculates the entropy scores of the fragments. These entropy scores are then used for computing a similarity distance matrix for fragments in a file-pair to classify similar (packed) files. Very good results are reported, but the experiment considered only three different packers and two XOR-based encoders for preparing the samples.

In [42, 43], authors elaborate a method to classify packing algorithms of a given executable by converting the entropy values of the executable file loaded into memory into symbolic representations, for which they used SAX (Symbolic Aggregate Approximation)[44]. The classification is done into packer families in [42], then in [43] into three categories: single-layer packing, repacking, or multi-layer packing. The experiments in [42] involve a collection of 324 packed benign programs and 326 packed malware programs with 19 unique packing algorithms. They report as results an accuracy of

2. Each byte is composed of 8 bits, thus it can take 256 values.

95.35%, a recall of 95.83%, and a precision of 94.13%. In [43], experiments involve 2196 programs and 19 unique packing algorithms, with results of accuracy of 97.5%, a precision of 97.7%, and a recall of 96.8%.

Furthermore, the tools PEiD, Yara and DIE (see Section 2.3.1) indicate that an executable is packed when its entropy is greater than a certain threshold (usually 7).

Broadly, approaches that rely only (or highly) on entropy to detect and classify packers are strongly based on these two assumptions:

1. If a binary file has high entropy score thus it is most likely packed.
2. If a binary file has low entropy thus it is most likely non-packed.

While the first assumption remains often true in practice, the second assumption has been severely criticized in the work [45]. Indeed, in the latter work, authors demonstrated that this measure can be altered (i.e., the entropy score value decreases) by different techniques that modify randomness. They particularly described two possible attacks: *Random byte insertion* and *Reduced source alphabet*. The first one consists in inserting strategically in the packed executable file some randomly-selected bytes from a reduced set of bytes, such that a considerably high number of bytes in the file pertain to that group. The second one consists in using only a subset of symbols in the source alphabet, such that the number of symbols necessary to represent the same information (i.e., the packed data) is higher.

In this thesis, we do not rely only on entropy for detecting and classifying packers, instead we rely on entropy as one of the elements in a feature vector, leaving to the ML algorithm to decide how relevant entropy is to detect and classify each specific packer family.

2.3.3 Machine Learning

This section starts by providing a background that is essential for both understanding the related work that rely on ML, and our ML-based techniques for detecting and classifying packers. We note on the one hand that we give just a brief overview of the working principle of supervised ML algorithms which we used in the first contribution of this thesis, considering that digging more deeply on these algorithms would be out of this thesis scope. On the other hand, in our second thesis contribution we design and implement our own incremental DBSCAN, thus a more extended description of the working principle of DBSCAN algorithm is needed.

Then, this section provides the related work that rely on ML for detecting and classifying packers, and discusses how our thesis is different from them.

2.3.3.1 Background on Machine Leaning

Machine Learning is a branch of Artificial Intelligence (AI) and computer science, seeking to get computers learn and act like humans do, i.e., learn by experience. It is based on algorithms that can learn and improve their learning autonomously from data without relying on rules-based programming [46] [3].

ML algorithms infer models based on a set of data samples (called also observations) described by a set of "features", forming as whole a "training set", in order to make decisions (e.g., predictions) without being explicitly programmed to do so. ML algorithms have been used during these last decades in a wide variety of applications, such as in biology, speech recognition, computer vision, and computer security.

In the context of computer security, ML algorithms continuously learn by automatically analyzing data and building models based on correlations that may be hidden at the scale of human eye, making possible the recognition of patterns and the prediction of threats in massive datasets, all at machine speed [47]. Thus offering a better security for users.

ML is generally based on four basic approaches: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning. The latter will not be developed because it is out of this thesis scope.

Supervised Learning supplies algorithms that learn a general rule that maps a set of samples (i.e., training set) provided with their labels (called "ground truth" [47]). Supplied algorithms are thus able to predict the label of a new observation wrt. what they learned during the training phase.

These supervised learning algorithms are deployed essentially for classification, but include also regression or active learning [48]. Classification algorithms (called also "classifiers") are used when the predicted outputs are restricted to a limited set of values, often just one representing the label of a given sample (e.g., packed or non-packed).

Many supervised algorithms are there in practice. In the context of this thesis, we explain briefly the working principle of the algorithms that have been used in our first thesis contribution, namely: algorithms based on decision trees and Naive Bayes.

3. This definition of ML is an aggregation of the references mentioned in [46].

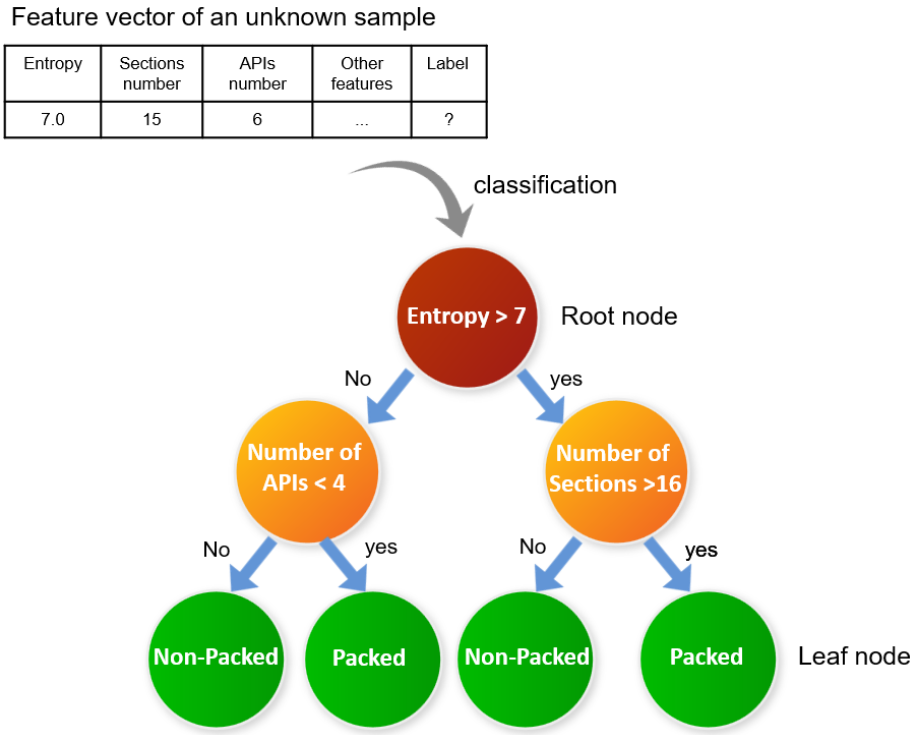


Figure 2.4 – Example illustrating the working principle of decision-tree-based classifiers.

(*i*) ***Decision Trees*** classifiers rely on a tree-like-structure as predictive model [49] In this tree-like-structure, each node represents a "test", which is called also "splitter", on the value of a specific feature (e.g., whether the entropy of packed binary is greater than 7.0 or not). Branches represent the outcomes of the tests being proceeded at the level of each node. Finally, each leaf node represents a class label (e.g., packed or non-packed), thus the final decision taken after testing all features values.

The whole path taken from the root node to the leaf node in order to classify an unknown sample, represents classification rules which are inferred statistically from the training set. Broadly, these classification rules take the form of:

if "condition 1" and "condition 2" and "condition 3" then label

We note that conditions are sequentially checked wrt. their importance in classifying a given sample. The condition 1 which relates to feature 1 (the root node, see Figure 2.4) is sequentially checked before the condition 2 which relates to feature 2 (the first branch of the root node), etc. This sequence marks the feature importance, and the sequence of features to be checked is decided on the basis of criteria like Gini Impurity Index or Information Gain metrics [49].

Multiple classifiers are based on the concept of decision tree: Simple Decision Tree [50], Extra-trees and Random Forest [51]. Obviously, Extra-trees and Random Forest are more complex than the Simple Decision Tree which constructs only one basic decision tree at the training phase. Indeed, Extra-trees and Random forest try mainly to limit the overfitting problem of the Simple Decision Tree as well as errors due to bias by constructing a multitude of decision trees at training phase, each based on a random subset of features. The class output is the class selected by most trees (i.e., vote system).

(*ii*) ***Naive Bayes*** classifier is an algorithm based on Bayes' theorem (see the formula 2.1 below) with a strong naïve independence assumptions between features [52].

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.1}$$

Where A and B are events and $P(B) \neq 0$. $P(A|B)$ is a conditional probability: the likelihood of event A occurring given that B is true. $P(B|A)$ is also a conditional probability: the likelihood of event B occurring given that A is true. $P(A)$ and $P(B)$ are the probabilities of observing A and B independently of each other.

In the context of this thesis, event A would represent a class variable (e.g., packed/non-packed or the packer used) that we call y, and event B would represent a vector that we call X, composed by a set of packer features $x_1, ..., x_n$. Thus, the formulas 2.1 above becomes:

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)} \tag{2.2}$$

In simple terms, Naive Bayes classifier assumes that the presence of a particular feature x_i in a class is unrelated to the presence of any other feature x_j.

Unsupervised learning supplies algorithms that learn from data without labels, in contrast to supervised learning. Supplied algorithm scans through data looking for any meaningful structures. The latter would constitute commonalities that allow splitting data into different groups.

A central application of unsupervised learning is Clustering. It consists in dividing a set of elements in subsets (called clusters) following some criterion. The elements can be treated individually (incremental clustering method) or they can be processed in batches (batch clustering method) [53]. While the batch method attempts to capture the underlying structure of the elements in a compact and efficient way, the incremental

method suits the scenario where new data arrive continually and recomputing the clusters from scratch becomes infeasible due to the volume of data. In particular, incremental clustering allows incremental learning, where knowledge is enhanced, integrated, adapted and evolved.

Clustering can be evaluated by intrinsic or extrinsic metrics [54]. Intrinsic metrics evaluate with some distance metric whether the elements in the same cluster are close while the elements in different clusters are distant. Extrinsic metrics evaluate the quality of a clustering by comparing it against a ground truth, also called gold standard, that represents the expected clustering.

In practice, many clustering algorithms are there. They can be primarily categorized into prototype-based, hierarchical-based or density-based [55]. The latter has the particularity of not being sensitive to noise and can deal with different cluster sizes and different cluster shapes.

(*i*) ***DBSCAN*** [56] is a typical density-based clustering algorithm widely used in many applications (e.g., malware clustering [57]). DBSCAN refers to density-based spatial clustering of applications with noise. It is designed to find arbitrary-shaped clusters and noise (i.e., outliers) in some space. Based on a set of points in some space, DBSCAN groups together points that are close to each other wrt. some distance measurement (e.g., Euclidean distance) and a minimum value parameter of the number of points. It marks as noise the points that stand alone in low-density regions (i.e., whose nearest neighbors are too far away).

Basically, DBSCAN algorithm requires two key parameters: *eps* and *minPts*.

- *eps* specifies the neighborhoods distance. That is, two points are considered to be neighbors if the distance between them is lower or equal to *eps*.
- *minPts* specifies the minimum number of points required to form a dense region, thus to form a cluster.

DBSCAN differentiates also 3 points categories: *core point*, *border point*, and *noise*.

- *Core point*. A point P is considered as *core point* if there are at least *minPts* (including the point itself) number of points within its neighborhood wrt. a radius *eps*. That is, two points are considered to be neighbors if the distance between them is lower or equal to *eps*.
- *Border point*. A point P is considered as *border point* if it is reachable from a *core point* wrt. a radius *eps*, and there are less than *minPts* number of points within its neighborhood wrt. a radius *eps*.

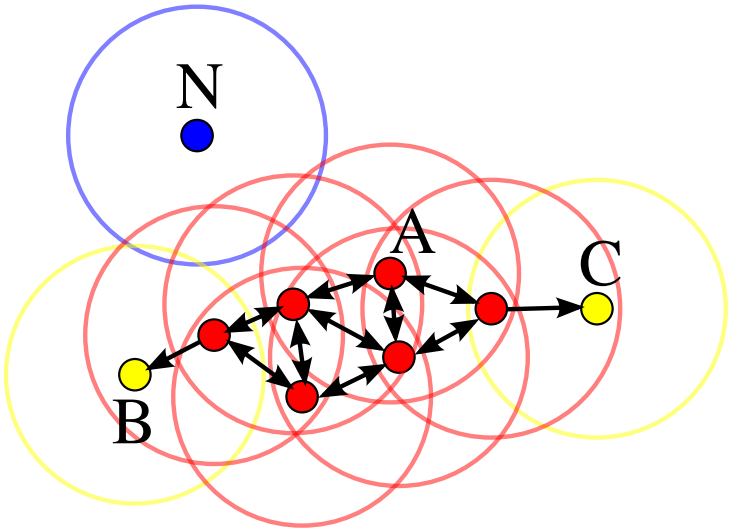


Figure 2.5 – In this diagram, minPts = 4. Point A and the other red points are core points, because the area surrounding these points in an *eps* radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable [58].

- *Noise point*. A point P is considered as a *noise point* if it is not a *core point* and not reachable from any *core point*.

For better clarity, Figure 2.5 is given to illustrate these points and parameters.

DBSCAN algorithm finds clusters starting with an arbitrary point P and retrieves all points density-reachable from P wrt. *eps* and *minPts* [56]. If P is a core point, this procedure yields a cluster. If P is a border point, no points are density-reachable from P and DBSCAN visits the next point in the given space. If not, the point P is marked as noise. During the formation of a cluster, all points within the neighborhood of the initial point (i.e., the core point) become a part of the cluster. Furthermore, if these new points are also considered as core points, points that are in their neighborhood are merged to the initial cluster being formed. The next phase consists in choosing randomly another point P' that has not been visited previously, then the same procedure described above applies. We note a point that has been marked as noise could be visited again during the procedure in order to check whether it can be merged to a given cluster. The whole procedure is completed when all points in the given space are visited.

Semi-Supervised Learning is somehow a mix between supervised learning where all training samples are labeled, and unsupervised learning where all training samples are unlabeled. In this approach, we combine a small amount of labeled data with a large amount of unlabeled data during training (often because the cost associated with the labeling process of large datasets is very high), thus leaving the ML algorithm the decision of how to explore data on its own, wrt. some assumptions (continuity assumption, cluster assumption, etc., [59]). For instance, samples which are close to each other are more likely to share a label, thus the ML algorithm would affect them the same label accordingly.

2.3.3.2 Related Work based on Machine Learning

Advanced approaches use ML techniques to detect and/or classify the packer. In this section, we present the related work that rely on ML for detecting and classifying packers and show how our thesis is different from them. The works are presented wrt. how close they relate to our two thesis contributions to facilitate showing the differences.

1) First Thesis Contribution.

Supervised learning is the most prevalent technique in literature [39, 60–67]. In this technique, ML algorithms build packer detectors and packer classifiers models by learning from a set of syntactic or behavioral features extracted from packed binaries and from corresponding labels provided as ground truth. These packer detectors and classifiers models are then used to detect and classify unlabeled binaries.

The challenge of packer detection was considered by [60] that tested a variety of machine learning approaches on a variety of features. Their best results were using Multi Layer Perceptron algorithm that achieved 10-fold cross-validation accuracy of 0.9995. Besides effectiveness in detection, the work claims a fast detection rate by relying on only 9 statically-extracted features relating to sections, entropies and import functions of the PE file. However, results on fast detection or computational cost of extracting features were not reported, thus not validated experimentally.

Machine learning on multiple features was also considered in [64]. To highly increase the accuracy of their packer detectors, they take from a large set of features (69 in total) the ones for which Information Gain exceeds a certain threshold. Their best detection results achieved 10-fold cross-validation accuracy of 0.9996 using Fuzzy Unordered Rule Induction Algorithm (FURIA) with Information Gain on a sample set of 63,000 samples of which 21,000 were packed.

In [39] the problem of packer detection was generalized to packer classification. Again

the approach was to test multiple algorithms, but used different features from those used in the works above. Indeed, their features measure essentially the amount of "randomness" in different parts of an executable program, following [37] that considers that the randomness distribution of each packer family exhibits a distinctive pattern. At the difference with [37], the features extraction is performed by employing a refined version of the sliding window randomness test with trunk pruning method. The experiments were conducted on a set 17,919 packed samples, including clean packed files and more than 17,000 malware samples gathered from the wild. Among the classifiers tested, K-Nearest Neighbors (KNN) algorithm performed the best classification results by achieving 10-fold cross-validation accuracy of 99.6% true positive rate, and only 0.1% false positive rate. Moreover, KNN took the least time to build a model on training data with only 0.02 seconds.

In [63] the packed binary is converted to Byte and Markov plots. The Byte plot represents a grayscale image where a byte value of 0 is black and byte value of 255 is white. The Markov plot is based on Markov model and uses byte level transitions to create a signature. These transitions are supposed to capture the encoding schemes used by a packer, thus to identify it distinctively from non-packed binaries (i.e., detection) and from other packers (i.e., classification). From these plots, a total of 534 features relating to image processing were extracted. Experiments are performed on real malware dataset from which 9 well known packers are selected with 5,000 random samples for each in their training and test sets. The performance of the system is assessed by comparing the two models: Byte and Markov plots using a binary-class SVM for the packing detection stage and multiclass SVM and Random forest for the packer classification stage. The result obtained wrt. Area Under Curve (AUC) metric is an accuracy of 95% on average with Markov plot for packer detection and an average of 80 % for packer classification with Random forest and Markov plot.

The major drawback of this work is the significant cost of converting each time an incoming binary into a plot before extracting features. Furthermore, a selection is not performed on the 534 extracted features which constitutes an additional significant cost in the step of detection and classification of samples.

In [66] authors introduce a framework called "BE-PUM" relying on concolic testing to generate a metadata signature as feature, to identify each packer family (and version) uniquely. This metadata signature represents the frequency vector of the numbers of occurrences of a set of well-known obfuscation techniques (e.g., anti-tracing, anti-tampering, etc.) found in the unpacking stub code. The set of obfuscation techniques

is carefully defined by observing more than 40 real world malware. The experiments were performed on 12,814 real malware with 12 packers. The training set is populated by samples of each of the 12 packers. The testing test is populated by samples that represent variants of the 12 packers included in the training set. The statistical Chi-square measure is applied as similarity metric on the obfuscation frequency vectors of test packers to classify the likelihood of their metadata signature wrt. the training set. Results showed that the proposed approach outperforms the classical byte signatures used by PEiD.

Like [63] the major drawback of this work is the cost to generate the metadata signature due naturally to using concolic testing (authors report that experiments costed 10 weeks, for only 12,814 samples).

The best packer detection results achieved are 10-fold cross-validation accuracy of 0.9996 using FURIA with Information Gain on 63,000 samples of which 21,000 were packed [64]. The best packer classification results achieved, based on 10-fold cross-validation, are 99.6% true positive rate and only 0.1% false positive rate [39].

Our first thesis contribution shows that our solutions are able to instead achieve an F-score up to 0.9999 on more than 280,000 samples, *but more importantly*, studies how these results depend on the construction of the ground truth, choice of metrics, and type of validation, and how these results hold in scenarios more realistic than k-fold cross-validation.

Furthermore, the results cited above focus only on increasing the effectiveness of detection and classification, but do not consider the efficiency of their systems when running to scale to millions of samples per day. Indeed, computational cost has largely not been considered as an aspect of packing detection and classification. In [68] they consider cost of feature extraction for use in malware classification, with packing detection playing a minor rôle and the independent cost for packing detection not being considered. In [39] the timing of building the model for classification was considered, but not the time to classify a sample. Our first contribution shows that feature extraction may dominate the classification time if features are not also selected taking into account their extraction time. Hence feature selection impacts efficiency in ways that were not considered by [39].

Finally, as far as we know, the efficiency in regularly retraining ML algorithms for packing detection and classification has not been considered in the literature.

2) Second Thesis Contribution.

The fact that most of works above rely on supervised learning makes them unable to identify new, totally unseen, packer families. Indeed, while supervised ML-based packing

detectors are able to detect new unknown packers because the problem is straightforwardly binary-class (packed, non-packed), supervised ML-based models classifying packer families bring a strong theoretical limitation: they are unable to classify new unseen classes, thus unable to cope robustly with the rapid evolution of packers over time. Our first thesis contribution offers our models regular and efficient retraining with new packer families (and variants). However, these retraining are still limited because supervised packer classifiers would still be unable to identify new packer families spreading constantly and rapidly in the wild, in the period of time occurring between each two retraining. This is where our second thesis contribution intervenes by proposing SE-PAC, a new Self-Evolving PAcker Classifier framework which relies on incremental clustering in a *semi-supervised fashion*, aiming to provide an effective, incremental, and robust solution to cope with the fast-paced evolution of packers.

Very few works [69, 70] in literature proposed solutions to (automatically) identify completely new unseen packer classes, and more generally to cope robustly with the quick evolution of packers over time.

In [69] control-flow graphs based on recursive traversal disassembly are extracted from the entry point function of the packed files. Then a set of normalization and sorting techniques are then applied on these graphs in order to be resilient against a set of well-known graph obfuscation techniques, before extracting a final signature of the sample. Finally, this signature can be compared to a set of signatures in a database of known packer signatures, to determine which packer is used. Their method was assessed on a dataset of only 7 well-known packers with 20 for each. Theoretically, for a new packer family, a new signature is generated and used to update the database with no human intervention. However, this was not validated in their experiments. Only variants of the same packer family are validated by tests.

Quite similarly to the work above, in [70] control-flow graphs are extracted from the entry point function of the packed files. Then graph matching techniques are applied to identify the critical parts of the graphs for enhancing the accuracy of similarities and optimizing the pairwise computations. Thus when an unlabeled graph comes, a graph similarity metric is used to measure the similarity between the critical parts of that graph and the critical parts of set of graphs present in the database. Their method is evaluated on three datasets: manually-packed benign applications, wild-packed malware and non-packed malware. The three datasets contain in total 39,692 non-packed malware and 15,998 packed samples with a total of 15 unique packers with different versions.

Evaluation shows satisfying results for identifying test packers that represent different versions of packer families present in the database. However, like [69], catching totally new packers is not validated in the experiments.

Our second contribution (SE-PAC) holds many differences with the works mentioned above. Firstly, although the continual update of the packing classification system is undertaken in the methods proposed in the works mentioned above, their main focus was essentially on bringing more resilient signatures as alternative of the classical byte-signatures matching which are unable to capture small differences (that could be maliciously introduced) beyond the matching rules for identifying variants of the same packer families Differently, our second thesis contribution (SE-PAC) focuses more on providing incremental clustering mechanisms for continually updating the packing classification system in order to cope with the fast-paced evolution of packers over time. Indeed, our framework "SE-PAC" is incremental and operates in real-time fashion, so we study more closely how updates take place and adopted a heuristic establishing a trade-off between the effectiveness and efficiency of the system.

Secondly, we derive in the context of packed binaries a composite pairwise distance metric which is able to combine any kind of features (e.g., numeric, strings, graphs, etc.). Thus our distance metric is fully extensible for any kind of other (future) features, in contrast to the distance metrics proposed in the works above.

Thirdly, regarding the evaluation, our test scenarios are more significant because they include two datasets containing in total more than $30K$ packed samples with a total of 32 unique packers (including custom ones), in contrast to [69, 70] which tests entail only 7 and 15 unique well-known packers respectively. In addition, our test scenarios do not include only different versions of packers but also totally new packer families. Moreover, following the evolution of the systems proposed above over time was not really considered. In [69] an updater module is added to integrate the feedback of classification, but the evolution of packer graphs is weakly followed in the experiments. Differently, we follow the evolution of clusters posture as long as packed samples come over time by a set of extrinsic and intrinsic measures, as well as examining more closely how to maximize the lifespan of our system for a better robustness.

Finally, the works mentioned above do not include a method to optimize the analysis of the packed samples found, while we propose a post-clustering selection strategy that provides for each cluster a small subset of relevant samples that can undergo deeper and more costly analyses, whose results can be extended to all samples in the cluster.

Chapter 3

A Study of Supervised Machine-Learning-based Packing Detection and Classification Systems

---

This chapter presents the first contribution of this thesis. We want to construct ML-based packing detection and classification models that are effective, efficient and robust in order to be integrated in a practical malware analysis workflow. To this end, we introduce in this first contribution a study which aims at understanding the impact of ground truth generation, ML algorithm selection, and feature selection on the effectiveness, efficiency, and robustness of supervised ML-based packing detection and classification systems, following the example of works on empirical testing of ML malware analysis including [3]. More precisely, we:

- Study ways to produce ground truth of different quality and size for training ML detection and classification algorithms. We apply 3 different methods of packing detection and classification on a database of 281,344 samples, then produce two different ground truths: a 3cons ground truth with higher-quality labeling but fewer samples following [39], and a 1cons ground truth with more samples but lower-quality labeling. We ask a research question on the relative effectiveness of algorithms trained on the two ground truths. While algorithms trained on 3cons seem to perform 1-2% better than algorithms trained on 1cons, we show that this is an artifact of k-fold cross-validation method.
- Extract a large number of features for ML classification and perform a careful feature selection based on both the features contribution to the algorithms effectiveness and the cost of extracting features from a sample. We find that some

features require $10^{-1} seconds$ to extract while others require $10^{-5} seconds$ or even $10^{-7} seconds$ to extract, hence feature selection strongly impacts classification and retraining costs.

- Perform a large scale hyperparameter optimization of ML algorithms, each depending on the choice of: ground truth, features, and scenario. We ask a research question on how effective and efficient our system is when optimizing for both F-score and classification cost instead of only for F-score. We find that decreasing the effectiveness by 1-2% can reduce the classification cost per sample by 17 to 44 times.
- Perform a robustness assessment against the evolution of packers over time by testing the best algorithms against malware samples gathered from the wild after the training phase. We ask a research question on how effectiveness decreases as malware and packing ecosystems evolve over time. We find that algorithms trained on 3CONS lose 6–30% of their effectiveness against 1–3% of the algorithms trained on 1CONS.
- Perform on the basis of the experiments above a retraining cost analysis evaluating which combination of algorithms and features has the best ratio of uptime to retraining cost. We ask a research question on whether simple algorithms with less features or complex algorithms with more features are more efficient to use. We find that the latter cost up to 60 times more to train for an uptime only 3–4 times higher than the former.

This chapter is organized as follows. Section 3.1 overviews the research methodology and questions asked. Section 3.2 describes the features used for packing detection and classification, the categories they were divided into for selection, and comments on their extraction costs. Section 3.3 details the datasets and methods used to generate the ground truth. Section 3.4 sets the evaluation metrics. Section 3.5 presents the experiments performed to answer the research questions, as well as the results. Section 3.6 discusses the experimental results. Finally, Section 3.7 concludes.

3.1 Methodology

This section overviews the Research Questions (**RQ**) considered in this study and the experimental methodology used to address them.

3.1.1 Supervised Machine Learning Detection and Classification Algorithms

Our approach to perform packing detection and classification is built by evaluating and choosing between multiple supervised ML algorithms [60, 64, 71].

We recall that in supervised ML, classification algorithms are trained on a ground truth, i.e., a set of fully-labeled data (see Section 2.3.3.1). In our case, each element in the ground truth is composed of a label describing the class of the element (packed or unpacked for detection and packer name for classification) and a vector of features describing the relevant characteristics of the element. A classification algorithm is trained on the ground truth to understand how the labels are connected to the corresponding features. Then this algorithm is tested on a new feature vector and having the algorithm guess the label of the element corresponding to the new feature vector. In our case, each element is a binary sample and its features describe properties that can be efficiently extracted from the binary. The features we chose are described in Section 3.2 together with their average extraction times.

The ML detection and classification algorithms considered here are two simple algorithms: Gaussian Naive Bayes and Decision Tree as well as two complex algorithms: Random Forest, and Extra Trees (see Section 2.3.3.1). The reason why we choose these algorithms is to show how the simplicity and complexity of ML algorithms could impact the effectiveness, efficiency, and robustness of packing detection and classification. The effectiveness of each algorithm is measured by the F-score metric, while its efficiency is measured as the inverse of the time it costs to predict the label of a sample. Section 3.4 provides more details concerning the metrics set for evaluation.

The generation of the ground truth is a problem in itself: large labeled databases of packed malware are largely unavailable and the tools used to label them often rely on different methodologies, so the labels generated from different tools for the same sample may be different. In [39], labeling a malware database is performed by using three sources of packer labels: Kaspersky, Microsoft, and the Computer Associates (CA) Threat Management Team. In addition, PEiD tool (see Section 2.3.1) is used to settle disagreements between sources: only samples for which all the tools agree on the label are kept. Moreover, only packer families for which there is at least 100 labeled samples are kept in the generated ground truth.

We follow a similar principle of annotating malware with three different sources. However, we construct two different ground truths based on whether the sources agree or not, and study how these two different ground truths impact the effectiveness and robustness of the detection and classification algorithms. Details on our ground truth generation are given in Section 3.3.

3.1.2 Feature Selection and Hyperparameter Optimization

Using more features does not necessarily increase the effectiveness of the algorithms, as some features can be misleading or insufficiently related to the class labels. Hence it is common in ML to experimentally determine which features to keep and which to discard, in a process known as feature selection. This process is important here since we want to optimize for speed against features that require a higher time cost for extraction, classification, and ML algorithm retraining. In addition, ML algorithms have hyperparameters that can modify their effectiveness and efficiency, so we need to find the optimal hyperparameters for each feature combination to guarantee a fair comparison. So the research question that arises is:

RQ1.1 How efficient and effective are packing detection and packing classification when optimizing for both F-score and classification cost instead of only for F-score?

For each of the scenarios of interest (packing detection, packing classification, or both at the same time) and each of the two ground truths, we test each algorithm on each combination of feature categories and hyperparameters. Then we keep the top 3 algorithms by F-score and the top 3 algorithms by ratio between F-score and total classification cost per sample (i.e., feature extraction cost plus algorithm classification cost). This provides insight as to how much F-score is lost when selecting algorithms and features that are optimized for both F-score and cost instead of only for F-score.

3.1.3 Robustness Assessment against the Evolution of Packers over Time

As the malware ecosystem evolves over time, new (variants of) packing techniques appear constantly and rapidly, impairing the effectiveness over time of the packing detection and classification algorithms. Therefore, regarding the robustness of these

algorithms against the evolution of packers over time, we derive the following research questions:

RQ1.2 How do algorithms trained on the 3CONS ground truth perform compared to algorithms trained on the 1CONS ground truth?

RQ1.3 How robust are the best algorithms found by feature selection and hyperparameter optimization against packed binaries collected from the wild after the training phase?

We test the robustness of our algorithms by training them with two different ground truth qualities and sizes for data collected from February to June 2017, and testing them on data collected from July to October 2017, to understand how their effectiveness degrades with time. This provides insights on how the ground truth quality and size impact the robustness of algorithms, how robust are the best algorithms, thus how often the best algorithms have to be retrained to keep the pace against the evolution of packers.

3.1.4 Retraining Cost Analysis

Packing detection and classification algorithms have to be retrained when their effectiveness drop to a certain point over time. Saving this retraining is important to ensure practical systems as parts of a malware analysis workflow. To save this retraining, we consider the uptime and time cost required to retrain the given ML algorithm, then we determine how many seconds of uptime are obtained for each second spent retraining the algorithm. Based on this ratio and upon the robustness assessment results, we evaluate the situation of a malware analyst choosing which algorithms and features to use, thus we derive the following research question:

RQ1.4 Which combination of algorithms and features yield the best ratio of uptime to retraining cost?

For illustration, we fix required F-score of at least 0.96 as required, implying that when the F-score drops below this point the ML algorithm must be retrained.

3.2 Feature Description and Selection

This section overviews the features used in the experiments, including citing other works that have used them where relevant, and explains how subsets of these features are selected for packing detection and classification.

A wide variety of features have been used in prior work on packer detection and classification (see Section 2.3.3.2), most of which detect possible artifacts of the packing process such as increased entropy and removal of metadata.

Since packing detection and classification typically play a major role in a larger malware analysis workflow, the focus in this study is on static features that can be extracted without dynamic execution, as the cost of extraction is critical to propose *efficient* solutions.

Static feature extraction tends to produce syntactic properties (see Section 2.1), which are severely limited for malware detection and classification. Dynamic feature extraction tends to produce behavioral properties which are much more expensive to extract (starting a sandbox for each sample, etc.), but have been shown to be more appropriate for malware detection and classification [72]. However, the limitations of syntactic properties for malware detection and classification stem precisely from the existence of packing techniques: the need for behavioral properties comes from the fact that syntactic properties would detect and classify the packer instead of malware. Since detecting and classifying the packer is exactly our goal, syntactic properties are fitting pretty well for such goals, as shown by the literature on packing detection and classification. In addition, our approach relies on combining these syntactic properties with ML, thus offering a better resilience compared to approaches based on merely syntactic signatures (see Section 2.3.1).

Nonetheless, despite the significant cost of extracting behavioral features, the latter have been used in some works [18, 73] for packer detection and classification. They are often used in approaches that dynamically analyze an unknown binary (malicious or benign), attempting to detect and classify packing behaviors in order to facilitate unpacking and retrieve the original entry point. We could include in the future *very lightweight* behavioral properties, which could give a better accuracy in detecting and classifying very sophisticated packers, capable of bypassing purely syntactic properties. Including these behavioral properties would not impact our feature selection, because the approach we explain below selects features regardless of their nature (syntactic, behavioral or both).

Due to the lack of extensive feature extraction cost analysis in prior works, our approach is to extract a large number of features (119) and use feature selection to find the ones with high contribution to detection and classification while having low extraction cost.

There are multiple approaches in literature [74] to select features in supervised ML. We broadly divide these approaches in two categories: *standalone-based* and *combination-based*. The first category of approaches evaluates the contribution of each feature standing alone to discriminate different ML classes, based on statistical criteria such as entropy, variance, etc. The main problem of such approaches is that they do not consider the relations that could occur when combining with other features. The second category of approaches evaluates the contribution of possible combinations of features to discriminate different ML classes. This second category is more robust than the first one, but it obviously suffers from the *combinatorial explosion*.

The number of all non-repetitive combinations of a given set S composed of n elements can be calculated by the following formula:

$$SumC_n = \sum_{k=1}^{k=n} \binom{n}{k} = \sum_{k=1}^{k=n} \frac{n!}{k!(n-k)!} \tag{3.1}$$

where n is the number of elements of the set S, k is the number of elements (or length equivalently) that can form non-repetitive combinations from n elements. We note that a combination of k-elements is considered non-repetitive only if it has at least one different element from other combinations, regardless of the order. Thus the sum of all possible non-repetitive combinations of different lengths k of elements n is represented as $SumC_n$. For 119 features, $SumC_{119}$ produces around $665{*}10^{33}$, which is computationally infeasible.

Therefore, we propose a trade-off between catching the links between features and producing a number of non-repetitive combinations that is computationally feasible. More precisely, we propose to divide the features into thematic-based categories, then produce all possible non-repetitive combinations of these categories. Our 119 features are here divided into 6 feature categories, thus the number of non-repetitive combinations of feature categories produces $SumC_6 = 63$.

We describe just below the 119 features and the thematic-based categories they belong to. Note that when a feature value cannot be extracted (e.g., entropy of a missing section), a constant outside the value range is used instead (e.g., -1 for entropy). In addition, Figure 3.1 summarizes the extraction costs for each feature category.

Byte Entropy – BE. The Byte Entropy (BE) feature category includes features that measure the Shannon entropy (see Section 2.3.2) of different parts of the binary file. There are 6 features in this category corresponding to the Shannon entropy of the:

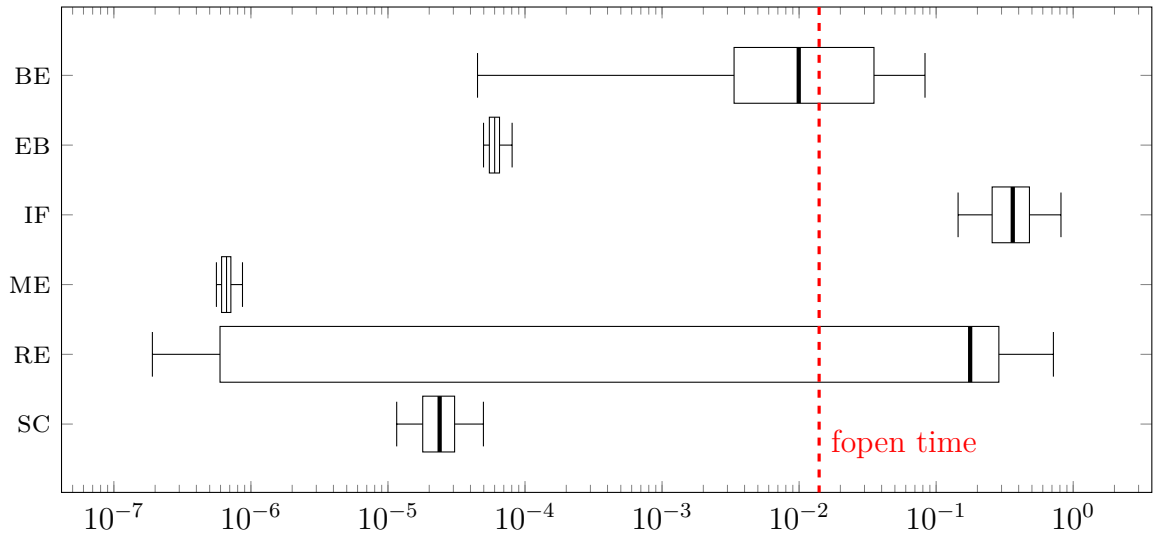


Figure 3.1 – Feature extraction costs for the feature categories. The dotted red *fopen time* line represents the empirically-calculated average time for opening the binary before extracting any feature, i.e., 14ms. Extraction cost of the EB, ME, and SC categories is negligible. Extraction cost of the BE and RE categories varies widely according to the size of the file, which sections are present, and whether it is stripped of its debug information and resources. Extraction cost of the IF category is consistently high.

- *.text* section.
- *.data* section.
- *.rsrc* section.
- PE header.
- Section containing the entry point.
- Entire file.

The intuition behind the features of this category is that since Shannon entropy corresponds with compression and encryption, higher Shannon entropy will correspond to packing. The various byte entropy features attempt to exploit which sections are most significant. Computing Shannon entropy of the whole file and its sections requires at least a full file scan to compute the byte frequencies. The extraction costs vary widely according to file size and which sections are present, as shown in Figure 3.1, and is usually comparable to the time required to open a file.

Entry Bytes – EB. The 64 Entry Bytes (EB) features are the first 64 bytes of the binary from the entry point. Each of these 64 features is the value of the byte converted to

a natural number. The intuition is that the bytes at the entry point often characterize the unpacking stub. Syntactic signatures are commonly based on the bytes after the entry point, as shown in Figure 2.2. These bytes can be read directly from the file with a direct memory access. The average extraction cost for this feature category is consistently negligible compared to the time required to open a file, as shown in Figure 3.1.

Import Functions – IF. The Import Functions (IF) feature category includes 5 features related to the import table and imported DLLs and functions. Related works have identified significant features here [36, 64, 68]. The work [64] gives a list of 16 API functions that are more common in packers and malware than in (non-packed) cleanware. The features in this category are:

- Number of imported DLLs.
- Number of imported functions in the import table directory.
- Number of addresses found in the import address table.
- Number of imported functions that appear in the [64] list:
  - GetProcAddress, LoadLibraryA, LoadLibrary, ExitProcess, GetModuleHandleA, VirtualAlloc, VirtualFree, GetModuleFileNameA, CreateFileA, RegQueryValueExA, MessageBoxA, GetCommandLineA, VirtualProtect, GetStartupInfoA, GetStdHandle, and RegOpenKeyExA.
- Ratio between the number of imported functions listed above and the total number of imported functions.

The intuition here is that packers often remove the import table to hide the packed program functionalities, and keep only the functions necessary to reconstruct it. Extracting these features requires scanning the import table and import address table when present. The average extraction cost for this feature category is in the tenth of seconds, as shown in Figure 3.1.

MEtadata – ME. MEtadata (ME) features are those extracted from the PE header that provide information about the program. The included 21 metadata features here are:

- The 8 PE characteristics.
- PE checksum.
- Base address (converted to decimal) of the:
  - Image.
  - The *.text* section.

- OS major version.
- OS minor version.
- Byte size of the:
  - Image.
  - *.text* section.
  - Headers.
  - Initialized data.
  - Uninitialized data.
  - Stack reserve.
  - Stack commit.
- Section alignment (i.e., size of a memory page).

All of these features have been used in prior works [36, 64, 68].

The intuition here is that this metadata provides information that may characterize a specific packer, or the information a packer provides to attempt to hide itself. The information for these features is available in the PE file header at specific addresses, so the features can be extracted very efficiently with direct file access. The average extraction cost for this feature category is consistently negligible compared to the time required to open the file, as shown in Figure 3.1.

REsource – RE. The REsource (RE) feature category includes 2 features relating to the resource (*.rsrc* and *.rdata*) sections of the file, following [68, 75]. The two included features are:

- Presence of the debug directory in *.rdata*.
- Number of resources defined in *.rsrc*.

The intuition for the presence of the debug directory is that packers can remove debug information to hinder reverse-engineering. The number of resources defined is a proxy to detect anomalous programs, such as when packers store their packed information in the resources section. Extracting these features requires checking the presence of the debug directory and parsing the *.rsrc* section to count the number of defined resources. The average extraction cost for this feature category is negligible if no resources exist, but can vastly increase if parsing a large resources section, as shown in Figure 3.1.

SeCtion – SC. The SeCtion (SC) feature category gathers 21 features that relate to the sections information. The majority of features here have been used in prior works [36, 64, 68].

- Ten integer features represent the number of sections of the PE file that are:
  - Standard.
  - Non-standard.
  - Executable.
  - Writable.
  - Executable and readable.
  - Executable and writable.
  - Readable and writable.
  - Executable, readable, and writable.
  - Have raw data size zero.
  - Have different virtual and raw data size.
- Seven boolean features relate to the existence of the following packing artifacts:
  - .text section not executable.
  - A non-.text section is executable.
  - *.text* section is not present.
  - Entry point not in *.text* or *.tls* section.
  - Entry point not in one of the standard sections.
  - Entry point not in an executable section.
  - Address not matching file alignment.
- The remaining 4 features are:
  - Ratio of standard sections to all sections.
  - Ratio between the raw data and virtual size of the section with the entry point.
  - Maximum ratio of raw data to virtual size.
  - Minimum ratio of raw data to virtual size.

The intuition here is that packers can use non-standard sections to change the program structure (e.g., UPX inserts sections named *.upx0*, *.upx1*, etc.). Moreover, packing introduces many artifacts into the sections of the program. All features in this category can be extracted from the PE header. The average extraction cost for this feature category is consistently negligible compared to the time required to open the file, as shown in Figure 3.1.

3.3 Datasets and Ground Truth Generation

Our experimentation rely on a malware feed dataset. This section details the origin of this dataset, collection period, and methods used to generate the ground truth.

One challenge in testing, training, and classification, is the lack of high-quality packed datasets – a lack of a ground truth that can be used to reliably train classifiers. To handle this challenge, we bootstrap two different ground truths by using three existing packing detection tools on our dataset: a 3CONS ground truth with the files that all three detection tools agree on (mimicking [71]), and a 1CONS ground truth with the files that are detected as packed by at least one of the detection tools (a superset of all files in the 3CONS ground truth).

We have built a dataset of 281,344 binaries kindly provided by Cisco and prof. Sandeep K. Shukla. The dataset has been created by capturing binary file streams available to our partners in the period from February to October 2017, and is thus representative of the stream of binary files that has to be analyzed by a medium-sized security company protecting a number of customers. Most of these binary files have been reported as being malicious by our partners, thus they represent real-world malware samples collected from the wild.

We ran three heterogeneous signature-based packing detection tools on our dataset: Packerid [1] [76], the Yara rules for packing detection curated by VirusTotal [32] [2], and a proprietary tool provided by Cisco that checks the binary hash against Cisco's database. If one or two tools detect a binary as being packed by a given packer (e.g., UPX), the binary is added to the 1CONS ground truth. When all three tools agree on a binary being packed by a given packer, we consider this to be strong evidence that the binary is indeed packed by this packer, and add it to both the 3CONS and 1CONS ground truths. If a file is detected as packed by two different packers or more, the file is discarded to avoid polluting the labeling in the ground truths. Note that detection tools either report the detected packing technique, or "unpacked" if none is detected.

The ground truth selection is summarized in Table 3.1. Table 3.2 reports the number of samples of each packer family in each ground truth. It is clear that the 3CONS ground truth has significantly less samples that the 1CONS ground truth, due to the rarity of an exact consensus between the three tools. In our experiments, we only use packer families

1. Cross-platform and open source alternative of the tool PEiD. It is implemented in Python and relies on PEiD rules.
2. The techniques used by these tools are explained in Section 2.3.1.

Table 3.1 – Generation of consensus (3CONS) and non-consensus (1CONS) ground truths based on three tools (Packerid, Yara, and a Hash-based proprietary tool). File1 is detected by two tools out of three as TheMida, hence is added to the non-consensus ground truths as a TheMida sample. File2 is detected by all tools as UPX, hence is added to the consensus ground truth as a UPX sample. File3 is not detected as packed by any tool, hence is added to both ground truths as an unpacked sample. File4 is associated to more than one packing technique, hence it is not added to any ground truth.

File	Detection results		Ground truth	Label
File1	Packerid Yara Hash-based	TheMida TheMida unpacked	1CONS	TheMida
File2	Packerid Yara Hash-based	UPX UPX UPX	3CONS 1CONS	UPX
File3	Packerid Yara Hash-based	unpacked unpacked unpacked	3CONS 1CONS	unpacked
File4	Packerid Yara Hash-based	UPX Armadillo unpacked	discarded	-

with at least 10 samples, meaning that a classifier trained on the 3CONS ground truth will not be able to correctly classify Armadillo, AutoIt, etc., and no classifier will be able to correctly classify PEPacK. This can be solved by adding samples of the required families to the database.

We note that it is not possible for us to evaluate the accuracy of the three techniques (Packerid, Yara, and Hash-based detection) against an independently-generated ground truth, since in the context of malware packers, the ground truth is usually built relying on such techniques. Therefore, evaluating the accuracy of these techniques on another ground truth produced by the techniques themselves would be unsound. However, these techniques do represent the state of the art of static signature-based packing detection tools used in practice by security researchers, so we expect them to be quite accurate in practice.

Note that the ground truths are very unbalanced towards unpacked and UPX-packed files. This reflects the class distribution in the data sources we used, and generally means that algorithms that are better at detecting unpacked and UPX-packed samples will have a higher F-score than algorithms that are better at detecting other packer families. This stems from the intent of being able to correctly classify as many samples as possible. Here

Table 3.2 – Number of samples for each packer family in the two ground truths. Only families with ≥ 10 samples are used, so algorithms trained on 3CONS will be able to identify less families than algorithms trained on 1CONS.

Packer	3CONS	1CONS	**Packer**	3CONS	1CONS
Armadillo	0	6,849	NsPacK	17	60
ASPack	6,037	6,172	NeoLite	2	104
ASProtect	179	206	PackMan	24	78
AutoIt	0	1,048	PEArmor	0	793
CAB	0	75	PECloak	0	21
cpEllie	0	119	PECompact	1,240	1,327
cpFlush	0	19	PENinja	0	18
cpGlyph	0	15	PEPacK	6	7
CreateInstall	0	12	PEtite	9	13
D1S1G	0	360	ProtectSW	0	629
DarkComet	0	27	RARSFX	0	65
DevCv	0	36	RLPack	1	123
dlThunder	0	1,192	SafeDisc	0	13
dlUpatre	0	628	StealthPE	0	321
dUP2XPatcher	0	15	TASM	0	101
eXPressor	12	14	TheMida	13	150
EXEStealth	1	91	UPack	62	115
FSG	13	17	UPX	4,857	63,402
InnoSetup	0	975	WinRAR	0	48
InstallShield	4	22	WinZip	69	141
MEW	217	234	WiseInstaller	14	72
MoleBox	7	49	YodaProtect	1	95
mPress	0	847			
NSIS	7	2,039	unpacked	188,729	188,729

the averaging technique for multiclass F-score (detailed in Section 3.4) treats all samples as equally important. This means classification on unpacked samples as "unpacked" is strongly favored here as this is by far the most dominant family of samples.

3.4 Evaluation Metrics

This section sets the metrics used to evaluate the effectiveness, efficiency, and robustness of supervised ML-based packing detection and classification systems.

The *effectiveness* of a trained algorithm is measured by its F-score. We recall that the

F-score metric is defined as:

$$\frac{2 * Precision * Recall}{Precision + Recall} \tag{3.2}$$

Precision represents the number of correctly predicted positive labels over the number of all positive labels, including those not labeled correctly (i.e., false positives). *Recall* represents the number of correctly predicted positive labels over the number of all samples that should have been labeled as positive.

For binary classification, which means packing detection, calculating the F-score is straightforward since only two classes (packed/unpacked) are considered, knowing that a positive label indicates that a sample is "packed".

For multiclass classification, there are various methods to compute the F-score of each class separately and aggregate them into a single score, knowing that a positive label indicates the "packer family" of the packed sample (e.g., UPX). In our case, we assume our dataset is representative of a malware source, including the highly skewed distribution of packed and unpacked samples and the high prevalence of samples packed by common packers instead of rarer ones (see Table 3.2). If the F-scores of each category were weighted equally, a single misclassified sample out of the five available for a rare packer would be considered as a 20% misclassification rate and significantly reduce the aggregated F-score. This would translate as a higher weight on the classification of rarer packers compared to more common ones. Instead, the F-score for the multiclass packing classification experiments is computed using the micro-average method, i.e., by considering all samples as equivalently important (independently from their class).

The micro-average F-score is defined as follows. Consider a data set A of "real" (sample, label) pairs. Classify the samples of A with the algorithm, obtaining the set B of "predicted" (sample, labels) pairs. Then the F-score is just the ratio of correctly labeled samples, i.e., $F(A, B) = \frac{|A \cap B|}{|B|}$. Note that with this averaging method F-score, precision and recall coincide.

The choice of this averaging method depends on the assumption that the distribution of the samples in the classes in the ground truth is representative of the real case. A different choice of F-score averaging function for multiclass classification can be used to model, for instance, that the analyst is more interested in correctly classifying rare packing techniques than common ones, or vice versa.

Ideally, the samples in set A should be separate from the samples used to train the algorithm. For our experiments, we use 5-round 80–20 cross validation, meaning that we randomly select 80% of the ground truth for training and 20% for F-score computation, we

repeat this operation 5 times and average the five F-scores to obtain the reported F-score. However, for the *robustness* assessment (described in Section 3.1.3), we test the best algorithms straightforwardly against samples gathered after the training phase, without resorting to k-fold cross-validation.

We note that in our tests we are not concerned on whether the analyzed samples contain malicious code or not: packed goodware is detected and classified in the same way as packed malware, and we have no interest distinguishing between the two.

Finally, the *efficiency* of a trained classifier algorithm is measured as the inverse of the total time it takes to label the sample. This time includes the time required to extract features from the sample (extraction time) and the time for the algorithm to predict the label (classification time).

3.5 Experimental Evaluation

This section starts with an overview of software and hardware implementations, then presents the procedure and experiments we have followed to implement and evaluate our approach, as well as the results we obtained.

For the feature extraction we rely on the PeLib C++ PE file manipulation library published by Avast [77] and based on the original PeLib library created by [78]. The PeLib library has been developed by Avast specifically to handle malware, including corrupted or exotic PE files produced by malware obfuscation and packing techniques. Our implementation of feature extraction in C++ allows very low feature extraction times.

The ML detection and classification algorithms considered for our experiments (Gaussian Naive Bayes, Decision Tree, Random Forest, and Extra Trees) are implemented from the numpy Python library [79] version 1.14.2 and the Scikit Python package [50, 51]. The averaging F-score method used in the experiments (see Section 3.4) is defined and implemented following the *F1-micro* in the Scikit Python package [80].

The experiments were conducted on multiple servers running in parallel, with each server also allowing code to run in parallel in order to signficantly reduce the execution time. Precisely, the experiments were conducted on 9 Linux servers running in parallel. Each server features 14-core processors running at 2 GHz, allowing up to 56 parallel threads, and 128 GB of RAM. Finally, we relied on the Joblib Python library [81] to handle the parallel processing of Python code.

3.5.1 Definition of Classification Scenarios

Packing detection and classification can be considered as two separate challenges, as shown in Figure 1.1. However, by manipulating the ground truth, we can construct classifiers that address either of the two challenges or both at the same time. We consider these as different scenarios.

- [**DET**] *Packing detection.* An algorithms trained for the DET scenario only determines whether a sample is packed or not. An algorithm can be trained for the DET scenario by relabeling as packed every sample that is not labeled as unpacked in the ground truth, thus discarding the information about the specific packer family.
- [**CLAS**] *Packing classification.* An algorithm trained for the CLAS scenario assumes that a sample is packed, and determines by which family. An algorithm can be trained for the CLAS scenario by removing every sample labeled as unpacked from the ground truth.
- [**BOTH**] *Packing detection and classification.* An algorithm trained for the BOTH scenario determines both whether a sample is packed or not, and by which packer family if it is packed. An algorithm can be trained for the BOTH scenarios by using the whole ground truth with samples labeled as unpacked or by family, as shown in Table 3.2.

Each of the algorithms for the three scenarios above can be trained on the 3CONS or on the 1CONS, thus creating six possible scenario–ground truth combinations. In what comes next, we will refer to them by their scenario name and ground truth name, e.g., DET–3CONS refers to the case in which the classifiers are trained for packing detection on the 3CONS ground truth.

3.5.2 Feature Selection and Hyperparameter Optimization

Section 3.2 details the 119 features we use, divided into six categories: Byte Entropy (BE), Entry Bytes (EB), Import Functions (IF), MEtadata (ME), REsource (RE), and SeCtion (SC). Testing the contribution of each category to the detection and classification would be insufficient, since they are most likely not independent. Hence, to determine which feature categories fit the effectiveness and efficiency requirements, we test every combination of feature categories. For six categories, this corresponds to 63 non-empty and

Table 3.3 – Hyperparameters used for algorithm optimization. All hyperparameter combinations for each algorithm have been tested.

Algorithm	**Hyperparameter**	**Values**
Naive Bayes	prior	Gaussian
Decision Tree	splitter	best,random
	maximum depth	10,15,...,40
	maximum features	0.1,0.3,...,0.9
	minimum sample split	2,3,10
	minimum sample leaf	1,3,5
	criterion	gini,entropy
Random Forest Extra Trees	number of estimators	5,10,...,50
	maximum depth	10,15,...,40
	maximum features	0.1,0.3,...,0.9
	criterion	gini,entropy
	bootstrap	true,false

non-repetitive subsets, as shown in Section 3.2. Combined to the six scenario–ground truth combinations defined at the end of Section 3.5.1, this corresponds to 378 combinations of scenario–ground-truth–feature-category. To perform a fair comparison, for each of these 378 combinations, we perform a hyperparameter optimization on each of the classification algorithms, i.e., we create a set of possible hyperparameters for each algorithm shown in Table 3.3 and we find the best combination of hyperparameters by testing them all. We measure F-score by k-fold cross-validation, average extraction and classification time per sample, and we rate the algorithms by the ratio of their F-score to extraction plus classification time.

Table 3.4a presents for each scenario and ground truth the three algorithm–feature categories with the highest F-score. This provides insight for each scenario into which features contribute to the algorithms and which do not.

We can observe the following facts:

- The F-score of the best algorithms for the 3CONS ground truth is always higher than the results for the best algorithms for the 1CONS ground truth on the same scenario. This is expected because the 3CONS ground truth has higher quality than the 1CONS ground truth, meaning that less files are labeled incorrectly in 3CONS than in 1CONS. However, we recall from Table 3.2 that 3CONS contains significantly less families than 1CONS. This means that algorithms trained on 3CONS will be less effective than algorithms trained on 1CONS when used against packed malware in

Table 3.4 – Best algorithms and feature categories (Byte Entropy BE, Entry Bytes EB, Import Function IF, MEtadata ME, SeCtion SC, REsource RE) for each configuration of scenario (detection DET, classification CLAS, or both BOTH) and ground truth (3CONS or 1CONS).

(a) Top 3 algorithms by F-score

Scen-GT	Name	Algorithm	Features	Ext (s)	Clas (s)	F-score
DET–3CONS	D3F1	Extra Trees	BE EB IF SC	0.4248	0.0021	0.9998
	D3F2	Extra Trees	EB ME IF SC	0.3922	0.0021	0.9998
	D3F3	Extra Trees	BE EB IF RE SC	0.6188	0.0015	0.9998
DET–1CONS	D1F1	Random Forest	BE EB ME IF RE SC	0.6188	0.0024	0.9897
	D1F2	Random Forest	BE EB ME IF	0.4284	0.0029	0.9897
	D1F3	Random Forest	BE EB ME IF RE	0.6187	0.0024	0.9895
CLAS–3CONS	C3F1	Random Forest	BE EB IF	0.4284	0.0019	0.9998
	C3F2	Random Forest	EB ME IF SC	0.3922	0.0038	0.9998
	C3F3	Random Forest	EB ME IF RE SC	0.5826	0.0050	0.9997
CLAS–1CONS	C1F1	Random Forest	BE EB ME IF RE SC	0.6188	0.0048	0.9997
	C1F2	Extra Trees	BE EB ME IF RE SC	0.6188	0.0061	0.9979
	C1F3	Extra Trees	BE EB ME SC	0.0502	0.0049	0.9978
BOTH–3CONS	B3F1	Extra Trees	BE EB IF	0.4284	0.0055	0.9999
	B3F2	Extra Trees	EB IF RE	0.5826	0.0058	0.9999
	B3F3	Extra Trees	BE EB ME SC	0.0502	0.0033	0.9999
BOTH–1CONS	B1F1	Random Forest	BE EB ME RE SC	0.2406	0.0046	0.9875
	B1F2	Random Forest	BE EB ME IF RE SC	0.6188	0.0053	0.9874
	B1F3	Random Forest	BE EB ME SC	0.0502	0.0037	0.9873

(b) Top 3 algorithms by F-score / (extraction cost + classification cost)

Scen-GT	Name	Algorithm	Features	Ext (s)	Clas (s)	F-score
DET–3CONS	D3R1	Decision Tree	ME SC	0.0140	0.0003	0.9984
	D3R2	Decision Tree	SC	0.0140	0.0003	0.9979
	D3R3	Decision Tree	ME	0.0140	0.0003	0.9942
DET–1CONS	D1R1	Decision Tree	EB ME SC	0.0140	0.0003	0.9765
	D1R2	Decision Tree	EB ME	0.0140	0.0003	0.9748
	D1R3	Decision Tree	EB SC	0.0140	0.0003	0.9749
CLAS–3CONS	C3R1	Decision Tree	ME SC	0.0140	0.0003	0.9979
	C3R2	Decision Tree	SC	0.0140	0.0003	0.9977
	C3R3	Decision Tree	EB ME	0.0140	0.0003	0.9989
CLAS–1CONS	C1R1	Decision Tree	ME SC	0.0140	0.0003	0.9935
	C1R2	Decision Tree	EB ME SC	0.0140	0.0003	0.9948
	C1R3	Decision Tree	EB SC	0.0140	0.0003	0.9938
BOTH–3CONS	B3R1	Decision Tree	SC	0.0140	0.0003	0.9989
	B3R2	Decision Tree	ME SC	0.0140	0.0003	0.9990
	B3R3	Decision Tree	ME	0.0140	0.0003	0.9950
BOTH–1CONS	B1R1	Decision Tree	EB ME SC	0.0140	0.0003	0.9747
	B1R2	Decision Tree	EB SC	0.0140	0.0003	0.9683
	B1R3	Decision Tree	ME SC	0.0140	0.0003	0.9630

the wild. This is explored more in detail in Section 3.5.3, and in general represents a caveat against relying uniquely on k-fold cross-validation for fields with evolving environments like malware and packing.

- The Random Forest algorithm and its Extra Trees variant completely dominate: no Naive Bayes or Decision Tree reached the top 3 by F-score in any scenario–ground truth combination. This validates the intuition that higher complexity algorithms are required to achieve very high detection and classification scores, due to the complexity of the packing problem.
- All the top algorithms require many different features to achieve the highest scores. As a consequence, the feature extraction times are relatively high, often in the order of half a second per file. The EB feature category appears in all the top 18 algorithms, the BE and IF categories in 14 of them, the ME and SC in 13, and the RE in 9. This provides insight on the contribution of the various feature categories to the algorithms.

Table 3.4b presents for each combination of scenario and ground truth, the three algorithm–feature categories with the highest ratio between F-score and extraction plus classification cost. This provides insight as to which features and algorithms achieve a high F-score while having a low cost.

We can observe the following facts:

- The Decision Tree algorithm completely dominates, appearing as the algorithm with the highest F-score to cost ratio in every scenario and ground truth. The relative simplicity of Decision Tree compared to Random Forest allows it to achieve a slightly lower F-score with a significantly lower classification time, i.e., 0.3 ms per sample.
- None of the used feature categories require any significant extraction time over the 14 ms required on average to read the file itself (as explained in Section 3.2). This restricts the used feature categories to SC (used in 14 of the top algorithms), ME (12 algorithms), and BE (8 algorithms).
- Comparing the F-scores with the best algorithms overall for each scenario in Table 3.4a, we note that the F-score of the best algorithms by F-score/cost ratio in Table 3.4b usually does not decrease by more than 1–2%. On the other hand, the sum of extraction and classification costs per sample are from 17 to 44 times lower. Thus, with accurate algorithm and feature selection, it is possible to decrease the classification times significantly while keeping a very high F-score (**RQ1.1**).

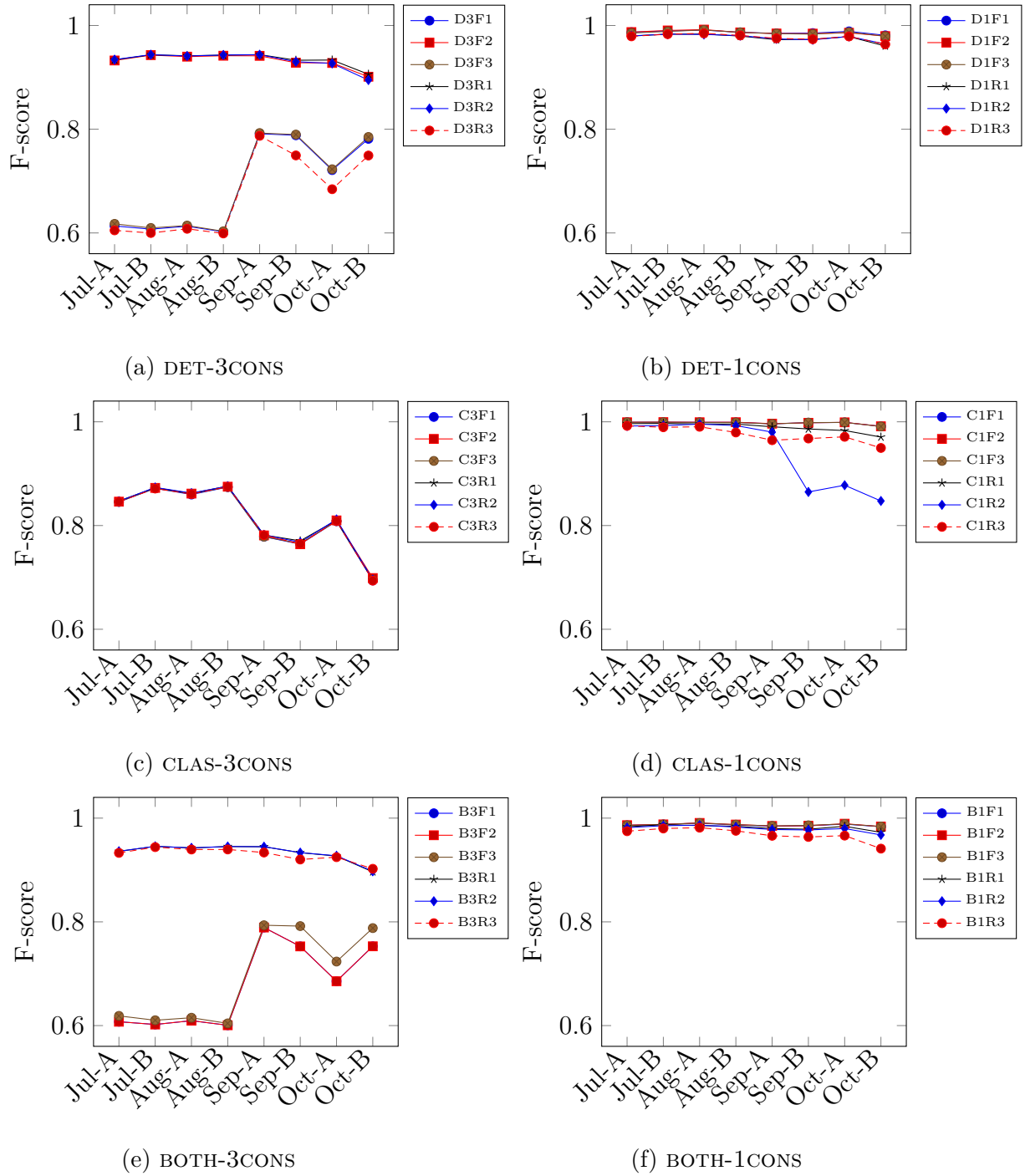
3.5.3 Robustness Assessment against the Evolution of Packers over Time

Since the malware and packing ecosystems evolve over time, new (variants of) packing techniques and families appear. To test the robustness of the packing detection and classification systems against this packers evolution over time, we are proposing, we perform a robustness assessment test over time. That is, we select the best algorithms for each scenario–ground truth combination, we train them with data from February to June 2017, and we test how they perform against data from July, August, September, and October 2017 in two-weeks intervals.

Figure 3.2 presents the results of the robustness assessment against the evolution of packers over time for the six combinations of scenarios and ground truths. The algorithms were trained on data from February to June as their ground truth, with the label of each graph indicating which consensus was used for the training. All algorithms were tested against data from July to October that was determined to be packed or not according to 1CONS, since 1CONS is *symbolically* considered in this context more similar to the malware and packing ecosystems in the wild than 3CONS. Hence, k-fold cross-validation was not used here.

We can observe the following facts:

- The algorithms trained on the 3CONS ground truth perform much worse than the algorithms trained on the 1CONS ground truth, losing 6–30% of their F-score (**RQ1.2**, **RQ1.3**). The algorithms trained on 1CONS lose only 1–3% of their F-score, except for the outlier C1R2 (**RQ1.3**).
  The reason is that, as shown in Table 3.2, the 3CONS ground truth contains less data and packer family labels than the 1CONS ground truth, hence such algorithms are not able to recognize a large amount of packer families. Indeed, the packing classification systems are supervised, thus theoretically unable to recognize new packer families that appear over time. For packing detection, the decrease in effectiveness does not come from a lack of family labels, because the problem is binary-class (packed, non-packed). However, the 3CONS ground truth contains less data, thus many observational feature values of packed binaries are discarded form the training, which has consequently an impact on packing detection.
  These results contrast with the results of Tables 3.4a and 3.4b in Section 3.5.2, where the algorithms trained on 3CONS had a higher F-score than the ones trained



(a) DET-3CONS (b) DET-1CONS

(c) CLAS-3CONS (d) CLAS-1CONS

(e) BOTH-3CONS (f) BOTH-1CONS

Figure 3.2 – Results of the robustness assessment against the evolution of packers over time. Each graph evaluates the F-score of the 3 best algorithms by F-score (Table 3.4a) and the 3 best algorithms by F-score/cost ratio (Table 3.4b) on a scenario-ground truth combination. Each algorithm is trained on data collected from February to June 2017 and tested on data collected on the first two weeks (A) and second two weeks (B) of July to October 2017.

on 1CONS. The reason is that the values in Section 3.5.2 are obtained by k-fold cross-evaluation, i.e., by testing and training on the same datasets. The important insight is that k-fold cross-evaluation favors scenarios with the construction of smaller ground truths and more precise labels, but at the cost of reducing the representativeness of the ground truth, thus training algorithms that will be ineffective against real data.

3.5.4 Retraining Cost Analysis

We perform a retraining cost analysis using the algorithms for the BOTH–1CONS scenario–ground truth combinations. Assume that an algorithm is useful while its F-score is at least 0.96, after which time the algorithm then has to be retrained. Consider the 6 algorithms whose robustness is plotted in Figure 3.2f. For each algorithm, we use the robustness assessment values to do a quadratic least squares regression and extrapolate the time at which the algorithm will reach an F-score of 0.96, requiring retraining. Such projection are presented in the second column of Table 3.5. The third column of Table 3.5 presents the retraining costs for the hyperparameter optimization for the algorithms from the experiments in Section 3.5.2. Finally, the fourth column of Table 3.5 presents the ratio between the uptime before retraining and the retraining time, i.e., how many seconds of uptime are obtained for each second spent retraining the algorithm. As the table shows, the simple Decision Tree algorithm with just the ME and SC feature categories has the highest ratio, since its very low retraining time cost means that it is more efficient to use it even if it has to be retrained more frequently than other algorithms (**RQ1.4**).

We note that the retraining cost analysis presented above considers for simplicity the time cost as the only cost of retraining, making it easy to evaluate compared to the uptime of the system. In reality, additional metrics could be used such: as the cost in kilowatts or currency of retraining the system, rollout costs for pushing an updated classifier, and various other factors. Such a detailed analysis would require additional information on the hardware and platform used, and possibly on the local energy and other infrastructure costs. For this reason such localized and precise costs are not considered in the scope of our retraining cost analysis.

Table 3.5 – Retraining cost analysis for the best algorithms for the BOTH–1CONS scenario–ground truth combinations, assuming that an algorithm has to be retrained when its F-score drops to 0.96.

Algorithm	**Uptime before F-score=0.96 (s)**	**Retraining cost (s)**	**Uptime to retraining cost ratio**
B1R3	7,410,960	5,750	1,289
B1R1	13,560,480	19,704	688
B1R2	11,274,120	18,990	593
B1F1	25,728,120	328,921	80
B1F2	22,048,920	362,979	61
B1F3	20,656,080	356,457	58

3.6 Discussion

This section first discusses the results in more details and summarizes the lessons learned. It then details the possible threats to the validity of our methodology and how they were addressed. Finally, it identifies the limitations and suggests potential improvements for future work.

3.6.1 Findings and Insights

On Feature Selection. We performed an evaluation of the cost and effectiveness of using different sets of features for supervised classification. Not all features have the same extraction cost, and training classifiers on large feature sets decreases their performance compared to training them on smaller feature sets. Hence, the efficiency of the classification is highly impacted by the features used. Table 3.4 shows that using all available features does not yield the highest F-score. This confirms that using less features can improve the extraction, training and classification times. To understand which features to exclude, we have examined not only their contribution to the F-score of the classification but also their cost. Similarly, we have evaluated complex algorithms like Random Forest with simpler algorithms like Decision Tree. The retraining cost analysis in Table 3.5 shows that when both effectiveness and efficiency are considered, then accepting a small decrease in effectiveness over time (uptime before F-score=0.96) can increase the efficiency (retraining cost) by orders of magnitude. This addresses research questions **RQ1.1** and **RQ1.4**.

This insight is of paramount importance to analysis and companies running large-scale ML-based malware and packing detection and classification systems. Due to the evolution of the malware and packing ecosystems, such systems require regular retraining on recent data representative of the current ecosystem. While carefully identifying effective algorithms and selecting features that are cheap to extract and evaluate, the system uptime to retraining cost ratio can be sharply increased, resulting in lower costs for the analyst and better protection for the users.

On Ground Truth. Many academic papers manually build a small-size ground truth of manually-verified samples. This is of course not possible for large-scale ground truths like the one used in this study. Due to the difficulty and sometimes inherent ambiguity of classifying samples, producing a highly reliable packing ground truth is extremely time-expensive and sometimes impossible. Hence, analysts and companies often have to rely on low-confidence ground truth, where the classification labels cannot be inherently trusted.

In this study we construct two ground truths of different sizes and reliability to understand the impact of the size and reliability of the ground truth on supervised classification: a 3CONS ground truth that is smaller but more reliable, and a 1CONS ground truth that is larger but less reliable. Importantly, some packer families that are present in the 1CONS ground truth had to be excluded from the 3CONS ground truth due to the lack of samples from such families that were classified with high enough confidence.

Table 3.4 shows that classifiers trained on the 3CONS ground truth achieve a higher F-score compared to classifiers trained on the 1CONS ground truth. However, the F-score of the classifiers are evaluated by k-fold cross-validation on the same ground truth used to train them, meaning that the classifiers trained on 3CONS are not evaluated on the packer families that appear in 1CONS but in 3CONS. This causes the classifiers trained on 3CONS to perform significantly worse than the classifiers trained on 1CONS in the robustness assessment in Figure 3.2, where all the packer families are considered. Research questions **RQ1.2** and **RQ1.3** are closely related to this result.

The robustness assessment over time is more representative of packing detection and classification in in-the-wild scenarios than k-fold cross-validation. Indeed, as explained in Section 3.4, k-fold cross-validation uses $\frac{k-1}{k}$ of the total size of data for training and $\frac{1}{k}$ for testing, for each of the k iterations. Thus, k-fold cross-validation assumes that the size of "known data", representing the training set, is much larger than the size of "unknown data", representing the testing set [3]. While this assumption might be correct for some

applications in in-the-wild scenarios, this is not valid for malware as well as packing detection and classification, where data in the wild evolve constantly and rapidly, thus making the set of "unknown data" much larger than the set of "known data".

On the one side, this shows that k-fold cross-validation method is not suitable for fields like malware as well as packing detection and classification. On the other side, this indicates that training classifiers on a larger, lower-reliability ground truth like 1CONS is preferable than training classifiers on a smaller, higher-reliability ground truth like 3CONS. Due to the evolution of the malware and packer ecosystems, wide coverage is necessary to properly train supervised classifiers, even at the cost of some label quality. This can be considered good news, since increasing the size of a database with samples with low-confidence labels is easier than increasing the confidence in the labels of the database.

3.6.2 Threats to Validity

As remarked in Section 3.3, we chose the three signature-based techniques used to annotate the database according to their widespread usage among security researchers and analysts. However, frequently the techniques were in disagreement on the labeling of a sample, not just whether a technique considers a sample packed and another considers it unpacked, but also cases where the three techniques labeled the same sample with up to six different packing families. Samples with such disagreements have been removed from the database, but this shows the fragility of these techniques against some packers. While on one hand this motivates the necessity of ML-based techniques like the ones proposed in this thesis, on the other hand it can be considered a warning on the unreliability of malware and packing classification ground truth.

As remarked in Sections 3.3 and 3.4, the creation of a database to use as ground truth requires some care to be aware of the representativeness and bias of the data. Additional bias is introduced by the choice of the effectiveness metric used to evaluate the classifiers (in this study, the F-score) and the averaging method for multiclass classification (in this study, all samples are considered equally important). The choice in this study reflects the fact that we give the same importance to false positives and false negatives in packing detection, and that we aim at correctly classifying the highest possible number of samples. Different choices may have led to different results. Analysts should always consider the origin of their data and their classification goals when choosing analysis parameters like the effectiveness metric and its averaging method.

The features used in this study have been divided into thematic-based groups (entropies, metadata, etc.) in Section 3.2. The grouping is necessary since features are not independent, so their effectiveness has to be evaluated on all possible combinations, and testing all combinations of 119 features is computationally infeasible, as explained in Section 3.2. However, dividing the groups by theme means testing together features with possibly very different extraction costs and effectiveness. For instance, Figure 3.1 shows that the cost of analyzing the resources section of the PE file varies from negligible to seconds. This is because if the resources section has been removed or is empty this is verified in negligible time, otherwise parsing it can require several seconds. Unfortunately, when not using the resource features we lose the very significant information on whether or not the resources section is empty. Future work should consider alternative feature groups.

The retraining cost analysis in Section 3.5.4 assumes that the analyst retrains a supervised classifier when its F-score drops to 0.96. The section shows how in this case it is more efficient to use simple algorithms and small feature sets to allow cheap and frequent retraining, increasing greatly the uptime to retraining cost ratio. However, it may be the case that the analyst requires a much higher F-score, e.g., 0.99, and this is attainable only by complex algorithm and large feature sets. While this does not invalidate our analysis, in this case the requirements would obviously not allow the cheap and frequent retraining we recommend.

3.6.3 Limitations and Future Work

A limit of supervised learning is to not be able to recognize classes that were not present in the ground truth. In the case of this study, this theoretical limitation does not affect the packing detection stage because the problem is binary-class (packed, non-packed). However, it affects the packing classification stage because packer families for which a classifier has not been trained will not be recognized. This limitation is the main cause of the ineffectiveness of classifiers trained on the 3CONS ground truth, as shown in Section 3.5.3.

More generally, unsupervised learning techniques like clustering should be used to provide information about malware packed with previously unknown packing techniques. In the next chapter, we present the framework SE-PAC which particularly relies on clustering to cover the theoretical limitations of supervised ML-based packing classifiers.

The construction of the ground truth has been shown to be fundamental in determining the effectiveness of the packing detection and classification processes. Additional packing detection techniques apart from the ones used in Section 3.3 could be used to improve the labeling of the ground truth. In particular, despite being costly, unpacking could be deployed to contribute in constructing the ground truth labels. In addition, it could be possible to rate the packing detection techniques with different confidence values based on their reliability, to better understand how to solve conflicting labeling in a more advanced manner than just the consensus/non-consensus paradigm used in this study.

Future work may examine the case of repacked malware, i.e., malware packed using multiple packers in sequence. In particular, the last packer may not completely overlay the outer layer of repacked malware, hence previous packer layers may still appear. We suppose that this multiform overlapping of multiple packers on the outer layer of malware is one of the main reasons for generating conflicting labeling (the ones we removed from the ground truth, as explained in Section 3.3). We refer to [82] for further discussion on this topic.

3.7 Conclusion

The study we presented in this chapter aims at understanding the impact of ground truth generation, ML algorithm selection, and feature selection on the effectiveness, efficiency, and robustness of supervised ML-based packing detection and classification systems, following the example of works on empirical testing of ML malware analysis including [3].

We find that the size of the ground truth is more relevant than its quality in in-the-wild test scenarios. Supervised ML algorithms can classify correctly only for classes they have been trained on, so having various classes and significant amount of samples for each class is more relevant than having a few classes of a few samples with highly reliable labeling. In particular, k-fold cross-validation method is not suitable for fields like malware as well as packing detection and classification where new types of samples and packing techniques appear constantly and rapidly in the wild. This contributes to explaining why algorithms can perform well in the lab and badly in the wild, as reported by [3].

We find that the number and extraction costs of the features used have a dramatic impact on classification and retraining times, and consequently on the viability of using ML algorithms. In fact, selecting features and algorithms for both effectiveness and

efficiency can greatly decrease classification and retraining costs against small decreases in effectiveness. In practice, this implies that using a simple algorithm (e.g., Decision Tree) on a reduced feature set and retraining it often results in an optimal expenditure of time compared to using a more complex algorithm (e.g., Random Forest) over many features and retraining it sparingly. This of course assumes that the minimum F-score requirements are not too high, in which case complex algorithms on large feature sets may be necessary. Our retraining cost analysis can be adapted as required considering cost and robustness.

We note that our results depend on some basic engineering choices, in particular the implementations of PE feature extraction and ML algorithms. More optimized implementations would possibly give better results in terms of efficiency.

We encourage other researchers and institutions to evaluate their packing detection and classification algorithms with the methodology presented in this chapter and to publish their findings.

Chapter 4

SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution

We have seen in the previous chapter that packers evolve, constantly bringing new classes or new variants of existing ones. Indeed, beside the many well-known packers in use (e.g., UPX, NsPack, ASPack), there is a growing trend for custom packers. The latter are developed either from scratch or partially from available ones (e.g., Vanilla UPX). Their usage has become so widespread that by 2015, Symantec detected their use in over 83% of all malware attacks [19]. Research works have also followed this evolution [18].

These new unknown packers complicate unpacking because the specialized unpacking function is unknown and generic unpackers are not always effective [83], which makes malware analysis and detection harder. Supervised ML-based packing detectors can detect new unknown packers because the problem is straightforwardly binary-class (packed, non-packed). However, we have seen previousely (see Section 3.5.3) that supervised ML-based models classifying packer families bring a strong theoretical limitation: they are not able to recognize new unseen classes. While we offer our models regular and efficient retraining, these retraining are still limited for supervised packer classifiers. The latter would still be unable to identify new packer families that appear constantly and rapidly in the wild, in the period of time occurring between each two retraining.

In this chapter, we particularly cover this theoretical limitation by introducing SE-PAC, a new Self-Evolving PAcker Classifier framework that copes with the fast-paced evolution of packers. SE-PAC constitutes the second contribution of this thesis.

Broadly, our framework relies on clustering in a semi-supervised fashion in order to cover the inability of supervised ML-based (and signature-based) systems to discover new classes. SE-PAC aims to provide an effective, incremental, and robust solution to cope with the rapid evolution of packers.

Our self-evolving technique consists in predicting incoming packers by assigning them to the most likely clusters, and relies on these predictions to automatically update clusters, reshaping them and/or creating new ones. Our system continuously learns from incoming packers, adapting its clustering to packers evolution over time.

The research challenges that we faced during the construction of SE-PAC concerned the similarity metric, optimization strategies for incremental update and post-clustering processing, and finally evaluation. We formulate these research challenges in terms of Research Questions (**RQ**), which are the following:

RQ2.1a How to define a pairwise distance metric when different types of features are extracted from packed binaries?

RQ2.1b How effective would such distance metric perform compared to other commonly used distance metrics?

RQ2.2 How the incremental update process can be optimized to reach a good trade-off between effectiveness and efficiency?

RQ2.3 Clustering is well-known to be an ill-posed problem for evaluation, so how the extrinsic quality of a clustering solution S can be evaluated wrt. the problem of the fast-paced evolution of packers?

RQ2.4 How effective and how robust is SE-PAC against packers evolution, particularly when dealing with binaries coming from the wild?

RQ2.5 How the post-clustering processing can be optimized?

The **contributions** that we precisely bring in this work are:

- We point out and experimentally show the importance of constantly updating the packing classification system.
- We introduce SE-PAC, a new end-to-end framework going from feature extraction, custom distance metric, to incremental clustering with a self-evolving classifier for packed binaries.
- We show how to combine different types of features in the construction of a composite pairwise distance metric, in the context of packed binaries (**RQ2.1a**). Furthermore, we show the efficiency of our composite pairwise distance metric by comparing it against simpler commonly used distance metrics (**RQ2.1b**).

- To decrease the update time of the incremental clustering procedure, we derive a methodology to extract representative samples for each cluster. Interestingly, the number of representatives extracted from each cluster is not fixed, but related to the number of samples in the cluster. Furthermore, we study this relation experimentally to derive a good trade-off between effectiveness and update time performance (**RQ2.2**).
- We show how to establish a good trade-off between the homogeneity of the clusters found and their number, to evaluate the extrinsic quality of a clustering solution S wrt. the problem of the fast-paced evolution of packers (**RQ2.3**).
- We propose a new post-clustering selection strategy that extracts a reduced subset of relevant samples from each cluster, to optimize the cost of post-clustering packer processing (**RQ2.5**).
- We support our findings with realistic experiments showing promising results for effectiveness and robustness (**RQ2.4**).

This chapter is organized as follows. Section 4.1 presents our methodology, and Section 4.2 our post-clustering selection strategy. Section 4.3 details the datasets and ground truth generation, and Section 4.4 the evaluation metrics. Section 4.5 presents the experimental setup and results, which Section 4.6 discusses. Section 4.7 concludes.

4.1 Methodology

This section starts with a brief overview of the methodology of our self-evolving classifier, before exploring each part in more details.

4.1.1 Overall Toolchain

Our approach comprises two phases (see Figure 4.1). First, the *offline phase* exploits and models all available knowledge, by using available packed samples as well as packer labels to tailor the generation of clusters. In the second phase, the *online phase*, the system self-evolves by incrementally updating the clusters as new samples, packed with either previously seen or unseen packers, are processed. Both phases are divided into three main steps: feature extraction, distance computation and clustering. Feature extraction and distance computation are identical for the offline and online phase, whereas clustering differs notably. Joining supervised learning in the offline phase and unsupervised learning in the online phase makes the whole system learn in a *semi-supervised* method.

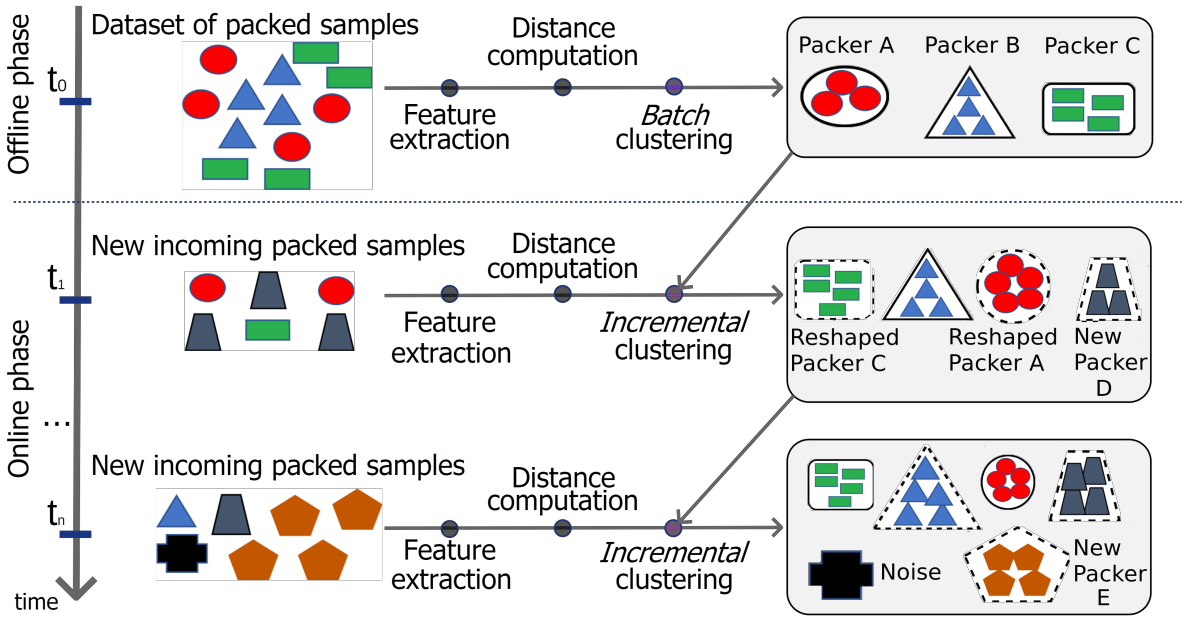


Figure 4.1 – Overall toolchain.

We rely on clustering because it can efficiently cluster similar samples that belong to the same packer family, *regardless whether the class was previously known*, and thus fits our goal of *discovering new unknown packer classes*. In particular, *incremental clustering* provides *incremental learning* where knowledge about packers constantly evolves, by creating new clusters for new families, or reshaping existing clusters according to new variants that can represent different versions, configurations, or polymorphic instances.

Initially, at t_0, **the offline phase** trains the self-evolving packer classifier and tunes parameters that will be used in the online phase. The initial dataset contains packed samples and the corresponding ground truth labels. Various static and dynamic features are extracted from each packed sample, to represent it by a *heterogeneous* features set. To define a unique pairwise distance for pairs of packed samples, each individual feature yields a *partial distance metric* that is averaged with all the others to obtain the final distance. Clustering is then used to group similar packed samples into clusters. The best clustering is found by tweaking the distance between packed samples according to the ground truth, so that the obtained clustering combines *homogeneity* (clusters do not mix packers from different families) with the most *coherent number* of clusters (the closest to the number of packer families in the ground truth). At the end of this phase, the parameters that produce the best clustering, as well as the corresponding clusters, are the first *clustering setup* which serves as baseline for the second phase.

The online phase $(t_1, ..., t_n)$ has successive self-evolutions, processing one packed sample at a time, for as long as the model produces accurate results before requiring full retrain or supervisor intervention. In this phase, the same tasks are repeated for features extraction and distances computation. Then starting with the clustering setup obtained from the offline phase, the system self-evolves by relying on incremental clustering that *dynamically* includes the incoming packed samples, thus creating new clusters for new families and reshaping existing clusters with new variants. Since our system classifies new incoming packed samples in real-time, it can be used in production, hence reinforcing the security of the user.

4.1.2 Feature Extraction and Selection

We take advantage of the 6 packer feature categories that we extracted in our ML study for packing detection and classification: *metadata* (21 features), *sections* (21 features), *entropies* (6 features), *resources* (2 features), *import functions* (5 features), and *entry bytes* (64 feature). All features are described in the previous Chapter, in Section 3.2.

However, we bring an important improvement over the *entry bytes* category representing the unpacking stub code by performing a *lightweight dynamic emulation* of the execution of the first instructions following the binary Entry Point (EP), instead of extracting 64 bytes *statically*. As instructions are fetched, the corresponding mnemonics are stored in a list, which is the assembly language (ASM) mnemonic sequence feature.

Dynamic emulation has the advantage to thwart unpacking stub codes that use obfuscation techniques to impede static analysis, such as far jumps, anti-disassembly and metamorphism (see Section 2.2.1). Although an attacker can engineer a bunch of code protection that foreruns the unpacking code routine, this would not mislead our step since this bunch of code protection would serve as well to identify the packer family.

Our improvement targets particularly the *entry bytes* feature category because the unpacking stub code is a strong characteristic feature [66, 69, 70, 84] for packer classification. The rational is that runtime packers often start executing the unpacking stub routine before reaching the malicious payload original entry point. Indeed, from the results that we obtained previously regarding the contribution of feature categories in classifying packer families (see previous Chapter, Table 3.4a for CLAS scenarios), we can observe that the category EB is mostly present in all the top-ranking scenario–ground-truth–feature–category. Hence, this validates the intuition of the prominent contribution of this feature category in classifying packer families.

So regarding this new context of features and ML algorithms, a selection of feature categories is performed again for representing packers. We test all possible non-repetitive category combinations in the offline phase, then we rank them conforming to the criteria introduced in the previous Chapter, that is effectiveness (AMI score in this context, see Section 4.4.1) and the ratio between effectiveness and efficiency (time cost on average of feature category extraction).

When we rank the combinations of feature categories wrt. effectiveness, the best combination we obtain is *sections* associated to *unpacking stub mnemonic sequences*, with an AMI score of 0.963 on average wrt. the scenarios we designed (described next in Section 4.5.1). The time cost of extracting the two feature categories was 0.244 seconds in total (0.014 seconds for *sections* and 0.23 seconds for *unpacking stub mnemonic sequences*). The best combination we obtain, by ranking according to the ratio between effectiveness and efficiency, is *sections* with an AMI score of 0.909 on average and a time cost of feature extraction of 0.014 seconds.

We notice that the ratio reduces significantly the time cost of feature category extraction, where we move down from 0.244 seconds to 0.014 seconds, but a the cost of a significant decrease in effectiveness, where we move down from an AMI score of 0.963 to 0.909. These results contrast the ones we obtained in the previous Chapter, in Table 3.4a, where the ratio provided a significant reduction in the time cost with, interestingly, a *neglected* loss in effectiveness.

In this context, we do not accept this significant loss in effectiveness, because the latter is favored as being a high priority of the system over the time cost of feature extraction. Hence, the ratio above no longer provides us an interesting trade-off. So for these reasons, we decided to go in the direction of the most *effective* combination of feature categories, thus selecting the association of the feature category *sections* and the feature category *unpacking stub mnemonic sequences* for representing packers. Besides this combination provides the best effectiveness, it also solidifies SE-PAC against obfuscation techniques that can deceive each category taken separately. Furthermore, it reduces ML over-fitting that could be caused by using each feature category alone.

Regarding the *efficiency* of SE-PAC, our efforts focused thus on providing a very *lightweight* extraction of the *unpacking stub mnemonic sequences*. The efficiency of the system concerns also the time cost to update a sample in a clustering procedure, so we present next in Sections 4.1.4.1 and 4.1.4.3 the optimization strategies we adopted to reduce that cost.

4.1.3 Composite Pairwise Distance Metric

Since extracted features are *numeric* (relating to PE sections) or *string sequences* (mnemonic sequences), we need to derive a unique pairwise distance metric able to combine both types. To this end, we derive a Gower distance [85], a composite metric overcoming the issue of mixed data type variables by being computed as the average of partial distances that range in $[0, 1]$ (**RQ2.1a**). Our work has two different partial distance metrics: *Manhattan* distance for numeric features and *Tappered Levenshtein* distance for mnemonic sequences. These two distances are then normalized, and their average computed, thus providing our composite distance metric. Formally:

$$Gower(i, j) = \frac{1}{p} \sum_{k=1}^{p} NormD(i, j)^{(f_k)} \tag{4.1}$$

i and j are the indices of two samples in the dataset, $NormD(i, j)^{(f_k)}$ is the normalized partial distance metric applied wrt. the data type of the k^{th} feature f, and p is the number of features. Each packed sample is represented by 22 features (a string and 21 numbers).

The Manhattan distance provides partial distance metrics for numeric features. The normalized partial distance of a numeric feature f between two samples of indices i and j is the ratio between the absolute difference of observations $x_i^{(f)}$ and $x_j^{(f)}$ and the absolute maximum range $R^{(f)}$ observed for f among all samples:

$$NormManh(i, j)^{(f)} = \frac{|x_i^{(f)} - x_j^{(f)}|}{|R^{(f)}|} \tag{4.2}$$

The Tapered Levenshtein distance provides partial distance metrics for ASM mnemonic sequences [84]. It has the advantage of quantifying the similarities between two ASM mnemonic sequences, in contrast to straightforward comparison that cannot capture small differences in the sequence. Furthermore, the *tapered* version proportionally decreases the weight of each element as they appear later in the sequence, punishing more differences in the beginning of the sequence and less in the end. In our work, the intuition behind *tapering* is that most of the times the unpacking stub routine is located directly at the binary EP, and thus the order in which the instructions appear is important. Nonetheless, since the length of the unpacking stub is a priori unknown, the focus is to fetch as few instructions as possible to attain the optimal balance between efficiency and information gain for packer classification. The normalized partial distance of a string

mnemonic sequence f between two packed samples i and j is formally defined as:

$$NormTapLev(i,j)^{(f)} = \frac{\sum_{k=0}^{L(S_i,S_j)-1} W(S_i,S_j)(k)(1-\frac{k}{L(S_i,S_j)})}{C} \tag{4.3}$$

where S_i and S_j are the respective mnemonic sequences of the two packed samples whose indexes are i and j. $L(S_i,S_j)$ is the maximum length between S_i and S_j. $W(S_i,S_j)(k)$ is a *factor* which is equal to 0 if the ASM mnemonic of S_i and S_j are equal at the position k, or equal to 1 otherwise. Finally, C is the maximum extraction length of any mnemonic sequence, used to normalize the distance into $[0,1]$.

Example 1. *Let C be set to 50, and S_i and S_j the following ASM mnemonic sequences, extracted from packed samples i and j:*

$$S_i = \{push, mov, push, push, push, mov, push, mov, sub\},$$
$$S_j = \{push, mov, mov, push, add, mov, add, mov, sub\},$$

then $L(S_i,S_j) = 9$ and $NormTapLev(i,j) = 0+0+(1-(2/9))+0+(1-(4/9))+0+(1-(6/9))+0+0 = 1.67/50 = 0.033$

4.1.4 Clustering: Batch and Incremental

In both offline and online phases we use DBSCAN (a background for the algorithm is provided in Section 2.3.3.1) as clustering algorithm, because it: (i) does not require the a priori number of clusters, which fits our ambition of discovering new packers; (ii) can find arbitrarily-shaped clusters, since the packers may be different in terms of complexity (see Section 2.2.1), packing techniques, or how they overlay the binary; (iii) has the notion of noise, which can have multiple interpretations in our context: rare (singleton) packers, outliers, or packers using very sophisticated obfuscation techniques and thus hard to group under clusters; (iv) gives control on parameters (*eps* and $minPts$), which allows parameter tuning in the offline phase.

4.1.4.1 Scattered Representative Points

To ease locating the clusters in the hyperplane while reducing comparisons complexity during the incremental update process, we select for each cluster a subset of its points

to represent its geometry. To this end, we target the *most scattered points* of the cluster, which we call *Scattered Representative Points* (SRPs), see Figure 4.2.

Finding the most scattered points of a cluster is akin to the *farthest neighbors traversal* problem: given a set of N points, find the X points that are the farthest apart from each other. There are heuristics in literature to solve this problem [86]. In our work, we trade-off between precision and performance, with a *greedy approach*: first take the two farthest points, then incrementally select points for which the sum of distances to the already selected points is the greatest, repeating this until a specific number of representative points (n_{rp}) have been selected. This n_{rp} is equal to $\left\lceil K\sqrt{N_c}\right\rceil$, where N_c is the number of elements of cluster c, K is a positive constant (that can be experimentally tuned), and $\lceil . \rceil$ is the round up number. This means that each cluster has a different n_{rp}, at least equal to 1, that grows dynamically with the number of elements of the cluster. For the sake of clarity, a pseudocode is given in Algorithm 1.

4.1.4.2 Batch Clustering in the Offline Phase

In this phase, DBSCAN parameter (*eps*) is tuned to achieve the best clustering wrt. the provided ground truth. The clusters are evaluated according to their *homogeneity* and *number* wrt. the number of packer families in the ground truth. The trade-off between these two constraints is established via clustering metric AMI (see Section 4.4.1).

The intuition behind the offline phase is to derive a generalized *eps* that is *learned* from the already available labeled samples. This experimental *eps* should work for a big variety of packers and give the incremental clustering step the ability to handle *new unseen packers*, hence the importance of providing a well-varied packer training set at the beginning. In practice, this variety represents various packer complexities, and techniques, e.g., self-installers, cryptors, compressors, protectors and virtualizers.

Parameter *minPts* is left as supervisor choice for the desired minimum number of samples in clusters. Assuming that some unknown packers (families) can be spread with very few elements (e.g., only 3), the value of this parameter should be low so that our clustering encompasses small clusters (in addition to larger ones).

4.1.4.3 Incremental Clustering in the Online Phase

The *online* phase is responsible for updating the existing clustering as new samples come in. To this end, we customize an incremental version of DBSCAN [87] including

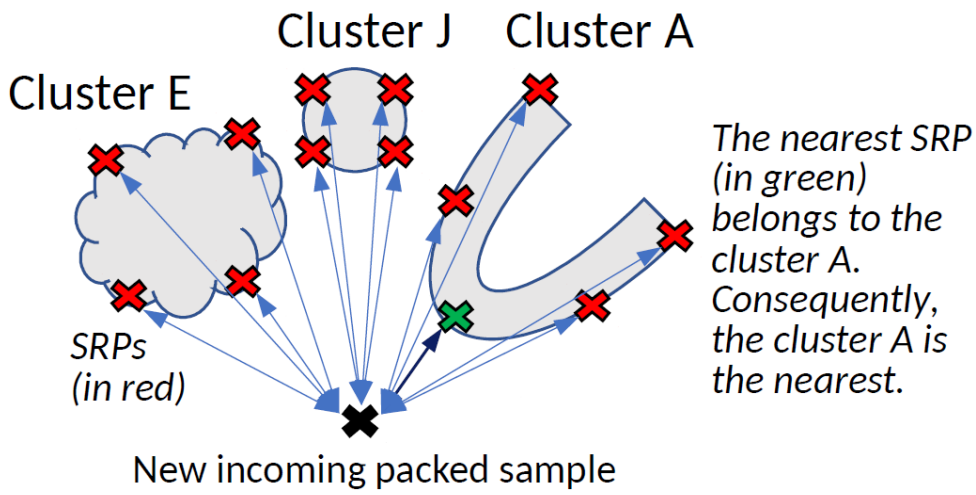


Figure 4.2 – Nearest cluster search in incremental update.

a specific update policy and a set of optimization techniques. This *update policy* covers three cases, checked in the following order:

A: The new sample joins the nearest cluster when at least $minPts$ sample, including the new one, are within radius eps.

B: A new cluster is formed when at least $minPts$ unclassified samples (noise), including the new one, are within radius eps.

C: The new sample remains unclassified when not A nor B.

A naive update would require computing distances between the new (sample) point and all the points in the clusters, resulting in a complexity of $\mathcal{O}(M \cdot N_c)$ per update, where M is the number of existing clusters and N_c the number of points in cluster c. However, using the *SRPs* in order to select the nearest cluster reduces the computation to $\mathcal{O}(M \cdot n_{rp})$ (**RQ2.2**), in our case $\mathcal{O}(M \cdot \sqrt{N_c})$ (see Section 4.1.4.1). The idea to optimize the computations is to get gradually closer to the hyperplane region where the potential points for case A are located. Therefore, in the first step, we compute the distances to the set of SRPs of each cluster, as illustrated in Figure 4.2, to find the nearest cluster. In the second step, we delimit the region of points around the nearest SRP for which distances have to be computed. By exploiting the *triangle inequality*, using the distances already computed between the nearest SRP and the points in the cluster, we identify the set of points that are *certainly closer* or *further* than *eps* from the new point. These points can thus be directly *accepted* or *rejected* (without distance computation) within the update policy. Then, only the remaining points (those we can not decide using the

Algorithm 1 Calculate_or_update_SRPs

Input:
C_j *with* $j \in \{1, 2, 3, ..., l\}$ {the cluster for which SRPs are calculated or updated}
M {the precomputed pairwise distances matrix}
K {the experimental K in $\left\lceil K\sqrt{|C_j|}\right\rceil$}

Output:
$SRPs_{C_j}$ {list of the best SRPs of the cluster C_j}

Function *calculate_or_update_SRPs*(C_j, K, M)

1: $n_{rp_{C_j}} \leftarrow \left\lceil K\sqrt{|C_j|}\right\rceil$ {calculating the number of SRPs for the cluster j}
2: $M_{C_j} \leftarrow M \setminus C_j$ {getting only the precomputed distances of the points belonging to the cluster C_j}
3: *points_remaining* $\leftarrow \{C_j\}$ {initializing the remaining points set with all the points belonging to the cluster j}
4: *solution_set* $\leftarrow$ *two_farthest_points*(M_{C_j}) {initializing the solution set with the two points having the greatest distance between them in M_{C_j}}
5: *points_remaining* $\leftarrow$ *remaining_points* $\setminus$ *solution_set* {removing the two points from the remaining points set}
6: **for** $i \in \{3, ..., n_{rp_{C_j}}\}$ **do**
7: *solution_set* $\leftarrow$ *solution_set* $\cup$ *point_with_max_sum_distances*(*solution_set*, *points_remaining*) {Adding to the solution set the point for which the sum of distances from it to all the points already in the solution set is the greatest, until a solution set of $n_{rp_{C_j}}$ points is reached}
8: *points_remaining* $\leftarrow$ *points_remaining* $\setminus$ *solution_set*
9: **end for**
10: $SRPs_{C_j} \leftarrow$ *solution_set*
11: *return* $SRPs_{C_j}$

Algorithm 2 Customized incremental DBSCAN

Parameters used:

M {the precomputed pairwise distances matrix}
$best\_eps$ {the best tuned eps in the offline phase}
$best\_minPts$ {chosen by the supervisor}
P_{new} {the point representing the new incoming sample}
C {list of clustes formed in the offline phase}
$Noise$ {list of points considered as noise in the offline phase}
K {the experimental K in $\lceil K\sqrt{|C_j|} \rceil$ with $j \in \{1, 2, 3, ..., l\}$}
$SRPs_{C_j}$ {list of SRPs of each cluster j}

Online Phase

$C_{near}, P_{SRP} \leftarrow min\_distance(SRPs_C, P_{new})$ {finding the nearest SRP (P_{SRP}) from P_{new} in all clusters, thus finding the nearest cluster}
$accepted\_points \leftarrow \emptyset$, $rejected\_points \leftarrow \emptyset$, $remaining\_points \leftarrow \emptyset$
if $M[P_{SRP}, P_{new}] \leq best\_eps$ **then** {attempting the triangle inequalities to reduce the number of points to process in the cluster}
    for $i \in \{1, ..., |C_{near}|\}$ **do**
        if $M[P_i, P_{SRP}] \leq best\_eps - M[P_{new}, P_{SRP}]$ **then**
            $accepted\_points \leftarrow accepted\_points \cup P_i$ {P_i is a cluster point}
        else if $M[P_i, P_{SRP}] > best\_eps + M[P_{new}, P_{SRP}]$ **then**
            $rejected\_points \leftarrow rejected\_points \cup P_i$
        else
            $remaining\_points \leftarrow remaining\_points \cup P_i$
        end if
    end for
else
    for $i \in \{1, ..., |C_{near}|\}$ **do**
        if $\{M[P_i, P_{SRP}] < M[P_{new}, P_{SRP}] - best\_eps\}$ **or** $\{M[P_i, P_{SRP}] > M[P_{new}, P_{SRP}] + best\_eps\}$ **then**
            $rejected\_points \leftarrow rejected\_points \cup P_i$
        else
            $remaining\_points \leftarrow remaining\_points \cup P_i$
        end if
    end for
end if
if $\{|accepted\_points| + 1) \geq best\_minPts\}$ **or** $\{Check\_minPts\_in\_eps(P_{new}, remaining\_points, best\_eps, best\_minPts) = True\}$ **then**
    $C_{near} \leftarrow (C_{near} \cup P_{new})$ {P_{new} joins the nearest cluster}
    if $\lceil K\sqrt{|C_{near}|} \rceil > \lceil K\sqrt{|C_{near}| - 1} \rceil$ **then**
        $calculate\_or\_update\_SRPs(C_{near}, K, M)$ {updating the set of SRPs of the modified cluster when its n_{rp} increases}
    end if
else if $Check\_minPts\_noise\_in\_eps(P_{new}, Noise, best\_eps, best\_minPts) = True$ **then**
    $C_{l+1} \leftarrow create\_new\_cluster(close\_noise\_points)$ {clustering noise points that are close to each other}
    $C \leftarrow C \cup C_{l+1}$ {updating the list of existing clusters}
    $calculate\_or\_update\_SRPs(C_{l+1}, K, M)$ {calculating the SRPs of the new formed cluster}
else
    $Noise \leftarrow Noise \cup P_{new}$ {adding the new incoming sample as noise}
end if
$update\_pairwise\_distance\_matrix(M, P_{new})$

triangle inequality) would require distance computation. A more detailed description of the possible cases is given in the Appendix 2.

If the new sample joins the nearest cluster (case A), the set of SRPs of the cluster may have to be updated. Such update, if performed frequently (e.g., at each cluster modification), would make the incremental process costly. Thus, to be efficient while still capturing the evolving geometry of the cluster, we limit the update of SRPs to be recomputed only when n_{rp} increases. For the sake of clarity, a pseudocode is given in Algorithm 2.

4.2 Post-Clustering Sample Selection

After some amount of updates, an analyst may want to select a subset of *Post-Clustering Relevant Samples* (PCRS) [1] from clusters found by SE-PAC in the online phase, for further packer analysis (see Figure 4.3).

Our strategy consists in representing the cluster by multiple connected regions from which we select PCRS. The diameter of these regions is set up by the analyst and allows for quicker or more detailed view on the clusters. The selected samples are ranked according to their region density in order to provide a measure of their *relevance* in the cluster. We call this measure *density marker*.

Our PCRS selection strategy assumes that any shape generated by DBSCAN consists of connected *core points* (see Figure 4.4). Core points that are close (within *eps* radius) are relatively similar. A core point can be selected to represent the points (including core points) comprised within its region. More precisely, for given a cluster $\mathcal{C}$, for each of its core points P_i our procedure visits, we identify the list of core points $\mathcal{P}_r^{(P_i)}$ directly reachable within radius r (selected by the analyst), i.e., $\mathcal{P}_r^{(P_i)} = \{P_{j,j\neq i} | d(P_i, P_j) \leq r\}$. The density marker of P_i is computed as $\rho_r(P_i) = \frac{|\mathcal{P}_r^{(P_i)}|}{|\mathcal{C}|}$. Our traversal procedure visits a core point P_j only if it does not belong to the list $\mathcal{V}$ of previously visited core points, nor to their reachable core points list, i.e., $P_j \notin \{\mathcal{V} \cup \mathcal{P}_r^{(P_k)} | P_k \in \mathcal{V}\}$. At the end of the procedure, the cluster is represented by the list of visited core points $\mathcal{V}$, for which samples with higher ρ_r are more *relevant* for further analysis (**RQ2.5**).

1. Not to be confused with SRPs.

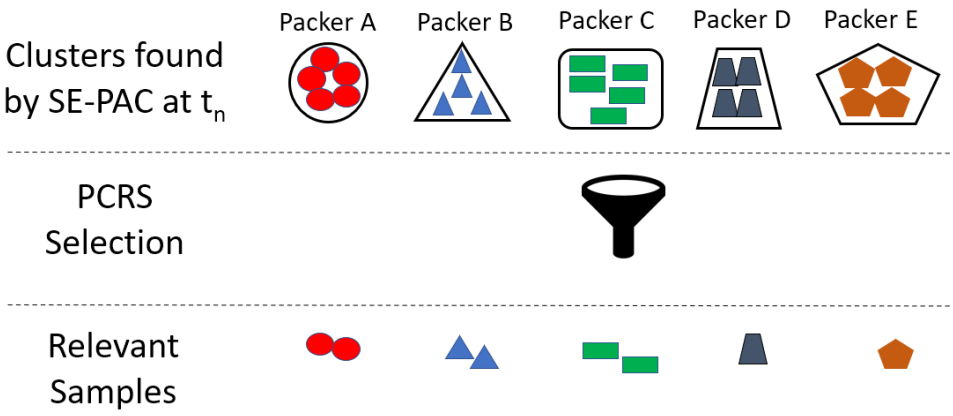


Figure 4.3 – PCRS selection from clusters found by SE-PAC in the online phase at t_n.

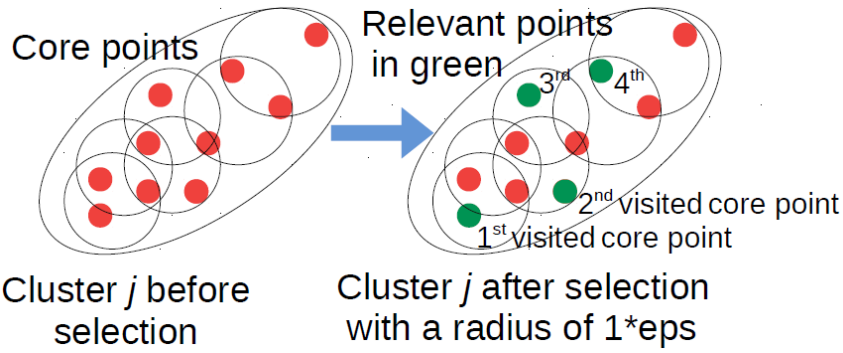


Figure 4.4 – PCRS selection strategy. Note that the selection of the first core point to visit is done randomly.

4.3 Datasets and Ground Truth Generation

In our experimentation, we rely on two datasets: a malware feed and a synthetic. For the malware feed, the origin, collection period, and the ground truth construction is described in the previous Chapter, in Section 3.3. So in the Section 4.3.1 below, we will just describe the differences that we mainly brought on this dataset for our experimentation. For the synthetic dataset, the same information is given next in Section 4.3.2.

4.3.1 Malware Feed

While the samples of this dataset are exactly the same as described previously, we brought some differences in the construction of the ground truth.

Indeed, we first added the well-reputed signature-based tool DIE (see Section 2.3.1) as additional tool for building the ground truth labels, then for many packer families, we

Table 4.1 – Malware Feed. Packers in blue italics are *specific* to this dataset. Packers in black belong to families *common* to both malware feed and synthetic datasets. "v?" is unspecified version. "x' is one or multiple sub-versions.

Packer	(version, # samples)	Total	Packers	(version, # samples)	Total
ActiveMARK	(v?, 2), (v5.x, 1)	3	*NsPacK*	(v?, 1), (v3.x, 13)	14
ASPack	(v1.08.x, 2), (v2.1, 13), (v2.12-2.42, 5,818)	5,833	Packman	(v1.0, 23)	23
ASProtect	(v1.0, 1), (v1.23-2.56, 174)	175	*PCGuard*	(v4.06, 3), (v5.0x, 1)	4
AutoIt	(v3, 1,036)	1,036	*Shrinker*	(v3.5, 3)	3
ExeStealth	(v2.74, 1)	1	*PEPACK*	(v1.0, 6)	6
eXPressor	(v1.3, 11), (v1.4.5.x, 1)	12	*PESpin*	(v1.3x, 6)	6
FishPE	(v1.3, 3)	3	Petite	(v2.1, 2), (v2.2, 7)	9
FSG	(v1.33, 2), (v2.0, 11)	13	RLPack	(v1.15-1.18, 1)	1
InnoSetup	(v?, 51), (v1.12.9, 1), (v1.3.x, 3), (v2.0.x, 6), (v3.0.x, 7), (v4.0.x, 4), (v4.1.4, 1), (v4.2.x, 4), (v5.0.x, 4), (v 5.1.x, 40) (v5.2.x, 35), (v5.3.x, 73), (v5.4.x, 45), (v5.5.x, 660)	934	PECompact	(v19x, 1), (v20x, 14), (v2.7x, 12), (v2.80x, 1), (v2.9x.x, 13), (v3.0x.x, 1,185)	1,226
InstallShield	(v?, 1), (v7.01.x, 1), (v9.00.x, 1), (v10.50.x, 1)	4	Themida	(v1.8.x-1.9.x, 12)	12
MEW	(v1.1-1.2, 217)	217	UPack	(v0.1x/0.20/0.21/0.24, 1), (v0.24-0.27/0.28, 19), (v?, 21)	41
MoleBox	(v2.3.3-2.6.4, 7)	7	WinRAR	(v?, 35)	35
NeoLite	(v1.0, 2)	2	UPX	(v0.6x, 2), (v0.7x, 5), (v1.0x, 40), (v1.2x, 2,157), (v1.9x, 14), (v2.0x, 114), (v2.9x, 6), (v3.0x, 1,599), (v3.9x, 609)	4,549
NSIS	(v?, 361), (v1.x, 5) (v2.0x, 47), (v2.1x, 20), (v2.2x, 28), (v2.3x, 33), (v2.4x, 977), (v2.5x, 43), (v3.0x, 397), (v9.99, 4)	1,916	WinZip	(v3.1, 66)	66
			Wise	(v?, 8)	8
Malware feed			All packers		16,159

selected from the feed the binaries for which there was a consensus of at least 3 tools out of 4 (DIE, Yara, Packerid, and the Hash-based proprietary tool of Cisco).

This 3/4 consensus is a trade-off between the size of the ground truth and its quality in terms of confidence, conforming to the conclusions made previously regarding the ground truth construction in in-the-wild scenarios (see Section 3.7). The goal of this trade-off is to include more families in the offline phase in order to get a more representative "eps", while keeping enough quality for training and evaluation.

So with this 3/4 consensus, we selected 16,159 binaries out of the whole feed dataset of 281,344 binaries. Table 4.1 summarizes selected packer families as well as the packer versions indicated by DIE to illustrate a possible (low-reliable) version labeling. We note that the packer version labeling is low-reliable because: 1) it is extremely hard in practice to meet a 3/4 consensus over the four tools on the exact version of the packer family; 2) our focus is classifying packers at the family granularity, so a low-reliable version labeling can just give possible explanations on why some packer families split across multiple sub-clusters instead of forming just one cluster family (as it will be shown next in Section 4.5.3.2).

We note that the complexity classes (see Section 2.2.1) of most of these packers range from I to III, like more than 85% of the worldwide packers evaluated in [18]. This means our packers are quite representative in terms of complexity. They also use diverse packing techniques: self-installers, compressors, cryptors, protectors, and virtualizers.

4.3.2 Synthetic Dataset

We created a second dataset of 18,798 packed binaries [2]. On a freshly installed 32-bit Windows 7, we collected 694 PE clean binaries mainly from the *system 32* folder and packed them with 31 public, commercial, professional and custom packers. The fact these binaries are cleanware and not malware is not a problem, since we focus mainly on packers, which are not necessarily malicious, as explained at the end of Section 3.4.

Packers "Custom Packer i", $i \in [1..10]$, come from public repository Github. Their families are not recognized by PEiD, Yara nor DIE. Although we could just use known packers (e.g., UPX, Armadillo) to simulate the arrival of new packer classes after training, these extra Custom Packers simulate more concretely the arrival of *new* packers. Indeed, in practice malware can simply be packed with a very recent custom packer, taking advantage

2. For access to this dataset, please contact me on this email: laminho@live.fr.

Table 4.2 – Synthetic Dataset. Packers in blue italics are *specific* to this dataset. Packers in black belong to families common to both malware feed and synthetic datasets.

Packer	**(version, # samples)**	**Total**	**Packers**	**(version, # samples)**	**Total**
Armadillo	(v2.52, 628)	628	MEW	(v1.1, 634)	634
ASPack	(v2.36, 633)	633	*mPress*	(v2.19, 593)	593
Custom Packer 1 [88]	(v1.0, 125)	125	NeoLite	(v2.0, 617)	617
Custom Packer 2 [89]	(v1.0, 22)	22	PackMan	(v1.0, 640)	640
Custom Packer 3 [90]	(v1.0, 277)	277	PECompact	(v3.03.23, 670)	670
Custom Packer 4 [91]	(v1.0, 648)	648	*PELock*	(v2.08, 621)	621
Custom Packer 5 [92]	(v1.0, 655)	655	*PENinja*	(v1.0, 666)	666
Custom Packer 6 [93]	(v1.0, 635)	635	Petite	(v2.4, 625)	625
Custom Packer 7 [94]	(v1.0, 651)	651	RLPack	(v1.21, 645)	645
Custom Packer 8 [95]	(v1.0, 651)	651	*telock*	(v0.98, 595)	595
Custom Packer 9 [96]	(v1.0, 547)	547	Themida	(v2.4.5.0, 612)	612
Custom Packer 10 [97]	(v1.0, 655)	655	UPack	(v0.39, 664)	664
eXPressor	(v1.8.0.1, 668)	668	UPX	(v3.91, 579) (v3.95, 579)	1,158
ezip	(v1.0, 597)	597	WinRAR	(v5.60, 694)	694
FSG	(v2.0, 630)	630	WinZip	(v5.0, 689)	689
			YodaCryptor	(v1.2, 653)	653
Synthetic dataset			All packers		18,798

of the relatively unknown packer class to evade malware detection systems before these are updated.

For all 31 packers, default settings were used when packing the binaries. Packing failed in some cases because the binaries were too small to be packed, or their PE structure could not be modified. Table 4.2 summarizes the number of samples of each packer family and the version used. Github references are given for custom packers.

This dataset is quite representative as well, because the packers complexity classes range from I to III (except Armadillo that has the class IV [18]), and their packing techniques are quite different.

Finally, the fact that the two datasets have their own specific packers and share packers

with common families (but not necessarily common versions) is particularly interesting for our evaluation, because it would exhibit the behavior of our system when facing the arrival of known packers as well as new, unknown ones. We explain more in details how we take advantage of this intersection of datasets for our evaluation in Section 4.5.1.

4.4 Evaluation Metrics

Both extrinsic and intrinsic metrics are used to evaluate clusters.

4.4.1 Extrinsic Metrics

Our ground truth packed samples are labeled by their packer families. Each family comprises many variants (different versions in Table 4.1; this could also be different configurations or polymorphic instances). Our feed dataset has no absolute ground truth, since labeling tools do not agree on versions. Variants may be far apart wrt. *eps*, so we do not punish a clustering procedure splitting a family into different clusters, provided sub-clusters contain elements of the same family.

In this context, the *homogeneity score* indicates how much a cluster contains samples belonging to a single family (class). Let T be the ground truth classes, and C be the predicted clusters by the clustering algorithm, then the homogeneity score h is given by:

$$h = 1 - H(T|C)/H(T) \tag{4.4}$$

where $H(\cdot)$ is entropy and $H(\cdot|\cdot)$ conditional entropy, where $h \in [0, 1]$. Low values indicate low homogeneity. h does not punish dividing one class in smaller clusters, so high homogeneity is easy to achieve with a large number of clusters. h is 1 if every element is clustered into its own size-1 cluster, so we use this metric to evaluate our clustering *homogeneity* only, not its global extrinsic quality.

The Normalized Mutual Information (NMI) trades-off between homogeneity of clusters and their number. We use the Adjusted Mutual Information (AMI), an *adjusted-for-chance* version of NMI highly recommended in the clustering literature [98], defined by:

$$AMI(T, C) = \frac{I(C, T) - E[I(T, C)]}{\sqrt{H(C) * H(T)} - E[I(T, C)]} \tag{4.5}$$

where $I(\cdot)$ is the mutual information and $E[\cdot]$ is the expectation. $AMI \in [0, 1]$, higher

values indicate more *homogeneous clusters* and/or a number of clusters closer to the number of packer families in T. AMI is 1 when T and C are identical and 0 when any commonality is due to chance. Thus this metric considers the clusters homogeneity while punishing clustering with a large number of clusters, since such clustering has a high $H(C)$ (**RQ2.3**).

4.4.2 Intrinsic Metrics

We use DBCV (Density-Based Clustering Validation)[99], which can validate arbitrarily-shaped clusters. This metric computes the density within a cluster (*density sparseness*) and the density between clusters (*density separation*). For a clustering $C = \{C_i\}, 1 \leq i \leq l$:

$$DBCV(C) = \sum_{i=1}^{l} {}^{|C_i|}/_{|O|}\, V_c(C_i) \tag{4.6}$$

where $|O|$ is the number of samples, and $V_c(C_i)$ is the validity index of cluster $C_i, 1 \leq i \leq l$, defined as:

$$V_c(C_i) = \frac{\min\limits_{1 \leq j \leq l, j \neq i} (DSPC(C_i, C_j)) - DSC(C_i)}{\max\left(\min\limits_{1 \leq j \leq l, j \neq i} (DSPC(C_i, C_j)), DSC(C_i)\right)}$$

where $DSPC(C_i, C_j)$ is the density separation between clusters C_i and C_j, and $DSC(C_i)$ is the density sparseness of cluster C_i.

$DBCV(C) \in [-1, 1]$, higher values indicate a clustering with high density within clusters and/or low density between clusters.

4.5 Experimental Evaluation

This section starts with an overview of software and hardware implementations, then presents the experimental evaluation we performed on our offline and online phases, and the results obtained.

All experiments were performed in Python 3.6.8 on a Linux server with four 14-core processors at 2GHz with 128 GB of RAM.

We used the framework Radare2 [100] to emulate the unpacking code execution and get the trace of ASM mnemonic sequences. Execution is stopped when the ASM sequence length reaches 50 mnemonics; this length is a trade-off between the relevant information

(unpacking stub code) and cost of extraction. In [84], up to 30 mnemonics are statically extracted. We slightly extended this length to 50 to improve the relevant information quality. We give in the Appendix 3 some examples of traces that we obtained from emulating the execution of the 50 first instructions of packed binaries. The average time needed to extract the ASM sequence is 0.23 seconds per sample.

We computed the PE sections features from the PE header of the packed file using a C++ PE parser able to handle (packed) malware samples [77]. The average time needed to extract these features is 0.014 seconds per sample, as reported in the previous Chapter, in the Table 3.4b

We largely modified the online implementation [87] of the incremental DBSCAN to fit our methodology. In particular, our modifications include the composite pairewise distance metric we set in Section 4.1.3 as well as the set of optimizations (SRPs and triangle inequalities) we introduced in both Section 4.1.4.3 and pseudocode 2. For the batch version of DBSCAN and evaluations metrics, we used Scikit-learn [101].

4.5.1 Scenarii Definition

The two datasets share common packers, but also have their own (see Tables 4.1 and 4.2). So we designed two scenarii to exhibits the behavior of our system when facing the arrival of known packers as well as new, unknown ones:

MF/S: we use the Malware Feed as training set in the offline phase, then the Synthetic dataset as test in the online phase.

S/MF: we use the Synthetic dataset as training set in the offline phase, then the Malware Feed as test in the online phase.

4.5.2 Offline Phase

This phase supervises the creation of clusters with each training set, given the ground truth, in each scenario. The goal is to fine-tune *eps* according to the AMI score, the best found *eps* will then be used all along the online phase against the testing set, in each scenario.

In both scenarii, we set the value of $minPts$ to 3 (see Section 4.1.4.2). For *eps*, we tune it over $[0.001, 2]$ with 0.00025 increments.

In this phase we select the most effective feature category combinations based on the best AMI score that we obtain. The selection is done in practice by testing all the possible

Table 4.3 – Summary of offline phase results.

Training set	**AMI**	**h**	**# clusters**	**best *eps***	***minPts***
Malware feed	0.941	0.987	38	0.08	3
Synthetic	0.985	0.993	43	0.06	3

Table 4.4 – Distance comparisons.

	AMI		
	Gower	**Euclidean**	**Cosine**
Scenario			
MF/S	**0.941**	0.890	0.887
S/MF	**0.985**	0.974	0.974

non-repetitive category combinations, as shown in the previous Chapter, in Section 3.2. The most effective one we get is "sections" associated to the "unpacking stub mnemonic sequences". So for reasons of simplicity, we will report only the results we obtained from the combination of these two categories.

Table 4.3 summarizes for each training set the best results achieved wrt. AMI score. Homogeneity h and number of clusters are given to explain the AMI score and detail the obtained clustering. Note that the best *eps* found slightly differs between the two training sets, due to the bias caused by the different variety of packers in each training set. The resulting clusters and their contents are presented in the column labeled "offline phase" in Tables 4.7, 4.8 and 4.9.

Finally, we show in this phase the effectiveness of the Gower distance that we derived for packed binaries, by comparing it over simpler commonly used distances, namely: Cosine and Euclidean distances. To do that, we start by encoding the feature categories which have different data types (numerics for sections and string sequences for unpacking code sequences) in order to have a homogeneous data type for our final feature vectors, thanks to the One-Hot-Encoder function of Scikit-learn [102]. Then we apply separately Cosine and Euclidean distances on these final feature vectors across the same interval of *eps* with the same value of increments.

Table 4.4 reports the results obtained from the comparisons. We can see clearly that the Gower distance that we derived for our packed binaries outperforms other distances. Therefore, these results validate the practical effectiveness of our pairwise composite distance metric (**RQ2.1b**).

Table 4.5 – Impact of n_{rp} on the effectiveness and update time per sample.

K	Scenario					
	MF/S			S/MF		
	AMI	h # clusters	update time (s)	AMI	h # clusters	update time (s)
10^{-2}	0.933	0.960 82	1.097	0.934	0.981 95	1.066
10^{-1}	0.936	0.959 80	1.190	0.935	0.981 95	1.125
1	0.945	0.960 76	3.245	0.937	0.981 91	3.035
10	0.948	0.960 73	22.950	0.937	0.981 91	22.378
100	0.948	0.960 73	51.196	0.937	0.981 91	49.544

4.5.3 Online Phase

In this phase, we first study the impact of n_{rp} on the effectiveness and efficiency of our system (**RQ2.2**). Then, we evaluate the effectiveness and robustness of our solution (**RQ2.4**). Finally, we test our PCRS selection strategy and discuss how it optimizes the cost of post-clustering analysis tasks (**RQ2.5**).

4.5.3.1 Scattered Representative Points

Here, we study the impact of n_{rp} (see Section 4.1.4.1) on *effectiveness* and *efficiency* (update time) of our model. Thus, we vary K in $\left\lceil K\sqrt{N_c}\right\rceil$ and then select a K that provides effectiveness while keeping our solution quite fast. For both scenarii, we try K among $\{10^{-2}, 10^{-1}, 1, 10, 100\}$, in order to largely vary n_{rp}. With our dataset, when $K = 10^{-2}$, n_{rp} is 1 (the minimum possible) for all clusters; when $K = 100$, n_{rp} equals the number of samples (the maximum possible) for all clusters.

Table 4.5 presents the obtained results. The update time is the average time in seconds to update the clustering when a new sample arrives, without considering features extraction time.

Impact on effectiveness. In Table 4.5, we observe in both scenarii that the higher the K, the slightly higher the AMI score, until stabilizing at $K = 10$. The AMI score is controlled by the ratio between homogeneity and number of clusters. Homogeneity stays stable when K increases, but the number of clusters decreases then stabilizes, so the AMI score increases then stabilizes. Thus, reaching $K = 10$, the n_{rp} becomes sufficient for

our model to achieve its best effectiveness. When SRPs are too few to well represent the geometric shape of some or all clusters, the model does not always find the nearest cluster (see Figure 4.2). Thus, the new sample stays "unclassified" or contributes to creating a new cluster, instead of joining the nearest existing cluster.

Impact on efficiency. In Table 4.5, the more K increases the more update time increases (drastically). For $K = 10^{-2}$, average update time is around one second. For $K = 100$, it is around 50 seconds, because many more comparisons with SRPs are performed to find the nearest cluster. These results confirm the paramount importance of SRPs to optimize computation in the update process.

Optimal K selection strategy. The optimal K trades-off between effectiveness and efficiency. This work aims to be *highly effective* and *quite fast*, so we discard all K values leading to an update time above 1.5 seconds. We then select the K with the highest AMI score in the solutions left. $K = 10^{-1}$ appears as the optimal solution for both our scenarii, and is thus used in our next experiments.

4.5.3.2 Effectiveness and Robustness of SE-PAC

Here, we evaluate in more details the effectiveness of our **SE-PAC** (**S**elf-**E**volving **PA**cker **C**lassifier) system on the various update uses-cases (see Section 4.1.4.3). Then we study how this effectiveness evolves over time in order to gauge the robustness of the model. The values of eps, $minPts$, and K are selected as previously described.

Time-flow of incoming samples. We simulate the arrival of the test packers over several months. Each month, a number of specific and common packers (see Tables 4.1 and 4.2) appear in each scenario. The *specific packers* represent *new packer classes*, and the *common packers* represent *variants*. The specific packers arrive in a random order, one specific packer appearing each month. The experimental test period covers 17 months (17 specific packers) for MF/S and 15 months (15 specific packers) for S/MF. The samples of each specific packer are equally distributed from the arrival month of their packer till the end of experiment: in MF/S, 125/17 samples of Custom Packer 1 appear in month 1, the rest is then equally distributed over the 16 months left; 22/16 samples of Custom Packer 2 appear in month 2, the rest is distributed over the 15 months left. The samples of common packers are equally distributed through the whole experimental period: in MF/S, packers ASPack and UPX appear monthly with a quantity of 633/17 and 1158/17 respectively.

Table 4.6 – Summary of final results.

Scenario	AMI	h	# clusters	DBCV
MF/S	0.936	0.959	80	0.285
S/MF	0.935	0.981	95	0.575

Table 4.6 summarizes the final results regarding AMI, homogeneity, number of clusters, and DBCV obtained after the whole update process. Figure 4.5 to 4.8 present the monthly evolution of those metrics. We now explain and discuss these results.

AMI, homogeneity, and number of clusters evolution. In MF/S, AMI stays high and quite constant over time. In S/MF, it slightly decreases over time (see Figure 4.5) because S/MF forms a higher number of clusters (which the AMI metric punishes, see Section 4.4.1) than MF/S (see Figure 4.7). Indeed, the scenario S/MF is more likely to classify an incoming packer to a new cluster, because the *eps* range (= 0.06) is smaller, so the incoming samples of some test packers may not be grouped under one cluster but form new additonal clusters instead. Therefore, higher homogeneity in S/MF is maintained since different packer families are more likely not to mix (see Figure 4.6).

DBCV evolution. The significant decrease of the DBCV score was expected (see Figure 4.8), since SE-PAC tends to find for some packers multiple but very close clusters, which thus strongly decreases their density separation. Being multiple, they strongly impact the global mean of the intrinsic clustering quality DBCV(C). This score is worse in MF/S than in S/MF, because the best *eps* (= 0.08) is higher in MF/S, hence the model tends to have lower density within clusters and higher density between clusters.

Tables 4.7, 4.8 and 4.9 present the final results obtained by SE-PAC after the offline and online phases, considering the content of clusters found and the DBCV score, for each packer family. We explain and discuss the results of this table next.

Misclassifications. They are marked in *italics*: e.g., in MF/S, one sample of Custom packer 2 is wrongly classified with both samples of Custom packer 10 and one sample of AutoIt, in cluster 47. Since *eps* tuning trades off between correct classifications and number of clusters, it may lead to misclassifications, which are reported in the offline phase. We chose the best AMI score. Experiment shows that scenario S/MF generates a smaller *eps* in the offline phase, resulting in less misclassifications during offline and online phases.

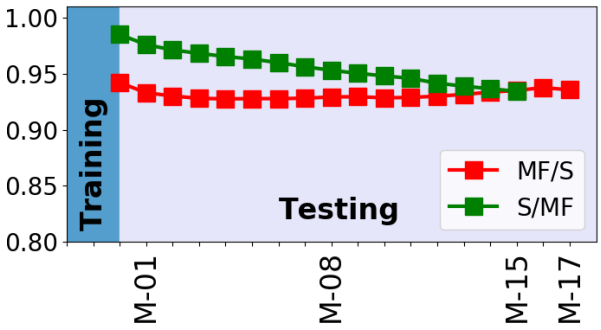


Figure 4.5 – AMI evolution.

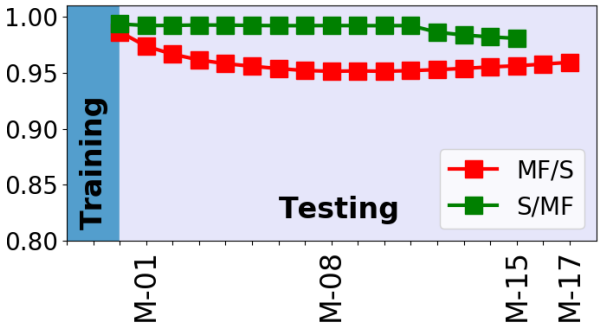


Figure 4.6 – Homogeneity evolution.

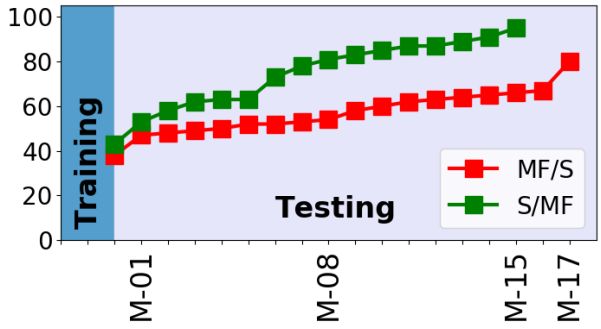


Figure 4.7 – # clusters evolution.

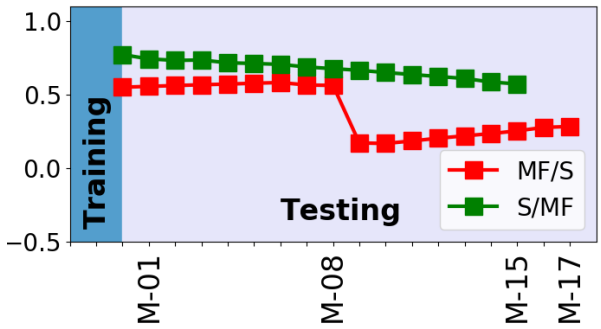


Figure 4.8 – DBCV evolution.

Table 4.7 – Cluster contents and DBCV score for each packer family, in both scenarii, after both offline and online phases, Part-1. "Not learned" in the offline phase column indicates training does not include the packer, so the packer is considered *specific*, thus new, when used as test packer. "No score" means there is no cluster to evaluate. Cluster ID "-1" means noise. Results in *italics* are misclassifications.

Packers	Content of clusters: (Cluster ID – # samples)				DBCV score per cluster: (Cluster ID – score)	
	Scenario MF/S		Scenario S/MF		Scenario MF/S	Scenario S/MF
	Offline phase	Online phase	Offline phase	Online phase	Online phase	Online phase
ActiveMARK	(-1 – 3)	(-1 – 3)	Not learned	(-1 – 3)	No score	No score
ASProtect	(4 – 173), (-1 – 2)	(4 – 173), (-1 – 2)	Not learned	(62 – 82), (73 – 9), (81 – 81), (-1 – 3)	(4 – 0.6)	(62 —0.1), (73 —0.1), (81 —0.1)
AutoIt	*(25 – 1)*, (36 – 1,027), (37 – 4), (-1 – 4)	*(25 – 1)*, *(36 – 1,027)*, (37 – 4), *(47 – 1)*, (-1 – 3)	Not learned	*(42 – 1,027)*, (90 – 3), (-1 – 6)	(25 – 0.6), (36 – 0.8), (37 – 0.2), (47 —0.2)	(42 – 0.6), (90 – 1)
ExeStealth	(-1 – 1)	(-1 – 1)	Not learned	(-1 – 1)	No score	No score
FishPE	*(20 – 3)*	*(20 – 3)*	Not learned	*(37 – 3)*	(20 —1)	(37 – 0.7)
InnoSetup	(32 – 870), (33 – 38), (34 – 8), (35 – 3), (-1 – 15)	(32 – 870), (33 – 38), (34 – 8), (35 – 3), (-1 – 15)	Not learned	(56 – 610), (57 – 260), (61 – 38), (80 – 8), (84 – 3), (-1 – 15)	(32 – 0.4), (33 – 1), (34 – 0.8), (35 – 1)	(56 – 0.1), (57 – 0.9), (61 – 1), (80 – 0.7), (84 – 1)
InstallShield	*(7 – 4)*	*(7 – 4)*	Not learned	(74 – 4)	(7 – 0.2)	(74 – 0.9)
MoleBox	(8 – 7)	(8 – 7)	Not learned	(85 – 7)	(8 – 0.8)	(85 – 0.9)
NsPacK	(9 – 9), (10 – 5)	(9 – 9), (10 – 5)	Not learned	(82 – 9), (92 – 5)	(9 – 0.8), (10 – 0.7)	(82 – 0.8), (92 – 0.7)
NSIS	*(7 – 3)*, (29 – 12), (30 – 1,730), (31 – 130), (-1 – 41)	*(7 – 3)*, (29 – 12), (30 – 1,730), (31 – 130), (-1 – 41)	Not learned	(65 – 6), (66 – 205), (67 – 112), (68 – 39), (69 – 52), (70 – 399), (71 – 17), (72 – 10), (76 – 22), (77 – 993), (78 – 8), (79 – 6), (-1 – 47)	(7 – 0.2), (29 – 0.4), (30 – 0.2), (31 – 0.5)	(65 – 0.9), (66 —1), (67 – 0), (68 —1), (69 – 0.2), (70 – 0.2), (71 – 0.5), (72 – 0.6), (76 – 0.5), (77 —1), (78 – 1), (79 – 0.7)
PEPACK	(17 – 5), (-1 – 1)	(17 – 5), (-1 – 1)	Not learned	(88 – 5), (-1 – 1)	(17 – 0.7)	(88 – 0.7)
PESpin	(27 – 3), (-1 – 3)	(27 – 3), (-1 – 3)	Not learned	(94 – 3), (-1 – 3)	(27 – 0.9)	(94 – 0.958)
PCGuard	(26 – 3), (-1 – 1)	(26 – 3), (-1 – 1)	Not learned	(93 – 3), (-1 – 1)	(26 – 1)	(93 – 1.0)
Shrinker	(28 – 3)	(28 – 3)	Not learned	(60 – 3)	(28 – 0.6)	(60 – 0.8)
Wise	*(7 – 8)*	*(7 – 8)*	Not learned	(83 – 8)	(7 – 0.2)	(83 – 0.6)

Table 4.8 – Extension of Table 4.7

Packers	Content of clusters: (Cluster ID – # samples)				DBCV score per cluster: (Cluster ID – score)	
	Scenario MF/S		Scenario S/MF		Scenario MF/S	Scenario S/MF
	Offline phase	Online phase	Offline phase	Online phase	Online phase	Online phase
Armadillo	Not learned	(38 – 627), (-1 – 1)	(0 – 628)	(0 – 628)	(38 – 0.5)	(0 – 0.6)
Custom Packer 1	Not learned	(65 – 123), (-1 – 2)	(5 – 123), (-1 – 2)	(5 – 123), (-1 – 2)	(65 – 1.0)	(5 – 1.0)
Custom Packer 2	Not learned	*(47 – 1)*, (54 – 16), (77 – 5)	(6 – 22)	(6 – 22)	(47 —0.2), (54 —0.5), (77 —0.3)	(6 – 0.9)
Custom Packer 3	Not learned	(48 – 277)	(7 – 277)	(7 – 277)	(48 – 1)	(7 – 0.9)
Custom Packer 4	Not learned	(53 – 646), (-1 – 2)	(8 – 648)	(8 – 648)	(53 – 0.3)	(8 – 0.5)
Custom Packer 5	Not learned	(62 – 651), (-1 – 4)	(9 – 654), (-1 – 1)	(9 – 654), (-1 – 1)	(62 – 0.6)	(9 – 0.6)
Custom Packer 6	Not learned	(58 – 635)	(10 – 635)	(10 – 635)	(58 – 0.8)	(10 – 0.8)
Custom Packer 7	Not learned	(49 – 646), (78 – 3), (-1 – 2)	(11 – 648), (12 – 3)	(11 – 648), (12 – 3)	(49 – 0.7), (78 – 1)	(11 – 0.7), (12 – 0.9)
Custom Packer 8	Not learned	(66 – 648), (-1 – 3)	(13 – 648), (-1 – 3)	(13 – 648), (-1 – 3)	(66 – 0.9)	(13 – 0.9)
Custom Packer 9	Not learned	(55 – 547)	(14 – 547)	(14 – 547)	(55 – 0.9)	(14 – 0.9)
Custom Packer 10	Not learned	*(47 – 655)*	(15 – 655)	(15 – 655)	(47 —0.2)	(15 – 1)
ezip	Not learned	(52 – 597)	(17 – 597)	(17 – 597)	(52 – 0.8)	(17 – 0.8)
mPress	Not learned	(64 – 593)	(22 – 593)	(22 – 593)	(64 – 1)	(22 – 0.9)
PELock	Not learned	(50 – 617), (79 – 3), (-1 – 1)	(30 – 617), (31 – 3), (-1 – 1)	(30 – 617), (31 – 3), (-1 – 1)	(50 – 0.8), (79 – 0.7)	(30 – 0.8), (31 – 0.8)
PENinja	Not learned	(63 – 661), (-1 – 5)	(32 – 663), (-1 – 3)	(32 – 663), (-1 – 3)	(63 – 0.6)	(32 – 0.6)
telock	Not learned	(61 – 595)	(35 – 595)	(35 – 595)	(61 – 0.5)	(35 – 0.5)
YodaCryptor	Not learned	(67 – 27), (68 – 259), (69 – 30), (70 – 79), (71 – 55), (72 – 59), (73 – 79), (74 – 4), (75 – 3), (76 – 22), (-1 – 36)	(2 – 427), (3 – 27), (4 – 3), (-1 – 196)	(2 – 427), (3 – 27), (4 – 3), (-1 – 196)	(67 – 1), (68 —0.2), (69 —0.2), (70 —0.3), (71 —0.3), (72 —0.2), (73 —0.2), (74 – 0.2), (75 – 0.2), (76 —0.2)	(2 —0.1), (3 – 1), (4 – 0.1)

Table 4.9 – Extension of Table 4.7

Packers	Content of clusters: (Cluster ID – # samples)				DBCV score per cluster: (Cluster ID – score)	
	Scenario MF/S		Scenario S/MF		Scenario MF/S	Scenario S/MF
	Offline phase	Online phase	Offline phase	Online phase	Online phase	Online phase
ASPack	(0 – 5,702), (1 – 9), (2 – 117), (3 – 3), (-1 – 2)	(0 – 5,702), (1 – 9), (2 – 748), (3 – 3), (-1 – 4)	(1 – 633)	(1 – 750), (43 – 4), (44 – 9), (45 – 5,282), (46 – 416), (91 – 3), (-1 – 2)	(0 – 0.5), (1 – 0.4), (2 – 0.7), (3 – 0.8)	(1 – 0.4), (43 – 0.9), (44 – 0.5), (45 – 0.4), (46 – 0.4), (91 – 0.8)
eXPressor	(5 – 11), (-1 – 1)	(5 – 11), (39 – 668), (-1 – 1)	(16 – 668)	(16 – 668), (58 – 11), (-1 – 1)	(5 – 1), (39 – 1)	(16 – 1), (58 – 1)
FSG	*(6 – 13)*	*(6 – 624)*, *(25 – 1)*, (40 – 7), (56 – 4), (-1 – 7)	(18 – 611), (19 – 7), (20 – 4), (-1 – 8)	(18 – 624), (19 – 7), (20 – 4), (-1 – 8)	(6 – 0.3), (25 – 0.6), (40 – 0.9), (56 – 0.9)	(18 – 0.1), (19 – 0.9), (20 – 0.5)
MEW	*(6 – 217)*	*(6 – 851)*	(21 – 634)	(21 – 851)	(6 – 0.3)	(21 – 0.9)
NeoLite	(-1 – 2)	(41 – 614), (59 – 4), (-1 – 1)	(23 – 612), (24 – 4), (-1 – 1)	(23 – 614), (24 – 4), (-1 – 1)	(41 – 0.6), (59 – 0.9)	(23 – 0.7), (24 – 0.9)
Packman	*(11 – 23)*	*(11 – 661)*, (-1 – 2)	(25 – 640)	(25 – 663)	(11 —0.1)	(25 – 0.1)
PECompact	(12 – 1,016), (13 – 16), (14 – 140), (15 – 16), (16 – 4), (-1 – 34)	(12 – 1,475), (13 – 16), (14 – 140), (15 – 16), (16 – 4), *(20 – 4)*, (42 – 43), (60 – 5), (-1 – 193)	(26 – 407), (27 – 40), (28 – 3), (29 – 8), (-1 – 212)	(26 – 486), (27 – 41), (28 – 3), (29 – 38), (47 – 838), (48 – 17), (49 – 140), (53 – 16), (63 – 16), (75 – 6), (86 – 24), (87 – 4), (89 – 7), (-1 – 260)	(12 —0.7), (13 – 0.7), (14 – 1), (15 – 1), (16 – 1), (20 —1), (42 – 0.9), (60 – 0.2)	(26 —0.9), (27 – 0.9), (28 – 1), (29 – 0.3), (47 – 1), (48 – 1), (49 – 1), (53 – 0.6), (63 – 1), (75 – 1), (86 —0.7), (87 – 1), (89 – 1)
Petite	(18 – 6), (-1 – 3)	(18 – 6), (43 – 625), (-1 – 3)	(33 – 625)	(33 – 625), (64 – 6), (-1 – 3)	(18 – 0.7), (43 – 9)	(33 – 0.9), (64 – 0.8)
RLPack	(-1 – 1)	(44 – 645), (-1 – 1)	(34 – 645)	(34 – 645), (-1 – 1)	(44 – 0.9)	(34 – 0.9)
Themida	(19 – 11), (-1 – 1)	(19 – 11), (45 – 612), (-1 – 1)	(36 – 612)	(36 – 612), (59 – 11), (-1 – 1)	(19 – 0.9), (45 – 0.9)	(36 – 0.9), (59 – 0.9)
UPack	(22 – 19), (23 – 21), (-1 – 1)	(22 – 19), (23 – 21), *(24 – 648)*, (46 – 16), (-1 – 1)	(40 – 648), (41 – 16)	(40 – 669), (41 – 16), (55 – 19), (-1 – 1)	(23 – 1), (24 – 1), (46 – 0.9)	(40 – 1), (41 – 0.9), (55 – 1)
UPX	*(11 – 5)*, *(20 – 4,487)*, (21 – 55), (-1 – 2)	*(11 – 5)*, *(20 – 5,066)*, (21 – 55), (51 – 6), (57 – 573), (-1 – 2)	(37 – 1,152), (38 – 6)	*(37 – 5,638)*, (38 – 6), (50 – 4), (51 – 55), (-1 – 4)	(11 —0.1), (20 —1), (21 – 0.8), (51 – 0.7), (57 —1)	(37 – 0.7), (38 – 0.6), (50 – 0.8), (51 – 0.8)
WinRAR	(24 – 30), (-1 – 5)	*(24 – 30)*, *(36 – 694)*, (-1 – 5)	(39 – 694)	(39 – 694), (54 – 30), (-1 – 5)	(22 – 1), (23 – 1), (36 – 0.8)	(39 – 1), (54 – 0.7)
WinZip	*(25 – 66)*	*(25 – 66)*, *(36 – 689)*	(42 – 689)	*(42 – 689)*, (52 – 66)	(25 – 0.6), (36 – 0.8)	(42 – 0.6), (52 – 0.6)

Seq 1:	pushal, call, pop, sub, mov, lea, mov, lodsb, dec, dec, nop, jmp, ror, dec, stc, jmp, sub, jmp, nop,rol, sub, ror, rol, nop, jmp, jmp, ror, nop, jmp, stosb, loop
Seq 2:	pushal, call, pop, sub, mov, lea, mov, lodsb, rol, jmp, dec, clc, jmp, rol, nop,ror, nop, jmp, jmp, add, dec, jmp, clc, jmp, add, xor, dec, jmp, add, stosb, loop
Seq 3:	pushal, call, pop, sub, mov, lea, mov, lodsb, jmp, stc, sub, ror, ror, jmp, ror, sub, ror, ror, jmp, jmp, clc, stc, add, rol, dec, dec, xor, nop, xor, stosb, loop

Figure 4.9 – Instruction substitution obfuscation technique used by YodaCryptor v1.2 packer to generate polymorphic instances of the unpacking stub code. Using the framework Radare2, the three sequences above were generated by emulating the execution of the unpacking stub code of three different binaries packed by YodaCryptor v1.2. The part in blue shows the instruction substitution obfuscation technique.

Specific packers. In both scenarii, despite some misclassifications, for most specific packers, including the custom ones, new clusters are created. This shows our system is able to identify new packers.

Common packers. They either joined their respective existing packer family clusters or formed new ones, or both of them. For example, in MF/S all samples of ASPack joined their family cluster 2, while all samples of eXPressor formed the new cluster 39. For FSG, many samples joined existing cluster 6, while the others formed new clusters 25, 40 and 56. Some common test packers did not join their existing family clusters mainly because their *versions* differ greatly (wrt. *eps*) from the one used in the training set. For example, in MF/S samples of Themida version 2.4.5.0 (see Table 4.2) used as test formed the new cluster 45, instead of joining their packer family cluster 19 that hosts a different version of the same packer. The same happened to packers eXPressor, Petite, WinRAR and WinZip.

Finally, multiple sub-clusters were formed for some packer families, because of: (*i*) Different *versions*: e.g., see UPX, NSIS and InnoSetup in Table 4.1; (*ii*) Obfuscation: e.g., YodaCryptor generates polymorphic instances of its unpacking code by using the *instruction substitution* technique that *tampers* quite arbitrarily the mnemonic sequence without affecting its behavior (see Figure 4.9), which makes the grouping of ASM sequences harder, causing additional clusters, and noise.

4.5.4 PCRS Selection

This experiment evaluates how our PCRS selection strategy (see Section 4.2) performs on the previously obtained clusters.

The total number of samples inside all these clusters is 34,613 in MF/S and 34,304 in

Table 4.10 – # of PCRS.

Radius r	#Samples before selection		# of PCRS		% Decrease	
	MF/S	**S/MF**	**MF/S**	**S/MF**	**MF/S**	**S/MF**
1 * eps			220	257	99.4%	99.3%
1.5 * eps	34613	34304	105	102	99.7%	99.7%
2 * eps			96	94	99.7%	99.7%

Table 4.11 – Overview of some clusters after PCRS selection (MF/S).

Cluster ID	(PCRS rank – density marker)
12	(1 – 57%), (2 – 32%), (3 – 4%),(4 – 3%), (5 – 1%), (6 – 1%), (7 – 0.4%), (8 – 0.3%), (9 – 0.3%), (10 – 0.1%) , (11 – 0.1%), (12 – 0.1%), (13 – 0.1%), (14 – 0.1%) , (15 – 0.1%), (16 – 0.1%) , (17 – 0.1%)
78	(1 – 50%), (2 – 21%) , (3 – 14%), (4 – 7%), (5 – 7%)
66	(1 – 100%)

S/MF. Selection radius r is given by $r = \alpha * eps$, where eps is the best eps found for each scenario, and α is evaluated over $\{1, 1.5, 2\}$.

Table 4.10 presents the results of our PCRS selection strategy on all the previously found clusters, with different radii r. For $r = 1 * eps$, the number of PCRS is 0.6% (220/34,613) of the total number of samples in clusters in MF/S, thus a percentage decrease of 99.4% (or one PCRS for 157 samples), and 0.7% (257/34,304) in S/MF, thus a percentage decrease of 99.3% (or one PCRS for 133 samples). This ratio tends to stabilize when the selection radius is enlarged, and the number of PCRS *converges* towards the number of clusters found.

Table 4.11 shows the results in MF/S of our PCRS selection strategy on three individual clusters, with $r = 1 * eps$. The PCRS are ranked by density marker. Clusters 12, 78 and 66 correspond respectively to PECompact and custom packers 7 and 8. The difference in number and distribution of the PCRS logically lies on how the samples of a cluster are scattered in the hyperplan. For example, cluster 12 needs 17 PCRS, because PECompact uses a random key to generate polymorphic instances of the unpacking stub.

4.6 Discussion

In this section, we first give further insights on our results. Next, we discuss the possibility of human interaction when using our approach over a large time frame. Then, we examine experimental properties influencing our results validity. Finally, we identify the limitations and propose improvements for future work.

4.6.1 Findings and Insights

Our results show that our incremental system stays valid against packers evolution over time (**RQ2.4**). In particular, keeping high homogeneity at the cost of a few additional clusters is favored (**RQ2.3**). In that sense, it appears that the S/MF scenario is the most suitable; Reducing *eps* would reduce the cost of supervisor intervention and ensure longer user protection.

Furthermore, in spite of a sharp decrease of the intrinsic quality of our model (see Figure 4.8), extrinsic quality remains effective (see Figure 4.5). Indeed, our update policy does not autonomously merge close clusters, which prevents potential upcoming misclassifications, at the cost of a few additional clusters.

Custom packers were accurately classified (see Table 4.8). This validates the effectiveness of the chosen features to represent packers. This also hints that these packers are simply inspired from well-known packers. Moreover, the difficulty to group some samples gives insights on whether the packer was originally developed for obfuscation purposes, like YodaCryptor or PECompact, or for mere compression and encoding tasks like WinRAR and WinZip. This could be further exploited by an analyst for *threat intelligence*.

Our PCRS selection strategy can reduce post-clustering complexity by 99% (see Section 4.5.4), a very important optimization for *concrete* tasks like packer analysis and/or unpacking. For instance, these optimizations could drastically reduce in practice the cost of manual reverse-engineering or automated packer analysis in well-instrumented sandboxes [103]. Furthermore, the density marker rank helps the analyst establish priorities on the most spread packer variants (**RQ2.5**).

Therefore, *with minimal cost*, the *packer classification system and unpacking can be updated*, which means updating the:

- *Ground truth labels*. When the cluster is modified, the prediction of labels is already performed by affecting the existing cluster label (e.g., UPX) to the new samples

that joined that cluster. When the cluster is new, the manual and/or automated analyses of the PCRS of the new packer would allow to generate a new label.

- *Unpacking systems.* The results of manual and/or automated analyses would give the malware analyst a better understanding of the packer in question, hence lets him on the one hand develop *specialized unpackers* to quickly and precisely unpack binaries packed with widespread packers, and on the other hand, develop effective heuristics to *generically* unpack binaries packed with packers spread in few quantities (see Figure 1.1).

In the context of the fast-paced evolution of packers, it might be costly to update for each new packer found its specialized unpacker. However, the post clustering updates do not necessarily aim to update for each new packer found its specialized unpacker, but rather to balance between updating the specialized unpackers and generic unpacking heuristics, wrt. how spread is the new packer found. Because we remind that the end-goal is to provide effective and efficient unpacking solutions in the context of an effective and efficient malware analysis toolchain (see Figure 1.1).

For better illustration, let's suppose that SE-PAC finds 3 new clusters which would represent 3 new packers A, B and C spread in the wild with a frequency of 10K samples per week, 10 samples per month, and 3 samples per month, respectively. So for such an example, to provide effective and efficient unpacking solutions for an effective and efficient malware analysis toolchain, it would be worthy to update a specialized unpacker for the packer A, in contrast to the packers B and C for which it would more efficient to update the generic unpacking heuristics such to include these two packers.

4.6.2 When and How to Retrain?

After a large amount of incremental updates, a human intervention may be useful, hence the question of when and how to retrain.

One approach is to exploit the extrinsic metrics that use the available ground truth to detect misclassifications. We distinguish two cases: (*i*) *misclassification of known packers*, where the error can be detected (e.g., the mix between WinZip and WinRAR packers in MF/S); (*ii*) *misclassification of unknown packers*, where the error is not detectable as long as the samples inside the clusters are not further analyzed and labeled.

Another approach relies on intrinsic metrics, like DBCV, that can indicate that some packers are very close, and thus prevent potential upcoming misclassifications.

These metrics would help to trigger an alarm for retraining. This retraining includes readjusting *eps*, reclustering from scratch, then continuing the incremental clustering.

4.6.3 Threats to Validity

Three elements of biases may have been introduced in our experiments: the training dataset, evaluation metric, and ground truth granularity. As shown in Section 4.5.2, different datasets output slightly different *eps*. In practice, we expect this radius *to stabilize* when the training datasets are more varied. Moreover, the selection of evaluation metrics influences naturally the view on the results. In this work, our aim is for most homogeneous clusters with a number of clusters that is as close as possible to the number of packers at the family ground truth granularity. Based on the argumentation on extrinsic metrics given in Section 4.4.1, we believe that our choices are valid to obtain accurate clusters of packer families. In general, the classification end-goals is the key to select those choices.

Our clustering updates do not autonomously merge or split clusters. While this provides our system a high time-resilience by prohibiting autonomous merging of close different packers, the number of clusters could quickly become larger if the number of variants of the incoming packed samples is large. Future work would consider including autonomous merging and splitting in our update policy. Such operations can be done by autonomously relabeling the clusters involved: merging would give the same ID to clusters that become very close to each other, while splitting would add new IDs for clusters resulting from the splitting of former clusters that became widely scattered.

Noise points do not impact our approach since they are not discarded but kept to be clustered or left unclassified. In practice, noise may represent hard-to-group samples due to obfuscation, or rare packers. Post-clustering analysis could reveal their exact nature.

4.6.4 Limitations and Future Work

While the tapered levenshtein distance we applied on our ASM sequences is able to group light polymorphic codes like those used by YodaCryptor, it remains fragile against *highly* polymorphic engines that generate very different variants of the same unpacking stub code. If these variants are few, the number of clusters generated remains acceptable; otherwise our clustering system would produce a very high number of clusters. Future work should consider the extraction of more semantic features and/or code-recompilation in order to better mitigate high and complex polymorphism.

In addition, future work may pay more attention to examine the case of repacked malware. Indeed, the overlapping of multiple packers in sequence on the outer layer of malware is not necessarily uniform. That is, the last packer may not completely overlay the outer layer of repacked malware, thus previous packer layers may still appear. Therefore, our features could be impacted by multiple packers. So it would be interesting to pay more attention on how such binaries impact the prediction of our classifier.

Finally, while the optimization techniques we adopted in the incremental update process improved significantly the update time performance of our incremental DBSCAN algorithm (as shown in Table 4.5), we believe that the scalability of SE-PAC can be further improved by relying on other engineering choices, namely the implementations of PE feature extraction and incremental DBSCAN algorithm. For the latter, relying on other engineering choices would mean resorting to lower-level languages and paralleling the architecture of the incremental DBSCAN.

4.7 Conclusion

This chapter presented SE-PAC, a *new self-evolving packer classifier* that deals with the issue of rapid evolution of packers. We derived a composite pairwise distance metric that is constructed from the combination of different types of packer features. We derived an incremental clustering approach able to identify both (variants of) known packers (families) and new ones, as well as automatically and efficiently update the clusters.

We evaluated our solution on two datasets: malware feed, and synthetic. The results showed that our classifier is effective and robust in identifying both known and new packer families. Indeed, our approach constantly enhances, integrates, adapts and evolves packer knowledge, making our classifier effective for longer times.

Moreover, we proposed a new post-clustering strategy that selects a subset of relevant samples from each cluster found, to optimize the cost of post-clustering processing.

We thus believe our work can help security companies, researchers and analysts to effectively, efficiently, and continually update their packing classification systems, specialized unpackers, and generic unpacking heuristics to ensure a better continuity of security for users over time.

Chapter 5

Conclusion and Future Work

This conclusion to the whole thesis starts by briefly recalling in Section 5.1 the thesis context and our objectives. It then summarizes in Section 5.2 the contributions we bring to the literature wrt. packing detection and classification problems, and how they fulfill to the thesis objectives. Finally, Section 5.3 starts by presenting possible improvements that can solidify directly our works, then discusses to what extent this thesis solves the malware packing problem in general and offers accordingly perspectives for future work.

5.1 Context and Objectives

Packing provides malware authors with an effective weapon to hinder static analysis and/or detection of their malicious codes from antiviruses, because the packed binary must be either unpacked or dynamically analyzed. Therefore, detecting, classifying, then unpacking a given packed sample is fundamental to be able to verify whether it is malicious or benign.

This thesis focuses on the packing detection and classification stages. It aims at providing effective, efficient, and robust packing detection and classification solutions to be practical parts of the malware analysis chain of an antivirus.

5.2 Contributions

In line with the thesis objectives, we add two contributions to the literature:

★ *Our first contribution* is presented in Chapter 3 where we introduce a study whose goal is to understand the impact of ground truth generation, ML algorithm selection, and feature selection on the effectiveness, efficiency and robustness of supervised ML-based packing detection and classification systems, following the example of works on empirical testing of ML malware analysis including [3]. Our findings are:

- We find that the size of the ground truth is more relevant than its quality for training supervised ML-based packing detection and classification algorithms, to perform more *effectively* and more *robustly* in production against real samples. In particular, the results of robustness assessment we reported in Section 3.5.3 show that k-fold cross-validation method is not suitable for fields like malware as well as packing detection and classification, where new samples and packing techniques appear constantly and rapidly in the wild. This constant and rapid evolution of the malware and packing ecosystems contribute to explaining the findings of [3], on the reasons why ML algorithms can perform well in in-the-lab scenarios and badly in in-the-wild scenarios.
- Furthermore, we show that selecting features and optimizing the hyperparameters of ML algorithms can greatly optimize the *efficiency* of our solutions. Indeed, the results we reported in Section 3.5.2 show that a minor decrease in effectiveness can reduce the detection and/or classification time per sample by up to 44 times.
- Finally, the results of the retraining cost analysis we presented in Section 3.5.4 show that simple algorithms with less features can be more *efficient* to use compared to complex algorithms with more features.

Despite the regular and efficient retraining we offer for our models, these retraining remain particularly limited for supervised packer classifiers. Indeed, the latter would still be unable to identify new packer families that appear in the period of time occurring between each two retraining, because of their theoretical inability to find new classes. Therefore, this theoretical limitation restricts specifically the *robustness* objective of our packing classifier solutions against the rapid evolution of packers over time. Our second contribution called "SE-PAC" arises from this situation.

★ *Our second contribution* is presented in Chapter 4 where we propose, design, and implement SE-PAC, a new Self-Evolving PAcker Classifier framework that relies on incremental clustering in a semi-supervised fashion, in order to cope with the fast-paced evolution of packers. More precisely:

- Our self-evolving technique predicts incoming packers by assigning them to the most likely clusters, and relies on these predictions to automatically update clusters, reshaping them and/or creating new ones. Therefore, SE-PAC continuously learns from incoming packers, constantly enhances, integrates, evolves, and adapts its clustering to packers evolution over time. The results

presented in Section 4.5.3.2 show that SE-PAC achieves the *robustness* objective by correctly identifying both known and new packer families over time, thus coping with the evolution of packers over time.

- We show how to combine different types of packer features in the construction of a composite pairwise distance metric. The results presented in Section 4.5.2, in Table 4.4, show that our composite distance metric outperforms simple distances.
- We derive an incremental clustering methodology which establishes a good trade-off between *effectiveness* and *efficiency*. The results we presented in Section 4.5.3.1, in Table 4.5, show that we reduce the update time per sample by 44 times on average.
- Finally, we propose a new post-clustering selection strategy which extracts a reduced subset of relevant samples from each cluster found, in order to optimize the cost of post-clustering packer processing. The results we presented in Section 4.5.4, in Table 4.10, show that our strategy deceases the number of samples by 99% on average.

We want to highlight that for both contributions, we supported our findings by realistic experiments, thanks to Cisco for having provided us the malware feed dataset. Furthermore, our two contributions can be generalized and applied to other problems, like other malware obfuscation techniques.

These two contributions led to the publication of two papers: [1] is the first journal paper we published, describing our first thesis contribution, and [2] is the second conference paper we published, describing our second thesis contribution.

Morever, we developed two tools: PE-PAC which implements the solutions we proposed in our first contribution, and SE-PAC which implements the solutions we proposed in our second contribution.

Through this thesis, we believe that our work will help to conceive and implement more *effective*, *efficient*, and *robust* packing detection and classification systems for malware analysis chains of antiviruses, to ensure a better security for users.

5.3 Future Work

Like any human work, some aspects of this thesis are still incomplete. In this section, we start by presenting possible improvements that can solidify directly our works. Then we step back and look from a broader view at the malware packing problem. We discuss to what extent this thesis solves the problem of malware packing in general, and identify potential research directions that can be tackled in the future.

Regarding our first contribution, a more comprehensive study could cover the following points:

- *Ground truth.* In addition to the signature-based packing detection and classification techniques used in Section 3.3, new ones could be used to improve the labeling of the ground truth. Indeed, despite the fact these signature-based tools have a widespread usage among security researchers and analysts, the techniques were many times in disagreement on the labeling of a sample, whether when it came to detect the packing (i.e., packed or non-packed) or classify the packer where for some cases the three techniques labeled the same sample with up to six different packing families. Possible perspectives for future would thus be to deploy unpacking as a next stage to contribute in constructing the ground truth labels. Moreover, a hierarchy of confidence values based on the reliability of the technique employed could be used to rate the packing detection and classification labels, i.e weighted voting, in order to better figure out how to solve conflicting labeling in a more advanced way than just the consensus/non-consensus paradigm used in our study.
- *Repacked malware.* The overlapping of multiple packers in sequence on the outer layer of malware is not necessarily uniform. In particular, the last packer may not completely overlay the outer layer of repacked malware, thus previous packer layers may still appear. We presume that this multiform overlapping of multiple packers on the outer layer of malware is one of the main reasons that provoked conflicting labeling when generating the ground truth. In our study, we removed from the ground truth many of these conflicting labeling which did not fit neither to 3CONS nor 1CONS. However, since these conflicting samples are likely repacked malware, hence they constitute a threat to the security of the user. While some works have already tackled this topic [82], we advise that future work examine more closely the morphology of repacked malware.

- *Feature selection.* The way features have been divided and grouped for selection is based on their themes (entropies, metadata, etc.) as described in Section 3.2. However, dividing the groups by theme means testing together features with possibly very different extraction costs and effectiveness. Future work should investigate alternative feature groups.

Regarding our second contribution, improvements could cover the following points:

- *Ground truth, repacked malware, and feature selection.* Our second contribution relies on many aspects of the first contribution, so improvements regarding the ground truth labels, repacked malware, and feature selection implies improvements of SE-PAC. Indeed, although SE-PAC works in an unsupervised fashion in the online phase, the offline phase remains supervised and thus still highly depends on the way the ground truth has been constructed. Furthermore, better combinations in dividing features and selecting them would naturally ease grouping packers in clusters. Finally, it would be interesting to pay more attention on how repacked malware would impact the predictions of SE-PAC, and accordingly how a hierarchy of prediction confidences can be associated.
- *Efficiency.* Our optimization techniques described in Section 4.1.4.1 and 4.1.4.3 reached an interesting reduction factor (44 on average) in update time performance. However, this reduction factor can be further improved by paralleling the architecture of the incremental DBSCAN and resorting to a lower-level implementation of the update function, which is actually implemented in Python 3.6.8. Such an improvement would allow SE-PAC to scale more efficiently to large amounts of incoming samples in production.
- *Incremental HDBSCAN.* The clustering algorithm HDBSCAN [104] has the ability to find clusters of data with very varied densities, so it theoretically improves DBSCAN on that specific point. In our context, this means that HDBSCAN is theoretically able to outperforms DBSCAN in terms of robustness on findings clusters of packers that may have very different densities, due to the polymorphic instances of the unpacking stub used in some packer families. An interesting recent incremental version of HDBSCAN has been published [105]. So it would be interesting to evaluate how this incremental HDBSCAN would perform in practice against the incremental version of DBSCAN we implemented, still in the context of the rapid evolution of packers.

Finally, to what extent does this thesis solve the malware packing problem in general? And what are the potential further research directions that can be undertaken accordingly in the future?

Malware packing is an open research problem that involves consequently open research problems at the level of packing detection, classification, and unpacking. This thesis focused on providing solutions for the packing detection and classification stages. We relied on machine learning to overcome the fragility of classical signature-based techniques, but more importantly, we studied and proposed machine-learning-based solutions which aimed to be *effective*, *efficient*, and *robust* to be practical parts of the malware analysis chain of an antivirus.

Nevertheless, malware and/or packer authors do not obey to any norms or standards when writing their packers, thus packing features as powerful as they are can be a subject of a cat-and-mouse game between these packers and antiviruses, to attempt to make the packing detection stage undecidable. These features can be updated in our models accordingly to the most up-to-date packing observations without altering the principles of our solutions. So we believe that the combination of updated features and machine learning is a good solution to cope with the imagination of the attacker and thus provide a better security for the user.

Finally, in this thesis we did not tackle the problem of *unpacking*, which is a part of the malware packing problem as whole. Therefore, in future work, we would like to improve the state of the art of unpacking techniques. In particular, a potential research direction would be to improve generic unpacking by relying on concolic execution to better the effectiveness of generic unpackers against many anti-analysis mechanisms. The latter tend to be more and more present in packed binaries specifically to prevent unpacking the malicious payloads. Thus, concolic execution could serve to avoid anti-analysis mechanisms and explore many execution paths, including the unpacking stub that brings to the malicious payload.

However, adopting concolic execution to explore the unpacking stub paths would pose a research challenge of effectiveness. We would have to find the right path that leads to unpacking the malicious payload. In addition, adopting concolic execution would particularly pose a great challenge of scalability, since many paths have to be explored and many heavy unpacking loops would have to be effectively and efficiently executed concolically. While concolic execution is widely used for malware, to the best we know, literature has rarely [106] addressed these challenges in the context of generic unpacking,

specifically. Even for the existing works [106], the results obtained are far from being satisfying and comprehensive.

A possible research track to defeat these challenges would be to learn from the concrete dynamic execution of single-path of various unpackers, in order to provide knowledge and heuristics to concolic execution engines, which could optimize the exploration of paths along with finding the right one. Still, other research tracks could be explored to offer solutions to these challenges, or even be compared with each other in terms of effectiveness and efficiency, in future work.

APPENDICES

1. PE File Format

This section details the Windows Portable Executable (PE) file format for which a summary is shown on the left column of Figure 2.1, considering that this file format is the focus of this thesis.

The PE format includes a PE header with the magic number (in ASCII) PE00, a file header, and an optional header. The file header includes metadata such as the number of sections in the file; the creation time; a pointer to the debug information (if any); the size of the optional header; and other characteristics. The optional header includes the major and minor OS version; the size of the code section, of all initialized and uninitialized data, of the whole image, and of the headers; the address of the image, code, and data sections, and entry point; the initial (commit) and maximal (reserve) stack sizes; a file checksum; and an array with pointers to other important parts of the file.

After the PE header the sections of the binary start. Standard sections (and their common name) include code (.text), initialized data (.data, .crt), uninitialized variables (.bss), resources (.rsrc), import table and address table (.idata), export table (.edata), relocations table (.reloc), thread-safe variables (.tls), and debug resources (.rdata). Each section includes information on its address, virtual and raw data sizes, and whether it is readable, writable, and/or executable.

2. Triangle Inequalities in the Cluster Update Policy (second step)

Let *eps* be the local radius of the nearest cluster, P_{new} the point representing the new sample, P_{SRP} the nearest SRP belonging to the nearest cluster to P_{new}, and P_i a cluster point.

The distance between P_i and P_{new} is unknown, however the distances $d(P_i, P_{SRP})$ and $d(P_{new}, P_{SRP})$ have already been computed. Using the *triangle inequality* and the

previously computed distances, the goal is to *accept* or *reject* the point P_i in the update process by proving that $d(P_i, P_{new}) \leq eps$ or $d(P_i, P_{new}) > eps$, respectively.

Case 1: if $d(P_{new}, P_{SRP}) \leq eps$ and $d(P_i, P_{SRP}) \leq eps - d(P_{new}, P_{SRP})$ then the point P_i is *accepted*.

Proof. Combining $d(P_i, P_{SRP}) \leq eps - d(P_{new}, P_{SRP})$ with the triangle inequality $d(P_i, P_{new}) \leq d(P_i, P_{SRP}) + d(P_{new}, P_{SRP})$, we have $d(P_i, P_{new}) - d(P_{new}, P_{SRP}) \leq d(P_i, P_{SRP}) \leq eps - d(P_{new}, P_{SRP})$, hence $d(P_i, P_{new}) \leq eps$. □

Case 2: if $d(P_{new}, P_{SRP}) \leq eps$ and $d(P_i, P_{SRP}) > eps + d(P_{new}, P_{SRP})$ then the point P_i is *rejected*.

Proof. Combining $d(P_i, P_{SRP}) > eps + d(P_{new}, P_{SRP})$ with the triangle inequality $d(P_i, P_{SRP}) \leq d(P_i, P_{new}) + d(P_{new}, P_{SRP})$, we have $d(P_i, P_{new}) + d(P_{new}, P_{SRP}) \geq d(P_i, P_{SRP}) > eps + d(P_{new}, PSRP)$, hence $d(P_i, P_{new}) > eps$. □

Case 3: if $d(P_{new}, P_{SRP}) > eps$ and $d(P_i, P_{SRP}) < d(P_{new}, P_{SRP}) - eps$, then the point P_i is *rejected*.

Proof. Combining $d(P_i, P_{SRP}) < d(P_{new}, P_{SRP}) - eps$ with the triangle inequality $d(P_{new}, P_{SRP}) \leq d(P_i, P_{SRP}) + d(P_i, P_{new})$, we have $d(P_{new}, P_{SRP}) - d(P_i, P_{new}) \leq d(P_i, P_{SRP}) < d(P_{new}, P_{SRP}) - eps$, hence $d(P_i, P_{new}) > eps$. □

Case 4: if $d(P_{new}, P_{SRP}) > eps$ and $d(P_i, P_{SRP}) > d(P_{new}, P_{SRP}) + eps$, then the point P_i is *rejected*.

Proof. Combining $d(P_i, P_{SRP}) > d(P_{new}, P_{SRP}) + eps$ with the triangle inequality $d(P_i, P_{new}) + d(P_{new}, P_{SRP}) \geq d(P_i, P_{SRP})$, we have $d(P_i, P_{new}) + d(P_{new}, P_{SRP}) \geq d(P_i, P_{SRP}) > d(P_{new}, P_{SRP}) + eps$, hence $d(P_i, P_{new}) > eps$). □

3. Examples of Radare 2 Traces for some Packed Binaries

***Synthetic datatset*:**

Sample packed with Armadillo [push, mov, push, push, push, mov, push, mov, sub, push, push, push, mov, call, xor, mov, mov, mov, and, mov, shl, add, mov, shr, mov, xor, push, call, xor, push, cmp, push, sete, push, call, test, mov, je, call, push, mov, mov, call, push, cmp, lea, jb]

Sample packed with ASPack [pushal, call, pop, inc, push, ret, jmp, call, pop, mov, add, sub, cmp, mov, jne, lea, push, call, mov, mov, lea, push, push, call, stosd, mov, scasb, jne, cmp, jne]

Sample packed with CustomAmberPacker [jmp, sub, mov, lea, mov, mov, jmp, mov, mov, sub, and, mov, mov, mov, lea, mov, mov, mov, mov, pushfd, pushfd, xor, popfd, pushfd, pop, xor, popfd, test, jne, mov, mov, mov, call, sub, mov, cmp, je, cmp, je, mov, mov, mov, call, sub, mov, mov, mov, mov, lea]

Sample packed with CustomOrigamiPacker []

Sample packed with CustomPackerSimple1 [push, mov, sub, mov, mov, mov, add, mov, mov, mov, mov, mov, mov, mov, mov, imul, mov, mov, shl, mov, mov, shl, mov, mov, imul, mov, lea, push, call, mov, mov, imul, mov, mov, shl, mov, mov, shl, mov, mov, imul, mov, lea, push, mov, push, call]

Sample packed with CustomPEPacker1 [pushal, call, pop, sub, lea, push, call, lea, push, push, call, mov, push, push, mov, add, push, push]

Sample packed with CustomPePacker2 [mov, mov, add, mov, xor, inc, cmp, jne]

Sample packed with CustomPetoyPacker [nop, nop, pushal, call, pop, sub, mov, or, je, inc, push, call, mov, push, push, push, push, call, push, mov, lea, push, push, push, call, push, mov, mov, mov, mov, shr, rep, mov, and, rep, mov, leave, ret, pop, lea, push, push, call, pushal, mov, mov, cld]

Sample packed with CustomSilentPacker [push, push, push, push, push, push, push, call, call, pop, ret, sub, mov, call, sub, sub, jmp, call, sub, mov, add, call, sub, movdqu, call, sub, mov, movdqu, aeskeygenassist, call, pshufd, vpslldq, pxor, vpslldq, pxor, vpslldq, pxor]

Sample packed with CustomTheArkPacker [push, mov, push, push, add, mov, mov, mov, mov, mov, lea, mov, call, push, mov, sub, mov, mov, mov, mov, mov, mov, mov, mov, mov, add, mov, mov, mov, mov, mov, cmp, je, mov, sub, mov, mov, mov, mov, mov, call, push, mov, sub, mov, mov, mov, mov, mov]

Sample packed with CustomUchihaPacker [nop, nop, nop, push, nop, nop, nop, push, pushal, mov, mov, cld, mov, xor, movsb, mov, call, add, jne, mov, inc, adc, ret, jae]

Sample packed with CustomXorPacker []

Sample packed with eXPressor [push, mov, sub, push, push, push, and, jmp, mov, add, mov, mov, mov, and, je, cmp, jne, cmp, jne, cmp, je, cmp, je, mov,

mov, cmp, jne, call, mov, mov, sub, ret, mov, push, push, push, push, call, mov, push, lea, push, push, call, lea, mov, mov]

Sample packed with ezip [jmp, push, mov, sub, push, push, push, lea, push, call, jmp, push, mov, sub, push, push, push, call, jmp, ret, mov, mov, mov, test, jne, push, pop, push, push, push, push, push, call, jmp, push, mov, movzx, cmp, jne, xor, pop, ret, add, test, jne, inc, jmp]

Sample packed with FSG [xchg, popal, xchg, push, movsb, mov]

Sample packed with MEW []

Sample packed with mPress [pushal, call, pop, add, mov, add, sub, mov, lodsw, shl, mov, push, lodsd, sub, add, mov, push, push, dec, mov, mov, jne]

Sample packed with NeoLite [jmp, mov, and, call, push, push, push, push, push, enter, push, sub, lea, mov, mov, call, push, push, mov, mov, mov, inc, test, je, xor, mov, inc, jmp]

Sample packed with Packman [pushal, call, pop, lea, add, mov, lea, push, pop, add, lodsd, dec, jne, mov, mov, mov, mov, push, push, push, push]

Sample packed with PECompact [mov, push, push, mov, xor, mov, push, inc, inc, outsd, insd, jo, arpl, mov, pop, pop, daa, add, fild, fisub, pop, adc, add, pop, scasb, xchg, in, loop, mov, in, push]

Sample packed with PELock [push, push, call, ret, mov, mov, call, cmp, jae, neg, add, add, test, xchg, mov, push, ret, mov, mov, mov, push, call, mov, mov, mov, mov, mov, mov, mov, and, mov, cmp, je, or, mov, shl, add, mov, xor, push, mov, call, cmp, jne, mov, jmp, push]

Sample packed with PENinja [mov, mov, call, push, xor, dec, mov, xor, xor, lodsb, xor, mov, mov, mov, mov, shr, rcr, jae, dec, jne]

Sample packed with Petite [mov, pushal, lea, push, push, push, push, push, call, mov, mov, lea, mov, push, call, push, cmp, jae, push, push, push, jmp, push, xor, xor, lodsb, xor, stosb, dec, jle, call, add, jne, mov, sub, adc, ret, jae]

Sample packed with RLPack [pushal, call, mov, add, cmp, jne, mov, mov, call, pushal, mov, add, mov, add, mov, add, mov, lea, jmp, cmp, ja, mov, mov, add, add, dec]

Sample packed with telock [jmp, cld, pushal, call, call, pop, sub, pop, je, mov, mov, clc, jae, add, lea, call, xor, inc, pop, jmp, aam, dec, jg]

Sample packed with Themida [push, push, push, call, pop, mov, inc, sub, sub, add, cmp, jne, mov, mov, push, push, push, push, call, push, mov, push, push, push, push, mov, mov, shr, mov, mov, test, je, xor, add, add, dec, jmp]

Sample packed with UPack [pushal, call, xor, pop, xchg, jecxz, sub, mov, lodsd, sub, lodsd, add, push, xchg, lodsd, xchg, rep, pop, lodsd, push, xchg, add, lodsd, loop]

Sample packed with UPX [pushal, mov, lea, push, or, jmp, mov, sub, adc, jb, mov, inc, mov, inc, add, jne, mov, add, jne, adc, add, jae, jne, xor, sub, jb, add, jne, adc, add, jne, adc, jne, inc, add, jne, adc, add, jae]

Sample packed with WinRAR [call, push, mov, sub, and, and, mov, push, push, mov, mov, cmp, je, lea, push, call, mov, xor, mov, call, xor, call, xor, lea, push, call, mov, lea, xor, xor, xor, cmp, jne, test, jne, mov, not, mov, pop, pop, mov, pop, ret, jmp, push]

Sample packed with WinZip [call, push, mov, sub, mov, and, and, push, push, mov, cmp, mov, je, push, lea, push, call, mov, xor, call, xor, call, xor, call, xor, lea, push, call, mov, xor, xor, cmp, jne, test, jne, mov, not, mov, pop, pop, pop, leave, ret, jmp]

Sample packed with YodaCryptor [pushal, call, pop, sub, mov, lea, mov, lodsb, sub, sub, sub, add, nop, jmp, ror, sub, xor, jmp, stc, add, stc, sub, jmp, sub, sub, sub, jmp, xor, add, stc, rol, stosb, loop]

***Malware feed*:**

Sample packed with ActiveMARK [mov, jmp, push, jmp, jmp, jmp, push, jmp, mov, jmp, jmp, push, jmp, jmp, mov, jmp, sub, jmp, mov, jmp, jmp, mov, mov, jmp, jmp, mov, mov, jmp, cmp, jge, mov, mov, jmp, jmp, mov, jmp, add, mov, jmp]

Sample packed with ASPack [pushal, jmp, call, pop, jmp, inc, jmp, sub, mov, add, sub, lea, push, call, lea, push, push, call, push, push, push, push, lea, push]

Sample packed with ASProtect [pushal, call, pop, sub, mov, add, sub, cmp, mov, jne, lea, push, call, mov, mov, lea, push, push, call, mov, lea, push, push, call, mov, mov, mov, jmp, push, pop, jmp, push, pop, push, pop, jmp, push, pop, push, pop, jmp, push, pop, push, pop, jmp]

Sample packed with AutoIt [call, mov, push, mov, sub, mov, and, and, push, push, mov, mov, cmp, je, push, lea, push, call, mov, xor, call, xor, call, xor, call, xor, lea, push, call, mov, xor, xor, cmp, jne, test, jne, mov, not, mov, pop, pop, pop, leave, ret]

Sample packed with ExeStealth [jmp, jmp, pushal, nop, call, pop, sub, mov, add, add, jmp, jmp, sub, add, jmp, add, sub, mov, nop, lea, mov, lodsb, add, ror, sub, clc, add, stc, clc, nop, clc, sub, sub, add, ror, ror, xor, ror, rol, add, sub, dec, stc, add, ror, sub, stosb, loop]

Sample packed with eXPressor [push, mov, sub, push, push, push, jmp, mov, sub, mov, cmp, je, mov, cmp, je, and, push, push, push, push, call, mov, push, push, push, call, mov, lea, mov, mov, movsx, cmp, je, mov, dec, mov, jmp]

Sample packed with FishPE [pushal, call, pop, push, mov, mov, mov, lodsd, lodsd, mov, add]

Sample packed with FSG [mov, lodsd, xchg, lodsd, xchg, lodsd, push, xchg, mov, movsb, mov]

Sample packed with InnoSetup [push, mov, add, push, push, push, xor, mov, mov, mov, mov, mov, mov, mov, mov, mov, mov, call, push, mov, xor, mov, push, call]

Sample packed with InstallShield [push, mov, sub, push, call, mov, test, jne, push, call, mov, push, mov, cmp, jne, cmp, jle, mov, test, je, cmp, jle, push, call, mov, jmp]

Sample packed with MEW []

Sample packed with MoleBox [call, pushal, call, call, call, push, mov, sub, push, push, push, call, mov, push, push, call, mov, mov, sub, mov, mov, sub, mov, mov, mov, mov, mov, mov, mov, mov, imul, mov, add, mov, imul, sub, mov, mov, add, mov, mov, mov, add, mov, mov, mov, add, mov]

Sample packed with NeoLite [jmp, mov, and, call, push, push, push, push, push, enter, push, sub, lea, mov, mov, call, push, push, mov, mov, mov, inc, test, je, xor, mov, inc, jmp]

Sample packed with NSIS [sub, push, push, push, push, call, mov, add, mov, mov, call, mov, push, push, call, push, call, push, push, call, cmp, jne, mov, jmp, mov, test, jne, push, call, mov, mov, jmp, cmp, je, cmp, jne, push, push, call, jmp, push]

Sample packed with NsPacK [pushfd, pushal, call, pop, sub, lea, cmp, je, mov, mov, sub, mov, add, lea, add, push, push, push, push, push, push, call, test, je, mov, call, pop, mov, add, push, push, call, push, mov, mov, mov, cld, mov, movsb, call, add, jne, mov, inc, adc, ret, jae]

Sample packed with Packman [pushal, call, pop, lea, add, mov, lea, push, pop, add, lodsd, dec, jne, mov, mov, mov, mov, push, push, push, push]

Sample packed with PCGuard [cld, push, push, call, pop, jmp, pushal, call, pop, jmp, inc, jmp, jmp, jmp, popal, pop, pop, jmp, pushal, call, add, mov, call, pushal, push, push, push, push, call, mov, push, lea, lea, push, push, call, pushal, mov, mov, cld, mov, xor, movsb, mov, call, add, jne, mov]

Sample packed with PECompact [mov, push, push, mov, xor, mov, push, inc, inc, outsd, insd, jo, arpl, nop, out, dec, sub, mov, xor, mov]

Sample packed with PEPACK [je, jmp, pushal, call, pop, sub, cmp, je, mov, mov, sub, mov, mov, mov, add, mov, cmp, jne, push, push, push, push, call, jmp, or, je, mov, mov, add, lodsd, or, je, mov, add, lodsd, mov, lodsd, or, je, push, push, mov, mov, rep, pop, push, push, push]

Sample packed with PESpin [jmp, pushal, call, mov, add, sub, dec, sub, pop, out, add, in, sahf, jne, xor, xor, test, mov, add, jmp, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add, add]

Sample packed with Petite [mov, push, push, push, mov, pushf, pushal, push, mov, add, push, push, call, mov, mov, add, push, push, call, mov, lea, mov, push, push, push, mov, mov, add, lea, mov, push, mov, mov, push, push, push, push, push, push, push, push, push, push]

Sample packed with RLPack [pushal, call, mov, add, lea, lea, xor, call, pushal, mov, or, je, popal, ret, jmp, cmp, jne, push, push, call, pushal, mov, mov, cld, mov, xor, movsb, mov, call, add, jne, mov, inc, adc, ret, jae]

Sample packed with Shrinker [cmp, push, mov, push, push, jne, push, call, push, push, call, ret]

Sample packed with Themida [mov, pushal, or, je, mov, mov, push, push, call, pushal, mov, mov, cld, mov, mov, inc, mov, inc, mov, add, jne, mov, inc, adc, jae, add, jne, jae, xor, add, jne, jae, add]

Sample packed with UPack [mov, lodsd, mov, xchg, movsd, xor, xor, stosd, dec, stosd, neg, mov, rep, shl, mov, rep, lodsd, push, xchg, push, lodsd, xchg, pop, lea]

Sample packed with UPX [pushal, call, pop, sub, push, lea, push, lea, or, xor, nop, nop, nop, nop, add, jne, mov, sub, adc, jae, mov, inc, mov, inc, jmp]

Sample packed with WinRAR [call, push, mov, sub, push, push, call, mov, ret, pop, pop, mov, cmp, je, mov, call, push, mov, call, push, mov, sub, push, push, lea, push, xor, push, mov, call, cmp, seta, mov, push, call, test, setne, mov, inc, cmp, jb]

Sample packed with WinZip [call, mov, cmp, je, mov, inc, cmp, je, cmp, je, inc, cmp, jne]

Sample packed with Wise [push, mov, sub, push, push, push, push, pop, push, mov, call, call, mov, mov, mov, cmp, jne, xor, cmp, je, cmp, je, mov, inc, mov, jmp]

BIBLIOGRAPHY

[1] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma, « Effective, efficient, and robust packing detection and classification », *in*: *Computers & Security* 85 (2019), pp. 436–451, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2019.05.007, URL: https://www.sciencedirect.com/science/article/pii/S0167404818311040.

[2] Lamine Noureddine, Annelie Heuser, Cassius Puodzius, and Olivier Zendra, « SE-PAC: A Self-Evolving PAcker Classifier against Rapid Packers Evolution », *in*: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, Virtual Event, USA: Association for Computing Machinery, 2021, 281–292, ISBN: 9781450381437, DOI: 10.1145/3422337.3447848, URL: https://doi.org/10.1145/3422337.3447848.

[3] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon, « Empirical assessment of machine learning-based malware detectors for Android », *in*: *Empirical Software Engineering* 21.*1* (2016), pp. 183–211, ISSN: 1573-7616, DOI: 10.1007/s10664-014-9352-6, URL: https://doi.org/10.1007/s10664-014-9352-6.

[4] *The Digital Revolution*, 2008, URL: https://web.archive.org/web/20081007132355/http://history.sandiego.edu/gen/recording/digital.html (visited on 10/2021).

[5] Vassilios Zoumpourlis, Maria Goulielmaki, Emmanouil Rizos, Stella Baliou, and Demetrios A Spandidos, « [Comment] The COVID-19 pandemic as a scientific and social challenge in the 21st century », *in*: *Molecular medicine reports* 22.*4* (2020), pp. 3035–3048.

[6] *Impact of COVID-19 on Cybersecurity*, 2020, URL: https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html (visited on 10/2021).

[7] *2020: The Year the COVID-19 Crisis Brought a Cyber Pandemic*, 2020, URL: https://www.govtech.com/blogs/lohrmann-on-cybersecurity/2020-the-year-the-covid-19-crisis-brought-a-cyber-pandemic.html (visited on 10/2021).

[8] Rabie A Ramadan, Bassam W Aboshosha, Jalawi Sulaiman Alshudukhi, Abdullah J Alzahrani, Ayman El-Sayed, and Mohamed M Dessouky, « Cybersecurity and Countermeasures at the Time of Pandemic », *in*: *Journal of Advanced Transportation* 2021 (2021).

[9] *CYBERCRIME: COVID-19 IMPACT*, 2020, URL: https://www.interpol.int/News-and-Events/News/2020/INTERPOL-report-shows-alarming-rate-of-cyberattacks-during-COVID-19 (visited on 10/2021).

[10] *Major ransomware attack against U.S. tech provider forces Swedish store closures*, 2021, URL: https://www.reuters.com/technology/cyber-attack-against-us-it-provider-forces-swedish-chain-close-800-stores-2021-07-03/ (visited on 10/2021).

[11] *Malware definition*, 2021, URL: https://www.lexico.com/definition/malware (visited on 10/2021).

[12] *Malware*, 2021, URL: https://www.av-test.org/en/statistics/malware/ (visited on 10/2021).

[13] *ENISA Threat Landscape 2020: Cyber Attacks Becoming More Sophisticated, Targeted, Widespread and Undetected*, 2020, URL: https://www.enisa.europa.eu/news/enisa-news/enisa-threat-landscape-2020 (visited on 10/2021).

[14] David Braue, *Global Ransomware Damage Costs Predicted To Exceed 265 Billion dollars By 2031*, Cybersecurity Ventures, 2021, URL: https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/ (visited on 10/2021).

[15] *THE COST OF CYBERCRIME, NINTH ANNUAL COST OF CYBERCRIME STUDY UNLOCKING THE VALUE OF IMPROVED CYBERSECURITY PROTECTION*, accenturesecurity, 2019, URL: https://www.accenture.com/_acnmedia/pdf-96/accenture-2019-cost-of-cybercrime-study-final.pdf (visited on 10/2021).

[16] Xufang Li, Peter KK Loh, and Freddy Tan, « Mechanisms of polymorphic and metamorphic viruses », *in*: *2011 European intelligence and security informatics conference*, IEEE, 2011, pp. 149–154.

[17] Mark Vincent Yason, « The art of unpacking », *in*: *Retrieved Feb* 12 (2007), p. 2008.

[18] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas, « SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers », *in*: *2015 IEEE Symposium on Security and Privacy*, IEEE, 2015, pp. 659–673.

[19] Balaji Prasad, *Cloak and Dagger: Unpacking Hidden Malware Attacks*, NortonLifeLock Inc, 2016, URL: `https://www.symantec.com/blogs/expert-perspectives/unpacking-hidden-malware-attacks` (visited on 10/2021).

[20] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar, « A Survey on Malware Detection Using Data Mining Techniques », *in*: *ACM Comput. Surv.* 50.*3* (June 2017), ISSN: 0360-0300, DOI: `10.1145/3073559`, URL: `https://doi.org/10.1145/3073559`.

[21] Andreas Moser, Christopher Kruegel, and Engin Kirda, « Exploring Multiple Execution Paths for Malware Analysis », *in*: *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007, pp. 231–245, DOI: `10.1109/SP.2007.17`.

[22] Jonathan A.P. Marpaung, Mangal Sain, and Hoon-Jae Lee, « Survey on malware evasion techniques: State of the art and challenges », *in*: *2012 14th International Conference on Advanced Communication Technology (ICACT)*, 2012, pp. 744–749.

[23] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st, Wiley Publishing, 2014, ISBN: 1118825098, 9781118825099.

[24] Michael Sikorski and Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st, San Francisco, CA, USA: No Starch Press, 2012, ISBN: 1593272901, 9781593272906.

[25] Jie Lin, Chuanyi Liu, Xinyi Zhang, Rongfei Zhuang, and Binxing Fang, « VMRe: A Reverse Framework of Virtual Machine Protection Packed Binaries », *in*: *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, IEEE, 2019, pp. 528–535.

[26] Rolf Rolles, « Unpacking virtualization obfuscators », *in*: *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[27] *BITDEFENDER ANTIVIRUS TECHNOLOGY*, 2007, URL: https://www.bitdefender.com/files/Main/file/BitDefender_Antivirus_Technology.pdf (visited on 10/2021).

[28] *PEiD*, 2017, URL: http://appnee.com/peid/ (visited on 10/2021).

[29] *YARA*, 2021, URL: http://virustotal.github.io/yara/ (visited on 10/2021).

[30] *Detect-It-Easy*, 2021, URL: https://github.com/horsicq/Detect-It-Easy (visited on 10/2021).

[31] *PEiD signatures*, 2020, URL: https://github.com/sooshie/packerid/blob/master/userdb.txt (visited on 10/2021).

[32] *VirusTotal Packer YARA Ruleset*, 2020, URL: https://github.com/Yara-Rules/rules/tree/master/packers (visited on 10/2021).

[33] *PCRE - Perl Compatible Regular Expressions*, 2015, URL: https://www.pcre.org/ (visited on 10/2021).

[34] *Detect-It-Easy signatures database*, 2021, URL: https://github.com/horsicq/Detect-It-Easy/tree/master/db (visited on 10/2021).

[35] R. Lyda and J. Hamrock, « Using Entropy Analysis to Find Encrypted and Packed Malware », *in*: *IEEE Security Privacy* 5.*2* (2007), pp. 40–45, ISSN: 1540-7993, DOI: 10.1109/MSP.2007.48.

[36] Xabier Ugarte-Pedrero, Igor Santos, Iván García-Ferreira, Sergio Huerta, Borja Sanz, and Pablo G. Bringas, « On the adoption of anomaly detection for packed executable filtering », *in*: *Computers & Security* 43 (2014), pp. 126 –144, ISSN: 0167-4048, DOI: https://doi.org/10.1016/j.cose.2014.03.012, URL: http://www.sciencedirect.com/science/article/pii/S0167404814000522.

[37] Tim Ebringer, Li Sun, and Serdar Boztas, « A fast randomness test that preserves local detail », *in*: *Virus Bulletin 2008*, Virus Bulletin Ltd, 2008, pp. 34–42.

[38] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee, « Generic unpacking using entropy analysis », *in*: *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, IEEE, 2010, pp. 98–105.

[39] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann, « Pattern Recognition Techniques for the Classification of Malware Packers », *in*: *Proceedings of the 15th Australasian Conference on Information Security and Privacy*, ACISP'10, Sydney, Australia: Springer-Verlag, 2010, pp. 370–390, ISBN: 3-642-14080-7, 978-3-642-14080-8, URL: http://dl.acm.org/citation.cfm?id=1926211.1926239.

[40] Munkhbayar Bat-Erdene, Taebeom Kim, Hongzhe Li, and Heejo Lee, « Dynamic classification of packing algorithms for inspecting executables using entropy analysis », *in*: *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, 2013, pp. 19–26, DOI: 10.1109/MALWARE.2013.6703681.

[41] Jithu Raphel and P. Vinod, « Information Theoretic Method for Classification of Packed and Encoded Files », *in*: *Proceedings of the 8th International Conference on Security of Information and Networks*, SIN '15, Sochi, Russia: ACM, 2015, pp. 296–303, ISBN: 978-1-4503-3453-2, DOI: 10.1145/2799979.2800015, URL: http://doi.acm.org/10.1145/2799979.2800015.

[42] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee, « Packer detection for multi-layer executables using entropy analysis », *in*: *Entropy* 19.*3* (2017), p. 125.

[43] Munkhbayar Bat-Erdene, Hyundo Park, Hongzhe Li, Heejo Lee, and Mahn-Soo Choi, « Entropy analysis to classify unknown packing algorithms for malware detection », *in*: *International Journal of Information Security* 16.*3* (2017), pp. 227–248.

[44] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu, « A symbolic representation of time series, with implications for streaming algorithms », *in*: *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, 2003, pp. 2–11.

[45] Xabier Ugarte-Pedrero, Igor Santos, Borja Sanz, Carlos Laorden, and Pablo Garcia Bringas, « Countering entropy measure attacks on packed software detection », *in*: *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 164–168.

[46] *What is Machine Learning?*, 2020, URL: https://emerj.com/ai-glossary-terms/what-is-machine-learning/ (visited on 10/2021).

[47] *What Is Machine Learning in Security?*, 2021, URL: https://www.cisco.com/c/en/us/products/security/machine-learning-security.html (visited on 10/2021).

[48] Ethem Alpaydin, *Introduction to Machine Learning*, The MIT Press, 2014, ISBN: 0262028182.

[49] Lior Rokach and Oded Z Maimon, *Data mining with decision trees: theory and applications*, vol. 69, World scientific, 2007.

[50] *Decision Trees*, 2020, URL: https://scikit-learn.org/stable/modules/tree.html (visited on 10/2021).

[51] *Ensemble methods*, 2020, URL: https://scikit-learn.org/stable/modules/ensemble.html (visited on 10/2021).

[52] P Davies, *Kendall's Advanced Theory of Statistics. Volume 1. Distribution Theory*, 1988.

[53] Margareta Ackerman and Sanjoy Dasgupta, « Incremental clustering: The case for extra clusters », *in*: *Advances in Neural Information Processing Systems*, 2014, pp. 307–315.

[54] *Evaluation of clustering*, 2008, URL: https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html (visited on 10/2021).

[55] PN Tan, M Steinbach, and V Kumar, « Chapter 8 Cluster analysis: basic concepts and algorithms », *in*: *Introduction to data mining, 6th edn. Peason Addison Wesley, Boston* (2006), pp. 486–568.

[56] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al., « A density-based algorithm for discovering clusters in large spatial databases with noise. », *in*: *Kdd*, vol. 96, 34, 1996, pp. 226–231.

[57] Joris Kinable and Orestis Kostakis, « Malware classification based on call graph clustering », *in*: *Journal in computer virology* 7.*4* (2011), pp. 233–245.

[58] *DBSCAN*, 2021, URL: https://en.wikipedia.org/wiki/DBSCAN (visited on 10/2021).

[59] Philippe Thomas, *Semi-Supervised Learning edited by O. Chapelle, B. Schölkopf and A. Zien*, Mar. 2009, URL: https://hal.archives-ouvertes.fr/hal-00372719.

[60] Roberto Perdisci, Andrea Lanzi, and Wenke Lee, « Classification of packed executables for accurate computer virus detection », *in*: *Pattern Recognition Letters* 29.*14* (2008), pp. 1941 –1946, ISSN: 0167-8655, DOI: https://doi.org/10.1016/j.patrec.2008.06.016, URL: http://www.sciencedirect.com/science/article/pii/S0167865508002110.

[61] T. Wang and C. Wu, « Detection of packed executables using support vector machines », *in*: *2011 International Conference on Machine Learning and Cybernetics*, vol. 2, 2011, pp. 717–722, DOI: 10.1109/ICMLC.2011.6016774.

[62] C. Burgess, F. Kurugollu, S. Sezer, and K. McLaughlin, « Detecting packed executables using steganalysis », *in*: *2014 5th European Workshop on Visual Information Processing (EUVIP)*, 2014, pp. 1–5, DOI: 10.1109/EUVIP.2014.7018361.

[63] Kesav Kancherla, J Kevin Donahue, and Srinivas Mukkamala, « Packer identification using Byte plot and Markov plot », *in*: *Journal of Computer Virology and Hacking Techniques* 12 (2015), pp. 101–111.

[64] Mohaddeseh Zakeri, Fatemeh Faraji Daneshgar, and Maghsoud Abbaspour, « A Static Heuristic Approach to Detecting Malware Targets », *in*: *section and Commun. Netw.* 8.*17* (Nov. 2015), pp. 3015–3027, ISSN: 1939-0114, DOI: 10.1002/section1228, URL: http://dx.doi.org/10.1002/section1228.

[65] Neminath Hubballi and Himanshu Dogra, « Detecting Packed Executable File: Supervised or Anomaly Detection Method? », *in*: *2016 11th International Conference on Availability, Reliability and Security (ARES)*, IEEE, 2016, pp. 638–643.

[66] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho, « Packer identification based on metadata signature », *in*: *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, 2017, pp. 1–11.

[67] Erik Bergenholtz, Emiliano Casalicchio, Dragos Ilie, and Andrew Moss, « Detection of metamorphic malware packers using multilayered LSTM networks », *in*: *International Conference on Information and Communications Security*, Springer, 2020, pp. 36–53.

[68] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto, « Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification », *in*: *CODASPY*, 2016.

[69] Moustafa Saleh, E Paul Ratazzi, and Shouhuai Xu, « A control flow graph-based signature for packer identification », *in*: *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*, IEEE, 2017, pp. 683–688.

[70] Xingwei Li, Zheng Shan, Fudong Liu, Yihang Chen, and Yifan Hou, « A consistently-executing graph-based approach for malware packer identification », *in*: *IEEE Access* 7 (2019), pp. 51620–51629.

[71] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann, « Pattern recognition techniques for the classification of malware packers », *in*: *Australasian Conference on Information Security and Privacy*, Springer, 2010, pp. 370–390.

[72] A. Moser, C. Kruegel, and E. Kirda, « Limits of Static Analysis for Malware Detection », *in*: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430, DOI: 10.1109/ACSAC.2007.21.

[73] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion, « Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost », *in*: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Toronto, Canada: ACM, 2018, pp. 395–411, ISBN: 978-1-4503-5693-0, DOI: 10.1145/3243734.3243771, URL: http://doi.acm.org/10.1145/3243734.3243771.

[74] Girish Chandrashekar and Ferat Sahin, « A survey on feature selection methods », *in*: *Computers & Electrical Engineering* 40.*1* (2014), pp. 16–28.

[75] Randall Balestriero, Hervé Glotin, and Richard G. Baraniuk, « Semi-Supervised Learning Enabled by Multiscale Deep Neural Network Inversion », *in*: *CoRR* abs/1802.10172 (2018), arXiv: 1802.10172, URL: http://arxiv.org/abs/1802.10172.

[76] Mike Sconzo, *Packerid*, 2020, URL: https://github.com/sooshie/packerid (visited on 10/2021).

[77] *PeLib*, 2020, URL: https://github.com/avast-tl/pelib (visited on 10/2021).

[78] Sebastian Porst, *PeLib*, 2005, URL: http://www.pelib.com (visited on 10/2021).

[79] *NumPy*, 2021, URL: http://www.numpy.org (visited on 10/2021).

[80] *Metrics and scoring: quantifying the quality of predictions*, 2020, URL: https://scikit-learn.org/stable/modules/model_evaluation.html (visited on 10/2021).

[81] *Joblib: running Python functions as pipeline jobs*, 2021, URL: https://joblib.readthedocs.io/en/latest/ (visited on 10/2021).

[82] Munkhbayar Bat-Erdene, Taebeom Kim, Hyundo Park, and Heejo Lee, « Packer Detection for Multi-Layer Executables Using Entropy Analysis », *in*: *Entropy* 19 (2017), p. 125.

[83] Igor Santos, Xabier Ugarte-Pedrero, Borja Sanz, Carlos Laorden, and Pablo G Bringas, « Collective classification for packed executable identification », *in*: *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*, 2011, pp. 23–30.

[84] Mike Sconzo, *I am packer and so can you*, 2015, URL: https://youtu.be/jCIT7rXX8y0 (visited on 10/2021).

[85] John C Gower, « A general coefficient of similarity and some of its properties », *in*: *Biometrics* (1971), pp. 857–871.

[86] Erhan Erkut, Yilmaz Ülküsal, and Oktay Yenicerioğlu, « A comparison of p-dispersion heuristics », *in*: *Computers & operations research* 21.*10* (1994), pp. 1103–1113.

[87] Chrysostomos Symvoulidis, *An Incremental DBSCAN approach in Python for real-time monitoring data*, 2019, URL: https://github.com/csymvoul/Incremental_DBSCAN (visited on 10/2021).

[88] *Amber*, 2018, URL: https://github.com/EgeBalci/Amber (visited on 10/2021).

[89] *Origami*, 2020, URL: https://github.com/dr4k0nia/Origami (visited on 10/2021).

[90] *Writing a simple PE Packer in detail*, 2019, URL: https://github.com/levanvn/Packer_Simple-1 (visited on 10/2021).

[91] *PePacker*, 2017, URL: https://github.com/SamLarenN/PePacker (visited on 10/2021).

[92] *PE-Packer*, 2020, URL: https://github.com/czs108/PE-Packer (visited on 10/2021).

[93] *PE Toy*, 2016, URL: https://github.com/qy7tt/petoy (visited on 10/2021).

[94] *Silent-Packer*, 2020, URL: https://github.com/SilentVoid13/Silent_Packer (visited on 10/2021).

[95] *theArk*, 2019, URL: https://github.com/aaaddress1/theArk (visited on 10/2021).

[96] *Simple-PE32-Packer*, 2018, URL: https://github.com/z3r0d4y5/Simple-PE32-Packer (visited on 10/2021).

[97] *xorPacker*, 2020, URL: https://github.com/nqntmqmqmb/xorPacker (visited on 10/2021).

[98] Nguyen Xuan Vinh, Julien Epps, and James Bailey, « Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance », *in*: *Journal of Machine Learning Research* 11.*Oct* (2010), pp. 2837–2854.

[99] Davoud Moulavi, Pablo A Jaskowiak, Ricardo JGB Campello, Arthur Zimek, and Jörg Sander, « Density-based clustering validation », *in*: *Proceedings of the 2014 SIAM international conference on data mining*, SIAM, 2014, pp. 839–847.

[100] *Radare2*, 2019, URL: https://rada.re/n/ (visited on 10/2021).

[101] *scikit-learn: Machine Learning in Python*, 2020, URL: https://scikit-learn.org/ (visited on 10/2021).

[102] *One-Hot-Encoder*, 2020, URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html (visited on 10/2021).

[103] *Deep Packer Inspector*, 2017, URL: https://www.packerinspector.com/ (visited on 2019).

[104] Leland McInnes, John Healy, and Steve Astels, « hdbscan: Hierarchical density based clustering », *in*: *Journal of Open Source Software* 2.*11* (2017), p. 205.

[105] Matteo Dell'Amico, « Fishdbc: Flexible, incremental, scalable, hierarchical density-based clustering for arbitrary data and distance », *in*: *arXiv preprint arXiv:1910.07283* (2019).

[106] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas, « RAMBO: Run-Time Packer Analysis with Multiple Branch Observation », *in*: *Detection of Intrusions and Malware, and Vulnerability Assessment*, ed. by Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, Cham: Springer International Publishing, 2016, pp. 186–206, ISBN: 978-3-319-40667-1.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Détection et classification d'empaquetage s'appuyant sur l'apprentissage automatique pour contrer la propagation des logiciels malveillants

Mot clés : Empaquetage, Logiciels malveillants, Détection, Classification, Apprentissage automatique

Résumé : Dans cette thèse, nous proposons des solutions de détection et de classification d'empaqueteurs effectives, efficaces et robustes, constituant des parties pratiques de la chaîne d'analyse de logiciels malveillants d'un antivirus.

Nos solutions apportent à la littérature deux contributions. Dans la première, nous introduisons une étude visant à mieux comprendre l'impact de la labellisation, la sélection d'algorithme d'apprentissage automatique et la sélection de caractéristique sur l'effectivité, l'efficacité et la robustesse des systèmes de détection et de classification d'empaqueteurs basés sur l'apprentissage automatique supervisé. Dans la seconde, nous proposons, concevons et implémentons SE-PAC (Self-Evolving PAcker Classifier), un nouveau framework auto-évolutif de classification d'empaqueteurs qui repose sur le regroupement incrémental de façon semi-supervisée, pour faire face à l'évolution rapide des empaqueteurs au fil du temps.

Pour ces deux contributions, nous menons des expériences réalistes montrant des résultats prometteurs en termes d'effectivité, d'efficacité et de robustesse pour la détection et la classification des empaqueteurs.

Title: Packing detection and classification relying on machine learning to stop malware propagation

Keywords: Packing, Malware, Detection, Classification, Machine learning

Abstract: In this thesis, we propose effective, efficient, and robust packing detection and classification solutions to be practical parts of the malware analysis chain of an antivirus.

Our solutions bring two contributions to the literature. In the first one, we introduce a study which aims at better understanding the impact of ground truth generation, machine learning algorithm selection, and feature selection on the effectiveness, efficiency, and robustness of supervised machine-learning-based packing detection and classification systems. In the second one, we propose, design, and implement SE-PAC, a new Self-Evolving PAcker Classifier framework that relies on incremental clustering in a semi-supervised fashion, in order to cope with the fast-paced evolution of packers over time.

For both contributions, we conduct realistic experiments showing promising results in terms of effectiveness, efficiency, and robustness for packing detection and classification.