Machine Learning and other Computational-Intelligence Techniques for Security Applications

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Control and Computer Engineering (31^{th}cycle)

Machine Learning and other Computational-Intelligence Techniques for Security Applications

By

Andrea Marcelli

******

Supervisor(s):
Prof. Giovanni Squillero

Doctoral Examination Committee:
Prof. Leonardo Vanneschi, Universidade Nova de Lisboa
Prof. Christine Zarges, Aberystwyth University

Politecnico di Torino
2019

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Andrea Marcelli
2019

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my parents and my sister.

Acknowledgements

I would like thank my supervisor, Prof. Giovanni Squillero, for guiding me during these years. He has been a mentor, a friend, and a source of inspiration. I thank him for the interesting discussions, the hours spent coding together and for the unconventional ideas.

I would like to thank Alberto Tonda, Prof. Corno, and Prof. Quer, for being always there, for their support and precious advices.

Thanks to all the Hispasec Team, who firstly believed in me and invited me in the beautiful Malaga. A big thanks to Antonio, Carlos, Daniel, Fernado D., Fernando R., Francisco, José, Miguel, Paco, Roman, and all the others employees. You have been my family for several months, offering such an incredible experience.

It is my privilege to thank Dario, Mariano, and Shock for the precious advice.

I am extremely thankful to my friends and colleagues Teo, Sebastiano, Lorenzo, Masoud, and Jetmir for helping me to survive these years and not letting me give up.

I profusely thank my friends Angelo, Luca and Chiara for the time spent together, and for the continuous support offered.

Most importantly, none of this would have been possible without the love and patience of my family that has been a constant source of support and strength all these years.

In full gratitude, I would like to acknowledge all the people that I did not mentioned earlier, but who likewise encouraged, inspired, and supported me.

Abstract

Machine learning and evolutionary computation are powerful tools that achieved incredible results in the most variegate fields. While the techniques are quite known, their application requires a deep knowledge in the field of usage. This thesis explores the application of computational intelligence methodologies to open problems in computer security, mainly in the field of malware families detection.

Malware is a big business. With hundreds of thousands of malware delivered every day, manual analysis in not an option. Malicious samples are commonly detected using a combination of techniques, ranging from machine learning to hash-based content. However, the industry mostly relies on signatures, which are patterns extracted from the code or behavior of selected samples. Generating effective signatures, with 0-false positives, and low false negatives rates, is a task that requires a considerable amount of time and resources from skilled experts, while automatically generating them is an open problem.

In this thesis, we propose a semi-supervised methodology for the automatic identification of malware families, used to safely extend experts knowledge on new malicious samples, and to reduce the amount of applications to manually analyze. Then, newly discovered samples are submitted to an automatic signature generation procedure, which produces a formal rule which has a limited risk of detecting false positives in the future, yet it is general enough to catch future threats.

The effectiveness of the approach is assessed running experiments on 1.5 million Android applications, the largest dataset ever used in a public research on Android malware. The procedure has been implemented in two frameworks which have been publicly released: *YaYaGen* for Android applications, and *YaYaGenPE* for Windows. Furthermore, since January 2018, part of the proposed approach is in use in Koodous, a collaborative analysis platform for Android, developed by Hispasec.

Astratto

Machine learning e algoritmi evolutivi sono potenti strumenti matematici che hanno permesso di raggiungere incredibili risultati nei campi più disparati. Nonostante le tecniche siano note da decadi, un utilizzo efficace richiede un'approfondita conoscenza del campo di applicazione. In questa tesi vengono esplorate alcune applicazioni di *Computational Intelligence* a problemi aperti di sicurezza informatica, soprattutto incentrati sul riconoscimento delle famiglie di malware.

Con un flusso di centinaia di migliaia di nuove applicazioni rilasciate ogni giorno, negli anni sono stati sviluppati diversi metodi in grado di automatizzare l'identificazione di malware, basati su una varietà di tecniche, che spaziano dal machine learning, al calcolo dell'hash di porzioni di codice. Lo standard industriale è bastato sull'utilizzo delle signature: regole contenti pattern unici estratti dal codice o dall'analisi di un'applicazione. Tuttavia, scrivere signature efficaci, con un basso numero di falsi positivi e negativi, richiede tempo e risorse.

In questa tesi, viene proposta una metodologia semi-supervised per identificare automaticamente le famiglie di malware, utilizzata sia per individuare nuovi esemplari, che per ridurre il numero di file da analizzare manualmente. In seguito, i campioni selezionati sono sottoposti ad un processo di generazione automatica della signature, che produce una regola abbastanza specifica da non generale falsi positivi, ma al contempo generica da individuare varianti future.

L'efficacia dell'approccio è stata verificata sperimentalmente, utilizzando 1.5 milioni di applicazioni Android, il più grande dataset mai utilizzato in una ricerca sul malware Android. Inoltre sono stati rilasciati due framework *YaYaGen* e *YaYaGenPE*, per la generazione automatica di signature rispettivamente per Android e Windows. Infine, a partire da Gennaio 2018, parte dell'approccio proposto è stato integrato in Koodous, una piattaforma di analisi collaborativa per applicazioni Android, sviluppata da Hispasec.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware is a long-standing well-known problem. Since the first computer virus, *Elk Cloner*, in 1982 the malicious-software scenario completely changed. Originally malicious programs were mostly developed for fun, where the complexity of the program was a showcase of the skill set of the author, and they even explicitly showed a message to inform about the computer infection.

As the computer and network infrastructure evolved, malware did too. Today's malware is more focused, either designed for the ultimate profitability, or as a pernicious spying tool supported by the state-sponsored attacks. Being able to promptly identify a malicious behavior is a well-studied, but still an open problem for researchers from both the academia and the industry world.

With the coming of mobile operating systems, iOS and Android above all, new opportunities arise for malware developers. A fully-connected device, in constant proximity of the user, with almost unlimited access to personal data is an extremely valuable target for an attacker. Although mobile OS development did benefit from decades of computer security experience of traditional operating systems, and they introduced strict security measures (such as sandboxing the applications execution, or asking before granting sensible permissions) the attack surface is so vast that the combination of technical vulnerabilities with social engineering attacks is still a powerful, yet effective, attack. Moreover, in the Android ecosystem, the situation is made worse by the open market model, where the fast availability of applications is a priority over the strict security check of the application behavior.

Malware authors exploit executable packing [1] and other code obfuscation techniques [2] to generate a large number of variants of the same malicious application. As a consequence antivirus (AV) software are struggling to keep their signature database up-to-date, and AV scanners suffer from a considerable quantity of false negatives. However, while malware variants can be generated at a high pace, they are likely to perform similar malicious activities when executed. One possible solution would be to automatically cluster such applications in families and focus the manual analysis on few archetypal samples, with the underlying assumption that malware bearing significant similarities are likely to derive from the same code base. Eventually, if a large number of malware belonging to the same family is identified, it may become possible to define a generic behavioral signature able to detect future variants with reduced false positives and false negatives.

In this thesis we tackle two well-known, but yet unsolved, problems in the malware research domain: the clustering of vast dataset of applications and the automatic signature generation of a malware family.

Firstly, we introduce a scalable semi-supervised system for the analysis of massive malware datasets based on careful feature engineering, and a standard density-based clustering algorithm. Then, starting from a cluster of malicious samples, we propose an algorithm that automatically generates a *family signature*. Thanks to exact and heuristic evaluations, such rules are *intelligible* and appear *reasonable* to human experts. Moreover, the algorithm guarantees zero false positives in the existing dataset, and limits the possibility of false positives in the future. The procedure includes an evolutionary-based approach, based on the *Selfish Gene* algorithm, to optimize automatically generated signatures in order to further decrease the number of false negatives, and detect future malware variants. The proposed algorithms are implemented in two frameworks, called *YaYaGen* and *YaYaGenPE*, to automatically generate signatures for Android and Windows binaries.

Finally, the thesis includes a study on one the most prominent threat in the Android ecosystem: the Android Banking Trojans, that is applications written with the specific purpose of stealing confidential information from victims bank accounts and on-line payment services.

Chapter 2

Background

2.1 Android Malware

Since August 2010, when the first Android malware *FakePlayer* was released, the number of new malicious samples steadily increased [3]. After eight years, malware programs are hundreds of times bigger than the old *FakePlayer*, hide their presence, use sophisticate anti-analysis tricks, and they can even secretly communicate through complex anonymous networks.

Differently from the iOS Apple Store, Android offers an open market model, where millions of applications are downloaded every day. Although applications undergo a mandatory review process before being published in the official Play Store, in order to confirm their compliance with Google policies [4], other third-party markets do not. Hence, a typical pattern among malware developers is to *repack* popular applications from Google Play Store, add malicious features, and finally distribute them to third-party app-stores with loose security checks, leveraging apps popularity to accelerate malware propagation. Moreover, automatic checks used by Google Play Store can be bypassed too, as shown by the malware campaign found by Check Point in 2017, where 41 malware apps from Google Play infected millions of devices [5].

In the personal-computer ecosystem, malware developers commonly exploit executable packing and other code obfuscation techniques to generate a large number of polymorphic variants of the same malicious application [1, 6]. As a consequence antivirus (AV) software are struggling to keep their signature database up-to-date, and AV scanners suffer from a considerable quantity of false negatives [7]. Malware developers commonly reuse and customize the code to fit different needs. For example, a developer may reuse the rootkit installation code, while replacing the modules that provide network connectivity to a Command-and-Control server [8].

By the end of 2010s, the Android ecosystem is facing a similar scenario, although the situation is made worse by the simplicity of malicious repackaging [9]. This is an alteration of the original application installation package (i.e., the `APK` file), where legitimate applications are reverse engineered, modified to include malicious code, signed one more time, and eventually distributed in open markets. Since applications mainly consist of bytecode, reversing the original application and adding new portions of code is relatively easy to implement, and ad-hoc tools exist to assist the procedure [10, 11].

2.1.1 Koodous

Koodous is an open platform for the analysis of Android applications which combines several state-of-the-art tools with the social interactions among analysts. Started in 2014, in 4 years it became the largest open repository of Android applications: its databases contain more than 40 millions of applications, among which more than 7 millions have already been identified as malicious. Fig. 2.1 illustrates the applications submission and detection trend from October 2014, until March 2017.

Koodous provides both analysis service and end-point protection: upon submission, each application is analyzed both statically and dynamically, and the final report is accessible through a web interface specifically designed to help analysts detect new malware threats. Analysis tools include a custom version of Androguard [12], CuckooDroid[1] and DroidBox [13].

Instead of relying on a closed group of expert malware analysts, Koodous takes advantage of an open community to identify malicious applications. Furthermore, in order to guarantee high quality results, manual detections are subject to reputation-based checking. The endpoint protection is guaranteed through an Android application, which backs to the cloud platform to detect most recent threats[2].

[1] https://github.com/idanr1986/cuckoo-droid
[2] https://play.google.com/store/apps/details?id=com.koodous.android

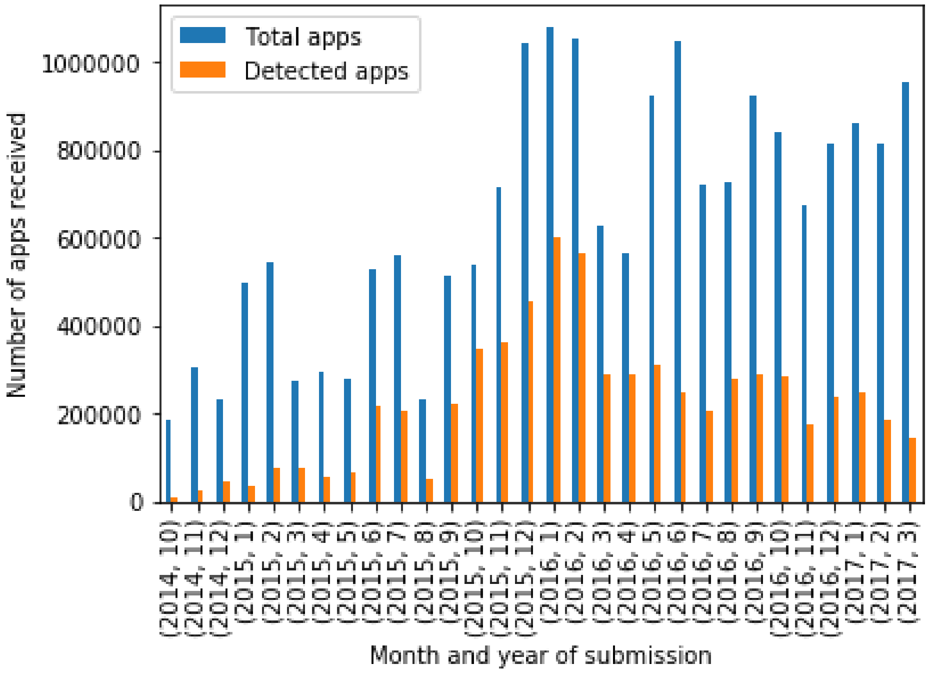

Fig. 2.1 Monthly trend of applications submitted and detected in Koodous from October 2014 until March 2017.

Koodous uses YARA (Section 2.4.1) to describe patterns for detecting malware application: since the creation of high-quality YARA rules requires a considerable effort, the platform also offer the possibility to identify malware through a simpler voting mechanism — an operation referred to as "triage". As of July 1, 2018, more than 2.5 millions applications are detected by triage[3].

Thanks to the collaboration with Hispasec, in this thesis Koodous has been used as a testbed to evaluate the proposed methodology. Moreover, since January 2018, part of the proposed framework has been integrated into the platform.

2.2 Clustering

Clustering is the technique to subdivide objects into groups according to the similarity in their feature set: objects within the same group are more similar to each other than those assigned to different groups. A feature set is used to describe data as

[3] For the up-to-date figure, visit https://koodous.com/apks?search=rating:%3C-1%20%26%20detected:1.

efficiently as possible, and it bridges the semantic gap between individual data objects and partitions with a higher-level of understanding. Similarity is the notion of closeness between two elements, and it is commonly measured through a distance function [14].

Typical cluster models include hierarchical-, centroid- and density-based methodologies. Hierarchical- and density-based approaches are following described: the former historically represents the most popular choice for malware analysis, while the latter is at the base of the proposed methodologies.

Hierarchical clustering

Hierarchical clustering takes a matrix of pairwise distances among objects as input, and produces a *dendrogram* as output, a tree-like data structure where leaves represent original objects, and the length of the edges the distance between clusters [15]. Hierarchical clustering is able to find clusters of arbitrary shapes, and can work on arbitrary metric spaces (i.e., it is not limited to distance in the Euclidean space) [16]. Differently from other, well-known clustering algorithms, such as k-means and DBSCAN, that only provide the clustering output, the resulting dendrogram shows a representation of how clustering evolved.

Two types of hierarchical clustering algorithms exist: divisive (top-down) and agglomerative (bottom-up). Divisive algorithms initially consider all elements in the same cluster and then proceed by splitting points into groups. On the other hand, agglomerative algorithms initially treat each element as a singleton cluster and then proceed by merging them.

Hierarchical agglomerative clustering (HAC) techniques use various criteria to decide locally, at each step, which clusters should be merged (or split for divisive approaches): the definition of cluster proximity differentiates the various agglomerative hierarchical approaches. Single and complete linkage, group average and Normalized Compression Distance (NCD) represent a few commonly used techniques to measure cluster proximity. Single linkage defines the distance between two clusters as the minimum distance between any two members of the cluster, while complete linkage is the exact opposite, characterizing the similarity between two clusters as the similarity between the two furthest elements inside each cluster. Both linkage typologies are local merge criteria, as they define similarity between two clusters by only considering a single element from each cluster: consequently, they can both lead to undesirable clusters. On the other hand, NCD uses the ability of compression functions (e.g., bzip2, gzip) to find similarities between two objects to define a distance between them [6].

Finally, to retrieve a flat clustering from a dendrogram, it is needed to decide at which level stop the merging of the clusters. Multiple general-purpose methods are described in the literature, such as cutting at a predefined constant depth, cutting links that are larger than a certain threshold, and cutting links that are considerably larger than their siblings [17].

Hierarchical algorithms are typically used whenever the underlying application, e.g., the creation of a taxonomy, requires a hierarchy. However, agglomerative hierarchical clustering algorithms are computationally and storage expensive, respectively $\mathscr{O}(n^2 \log n)$ and $\mathscr{O}(n^2)$ [18], while divisive clustering with an exhaustive search are computationally even worse $\mathscr{O}(2^n)$.

Density-based clustering

Density-based approaches locate regions of high density, surrounded by regions of low density, with Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [19] being the most well-known technique of this kind. Density is evaluated on user-specified parameters ε and *min points*. A dense region in the data space is a n-dimensional sphere with radius ε and at least *min points* objects inside. DBSCAN iterates over data objects in the collection by analyzing their neighborhood and classifies objects as being (i) inside a dense region (a core point), (ii) on the edge of a dense region (a border point), or (iii) in a sparsely occupied region (a noisy or outlier point). Any two core points that are close enough (within a distance ε of one another) are associated to the same cluster. Any border point close enough to a core point is put in the same cluster as the core point. Finally, outliers are those far from any core point. Since outliers are isolated, the algorithm does not produce a complete clustering.

Differently from other clustering methods, density-based algorithms can effectively discover clusters of arbitrary shape and filter out outliers, increasing cluster homogeneity. Additionally, the number of expected clusters to be found in the data is not required; in many practical cases, such as the automatic discovery of malware families, the number of clusters is hard to guess a priori, as the goal is to discover groups of similar applications without any prior knowledge about their composition. In low-dimensional spaces, the time complexity of DBSCAN can be as low as $\mathcal{O}(n \log n)$, while its space requirement is $\mathcal{O}(n)$, making it applicable to large datasets.

In 2013, Campello et al. [20] proposed a new density-based algorithm that converts the original DBSCAN into a hierarchical clustering algorithm. For its speed and robustness, Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) is ideal for exploratory data analysis [21]. Matter-of-factly, this technique performs DBSCAN over several values of ε, and integrates the results to find a clustering that provides the best stability over ε. This allows HDBSCAN to find clusters of varying densities, and be more robust to parameter selection. Moreover, HDBSCAN also supports the Global-Local Outlier Score from Hierarchies (GLOSH) outlier detection algorithm: during the fitting phase, each data point is associated to a score that represents its likelihood of being an outlier; at the end of the process, outliers are selected via upper quantiles [22].

2.2.1 Clustering Evaluation

As clustering is an unsupervised learning task, analyzing the validity of its results is intrinsically hard due the difficulty of establishing a *ground truth*. Indeed, clustering itself is inherently ill-posed, in the sense that there is no single criterion that measures how well a clustering of the data corresponds to the real world [23]. Cluster validity analysis often involves the use of *subjective criteria of optimality* specific to a particular application. Therefore, no commonly accepted standard of validating the output of a clustering procedure exists [24].

In real-world applications, it is often completely infeasible to manually investigate the results of a clustering, making necessary the definition of automatic measures [14]. Helpful metrics to determine the quality of a clustering process are commonly classified in *internal* and *external* indexes.

Internal indexes evaluate both cluster *cohesion* (e.g., compactness and tightness), which determine how closely related the objects in a cluster are, and cluster *separation* (e.g., isolation), which determine how distinct or well-separated a cluster is from others. Internal indexes are often called *unsupervised measures*, since they only use information derived from the data set.

On the other hand, external indexes, known as *supervised measures*, use a reference set as a means of quality control for the setup of the clustering algorithm [14].

For instance, the *Adjusted Rand Index* (ARI) computes a similarity measure which consider all pair of samples, counting pairs that are assigned in the same or different clusters in the predicted and true clusterings. The *Homogeneity* index measures if each cluster contains only member of a single class, while *Completeness* if members of a given class are assigned to the same cluster. Finally the *V-measure* is the harmonic mean between homogeneity and completeness. All of them are bounded score between 0.0 and 1.0, where 1.0 stands for a perfect and complete labeling.

Malware clustering validation

In the field of malware analysis, clustering validation is made further complex by the intrinsic difficulty of establishing a reliable ground truth. Firstly, malware analysis is challenging and it gets more difficult when anti-analysis, triggering sequences and dynamic code loading techniques are in place. Secondly, not even a manual categorization would provide a reliable partition, since most of the malware could not be unequivocally assigned in categories; not to mention the unrealistically high amount of time it would require.

As a reference set is not available, one possibility is to take advantage of labels assigned to each malware sample by several antivirus scanner. The availability of services that specifically provide these results, e.g. VirusTotal[4], eases the procedure. However, there is an intrinsic complexity in defining a unique labeling schema, since most of the malware result in being marked as belonging to one malicious category only. As a matter of fact, Bailey et. al. [25] showed that antivirus labeling fails in satisfying three fundamental criteria: consistency among different products, completeness in malware tagging, and conciseness in label semantics. One possible explanation is that signatures used in the malware-matching algorithms mostly evaluate static properties of the binary, in contrast to behavioral properties: the result is that the families found using static features might be quite different from the

[4] https://virustotal.com/

ones established using behavioral features. Moreover, different AV products apply different criteria and granularity to rule generation, resulting in inconsistent results.

Despite the complexity and intrinsic challenges of the procedure, given the importance of automatically building a malware reference dataset to evaluate clustering results, the problem was directly tackled in different researchers, such as VAMO [26] and AVclass [27].

In the literature of malware clustering, several techniques have been proposed. In [28] and [29], precision and recall have been used as a way of determining the quality of the proposed feature set by conducting analysis on an already labeled dataset. However, in practice this approach requires to define a manual mapping between labels assigned by different AVs. Nevertheless, as the dataset size increase this method becomes hardly sustainable and quite costly. In the same way, ClusTheDroid [14] used a reference set developed through manual analysis [30].

On the other hand, Apel et al. [6] choose to take into consideration the amount of “shared behavior” that can be found among different analysis traces within the same cluster of applications. In practice, each system call is modeled as a single character, and the evaluation is computed in linear time finding all substrings in a generalized suffix tree, using the algorithm described in [31]. The main limitation of this technique is related to the choice of the reference dataset, since Apel et al. use an artificial dataset starting from three real-world malware traces, then divided into blocks of system calls and randomly permuted.

Differently, Perdisci et al. [16] tackle the problem of analyzing the validity of malware clustering results without exploiting any manual mapping of AV labels. Their approach is based on a measure of cohesion and separation of each cluster, in terms of agreement between labels assigned to the malware samples by cluster analysis and those assigned by multiple AV scanners. However, since AV labels have been shown to be inconsistent [25], the measures of cluster cohesion and separation only give an indication of the validity of the clustering results.

2.2.2 Clustering applied to malware analysis

The first attempt to automatically group computer malware based on their behavior dates back to Lee and Mody [32], who use a sequence of runtime events (e.g., registry and file system modifications) to cluster similar programs. As a similarity measure, they choose a variant of the *edit* distance, resulting demanding in term of computational resources, since it has a computational complexity $\mathcal{O}(n^2)$ in the number n of features.

Later, Bailey et al. [25] propose a system for automated malware classification and analysis as a remedy for the inconsistent and incomplete labeling that commonly affect traditional antivirus. By applying *single-linkage* Hierarchical agglomerative clustering (HAC) with Normalized Compression Distance (NCD) and using *inconsistency measure* as a cutting criteria, Bailey et al. are able to automatically categorize malware profiles into groups that reflect similar classes of behaviors in terms of system state changes. While results are generally affected by the restriction of dynamic analysis, for the first time they introduce the idea of "detection through clustering", exploited in our proposed framework.

In their work, Apel et al. [6] study which combination of metrics (i.e., *Edit Distance*, *Approximated Edit Distance with Blockwise Hashing*, *NCD* and *Manhattan Distance*) and n-gram features are mostly appropriate for determining relations between malware samples. They define three different criteria to support their evaluation (i.e., appropriateness, computable efficiency and local sensitiveness), using *single-linkage* HAC as clustering algorithm. Experimental results show that Manhattan Distance along with 3-grams deliver the best results, while NCD and Edit Distance generally perform poorly.

Neither Lee and Mody [32], nor Bailey et al. [25] have any specific solution to large-scale clustering. On the other hand, Bayer et al. [28], Rieck et al. [33], and Jang et al. [29] directly address the problem of managing large datasets, developing methods to scale the clustering process.

Bayer et al. [28] propose a scalable malware clustering approach using a combination of approximate and hierarchical clustering with Local Sensitive Hashing (LSH) [34] to significantly reduce the number of distance computations. By extending Anubis [35], they are able to extract detailed behavioral-reports based on taint tracking results and network captures from malware execution. In particular, the taint engine allows them to map low-level operations (e.g., system calls) to operating system objects (e.g., registry keys and files). By deploying LSH, Bayer et al. are capable of clustering 75,000 samples in less than 3 hours. By contrast, Rieck et al. [33, 36] proposes an incremental approach, where they alternate a prototype-based clustering algorithm with a classification step, eventually reducing the runtime complexity by performing clustering only on representative samples.

Jang et al. [29] develop BitShred as remedy to the problem of clustering large data sets with high-dimensional feature sets. They propose to use feature hashing to reduce the dimensionality of high-scale feature sets, while reducing the computational cost of the calculation of the Jaccard index using an approximated version that exploits bit-vector arithmetic. However, since BitShred simply relies on a static analysis approach, results are susceptible to binary level obfuscation.

In 2010, Perdisci et al. [16] propose a network-based version of a behavioral malware clustering system, relying on a three-step clustering refinement process, starting from the analysis of malicious HTTP traces. The first phase consists in a *coarse-grained clustering* where malware samples are grouped together according to simple statistical similarities; subsequently, a *fine-grained clustering* further splits samples considering structural properties of HTTP queries. In the final step, fine-grained clusters whose centroids are close to each other are merged together. The system is tested on HTTP traces generated from 25,000 applications using single-linkage HAC and the Davies-Bouldin (DB) validity index [37] as cutting criteria. While the underlying idea of a multi-step clustering refinement process is quite interesting, this practically results in the biggest limitation to the scalability of their work. Moreover, Perdisci et al. limit behavioral analysis to HTTP-based malware only, which in practice can be easily bypassed by using an encrypted protocol (e.g., HTTPS).

In 2013 Hu et. al [38] present MutantX-S, focusing on malware comparison and triage on a large scale. Their system falls into the static-analysis category, since it relies on features extracted from the malware instructions. MutantX-S can efficiently cluster a large number of samples into families based on program static features, by extracting N-gram features directly from the x86 opcode sequences and exploiting a feature hashing technique to reduce features dimensionality, thus significantly lowering the memory requirement and computation costs. MutantX-S adopts the same prototype-based algorithm of [36] because of its efficiency and explicit expression of malware features.

In the Android context, ClusTheDroid [14] is the first research to combine behavioral analysis and clustering to specifically target Android malware. The goal is both to develop a tool, and to evaluate clustering alternatives. Finally they focused on *single and complete linkage* HAC, using a feature set composed of 38 numerical quantities extracted from the CopperDroid [39] report, and weighted according to a three-level interpretation of malware behaviors.

2.3 Evolutionary Strategies

Evolutionary algorithms are based on the idea that given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection, known as survival of the fittest, which also generates a raise in the fitness of the population. The fitness is the result of a quality function applied to an individual of the population (i.e., candidate solution) to evaluate it. On the basis of these fitness values, some of the best candidates are chosen to seed the next generation by applying recombination and/or mutation. Recombination is an operator that is applied to two or more selected candidates (the so-called parents), producing one or more new candidates, the children. On the other hand, mutation is applied to one candidate individual and results in a new one. This process can be iterated until a candidate solution with sufficient quality is found or a previously set computational limit is reached.

There are two main forces that form the basis of evolutionary systems: variation operators, that create the necessary diversity within the population, and selection, that acts as a force increasing the mean quality of solutions in the population. The combined application of variation and selection generally leads to improving fitness values in consecutive populations [40].

2.3.1 Selfish Gene

In canonical *evolutionary algorithms* (EAs) [41], an *individual* encodes a candidate solution and the set of all individuals that have a role in the evolutionary process is called *population*. In *estimation of distribution algorithms* (EDAs) [42], on the contrary, candidate solutions are not explicitly stored, but the population contains the distributions, and possibly the relationships, of the variables.

The *Selfish Gene* algorithm (SG) is an EDA proposed more than two decades ago [43], it was inspired by a tough experiment suggested by Richard Dawkins in his celebrated book [44]. It exploits an univariate discrete probabilistic model—a vector that stores the marginal probabilities of allele values for each gene position, independently of other gene positions—and is akin to other algorithms, such as the equilibrium genetic algorithm (EGA) [45], the population-based incremental learning (PBIL) [46], the univariate marginal distribution algorithm (UMDA) [47], and remarkably similar to the compact genetic algorithm (cGA) published the same year [48].

Like the cGA, the SG algorithm iteratively samples two solutions at a time and conducts a competition between them. It then updates the probabilistic model by rewarding the allele values contained in the winner and penalizing the allele values contained in the loser of the competition. The strength of the reward/penalty for each gene position i is determined by a parameter ε_i, which usually depends on the number of different alleles that can occupy the i-th locus.

The original SG was quite simple to implement and efficient in finding optima, yet far more robust than pure hill climbing. It was exploited by practitioners in some real-world applications, such as CAD problems [49], by scholars for various test benches [50], and few new approaches derived from it [51]. In 1999 it was enhanced to tackle highly deceptive functions at the expense of a significant loss in performance [52].

2.4 Signature

Early AV products used the hash value of an application to detect malicious software. However, every modification in the source code, as tiny as one byte, results in a detection evasion. Today's signatures are pattern-matching rules commonly defined on static or dynamic properties of applications under analysis and, even though they are assisted by heuristic and AI-based solutions, still represent the most reliable (i.e., with the lowest false positives) antivirus technology.

Today's there are two main standard languages used: ClamAV [5] and YARA [6]. In this thesis we will focus on YARA, as it is more flexible and easy to adapt to different platforms and analysis through custom modules.

[5] https://www.clamav.net/

[6] https://github.com/VirusTotal/yara

2.4.1 YARA

YARA is a pattern matching tool, designed to provide a fast matching between signatures (i.e., YARA rules) and multi-platform binaries. YARA has been described to be the equivalent for binaries of *what is SNORT to network traffic*. The software has been originally developed by Víctor M. Álvarez [53], and it is now owned by VirusTotal.

The name YARA indicates both the signatures, also known as YARA rules, and the name of the tool to match a signature with a binary. The tool internally compiles the rule, and implements an efficient pattern matching mechanism using a custom stack-based virtual machine. The output is the list of the rules names that matched a particular binary.

One of the biggest advantages provided by YARA is the flexibility to extend core functionalities through custom modules written in C [7]. YARA already includes modules to correctly parse and process the PE, ELF, MACHO and .NET file formats, but custom ones can be used to enrich the semantic of the detection, introducing platform-specific matching attributes. For instance, the PE module has been designed to parse the PE header of an executable or DLL for Windows, and it provides specific attributes to match corresponding fields of the executable header.

YARA rules are language-specific signatures, and are composed of three main section. The first one, *meta* is where additional information is added. This section is skipped during the rule evaluation, and it is commonly used only to enrich the rule with a description of the desired behavior and detected samples. The strings section is where hexadecimal or strings pattern are placed. Each pattern is associated to a variable, which is then used in the last section, *condition*, to express the matching logic. In case custom modules are used, this section is used to specify the usage of those attributes. An example of YARA rule with the PE module is presented below.

One major limitation of YARA is that rules are applied to the binary as it is, without any prior transformation. This limits the rule detection effectiveness in case of packed samples, and any obfuscation techniques.

Interestingly, YARA rules have several applications other than malware identification: among the others, rules can be used to identify indicators of compromise,

[7] https://yara.readthedocs.io/en/v3.10.0/writingmodules.html

shellcode, compilers, and packer software. Moreover, open and public repositories of YARA rules exist, and are daily updated.

```
import "pe"

rule silent_banker : banker
{
    meta:
        description = "This␣is␣just␣an␣example"
        thread_level = 3
        in_the_wild = true
    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
    condition:
        ($a or $b or $c) and pe.number_of_sections == 1
}
```

2.5 Automatic Signature Generation

A number of prior works propose systems to automatically generate different types of network signatures to identify malicious traffic.

Honeycomb [54], Autograph [55], and EarlyBird [56] propose the generation of signatures comprising a single contiguous string (i.e., token). Later on, PAYL [57], Nemean [58], Hamsa [59] and Botzilla [60] introduce more complex methods based on the token subsequence signatures.

Other researchers like ProVex [61], AutoRE [62], ShieldGen [63], and [64] also tackle the problem of automatically generating network signatures, although their applicability is specific to the network traffic detection.

In 2005, Newsome et. al. introduces Polygraph [65], a system which exploits the Token-Subsequence algorithm to automatically obtain IDS signatures to match polymorphic worms. Polygraph is tested against three real-world exploits and is able to successfully generate HTTP and DNS signatures with a low false positive rate.

Perdisci et al. [16] also tackles the problem of automatically generate network signatures for cluster centroids, with the aim of deploying them into an IDS at the edge of a network in order to detect malicious HTTP traffic. Since malware samples may contact legitimate websites for malicious purposes, instead of pre-filtering HTTP traffic against legitimate websites, authors apply a pruning process by testing the signature set against a large dataset of legitimate traffic, while discarding signatures that generate false positives, although such an approach is as effective as it is the legitimate traffic available.

In the Android context, Faruki et al. [66] propose *AndroSimilar*, a statistical signature-based solution that generates variable-length signatures for the application under test and identifies malware on the basis of a similarity percentage with a dataset of known malicious samples.

Another approach is presented in *DroidAnalytics* [67], a signature-based analytic system, which extracts and analyzes applications at opcode level. Firstly, a three-level signature (i.e., methods, classes, application) is generated by combining the API call traces, then the malware is associated to a family according to its malicious content.

While [66] shows robustness against control-flow obfuscation, junk method insertion and string encryption, [67] could fail in the detection of repackaged malware. On the the other hand, both solutions are affected by a high false-positive rate due to the wrong choice of signature patterns available in both malicious and benign applications.

2.5.1 Tools

Along with the proposed approaches in the literature, several tools have been developed to automate the process of generating the signatures, balancing the need of generality to catch future samples, with the requirement of avoiding false positives detections. Given that YARA is the de-facto standard language used to write signature, several tools directly tackle the signature generation into the YARA format.

YaraGenerator

In 2013, Chris Clark develops *YaraGenerator*[8], a Python program which automatically generates YARA rules by sampling a small subset of common strings between malware, while blacklisting goodware ones. Although the tool is designed to work with any type of malicious file, in order to increase the efficacy of the results, specific dataset of goodware strings are available for several file formats (e.g., Windows executable, PDF, email and office document).

YarGen

YarGen[9] is a Python tool developed by Florian Roth to automatically generate YARA rules by combining the topmost malware strings, while removing those that also appear in goodware files. By using fuzzy regular expressions, each malware string is assigned a score proportionally to the inverse of its frequency, and the "Gibberish Detector" allows to select real language over character chains without any meaning. The tool also exploits a naive-bayes-classifier to classify candidate strings, avoiding compression or encryption garbage in favor of more generic strings. Finally, each rule is created by combining the 20 strings with the highest score. The result of the generation process may be a single rule, specific to one sample, or a *super rule*, catching malware variants and groups.

YaBin

YaBin[10] is a tool developed by Christopher Doman, from AlienVault antivirus company. The signatures are directly generated using the YARA syntax using a simple combination of function prologues extracted from a lightweight static analysis of the executable code.

The function extraction is based on the assumption that different compilers use different unique pattern to define the function prologue, hence using a simple set of regular expression, it is possible to get the binary functions without the need of disassembling the entire binary. Then, the efficacy of the tool relies on the availability

[8] https://github.com/Xen0ph0n/YaraGenerator
[9] https://github.com/Neo23x0/yarGen
[10] https://github.com/AlienVault-OTX/yabin

of a huge whitelist of about 5 millions function prologues extracted from goodware binaries.

YaBin does not perform any optimization of the rule, hence if the samples slightly differ, one rule for each sample is usually generated. Finally, As the author declares, the tools is designed to work on unpacked binary samples, otherwise the signature generated could lead to false positives as it could match the packed code.

BASS

BASS [68] is a tool developed by Cisco Talos to generate ClamAV logic signatures. Although the ClamAV syntax differs from YARA, signatures can be usually converted in the two formats.

BASS uses a very detailed approach to find shared portion of code among samples, exploiting a clustering phase through binary-diffing technique and the Longest-Common-Subsequences (LCS) algorithm to generate the final signature.

The complexity of the algorithms involved in the procedure is the main downside of the proposed approach, which finally limits to few tens the number of samples that can be processed together. Indeed, the clustering phase, which is based on a divisive hierarchical approach, requires to compute the similarity among all the pairs of binaries. Then, the LCS algorithm finds the sequence of not necessarily contiguous characters that appear in the same relative order in both binary sequences.

Generated signatures are checked against an internal database of goodware, and in case of a false positive, the entire process is repeated, but excluding the unwanted matching.

Chapter 3

Proposed approach

Part of the work described in this chapter has been previously published in *"Countering Android Malware: A Scalable Semi-Supervised Approach for Family-Signature Generation", IEEE Access, 2018 [69].*

3.1 Problem Statement

The growth of malware created a major challenge for AV vendors to efficiently handle new samples and accurately label them. Due to the practical impossibility of manually analyzing thousands of suspicious samples received every day, a large fraction of them is left unlabeled, delaying the signature generation.

While malware variants can be generated at a high pace, they are likely to perform similar malicious activities when executed. Hence, one possible solution is to automatically cluster the applications into families and focus the manual analysis on few archetypal samples, with the underlying assumption that malware bearing significant similarities are likely to derive from the same code base [38]. Furthermore, new samples that belong to a known family can be automatically labeled, and existing signatures and other mitigation techniques could be easily extended to cover the new threats too.

Eventually, if a large number of malware belonging to the same family is identified, it may become possible to define a generic behavioral signature able to detect future variants with reduced false positives and false negatives [16]. Therefore, a precise and robust clustering is crucial to help AV companies categorizing the large amount of samples, avoiding duplicate work, and allowing analysts to prioritize their limited resources on novel and representative samples [29, 28].

In this chapter, we describe a semi-supervised system for the analysis of massive datasets of malicious applications. We introduce a platform that is able to suggest new families of applications to human experts, and which also generates an intelligible signature, in the form of a YARA rule [70], to identify family members with high precision. The proposed methodology explicitly minimizes false positives, a business hazard and a reputation blow for AV vendors. Finally, the approach aims to alleviate human experts from the burden of manually inspecting thousands of malware, while letting the system take critical decisions.

The main contributions of the proposed approach can be summarized as:

- We introduce a scalable system for the analysis of massive malware datasets based on careful feature engineering, and a standard clustering algorithm. The mechanism is demonstrated to be robust and able to overcome the well-known limitations of traditional signature-matching mechanisms.
- We propose an algorithm that, starting from a cluster of samples, generates its *family signature* as a YARA rule. Thanks to exact and heuristic evaluations, such rules are *intelligible* and appear *reasonable* to human experts. Moreover, the algorithm guarantees zero false positives in the existing dataset, and limits the possibility of false positives in the future.
- We present an evolutionary-based approach to optimize automatically generated signatures in order to further decrease the number of false negatives, and detect future malware variants.
- We have implemented the proposed algorithms in two frameworks, called *YaYaGen* and *YaYaGenPE*, to automatically generate signatures for Android and Windows binaries.

Since the 2000s, researchers from the academia proposed several approaches based on machine learning aiming at completely replacing humans in the malware analysis process. In most of the cases, such proposals fell back into mere classification, also known as *supervised* machine learning. The drawbacks included the need of large amount of accurately labeled, i.e., already analyzed, data, and hard to control false positives, a major cause of concern for all the AV vendors. As a result, AV companies developed systems mostly based on the reliable signature-detection mechanism. Even though signatures suffer from the so-called “specificity” problem, and new ones need to be frequently generated, they have been demonstrated effective, scalable, and almost unaffected by false positives.

The proposed framework is *semi-supervised* and introduces essential improvements in the identification of similar applications and the generation of family signatures. It combines the scalability of fully automatic techniques for clustering and the optimization of new family signatures, while it exploits manual analysis, inherently more flexible and accurate, in few crucial steps, such as the validation of newly discovered malware families.

Traditionally, the effort of automatically classifying and analyzing malware focuses on *content-based* signatures that specify binary sequences. Indeed, content-based signatures are inherently vulnerable to malware obfuscation: even if all variants of a malicious application share the same functionalities and exhibit the same behavior, they can have tiny different syntactic representations. As a consequence, a huge number of signatures needs to be created and distributed by AV companies.

On the other hand, a rule that automatically identifies the behavior of a family of samples would be the first step towards the creation of true *family signatures*. Such a signature would match all samples of a family, and would significantly help to reduce the number of signatures required to cover it. Moreover, as new samples could be mapped to a family behavior already known, the time and effort required to analyze and reverse engineer new samples would be reduced.

Differently from the previous approaches, the proposed system generates effective, precise and descriptive rules using the properties directly extracted from both static and dynamic analyses. While aiming at reducing false positives and false negatives, it also exploits a heuristic measure to emulate how expert analysts write existing signatures.

3.2 Clustering

3.2.1 Iterative clustering

Clustering provides a mechanism to automatically categorize applications into groups that reflect their similarity, both in source code and runtime behavior. Ideally, the clustering algorithm to use should meet the following requirements:

- The algorithm should be able to find clusters of any shape and it should be able to identify outliers, because real data has outliers.
- The number of clusters should not be defined a priori, because the composition of the data is not known a priori and there is not preprocessing step that gives any hint on the number of different families.
- The algorithm should be able to scale in order to meet the necessity of processing millions data points in input. As a matter of fact, AV vendors are currently required to process about 1 million new application every day to find new malware samples, and existing malware dataset consist of several millions of samples.

HDBSCAN, a density-based algorithm, was chosen as it fits most of previous requirements.

Density-based clustering algorithms locate region of high-density in the feature space, moreover they can effectively discover clusters of arbitrary shape and filter out outliers, eventually increasing cluster homogeneity. Additionally, the number of expected clusters to be found in the data is not required: our aim is to discover groups of similar applications without any prior knowledge about their composition, otherwise the number of clusters is hard to guess a priori.

Differently from most of the previous works [25, 6, 28, 16, 14] that rely on the HAC algorithm (which is both computationally and storage expensive, respectively $\mathcal{O}(n^2 \log n)$ and $\mathcal{O}(n^2)$ [18]), in low-dimensional spaces HDBSCAN has an average complexity of approximately $\mathcal{O}(n \log n)$, while its space requirement is $\mathcal{O}(n)$, making it applicable to moderately large datasets [71]. Furthermore, differently from [36], we devise an iterative clustering approach where HDBSCAN is iteratively applied over the entire dataset, without the needed of alternate any classification step, finally discovering precise families of applications with a shared behavior.

As the number of samples in malware datasets is in the tens of millions, through the *iterative* process the original dataset **D** (Formula 3.2) is divided into m chunks (Formula 3.1) $\mathbf{d}_i$ of fixed size N. The methodology used to divide the original data into the m chunks does not influence the results.

$$m = \left\lceil \frac{|\mathbf{D}|}{N} \right\rceil \tag{3.1}$$

$$\mathbf{D} = \bigcup_{i=0}^{m-1} \mathbf{d}_i \tag{3.2}$$

The parameter N balances the quality of the results with the time required for the analysis, and can be set experimentally according to the available resources.

HDBSCAN is applied to each chunk of data $\mathbf{d}_i$ finding, at each step, a set of clusters $\mathbf{c}_i$ and a set of outliers $\mathbf{o}_i$. Finally, all the outliers **O** (Formula 3.3) are processed together through another clustering iteration in order to find even those small groups of applications whose samples are spread through several chunks of data. In the end, $m+1$ total iterations are required to complete the process.

$$\mathbf{O} = \bigcup_{i=0}^{m-1} \mathbf{o}_i \tag{3.3}$$

Since the clustering on the first m chunks of data can be executed in parallel, the benefit of the iterative approach is the huge reduction in the analysis time. On the other hand, few applications could be misclassified as outliers and the same group of similar applications could be found multiple times, although, as shown in section 3.2.2, those corner cases do not limit the framework efficacy.

The following description refers to to the clustering of Android applications, although the proposed approach is generic, and it can be applied also to other datasets.

Features selection

An accurate features selection is a crucial step in every machine learning approach. As suggested in [16], we exploit aggregate information: from the analysis result of each application, we extract a subset of “statistical” properties, meant as quantitative measure of a malware behavior. Indeed, we experimentally found that exploiting statistical similarities among applications, rather than “structural" properties which exactly describe the malicious behavior, does not effectively alter the results, while at the same time, significantly reduces the amount of data to process.

Starting from a set of n analysis reports provided by Koodous 2.1.1, each report r_i is translated into a feature vector $v_i = (f_0, \dots, f_{34})$ containing 35 statistical properties extracted from the results of the static and dynamic analysis. These properties are summarized in Table 3.1 and represent the standard type of information extracted in the field of malware analysis.

In more detail, the static analysis performed by Androguard extracts the features from the Manifest file (i.e., number of activities, permissions, receivers, filters), and the source code analysis. The former allows to unveil similarities among applications based on the software architecture used to develop the application, while the latter models each application extracting portions of code related to suspicious API call (e.g., number of calls to SMS API, or IMEI, or other network related methods). On the other hand, the dynamic analysis extracts features that model the application interaction with the surrounding operating system both at file system and network level extracted by DroidBox (e.g., files written, usage of cryptography, SMS sent), and the network information extracted by CuckooDroid (e.g., number of DNS resolved, HTTP requests).

Because the range of each feature is quite different, the dataset is firstly normalized so that the features have mean equal to zero and variance equal to one. Since the choice of the distance to use during cluster analysis is tied to the type and the dimension of selected features, we experimentally found that the combination with the Euclidean distance delivered the best performances.

Table 3.1 List of the 35 statistical properties extracted from the analysis result of each APK file. Features are grouped according to the type of analysis. Static features are extracted using Androguard both parsing the Manifest file and looking for interesting API calls in the decompiled source code. Dynamic features are extracted using DroidBox and CuckooDroid from the dynamic analysis of the application.

Analysis method	Sofware	Statistical property
Parsing Manifest file	Androguard	Filters Activities Receivers Services Permissions
Statically from APK	Androguard	Accounts Advertisement Browser history Camera Crypto functions Dynamic broadcast receiver Installed applications Run binary MCC ICCID IMEI IMSI SMS MMS Phone call Phone number Sensor Serial number Socket SSL
Dynamically	DroidBox	Files written Crypto usage Files read Send SMS Send network Recv Network
	CuckooDroid	HTTP request Hosts Domains DNS

3.2.2 Extending Malware Detection

Starting from millions of samples, the iterative clustering (Section 3.2.1) identifies clusters of strongly related applications. The concept of malware family is not uniquely defined, and it may vary according to the properties used to identify it: a malware *family* can coincide with a single cluster, or it could include among multiple clusters. In the following we identify a malware family as a group of applications that share similar static properties, the same features used in the clustering process. The terms malware families and clusters are used interchangeably.

In some cases, by combining this result with the information already available in the database of an antivirus, like Koodous, the families may be automatically labeled, as they extend either known threats or legitimate software. In the other cases, experts are required to manually evaluate the family, but they need to analyze only few representative samples of the group and not all applications, therefore drastically reducing the time required by the analysis. This process exploits the *clustering assumption* of the semi-supervised learning algorithms, which states that two points which are in the same cluster, that is which are linked by a high density path, are likely to share the same label. In such a way, the partial information of few labels extracted from each cluster can be used to increase the knowledge of all the applications within the same group.

The set of all applications in the Koodous dataset **K** may be partitioned into three subsets $\mathbf{K} = \{\mathbf{S} \cup \mathbf{T} \cup \mathbf{U}\}$ corresponding to the applications detected by signatures (**S**), detected by triage only (**T**), and undetected (**U**); applications detected both by signatures and in the triage phase belong to the **S** set. Such a partition does not reflect a peculiarity of the Koodous dataset. Indeed, the usage of a *staging area* **T**, where suspicious samples which are pointed out in the triage by several techniques (e.g., heuristics, clone detector, suspicious libraries or network traffic) are waiting further analysis, is common in AV laboratories.

As shown in Figure 3.1 It is possible to classify a family according to the different subsets its applications belong to. The resulting seven different types of family correspond to the power set P(**K**), excluding the empty set: $\{ \{\mathbf{S}\}, \{\mathbf{T}\}, \{\mathbf{U}\}, \{\mathbf{S},\mathbf{T}\}, \{\mathbf{S},\mathbf{U}\}, \{\mathbf{T},\mathbf{U}\}, \{\mathbf{S},\mathbf{T},\mathbf{U}\} \}$

- **Type 1** $\{\forall s \in \mathbf{F}^{(1)} \mid s \in \mathbf{S}\}$. The family is composed of applications that have been already detected by YARA signatures. No further action is required,

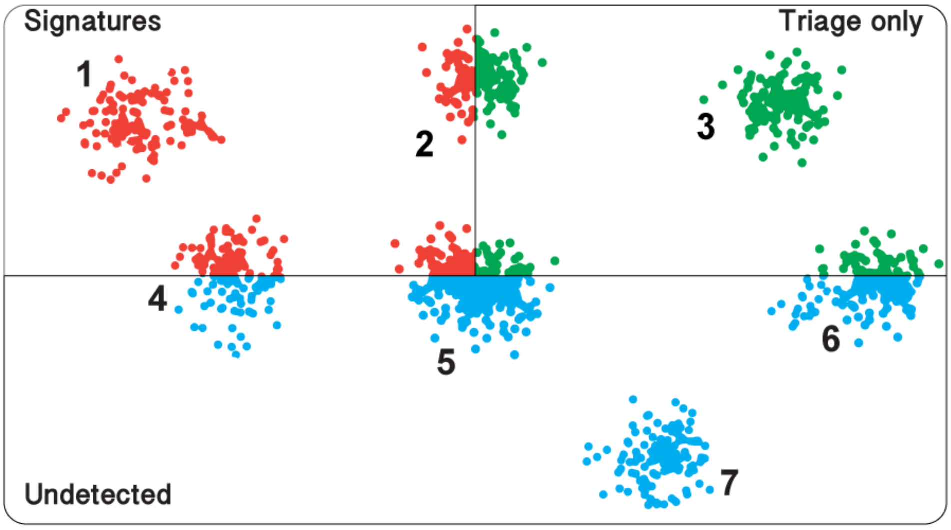

Fig. 3.1 The figure illustrates the subdivision of the applications in database and the seven type of families (i.e., clusters) that can be automatically inferred by the proposed approach. The database is divided in three macro areas according to the type of detection: applications detected by signatures, by triage only, and undetected. Each point in the figure represents an application, and each numbered group represents one of the seven cases identified by the proposed approach.

although the generated family rule may still be effective to generalize the detection.

- **Type 2** $\{\forall s \in \mathbf{F}^{(2)} \mid s \in \mathbf{S} \cup \mathbf{T}\}$. The family includes applications already identified as malicious either by YARA, or during the triage process. The correctness of the detection is either guaranteed by the existing signatures, or by the triage process (i.e., the community votes); thus a new YARA rule matching all the applications in the family can be automatically generated and added to the detection system without further manual check.

- **Type 3** $\{\forall s \in \mathbf{F}^{(3)} \mid s \in \mathbf{T}\}$. The family is composed of applications that have been detected through the triage process only. The correctness of the detection is guaranteed by the triage process, and as in the previous case, a new YARA rule can be automatically generated and added to the detection system without manual intervention.

- **Type 4** $\{\forall s \in \mathbf{F}^{(4)} \mid s \in \mathbf{S} \cup \mathbf{U}\}$. The family combines applications detected by existing signatures with undetected ones. In order to avoid false positives, the correctness of the family must be manually validated before generating a family signature.

- **Type 5** $\{\forall s \in \mathbf{F}^{(5)} \mid s \in \mathbf{S} \cup \mathbf{T} \cup \mathbf{U}\}$. The family combines applications detected both by signatures and by triage only, with undetected ones. As in the previous case, in order to guarantee complete correctness the family must be manually validated before generating a signature.
- **Type 6** $\{\forall s \in \mathbf{F}^{(6)} \mid s \in \mathbf{T} \cup \mathbf{U}\}$. The family combines applications either detected by the triage process only with undetected ones. As in the two previous cases, the family must be manually validated before generating a family signature.
- **Type 7** $\{\forall s \in \mathbf{F}^{(7)} \mid s \in \mathbf{U}\}$. The family is composed of undetected applications, hence no classification can be automatically inferred. However, as all the applications within the cluster show strong similarities, the analysis of few representative samples shall be sufficient to classify the whole cluster as malware or goodware.

Such an approach offers apparent benefits: the need for human intervention is often limited to the simple validation of the discovered family, while the need for full analysis is reduced to few representative samples. The identification of families with only partially detected applications, either by signature or during the triage process, allows to discover false negative and new 0-day malware.

In Koodous, the triage process makes it possible to quickly identify threats without the burden of creating signatures, although it has the drawback of potentially leaving others similar applications undetected. Our frameworks may automatically convert all the knowledge about single, unrelated threats into more reliable signature, potentially able to discover newer variants as well.

Finally, among **Type 7** families, the system is able to identify groups of legitimate software, for example finding applications written by the same developer or using the same framework. This result was proved to be of practical importance to limit and correct false positive detections.

3.3 Automatic Signature Generation

We developed an automatic procedure that, starting from each family of applications identified as malicious, eventually produces a family signature (i.e., a YARA rule) to precisely match them. The program has no requirements on the origin of the set: it could be the result of automatic clustering or manual selection, although the more the applications in the set are related, the more the rule will be able to catch new variants. The system could generate family signatures for legitimate applications as well, but they would be of no use.

The proposed system has been designed in accordance to the following guidelines:

- The algorithm should be scalable, and it should be able to generate a signature to cover hundreds to thousands of samples in input.
- The process to generate a signature should be fast, taking less than five minutes for 100 samples (running on a single core).
- The signature should limit, as much as possible, false positives (i.e., the matching of goodware application).
- The efficacy of avoiding false positives should not be related to the number of samples in input.
- The signature should be able to generalize and catch other malware variants from the same family too.

Authoring an effective signature requires a considerable effort and experience. Good signatures are compact, and they have the ability to generalize, that is, to identify all known variants of the malware and even possible new ones. Moreover, they do not yield false-positive results by detecting non-family members, and finally, they appear intelligible to human experts and are almost self-explanatory.

Traditionally, a signature is defined on unique strings or binary patterns found in malware but not present in legitimate programs. Moreover, approaches like yarGen (Section 2.5.1) or YaBin (Section 2.5.1) strongly rely on the completeness of a database of white-listed strings or patterns. Quite differently, we generate precise, descriptive rules using the structural properties extracted from the static and dynamic analysis of the applications. Indeed, our system identifies an optimal set of clauses that match all the target applications, while yielding to no false positive in the current database. Furthermore, thanks to some heuristics (discussed in Section 3.3.3), the rule has a good ability to generalize, a low risk of detecting false positives in the future, and it appears *reasonable* to the eye of the human experts.

The set covering problem

The idea behind the signature generation is as simple as find the intersection of to set. Starting from two applications, *Sample1* and *Sample2*, each can be represented as a grid of blocks, as illustrated in Figure 3.2, where each block is one of the feature extracted during the application analysis. For the sake of simplicity, the number of blocks in the figure is the same, but this is not a requirement of the procedure. Feature from the same type (e.g., green and orange ones) that are shared among the two samples represent one simple solution to the signature generation problem, as represented in the rightmost features block in the figure.

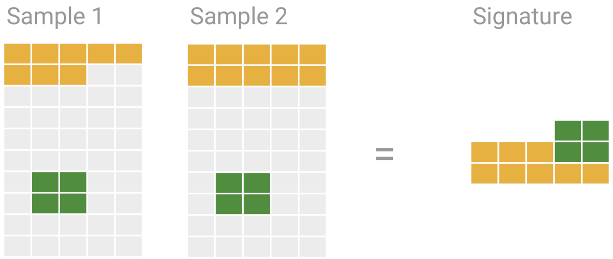

Fig. 3.2 Visual representation of the features from two sample applications *Sample1* and *Sample2*. The rightmost block is one of possible resulting signature.

However, as illustrated in Figure 3.3, real scenarios are much more complex, since several types of features are involved, and no single pattern can be easily extracted from the pool of input samples. One possible solution is presented in Figure 3.4 where five combination of patterns are extracted, and they represent one of the valid solution of the signature generation problem.

More in general, the problem of generating a signature, that cover a set of input samples, can be reduced to a variant of the well-known set cover problem[72]. Differently from the original problem, rules may overlap, hence a samples can be covered by multiple rules. The ultimate goal is to find the minimal set of rules that

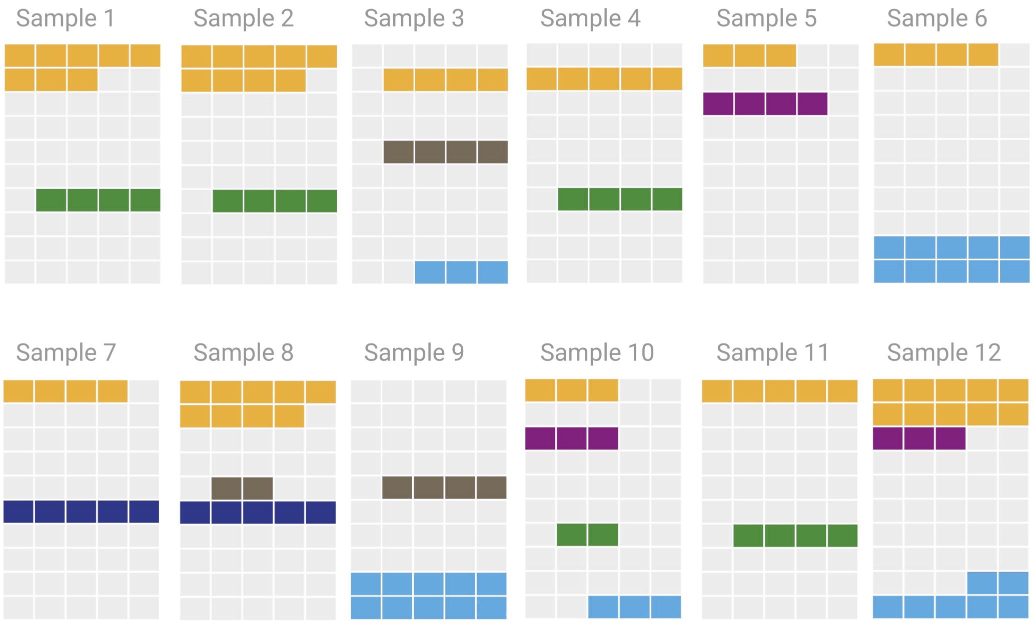

Fig. 3.3 Visual representations of different types of features from an input cluster of samples.

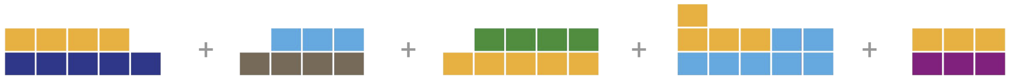

Fig. 3.4 A possible solution for the signature generation problem from the previous example.

cover all the samples, without producing false positives (i.e., rules should not match any sample within the goodware set), but limiting false negatives too.

Formal notation

A signature S can be defined as the d disjunction of n clauses.

$$S = \bigvee_{i=0}^{n} c_i \tag{3.4}$$

A clause c_i is a finite conjunction of m_i literals.

$$c_i = \bigwedge_{k=0}^{m_i} l_{k_i} \tag{3.5}$$

In the present context, a literal l_{k_i} is a single feature resulting from the analysis of the application.

An example of literal is "android permission.INTERNET", that is the Android permission to access Internet, while an example of clause defined by the conjunction of three Android permissions is shown below:

```
"android.permission.INTERNET" and
"android.permission.READ_EXTERNAL_STORAGE" and
"android.permission.SEND_SMS"
```

Introducing the automatic process

The process of automatically generating a signature consists of three steps: a reasonable signature composed of a small number of clauses is generated; the signature is checked against the full database of applications, and false positives are identified; in case of unwanted detections, the generation procedure is run again, but explicitly taking into consideration the false positives discovered in the second step.

Figure 3.5 exemplifies the idea of the process of generation of a signature for two malware m_a and m_b, and two legitimate applications g_a and g_b. In the first phase, the algorithm defines a signature $Y = r$, where r is a single clause composed by the common features between the two malware: $r = m_a \cap m_b$. Indeed, a rule Y detects an application m only if Y is a subset of m: $Y \subseteq m$.

During the second phase, the rule Y is checked against the complete database, where it generates two false positives matching two legitimate applications g_a and g_b. The clause r is therefore *too generic* to be used as a signature.

As it is not possible to find features common to malware that do not matches legitimate applications $(m_a \cap m_b) \setminus g_a = \varnothing$ and $(m_a \cap m_b) \setminus g_b = \varnothing$, the third step generates a signature with the disjunction of two clauses $Y^* = (r \wedge r_a) \vee (r \wedge r_b)$.

3.3.1 A dynamic greedy algorithm

The pseudo code of the algorithm responsible for the signature generation is reported in Algorithm 1: at first it determines a suitable set of clauses (function *Clauses*), then picks a subset of them of variable size to build an optimal family signature (function

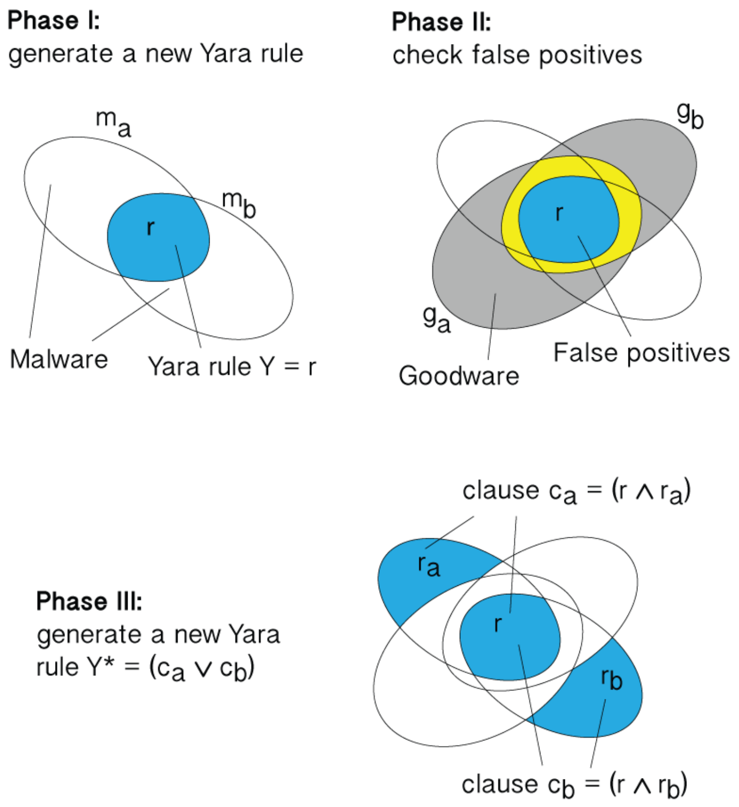

Fig. 3.5 Schema of the process of generation of a YARA rule. In the first phase a signature $Y = r$ is defined for malware m_a and m_b. In the second phase Y is checked against a dataset of goodware (g_a and g_b). Finally, in the third phase, a new signatures $Y^* = (r \wedge r_a) \vee (r \wedge r_b)$ is created to avoid the false positive detection of g_a and g_b.

Clot). Lines 2 and 3 correspond to the first phase of Figure 3.5; line 4, to the second; lines 5 and 6, to the third.

Algorithms 2 and 3 add more details about the procedure: the function *Clauses* extracts the clauses that can be used to build the signature, and it is based on a heuristic algorithm. First, each malware application r_i in the target set R is transformed into a single clause y_i able to detect it using all available literals. Such clauses are not directly usable, but are the starting point of the interactive procedure for building the set of optimal clauses H: in each step, the least generic y_i is selected and compared against all clauses in H calculating the common features z_i; the least generic of these z_i is eventually considered for inclusion in H.

Algorithm 1 Automatic YARA rule generation

```
procedure GENERATESIGNATURE(R)
    $C \leftarrow \mathrm{Clauses}(R, \varnothing)$
    $Y \leftarrow \mathrm{Clot}(R, C)$
    $G \leftarrow \mathrm{GetFalsePositives}(Y)$
    $C^* \leftarrow \mathrm{Clauses}(R, G)$
    $Y^* \leftarrow \mathrm{Clot}(R, C^*)$
    $\mathrm{DumpAsYARARule}(Y^*)$
```

Algorithm 2 Clauses extraction

```
 1: function CLAUSES(R, G)
 2:     $Y \leftarrow \{\mathrm{Features}(r)\ \forall r \in R\}$
 3:     for all $r \in R$ do
 4:         $Y \leftarrow Y \cup \mathrm{SelectedClauses}(r)$
 5:     $H \leftarrow \{\mathrm{Features}(r)\ \forall r \in R\}$
 6:     while $|H| > 0$ do
 7:         $h \leftarrow \mathrm{LeastGeneric}(H)$
 8:         $Z \leftarrow \{\mathrm{CommonFeatures}(h, y)\ \forall y \in Y\}$
 9:         $z \leftarrow \mathrm{LeastGeneric}(Z)$
10:         $F = \{r \in G \mid \mathrm{Det}(z, r) = \mathtt{True}\}$
11:         if $F = \varnothing$ and $z \notin Y$ and $\mathrm{Quality}(z) > T_q$ then
12:             $H \leftarrow H \cup \{z\}$
13:             $Y \leftarrow Y \cup \{z\}$
14:         $H \leftarrow H \setminus \{h\}$
15:     return Y
```

The rationale is to build Y by adding clauses progressively less specific (i.e., checking fewer features), but still usable in signatures. Line 10 computes the set F of application from G detected by the candidate clause; as G is the set of all potential false positives, if F is not null the clause is too generic to be usable. Additionally, the function Quality$(\cdot)$ performs a heuristic evaluation of the clause: if the quality is below a certain threshold T_q, the rule is so generic that it is likely to create false positives in a near future — see 3.3.2 for more details. For each application, few *not-too-generic*, heuristically selected clauses are also included (i.e., r_a and r_b in the example shown in 3.5).

The function *Clot* (Algorithm 3) implements a dynamic greedy algorithm for building the signature as a disjunction of clauses. It iteratively adds one clause to Y from a set C until all applications in R are detected by at least on clause in Y.

Algorithm 3 Clauses selection

```
procedure CLOT(R, C)
    $Y \leftarrow \varnothing$
    $D \leftarrow \varnothing$
    while $R \neq C$ do
        if $\exists r \in R \setminus D : \text{Critical}(r) = \texttt{True}$ then
            $\bar{r} \leftarrow \text{GetCritical}(R \setminus D)$
            $Z = \{z \in C \mid \text{Det}(z, \bar{r}) = \texttt{True}\}$
        else
            $Z = \{z \in C \mid \exists r \in R \setminus D : \text{Det}(z, r) = \texttt{True}\}$
        $Y \leftarrow Y \cup \{\text{MostUseful}(Z)\}$
        $D \leftarrow \{r \in R \mid \nexists y \in Y : \text{Det}(y, r) = \texttt{True}\}$
    return Y
```

In an iterative way, *Clot* first picks out all clauses that detect at least an application not yet detected by any rule, with the only exception that, if an application can be detected by only one clause, that clause is the only one picked. Then the algorithm selects among this group the clause that is able to detect more applications in the original target set R.

An example of an automatically generated YARA rule for the *Syringe* Android malware family is shown below[1]. It may be noted that the statistical features exploited during clustering (Section 3.2.1) are usually not used in the in the rule, as they would result in over-complicated rules hardly understandable by humans.

```
rule YaYaSyringe {
    condition:
        androguard.filter("action.BATTERYCHECK")
        and androguard.permission("SYSTEM_ALERT_WINDOW")
        and androguard.url("http://s.adslinkup.com/v2")
        ...
}
```

[1] The complete version of the rules is available on Koodous at https://koodous.com/rulesets/3243

3.3.2 Rule quality

A heuristic evaluation is used to reduce the risk of false positives in the future and to increase the perceived quality of the rule. We defined a heuristic score $\mathscr{S}(\cdot)$ for a rule, inversely related to its generality. More formally, let associate each literal l to a score $\mathscr{S}^*(l)$ that measures how specific the literal is. The score of a clause c_i is the sum of the scores of the n_i literals composing it: $\mathscr{S}(c_i) = \sum_{k=0}^{n_i} \mathscr{S}^*(l_{i_k})$. The sore of a rule r is the minimum among the scores of its clauses: $\mathscr{S}(r) = \min_{\forall i} \mathscr{S}(c_i)$.

The higher the score, the more a rule is specific and less susceptible to generate false positives. On the other hand, the lower the score, the more a rule will be able to generalize, while being more prone to generate false positive in the future. High quality signatures require an optimal balance between generality and specificity, and this is one of the main challenges in automatic signature generation. We use two threshold $T_{\min}$ and $T_{\max}$, where the lowest is the minimum score that a rule needs to be valid, and the highest is used in the optimization process to avoid overly-specific rulesets.

All the clauses in YARA rules created by expert analysts are valid, that is, the score assigned to literals must guarantee that $\forall r \in \mathbf{R}_{\text{expert}} : T_{\min} \leq \mathscr{S}(r) \leq T_{\max}$. We consider invalid the rules containing a clause mentioning only Android official permissions and intent filters, or containing a clause composed of a single literal, with the exception of accessing an URL that have been detected as malicious by VirusTotal or similar services. Then, we exploit the simplex method as a mean to automatically define $\mathscr{S}^*(\cdot)$ starting from the existing ruleset.

The simplex method is a linear programming technique, which refers to the problem of optimizing a linear *objective function* ζ of m variables x_i subject to a set of n linear inequality constraints. In *standard form*, the problem of finding an optimal set of weights for m literals can be expressed as:

$$\min \zeta = \mathbf{c}^{\mathrm{T}} \times \mathbf{x} \tag{3.6}$$

$$\text{s.t} \; -\mathbf{A} \times \mathbf{x} \geq -\mathbf{b}, \; \mathbf{x} \geq 0 \tag{3.7}$$

where $c_i = 1, \forall i = 1 \ldots m$, since the objective function ζ minimize the number of literals in each clause, $\mathbf{x} \in R^m$ is a vector of m unknown weights, and $b_i = T_{min}$,

Table 3.2 Details about the number of rules, clauses, and unique clauses analyzed to find the optimal score for each literal.

	Num. of YARA rules	Num. of DNF clauses	
		Total	Unique
Koodous public rules	348	788	104
Yara-Rules on GitHub	348	697	48

$\forall i = 1 \dots n$, as we want each existing literal combination to satisfy the minimum score of all existing rulesets.

Finally $\mathbf{A}$ is a $n \times m$ matrix that put into relation each clause with their own literals:

$$\mathbf{A} = \begin{bmatrix} l_{11} & l_{12} & l_{13} & \dots & l_{1m} \\ l_{21} & l_{22} & l_{23} & \dots & l_{2m} \\ \dots\dots\dots\dots & & & & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nm} \end{bmatrix}$$

where $l_{nm} = 1$ if l_{nm} is a literal of the clause c_n, otherwise $l_{nm} = 0$. In order to get the list of all the n existing clauses c_i, we firstly reduced all the available YARA ruleset in the Disjunctive normal form (DNF).

We arbitrarily set the values $T_{min} = 400$ and $T_{max} = 700$ for the two thresholds. Table 3.2 reports the details about the rules, clauses, and unique clauses that have been analyzed, using the YARA rules from both Koodous[2] and the *Yara-Rules* repository on GitHub[3]. Table 3.3 show the final result, where each literal is assigned a distinct weight.

3.3.3 The optimization phase

Part of the work described in this section has been previously published in *"Evolutionary Antivirus Signature Optimization", Congress on Evolutionary Computation, 2019 [73].*

[2] https://koodous.com/rulesets

[3] https://github.com/Yara-Rules/rules/tree/master/Mobile_Malware

Table 3.3 Weights assigned to each type of literal as a result of the simplex method optimization. Weights are used by the automatic procedure to generate new YARA rulesets.

Module Name	Literal type	$\mathscr{S}^*(\cdot)$
Androguard	App name	100
	Package Name	100
	Certificate SHA1	150
	Certificate Subject	100
	Certificate Issuer	100
	Main Activity	50
	Activity	150
	Service	150
	Broadcast Receiver	100
	Intent Filter	150
	Content Provider	80
	Functionality	15
	URL	400
	Permission Normal	7
	Permission Dangerous	80
	Permission Not third party	50
	Permission System	80
	Permission with Typos	150
	Permission non standard	50
Cuckoo	DNS lookup	400
	HTTP request	400

Valid rules have a score between T_{min} and T_{max}, which are defined to avoid too generic or too specific signatures. However, thousands of different combinations of literals can lead to a valid score. The goal of the optimization phase is to find, among the valid configurations, the most successful one that minimize the number of false positives (FP) and false negatives (FN). The proposed optimization approach is based on an *estimation of distribution evolutionary algorithm*, which has been presented in Section 2.3.1.

The evolutionary optimization phase

The technique to optimize a signature relies on the Selfish Gene (SG) evolutionary algorithm, with an unorthodox fitness function given by unsystematic human expert knowledge coded as a set of heuristic rules[4]. In the current application, the genome is the signature to optimize and the loci are the literals of the signatures. Each locus may contain two alleles (*true* and *false*) specifying whether the attribute is used or not in the final signature.

The metrics we used to compare two candidate solutions are the following, listed in order of priority:

1. the number of malware correctly detected by the candidate solution among the ones used to generate the candidate signature, with the aim of achieving 100% coverage;
2. a set of manually-defined heuristic rules, gathered from human experts;
3. the score of the candidate optimized signature;
4. the number of attributes contained inside the candidate signature.

The first value is to be maximized while the third and forth minimized.

The algorithm starts considering two candidate solutions and comparing the number of malware they correctly detect. The signature that matches more malware than the other one (and so, probably even more than the original one) will be considered the most powerful. Otherwise, if the number of matches is equal the algorithm proceeds further in the comparison using the heuristic rules. If the number of malware and the heuristic rules are still not sufficient to establish which signature is better, then the algorithm also evaluates the score and the number of attributes, finally choosing the candidate signature with the lower values.

Since the score of the signatures must be between the two thresholds ($T_{\text{min}} < S^{\sigma} < T_{\text{max}}$), a penalty is assigned to signatures that are outside these limits. A candidate solution receives a huge penalty if the signature becomes too general going under T_{min}, and a smaller penalty if the signature becomes too specific going beyond T_{max}. Penalties are different because, for the endpoint malware detection, it is preferable to have too specific signatures and miss some samples rather than detect false positives.

The heuristic rules mentioned in the comparison metrics have been obtained with the help of experts if the field of Android malware: analysts from Koodous have been

[4] Note that the word *rule* in this context does not refer to signature.

interviewed and a set of *rules of thumb* was created to determine which signature is best between two. We used the following list of rules, although the approach is note limited to these only:

- having URLs is better than not having them;
- not having embedded piece of codes related to TLS code (i.e., secure network communication) is better than having it;
- the more features categories the better;
- the more code features the better;

To make the empirical rules even more similar to the decision-making process of a human we decided to apply a tolerance on the comparisons implemented by the last two rules. For example, signature A is assumed better than signature B if A has at least 3 feature categories more than B. In some cases it is not possible to establish which signature is better, this happens for example when both the candidate solutions are composed only by URLs or both only have the TLS code.

This set of heuristic rules is able to simulate the human process of choosing between two signatures, but cannot perform like a traditional fitness function where individuals can be ordered through the evaluation process. That is, if an individual i_a is preferable to the individual i_b, and i_b is preferable to i_c, this does not imply that i_a is preferable to i_c:

$$(i_a \geq i_b) \wedge (i_b \geq i_c) \not\Rightarrow \; i_a \geq i_c$$

As an example, if we assume that the tolerance on the *code features* rule is 3 and the tolerance on the*feature categories* rule is 10. If $S1$, $S2$, $S3$ are three signatures with the following characteristics:

$$\begin{aligned}
&S1 = 5 feature\ categories;\\
&S2 = TLS + 10\ feature\ categories + 2\ code\ features;\\
&S3 = TLS + 15\ feature\ categories + 4\ code\ features;
\end{aligned}$$

we obtain:

$$S1 > S2; \quad S2 = S3; \quad S1 < S3.$$

Due to this peculiarity a problem arises in the management of the candidate solution archive.

Candidate solution Archive In traditional EDA, when two candidate solutions are generated and compared, they have to compete with a set of solutions that are considered equally good. This pool of individuals is called *archive* [74]. All the individuals contained inside the archive are candidates to be returned as the final solution. If the fitness value of a candidate is greater or equal to all the solutions included in the archive, that candidate solution becomes part of the archive. This mechanism requires an absolute order among the solutions.

By contrast, in the proposed approach, due to the introduction of the heuristic rules, the transitive property is lost, so is the absolute order. The way in which the optimizer selects the individuals to keep in the archive is based on a tournament-based approach. After each comparison, each individual inside the archive receives a score. In particular, it gets:

- 3 points, if it is better than the other one;
- 0 points, if it is worse than the other one;
- 1 point, if it is not possible to establish which individual is better between the two.

This mechanism is executed in a round-trip way, so that each pair of individuals is compared twice. At the end, the individual that is stored in the archive is the one with the highest score. If there is more than one individual with the highest score, then all of them are kept in the archive.

3.3.4 YaYaGen

The approach presented in 3.3.1 has been implemented in a framework called *YaYaGen*, which is an acronym for *Yet Another YARA rule Generator*. The framework has been publicly released on GitHub [5], after the presentation "Looking for the perfect signature: an automatic YARA rules generation algorithm in the AI-era" at BSidesLV and Def Con 26, in August 2018.

[5] https://github.com/jimmy-sonny/YaYaGen

YaYaGen is an automatic procedure, that starts from a set of Koodous reports, either identified as a malware family, or by any other mean, and eventually produces a signature in the form of a YARA rule that can be seamlessly used in Koodous. YaYaGen analyzes the reports of the target applications, extract the analysis attributes, and identifies an optimal attribute subsets that are able to match all the targets; moreover, thanks to a heuristic measure, the generated signature has a limited risk of detecting false positive in the future, yet it is general enough to catch future threats.

3.4 Extension to Windows Malware

The approach to automate the signature generation for Android malware has been extended to Windows malware too, and YaYaGenPE is an extension of the original YaYaGen (Section 3.3.4). Following the same principles of scalability and accurate malware identification, YaYaGenPE has been specifically designed to automatically generate signatures for Portable Executable (PE) files, that is the executable file format of Windows binaries and libraries (DLLs).

YaYaGenPE shares many of the ideas at the base of YaYaGen, but the extension to Windows required a substantial amount of work for the code development, which has been partially conducted by Luca Cetro during his MSc Thesis *"Automatic Malware Signature Generation", Politecnico di Torino, 2018.*

3.4.1 YaYaGenPE

Differently from the previous approaches that generate signatures for Windows malware, YaYaGenPE introduces the following main advantages:

- The efficacy of the rule generation engine does not rely on an extensive goodware database. Rules are generated through an optimization phase which combines hundreds of simple features in complex and rare clauses that limit the number of false positive detections.

- The ability of generalizing the detection to other malware variants of the same family is directly related to the quality of the input set. The more the samples of the same family are provided in input, the more the framework is able to

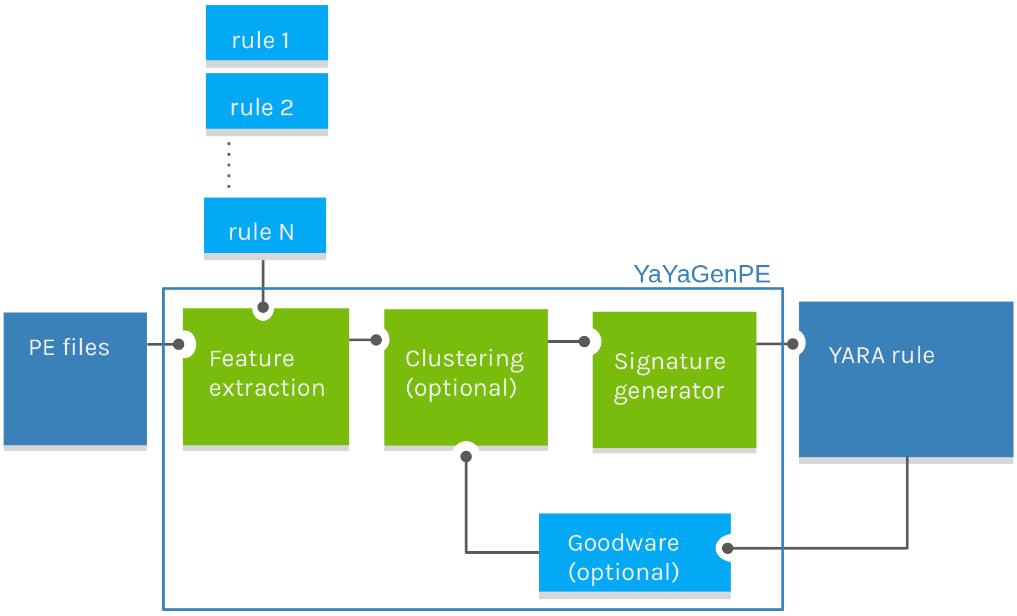

Fig. 3.6 Work flow of a YARA rule generation in the YaYaGenPE framework.

select the relevant features that distinguish that particular family from all the others.

As illustrated in Figure 3.6, YaYaGenPE mainly consists of four steps:

- Feature extraction: samples in input are processed in order to extract useful features for the next rule generation phase. As the rule generation process is completely independent from the specific type of features in input, anything of semantically relevant can be selected in this phase. This include both simple

Table 3.4 Comparison of state-of-the-art automatic signature generation approaches for Windows binaries.

	YaraGenerator	YarGen	YaBin	BASS	YaYaGenPE
Language	YARA	YARA	YARA	ClamAV	YARA
Features	Strings	Strings	Binary	Binary	PE header
Algorithm	Intersection	Whitelist	Whitelist	LCS	Set covering
Input coverage	No	No	Yes	Yes	Yes
Clustering	No	No	No	Yes	Yes
Scalable 100+	Yes	Yes	Yes	No	Yes
Low FNs	No	Yes	Yes	Yes	Yes

feature, such as patterns extracted from the PE header or code, and complex features, such as other rules that match the samples in input. In the latter case, rules are preprocessed in order to extract the literal from which are composed, where each of them is later treated as a simple feature.

- Clustering: finding subgroups of similar samples is an optional step that guarantees the scalability of the approach. It allows to keep the approach scalable, even if thousands of samples are provided in input. Different clustering methods can be used, but the Unsupervised Decision Tree is the preferred one.
- Rule generation: a set of signatures that cover the input samples is automatically generated as a result of an optimization phase that reduce the original problem to a variant of the set-covering problem. As the original YaYaGen, two algorithms are proposed, the pure *greedy* and the *clot*. Optionally, in case of false positives detection, the rule generation can be repeated to exclude the unwanted detections.
- Syntax conversion: the generated rule is converted into the specific signature syntax, where the YARA language is the default choice.

Each phase is detailed in the following sections.

Feature extraction

Each automatically produced signature is a combination of simple features extracted from the samples in input. The rule generator mechanism used by YaYaGenPE is completely independent from the type of features provided, hence finding the most effective type of feature is a crucial step to ensure the final high quality of the signature.

Features can be simple terms extracted from the binary, such as PE header files, strings and binary patterns. On the other hand, complex features, such as existing signatures that match the file in input, can be used too. In case of complex feature set, those will be preprocessed in order to extract their basic elements (i.e., simple features) to be used in the following steps of the rule generation process.

Simple Features YaYaGenPE uses by default all the header fields extracted from the PE header []. It consist of hundreds of entries, and apart from the *AddressOfEn-*

tryPoint and the *Import Address Table*, most of the PE header fields are not modified by packers. The *imphash* [75] and the *overlay* [76] are extracted too, although they are not part of the standard.

Since the specification of some fields of the PE header are loose, and different tools implements the extraction of those fields in different ways, it is important to ensure that the value of each field of header file is the same in both the extraction and checking phase.

YARA allows the feature extraction from specific file format, such as the PE file, through custom modules which are added at the compilation time of the tool. In order to ensure the equality among the features extracted and those checked, a modification to the YARA tool, and respective Python bindings has been implemented.

Complex Features Complex features include all of those features that cannot directly extracted from the file under analysis, but requires an external source of information to enrich the available data. After a pre-processing phase, complex-features are translated in set of simple features that can be used to construct a malware signature. In the context of malware, thousands of signatures are freely available, and even if none of them entirely match the sample analyzed, they are usually made of tens of tens of literals. By extracting the literals from available rules, and filtering those that match the samples, it is possible to enrich the set of features. The idea behind the use of existing ruleset is that they encode the expert knowledge of humans who analyzed the malware and extracted those patterns that are characteristics of a specific malware threat [6].

Clustering

For windows binaries, an optional clustering phase is devised to ease the generation of the rule before applying the *clot* algorithm. Given the rich feature set of each application, in order to meet the scalability requirement and being able to process thousands of files in input, an unsupervised clustering algorithm is used to identify cluster of samples.

[6] A YARA rule parse has been implemented by Luca Cetro in his MSc thesis “Automatic Malware Signature Generation”

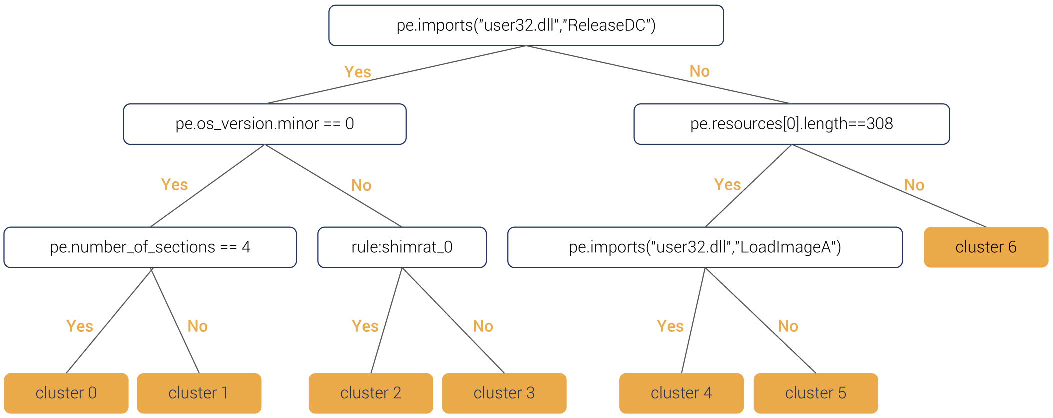

Fig. 3.7 Example of a binary tree constructed by the UDT clustering algorithm. Each path highlights the boolean value of the selected features.

Besides the iterative clustering (Section 3.2.1), an approximated Unsupervised Decision Tree (UDT) algorithm is presented. As the UDT finds at each iteration a binary split, according to the value of a single feature, it has the main advantage of generating part of the signature while producing the clustering. Indeed, the path the connects the root of the tree to the leaf (i.e., the cluster) represent a logical expression that can be easily translated into a valid signature.

The algorithm selects the best splitting feature according the the one that maximize the distance among the cluster centroids, which calculation is approximated. Experimentally, we found that the Russell-Rao and the Jaccard distances delivers the best results. The tree generation ends when the distance among centroids is lower than an experimentally defined value.

Figure 3.7 shows an example of a binary tree generated by the UDT clustering algorithm. The figure shows for each path the boolean value of the feature selected at each level of the tree. The entire path from the root of the tree to the selected cluster can be easily translated into a boolean expression, as shown in Figure 3.8.

Signature Generation

Starting from the clusters identified at the previous step, or directly from the all set of samples in input, the algorithm that generate the signatures follows a similar approach to the one adopted by YaYaGen framework (Section 3.3.1). Indeed, the presented algorithms are general, and independent from the specific types of features

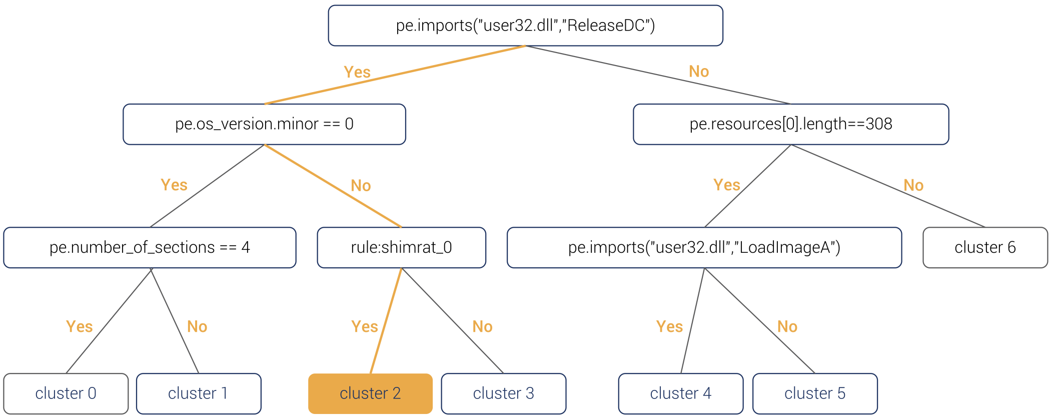

Fig. 3.8 Example of path from the root the leaf of binary tree created by the UDT clustering algorithm. The path will be translated into a boolean expression and will be integrated into the rule of the selected cluster.

extracted. The problem of generating a minimal set of signatures from the samples in input can be reduced to a variant of the set coverage problem, where sets can overlap, but cannot cover other point (i.e., samples) which otherwise will lead unwanted detections.

The scoring system The rule generation process requires the definition of two thresholds, MIN_T and MAX_T, which identifies the limits of a rule in terms of generality and specificity. The choice of these two parameters is critical, and the effectiveness of the generated rules is strongly related to their value.

Malware industry commonly identifies two type of rules, each with its own requirements:

- *Hunting rules* requires a low number of FNs, in flavor of a higher number of FPs. The goal of the rule is the investigation of a particular threat. Each detected samples will be further manually analyzed, looking for variants of a particular malware.
- *End-point rules* are used for the customer (i.e., endpoint) protection and FPs are usually not accepted. Rules should be very specific to the type of threat, but should also limit the FNs originated by new variants of the malware.

An appropriate choice of MIN_T and MAX_T make YaYaGenPE suitable for both scenarios.

Differently from the Android scenario, a manual procedure has been adopted to set the threshold values. The value of MIN has been arbitrary decided after a precise static analysis of thousands of malware samples, and the scoring system has been simplified, making each feature counts one, that is considering them equally important.

Given the high cardinality of the feature set, with hundreds of features extracted from the PE header, and thousands from existing rules, finding their value through an optimization procedure, as proposed in the Android case (Section 3.3.2), is extremely challenging. Moreover, the problem cannot be simplified, as there is no unique way to categorize those features in group of semantically similar features, and there is the lack of a huge dataset of labeled malware which presents different characteristics in respect of the selected features.

Syntax conversion

The set of features that describe a rule is finally converted into the specific signature syntax. The choice of the syntax is strongly related to the engine that will be used for checking the samples. YaYaGenPE automatically converts the rules into valid YARA signatures.

For each feature a corresponding YARA literal is created, and in the condition section it is added the logical expression corresponding to the logical “AND” of each literal. If properties from custom modules are used, those are directly inserted in the condition section, according to YARA specification. Finally, several signatures can be included in the same YARA rule, if concatenated by the “OR” logical operator.

3.5 Limitations

A major limiting factor of the described semi-supervised approach is represented by the ability to extract meaningful information regarding the malicious behaviour of the applications under analysis. Indeed, the accuracy of the analysis directly affects the clustering results and the automatic rule generation process. The Android platform lacks of mature reverse engineering tools compared to the ones used for x86 malware [77]. Since each malware is different, automatically finding the malicious code by means of static analysis is difficult, because it is mixed with benign code;

moreover dynamic code loading and reflections make the analysis even harder. Unfortunately, most malware include trigger-based anti-analysis techniques that delay or hide their malicious activities at the first application run or in an emulated environment. For instance, the family of applications known as *DroidKungFu*[7] uses a time bomb of 240 minutes to schedule the execution of its malicious code, indeed a simple dynamic analysis fails to observe interesting behaviors. However, in this research we do not address problems related to application analysis, as we focus on the detection of new samples and the automatic generation of new signatures.

Evasion attacks, such as noise-injection attacks [78] and other similar approaches [79–82] may affect the correctness results of the clustering and the signature generation. Those attacks rely on the ability of injecting, in the analysis platform, applications specially crafted to mislead the clustering process and the generation of a good detection model.

In the described system, an attacker could exploit such attacks by injecting specially crafted applications with the final goal of generating a false positive or a false negative detection. However, in both cases we assume that the detection information of already known threats (identified through signatures or by triage) cannot be maliciously tampered, thus new injected families will result in a Type 4, 5, 6 or 7, hence will be subject to manual validation.

If the attacker wants to deliberately generate a false positive, several malicious applications whose statistical properties are similar to a target goodware can be injected. Since a false positive detection mainly generates a disruption to a third party service, causing a reputation fail for the AV solution, the magnitude of the echo is proportional to the diffusion of the target goodware. As a matter of fact, the analyst will be alerted by such a family.

On the other hand, if the goal is to generate a false negative, the attacker could inject several goodware with the same statistical properties of a target unknown malicious app. Such a family could be misclassified as a completely goodware even after the validation process, as the manual analysis focus only on few samples. However, such a situation applies only as far as the malware is a zero-day, and no specific knowledge about that threat is available. The identification of zero-day malware is a challenging and an open-research problem in the security community.

[7] Sample MD5: `7f5fd7b139e23bed1de5e134dda3b1ca`

Finally, the proposed system strongly relies on the information provided by the platform to automatically extend the detection to new applications and identify new potential malware families. It is a prerequisite that this information is not tampered by any malicious actor. Although Koodous provides protection mechanism for both YARA rules (rules before becoming active undergo a review process) and the triage process (community members are subject to a reputation check), it is not intent of this research to tackle those issues, leaving their study to future works.

Chapter 4

Experimental Results

Part of the work described in this chapter has been previously published in *"Countering Android Malware: A Scalable Semi-Supervised Approach for Family-Signature Generation", IEEE Access, 2018 [69].*

4.1 Android applications

4.1.1 Android dataset

As a case study we used a dataset of 1.5 million Android applications collected over the 2016. The dataset is recent and diverse in the set of attack vectors it represents: in order to have the same ratio between detected and undetected applications as in Koodous, we sampled a subset of 1 million apps[1]. As result, the dataset under analysis is composed by 65% undetected applications, 31% detected by signatures, and 4% detected through triage only.

4.1.2 Clustering

HDBSCAN has two parameters that mostly influence the results of the clustering: *min cluster size* (*mss*) determines the smallest size of a cluster, while *min samples*

[1] In order to ensure the quality of the results and avoid artifacts, the sampling of 1 million applications have been repeated three times: in all the cases the proposed techniques showed coherent results.

Table 4.1 Comparison of Homogeneity (Hom.) and Completeness (Comp.) index values between the families inferred by the clustering process (using both the iterative clustering with different chunk sizes N, and the *non*-iterative version), and the families labels extracted from Koodous and VirusTotal.

	Koodous labels		VirusTotal labels	
N	Hom.	Comp.	Hom.	Comp.
50k	0.96	0.36	0.85	0.49
100k	0.96	0.35	0.85	0.49
200k	0.96	0.35	0.85	0.50
non-iterative	0.92	0.36	0.78	0.50

(ms) how conservative are the results. A higher value of *min samples* restricts clusters to more dense areas, but it also increases the number of outliers. We use $mss = 3$ and $ms = 1$; in other words, we considered only malware clusters containing a minimum of three samples as representative of a malware family.

We used a high-performance, open-source implementation of HDBSCAN in Python from Leland McInnes [83]. All experiments were performed on a 6-core Intel Xeon (CPU E5-1650 v2 @ 3.50GHz), with 128 GB of RAM, although HDBSCAN only used up to four cores and 6 GB of RAM in each run.

The quality of the clustering results is evaluated as a measure of the ability of correctly extending malware detection to undetected applications. However, given the difficulty of establishing a reliable ground truth in the field of malware analysis, evaluating the results was challenging. Finally, for the clustering validation we used all the available information: detection results and AVs labels extracted from VirusTotal reports, and signature labels extracted from existing YARA rules in Koodous.

Since clustering exploits the relationship between statistical similarities among applications, in contrast to the structural properties commonly used in AVs signatures, no one-to-one correspondence between clusters and AV labels is expected, however by combining several indexes we deliver a trustworthy quality measures of clustering performances. In order to estimate cluster assignment, we adopt the Adjusted Rand Index in combination with other external indexes as proposed by Rosemberg et al. [84]:

- *Adjusted Rand Index* (ARI) is defined as the number of pairs of items that are either both in the same cluster or both in different clusters in the two partitions, normalized over the total number of pairs of items. The index lies between 0 and 1: when two partitions agree perfectly, the Rand index achieves the maximum value 1, and more in general a larger adjusted Rand index means a higher agreement between two partitions. Moreover, ARI supports the measure of the agreements even when the compared partitions have different numbers of clusters

- *Homogeneity* (Hom.), which measures whether its clusters contain only data points which are members of a single class

- *Completeness* (Comp.), which measures whether all the data points that are members of a given class are elements of the same cluster

- *V-measure* (V-ms.), measured as the weighted harmonic mean of homogeneity and completeness; this is useful since homogeneity and completeness of a clustering solution run roughly in opposition: increasing the homogeneity of a clustering solution often results in decreasing its completeness.

Table 4.1 compares homogeneity and completeness index values between the families (i.e., clusters) inferred during clustering process, and the families labels extracted from Koodous signature names and VirusTotal AV labels[2]. Results are compared using both the iterative clustering, with different chunk size N, and the *non*-iterative version.

Since AVs listed in VirusTotal commonly use different names to identify the same type of threat, we took advantage of AVclass [85], an automated labeling tool that, given the labels of multiple antivirus engines, returns the most likely family names for each sample, focusing on normalization, removal of generic tokes and alias detection. The implementation is open-source, available on GitHub [86], and provides VirusTotal integration.

Interestingly, all the cases reported in Table 4.1 show very high homogeneity value, which indicates that malware families identified by AVs signatures are further split in finer partitions during the clustering process. Moreover, precise clusters increase the effectiveness of the following automatically generated signatures.

[2] The comparison with VirusTotal AV labels is limited to 100,000 randomly selected applications.

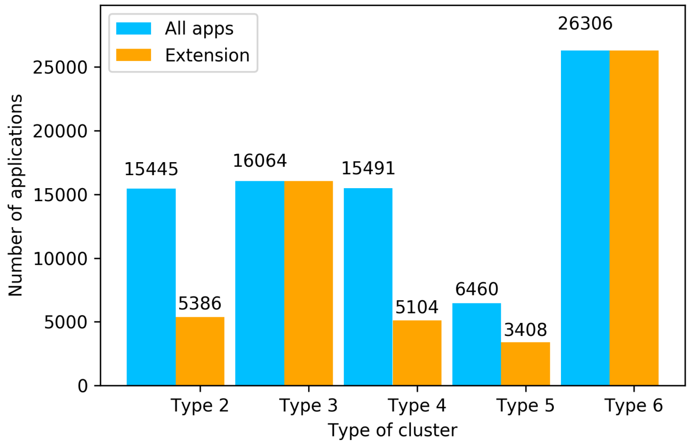

Fig. 4.1 Number of total applications, and newly automatically inferred detections, for each type of malware family (Type 2...6). Results refer to the iterative clustering approach, using chunk size $N = 100k$, over a dataset of 1 million applications.

Table 4.2 Number of families automatically inferred by the clustering algorithm (using both the iterative clustering with different chunk sizes N, and the *non*-iterative version), using dataset of 1 million applications. Results are gathered for each type of malware family (Type 2...6).

N	Type 2	Type 3	Type 4	Type 5	Type 6
50k	1,890	2,949	1,467	463	2,846
100k	1,477	2,439	1,519	500	3,385
200k	1,193	2,203	1,436	536	3,133
non-iterative	435	1,046	2,126	536	4,629

Extending malware detection

Fig. 4.1 illustrates the result of the automatic detection extension for the 1 million applications under analysis: each bar in the plot is related to a family type (refer to Section 3.2.2 for an accurate description of each type of malware family), illustrating both the total number of applications, and the number of those automatically identified as malicious. Results are obtained using the iterative clustering approach, with chunk size $N = 100k$. Note that Type 1 and 7 families are not shown, as the first consist of application that are already completely detected by signature, while the latter include families found within unknown applications, hence no direct information about their composition can be automatically inferred.

Table 4.2 is complementary to Fig. 4.1, as it compares the number of families, for each family Type, using both the iterative clustering with different chunk sizes N, and the *non*-iterative version.

Among the clusters of Type 2 and 3, the system automatically identifies a total of *21,450* new malicious applications that will be automatically covered by new signatures, without requiring any human intervention. In more detail, *5,386* applications (Type 2) are found within clusters with other apps already detected by YARA signatures; while *16,064* applications (Type 3) are assigned to clusters purely made of applications detected during the triage phase only. As matter of fact, generating new family signatures for these applications allows to transform the knowledge of existing threats into a more reliable and scalable form of detection, without affecting the precision of the results: all those applications have been already identified as malicious by the community of malware experts.

On the other hand, *34,818* applications are assigned to families Type 4, 5 and 6: *20,464* are the newly identified potential threats, since previously marked as undetected. In this case, the proposed framework allows an easy identification of hard to find potential threats, reducing the human intervention from the manual analysis of thousands of applications to the validation of a very fewer number of families where applications reflect a similar behavior, eventually speeding up the procedure of new malware discovery. For example, the system identified a total of 500 families for the Type 4 (refer to Table 4.2, second row) reducing of an order of magnitude the need of manual analysis, as a detail analysis of a malicious application could take few hours, this approach results in a huge time saving.

Evaluation of malware detection extension

Aiming at evaluating the detection extension performance in a real-world case, we evaluate how the proposed system is accurate in relationship to the information of the detections available in VirusTotal. We choose VirusTotal as a well-known and trustworthy source of information about existing threats since it collects the detection results from tens of independent AV companies. Moreover, recently other researchers used the same metric [87].

In order to evaluate the detection extension results, we firstly assessed how precisely Koodous detects malware samples, and how effectively covers all the malware

Table 4.3 Comparison of the detection results between VirusTotal and two datasets of 50,000 applications, respectively undetected (und.) and detected (det.) by Koodous. Columns indicate the number of applications unknown (unk.), undetected (und.), detected by at least one AV (det.), and detected by more than three AVs, as reported by VirusTotal.

	VT unk.	VT und.	VT det.	VT det. >3
Koodous det.	18	72	49,910	49,717
Koodous und.	3,449	12,508	34,043	28,166

Table 4.4 Evaluation of the accuracy of the clustering system to automatically identify groups of malicious applications, by comparing the detection of the new applications with VirusTotal. Columns *Correct* and *Incorrect* respectively reports the number of applications correctly or wrongly classified, while *Min* and *Error* illustrate the minimum precision and the maxim error of the proposed approach. Results are reported using both the iterative clustering with different chunk sizes *N*, and the *non*-iterative version.

N	Correct	Incorrect	Min %	Error %
50k	7,493	254	91.23	2.91
100k	12,502	877	86.04	6.03
200k	13,628	917	89.54	6.18
non-iterative	14,619	1,109	87.93	6.67

variants. Starting from two randomly sampled subsets of 50,000 applications, respectively originally undetected and detected in Koodous, we cross-checked their maliciousness using VirusTotal. Results are illustrated in Table 4.3. The first line of the table (Koodous det.) shows that among detected applications, Koodous has 100% of precision, and very high recall (99.8%), as almost all Koodous detected applications are completely identified as malware by traditional AVs too, while only 100 applications (the 0.2% of the dataset) are unknown or undetected by VirusTotal. However, the second line of the table (Koodous und.) shows a very low accuracy (27.8%), as a consequence of a major diversity in the detection ratio among the applications undetected by Koodous and VirusTotal. Although such a difference could be partially explained by the different policies that traditional AVs use in identifying a malicious application, particularly regarding adware, this result further motivates the need of an automatic mechanism to increase the number of correct detections in Koodous.

With the awareness that VT detection results are not completely reliable, we only considered those clusters for which the VT information is available for all the applications. In order to calculate the accuracy of the proposed system, we adopted the following metrics:

- if the system proposes an extension to a malware family where all the applications are detected by VT, we consider the extension as *correct*;
- if the system proposes an extension to a family where all the applications are undetected by VT, the extensions is considered as *incorrect*;
- if the system proposes an extension to a cluster that mixes applications partially detected and undetected by VT, the result is considered *unknown*.

Table 4.4 illustrates the results. For each clustering experiment, each line of the table reports the number of applications that have been correctly or wrongly classified, according to the type of the cluster to which they were assigned. Without any human intervention, the system scores a minimum accuracy that ranges from 86.04% to 91.23%, and it has a worst case error of the 6.18%. A further manual inspection of the results revealed that several families completely undetected by VT are mostly related to aggressive adware samples, whose classification is subject to different considerations. Furthermore, results show that a smaller chunk size increase the precision the detection, reducing the error, although the absolute number of applications automatically extended is smaller. Accordingly, the chunk size can be set in accordance with the needs of the system.

Example of manual analysis of a malware family

Table 4.5 shows an example of a Type 4 malware family. As the first two samples are already detected by the signature *Xynyin.Trojan*[3] in Koodous, the system proposes to extend the detection to the other applications of the same cluster. The comparison of the detection results with VirusTotal[4] shows that all but one application are already detected, while a manual analysis of *Leagueoftankheroes3D*[5] confirm its affinity to the Xynyin malware family[6].

[3] https://koodous.com/rulesets/1225

[4] Detection results refer to 15 Nov 2016

[5] MD5: `695d6b9f97a9e992f8e321d36509c080`

[6] From 24 August 2017, VirusTotal's AVs detect the application too.

Table 4.5 Example of a Type 4 malware family. As the first two samples are already detected in Koodous by the YARA rule *Xynyin.Trojan*, the system identifies other applications within the cluster as potentially malicious too. The comparison with VirusTotal (the number of detection is reported) and a manual analysis confirm the accuracy of the system.

	Detected	
MD5	Koodous	VT
`998faf5e7a0d45f6ad60903bc5d60817`	Yes	12
`5a8dd85a5707f520563069bf536f9d5f`	Yes	19
`695d6b9f97a9e992f8e321d36509c080`	No	0
`304754e9f8f95228af0e7118d62e999f`	No	12
`805d8770d6314f5adad266ddaba610e1`	No	10
`23863ddba21b96aea3e8b2cc120bb2b2`	No	12

Table 4.6 Comparison of the clustering results using using both the iterative version with different chunk sizes *N*, and the *non*-iterative one. Column *Time* indicates the time (in seconds) required by the clustering process, while column "Outliers" reports the number of outliers found at the end of the iterations.

N	Time (s)	Outliers
50k	5,746	64,553
100k	6,408	65,685
200k	10,573	67,081
non-iterative	16,592	119,919

One of the major benefit of a semi-supervised system is to limit the detection of false positives, and the operation is further simplified since the analysts should only focus on groups of similar applications, without considering single samples. As useful side effect, the system could be also used to improve the precision of the results, by reducing false positive detections for those families of applications that have been partially miss-classified by existing signatures.

The Iterative algorithm

The adoption of the iterative approach brings a number of benefits: it proved to be essential in order to analyze millions of applications, and the resulting number of outliers, as illustrated in Table 4.6, is much lower than what was obtained by clustering all applications together. The time required by the clustering phase is

Table 4.7 Indexes comparison of the clustering label inferred by the iterative approach (with different chunk sizes N) using the assignment produced by the *non*-iterative version as a reference.

N	ARI	Homogeneity	Completeness	V-Score
50k	0.26	0.92	0.78	0.85
100k	0.27	0.93	0.81	0.86
200k	0.29	0.94	0.84	0.89

proportional to the chunk size and it is up to one order of magnitude lower than in the *non*-iterative case.

The adoption of the iterative approach does not affect the quality of the results, even though using a bigger chunk size results in a greater number of new detections.

Table 4.7 compares the iterative approach using as a reference clustering assignment the one produced by the *non*-iterative version. A relatively low ARI value indicates a difference in the clustering assignment between the two approaches, while a very high homogeneity value, compared to completeness, is a clear sign of a finer cluster partitioning. In other words, using the iterative approach the quality of the information is not compromised, although the resulting clusters are smaller, hence less likely to contain enough applications that span different detection areas, finally resulting in a lower extension. A bigger chunk size lowers the differences between the iterative and the *non*-iterative assignment, as shown by an increasing *V-score* value. Eventually, if a large enough chunk size is used, the iterative approach produces almost the same results as the *non*-iterative one, while generally finding a higher number of clusters, as illustrated in Table 4.2, and a less outliers, Table 4.6.

Finally, in order to further test the scalability of the proposed method, we successfully applied the algorithm on a very large dataset of 10 million applications, using a chunk size $N = 500k$.

4.1.3 Android signatures

In order to evaluate the effectiveness of the automatic signature generator, we compare the detection results of several YARA rules automatically generated by the proposed algorithm with existing rulesets created by expert analysts.

Table 4.8 reports the results of the rules detections on a dataset of 1.5 million applications: in all the cases, the automated generated rules [7] performed better than the one authored by humans, increasing the detection from the 8.2% up to 131.2%, without generating any false positives.

Referring to Section 3.3.2, in all the cases the rule generation process stopped at the second step, as none of the new rules produced any false positives in the current dataset of applications. A further manual analysis of the detected applications, confirmed that no false positive was generated.

As shown in Table 4.9, the time required to generate a rule for few hundreds malware is always less than a minute, although when the target increases to a few thousands applications, the time required grows up to several minutes, as the most expensive part of the process is the check for false positives against a reference dataset. This is not considered a limitation, since all the process is automatic, and given the goodness of the results, it is of invaluable support for the family signature generation process.

Table 4.9 reports the number of literals (i.e., application features) and the final score for each generated YARA rule: referring to Section 3.3.2, each score is higher than the minimum threshold $T_{min} = 400$, satisfying the minimum requirement for acceptability in order to avoid false positive detections, and lower than the maximum threshold $T_{max} = 700$, as a result of the optimization process to increase the rule generality and therefore the ability to catch future malware variants.

In order to increase the effectiveness of a rule, URLs are included only if are known to be malicious, like in case of *http://s.adslinkup.com/v2* for the *Syringe* malware family. Moreover, aiming at identifying malware with very high precision and avoiding false positives, whenever available, the automatic signature generator includes those attributes extracted from the application analysis that contains a typing mistake. For instance, the rule *YaYaMetasploit1*[8] includes a wrong permission `ACCESS_COURSE_LOCATION` instead of the correct one `ACCESS_COARSE_LOCATION`. Given the difficulty of reproducing such an uncommon mistake, we consider this feature as a hard indicator of the maliciousness of a sample.

[7] Example rulesets could be found at the following address: https://koodous.com/analysts/YaYaGen/rulesets

[8] https://koodous.com/my_rulesets/3466

Table 4.8 Comparison of detection performances of human authored YARA rules (Original) with automated generated ones (Auto). Last column reports the improvement (in percentage) for the newly generated rules. Detections are tested on a dataset of 1.5 million applications.

Rule name	Detections		
	Original	Auto	Improvement
SmsSender	539	1,004	+86.3%
Syringe	220	315	+43.2%
HummingBad2	136	257	+89.0%
Marcher2	559	652	+16.6%
SMSReg	159	172	+8.2%
VolcmanDropper	186	430	+131.2%
FakeGoogleChrome	516	822	+59.3%

Table 4.9 Comparison of the number of literals, score and time (in seconds) required to generate each YARA rule.

Rule name	Literals	Score	Time (s)
SmsSender	15	412	43
Syringe	19	574	48
HummingBad2	12	599	52
Marcher2	20	686	49
SMSReg	34	537	42
VolcmanDropper	10	439	13
FakeGoogleChrome	15	407	43

4.1.4 Signature optimization

Table 4.11 shows a comparison of the scores of the signatures in the three cases of no optimization (original version of YaYaGen, *Not Optimized* column), optimization with Hill Climber Optimizer (*HC Optimizer* column) and optimization with Evolutionary Optimizer (*SGX Optimizer* column). Since the score of a rule is directly related to the accuracy of the malware detection, the results show that both the optimization methods succeed in reducing the score within the range [T_{min}, T_{max}] that is defined as optimal ($T_{min} = 400$, $T_{max} = 650$), but the SGX-Optimizer reaches the best results using a combination of sub-optimal literals defined through heuristic rules (Section 3.3.3).

Results are computed on the average of 10 independent tests performed on 10 different clusters. Each cluster contains a different number of applications which can vary between 3 and 20, but the exact number of elements of a cluster is shown in the *TPs* column.

On average the Selfish Gene algorithm is able to reach a lower score, with a higher number of literals for the rules optimized. The execution time of the SGX is fixed to a maximum of five minutes.

Table 4.10 Comparison of the scores of the signatures in the three cases of no optimization, hill climber (HC) and evolutionary optimization (SGX).

Cluster	TPs	Non Optimized		HC Optimizer		SGX Optimizer	
		Literals	Score	Literals	Score	Literals	Score
Cluster10	4/4	260	17614	9.00	625.00	11.50	401.70
	4/4	260	17614	9.20	597.00	12.60	401.80
Cluster20	3/4	64	2073	20.20	612.30	39.20	555.00
	1/4	59	1907	19.80	620.40	39.30	550.50
	1/4	59	1907	20.80	615.40	38.30	536.20
	3/4	64	2073	18.20	620.50	40.90	583.90

4.2 Windows malware

Part of the experiments described in this section have been implemented by Luca Cetro in his MSc Thesis *"Automatic Malware Signature Generation", Politecnico di Torino, 2018.*

4.2.1 Windows dataset

In order to validate the proposed approach for the automatic signature generation on Windows malware, a dataset of 6,881 malicious binaries and 3,413 goodware samples has been used. Malicious samples have been kindly provided by VirusTotal through their academic program, while goodware samples have been manually selected among common user and system applications, extracted from a Windows 10 installation.

Malware samples are provided with a report of analysis, which include the detection information of about 60 antivirus products. Samples have been automatically labeled through AVclass [27], which selects the most frequent name from

Table 4.11 Comparison of the scores of the signatures in the three cases of no optimization, hill climber (HC) and evolutionary optimization (SGX).

Cluster	TPs	Non Optimized		HC Optimizer		SGX Optimizer	
		Literals	Score	Literals	Score	Literals	Score
Cluster1	2/3	64	4263	10.30	608.70	21.00	400.00
	1/3	165	7526	13.80	592.60	17.40	440.00
	2/3	64	4263	9.40	603.10	21.70	409.30
	2/3	18	440	-	-	-	-
Cluster2	3/3	15	1145	8.00	651.50	5.00	405.00
	3/3	15	1145	8.70	630.00	5.00	406.00
Cluster3	4/4	260	17614	9.00	625.00	11.50	401.70
	4/4	260	17614	9.20	597.00	12.60	401.80
Cluster4	3/3	163	12015	8.80	597.30	10.30	408.30
	3/3	163	12015	9.10	616.10	10.10	406.50
Cluster5	3/4	64	2073	20.20	612.30	39.20	555.00
	1/4	59	1907	19.80	620.40	39.30	550.50
	1/4	59	1907	20.80	615.40	38.30	536.20
	3/4	64	2073	18.20	620.50	40.90	583.90
Cluster6	4/4	30	904	21.50	610.10	20.30	411.70
	4/4	30	904	20.90	611.20	19.60	400.60
Cluster7	4/4	157	12215	8.70	615.00	10.00	405.00
	4/4	157	12215	7.90	620.00	10.00	405.00
Cluster8	14/15	13	620	-	-	-	-
	1/15	91	2512	19.80	603.90	41.40	611.30
	3/15	21	400	-	-	-	-
	14/15	12	435	-	-	-	-
Cluster9	8/8	128	10725	7.70	608.50	21.80	412.00
	8/8	128	10725	7.20	619.50	22.70	425.50
Cluster10	7/20	20	421	-	-	-	-
	16/20	24	438	-	-	-	-
	1/20	37	750	30.50	617.50	29.20	405.20
	1/20	37	750	30.00	596.50	29.10	405.10
	18/20	22	408	-	-	-	-
	17/20	19	406	-	-	-	-

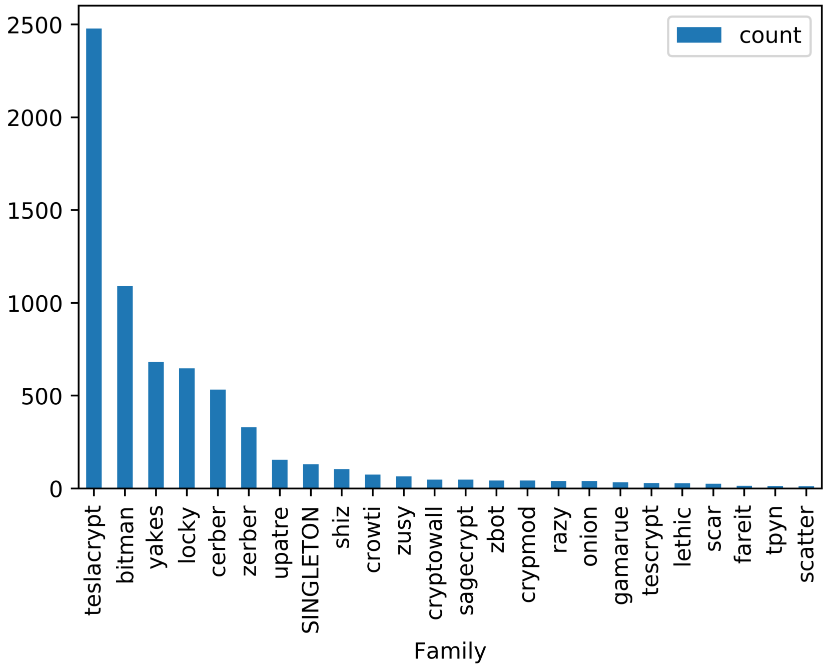

Fig. 4.2 Distribution of malware families with more than 10 samples from the VirusTotal dataset.

the antivirus signature names after a pre-processing step where standard names are removed (e.g., Trojan, Agent). Figure 4.2 illustrates the distribution of the families with more than 10 samples in the dataset.

Dataset analysis

Detailed information about the samples in the dataset, such as the packer detection or the compiler version, have been automatically extracted using an open dataset of YARA rules available on GitHub[9], where rules related to windows malware and packer detection have been manually selected. Results show that about the 63% of the samples match a packer rule.

[9] https://github.com/Yara-Rules/rules

Packer	Matched samples
Armadillo	2537
NSIS	338
PureBasic	105
PCGuard	95
aPLib	58
PECompact	48
Yoda	30
UPX	27
ASProtect	12
ASPack	5
VMProtect	4

Table 4.12 Number of matching samples for each packer rule.

About the 37% of the samples resulted to be packed by Armadillo, a well known commercial packer. The second most frequent match is NSIS (5%), an installer [10] which was reported to be used in ransomware attacks[11].

4.2.2 PE signatures

In order to test automatically generated rules for PE files, five criteria have been defined and addressed by the experimental results.

- Cluster quality, using the v-measure and homogeneity metrics.
- True positives, that is the number of malware samples from a specific family covered by the rule.
- False positives, that is the number of goodware samples misclassified as malicious.
- Dataset coverage, that is the total number of malware samples from the dataset under study that have been covered by the rule.

[10] https://nsis.sourceforge.io/Main_Page

[11] https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/ransomware-families-use-nsis-installers-to-avoid-detection-analysis/

- Packer resistance, that is the ability of the rule of matching malware samples, even though malware has been packed.

In the following results, the proposed approach, YaYaGenPE, is compared with two state of the art approaches YarGen (section 2.5.1) and YaBin (section 2.5.1), using several algorithm configurations. YaraGenerator (section 2.5.1) has been discarded, as it only managed to generate few signatures for few cases. On the other hand, BASS (section 2.5.1) cannot be used to generate signatures for more than few tens of samples, making such an approach non applicable in the performed experiments.

All experiments here reported were performed on a 6-core Intel Xeon (CPU E5-1650 v2 @ 3.50GHz), with 128 GB of RAM. However the proposed algorithms were executed on a single core: an optimized implementation of a multi-threads version is left as future work.

The following tables show the result of the tests which aim to compare several algorithm configurations, and the legend here reported includes the acronyms used to identify different configurations:

- U: unsupervised decision tree (clustering algorithm)
- HD: HDBSCAN (clustering algorithm)
- C: *clot* algorithm (rule generation subsection 3.3.1)
- G: pure greedy algorithm (rule generation, alternative to *clot*)
- RUL: complex features (i.e., other signatures) are used additionally to PE header ones.
- GOODWARE refers to the usage of goodware applications in the clustering process.

Table 4.13, Table 4.14, and Table 4.15 compare the number of rules, true positives (TP), and false positives (FP), for three malware families of different sizes. In particular, Table 4.13 refers to Cryptowall, a small malware family of 47 samples[12],

[12] There are thousands of different malware variants for each malware family here reported. The terms *small*, *medium*, and *large* refer to the size of the family within the dataset of study.

Table 4.13 Comparison of number of rules generated, false positives and true positives for the Cryptowall malware family.

Tool	Parameters	Num rules	FPs	TPs
YaYaGenPE	U + G	29	0	76
	U + G + RUL	31	0	75
	HD + G + RUL	23	0	80
YarGen	RUL Z0	53	1	130
	RUL Z0 + OPC	53	0	86
YaBin	Yara (-y)	36	0	76
	YaraHunt (-yh)	36	10	194

Table 4.14 Comparison of number of rules generated, false positives and true positives for the Cerber malware family.

Tool	Parameters	Num rules	FPs	TPs
YaYaGenPE	U + G	65	2	854
	U + G + RUL	64	2	896
	HD + G	137	0	768
YarGen	RUL Z0	328	7	705
	RUL Z0 + OPC	321	4	687
YaBin	Yara (-y)	157	0	737
	YaraHunt (-yh)	157	16	937

while Table 4.14 refers to Cerber, a malware family with 533 samples. Finally, Table 4.15 refers to a big malware families, with 2478 input samples. Results show that the number of generated rules by YaYaGenPE, regardless the algorithm configuration, is always lower than the other approaches. Such a difference is more relevant in case of thousands of inputs, such as in the Teslacrypt example, where the number of generated rules is about one order to magnitude less than the other approaches. On the other hand, the number of true positives and false positives is comparable with the other approaches.

Table 4.16 shows the result of the false positives test: a low number of unwanted detections is a key property of an effective signatures. In order to test the signatures, the *retrohunt* service of VirusTotal has been used: it tries to match a rules with about 100TB files [13], which corresponds to tens of millions of binaries. Only in the case of

[13] The size of the dataset is the same in all the runs of the retrohunt service, but the actual files can change in each run.

Table 4.15 Comparison of number of rules generated, false positives and true positives for the Teslacrypt malware family.

Tool	Parameters	Num rules	FPs	TPs
YaYaGenPE	U + G	497	0	3349
	U + G + RUL	493	0	3373
	HD + G	837	0	3237
YarGen	RUL Z0	2782	2	3367
	RUL Z0 + OPC	2760	0	3226
YaBin	Yara (-y)	1166	0	3172
	YaraHunt (-yh)	1166	68	4027

Table 4.16 Comparison of the false positives test for automatic generated rules for several malware families.

Family	Algorithm	Input size	Total matches*	FPs
OlympicDestroyer	U + G + RUL	22	143	0
Sagecrypt	HD + C + RUL	47	136	0
Crowti	U + G + RUL	75	66	0
Scatter	U + G	12	57	8
Scatter	U + G + RUL	12	35	4
Shiz	U + C + RUL	104	12	0

the Scatter family, the rules matched 8 unwanted applications, which is lowered to 4 if other complex features (i.e., existing ruleset, as described in subsection 3.4.1) are used. It is also possible to note how the generated signatures were able to extend the detections to other variants of the same family, extending the detection from two to six times the number of the original input samples.

Table 4.17 reports the result of the packer resistance test. Packers are commonly misused by malware developers to easily create new variants of the malware, increasing the difficulty of the detection and of the analysis. If signatures are automatically created on the code of the packer, they will match any sample which is packed with the same mechanism, regardless of being goodware or malware. In order to perform this experiment, each application of the goodware dataset has been packed with UPX, a well-known packer. The results show that rules created on malware packed samples with UPX do not match any binary of the goodware dataset. One possible explanation of the results, is that most of the fields of the PE header are left untouched by the majority of the packers.

Table 4.17 Test on false positives detection of packed samples. As the table shows, none of the rule matches any goodware packed sample.

Algorithm	rule:Cerber	rule:Locky	rule:Upatre	rule:Zerber
U + G	0	0	0	0
U + G + RUL	0	0	0	0
U + C	0	0	0	0
U + C + RUL	0	0	0	0
HD + G	0	0	0	0
HD + G + RUL	0	0	0	0
HD + C	0	0	0	0
HD + C + RUL	0	0	0	0

Table 4.18 Number of rules, average number of literals, and time necessary to cover the 3 malware families Fareit, Zerber, and Teslacrypt

Family	Size	Algorithm	Num. rules	Num literals (avg)	Time
Fareit	14	U + G	5	594	30s
Zerber	329	U + C + RUL	36	163	5m
		HD + C + RUL	86	187	5m
Teslacrypt	2478	U + G + RUL	493	381	3-4h
		HD + G+ RUL	850	336	3-4h

Table 4.18 shows the number of rules, the average number of literal for each rule, and the time necessary to automatically compute the rules for the three malware families in input. Several algorithm configurations are reported. The rule generation time spans from 30 seconds with 10 input samples, up to 4 hours if over 2 thousands samples are used. Although the generation time is a nonlinear function of the number of samples, it makes the approach applicable in practice even if large dataset are provided in input.

Table 4.18 shows that on average both the unsupervised decision tree and HDBSCAN produce clusters of five samples each, but the latter identifies about the 20% of the input points as outliers.

Finally, a detailed comparison of the execution time of the various algorithm configurations is reported in Table 4.19. UDT clustering is slower than HDBSCAN, taking about the double of the time to terminate, but it globally delivers better results. Similarly, using existing rulesets, which has the effect of increasing the number of

Table 4.19 Detailed comparison of the execution time for each algorithm configuration of the proposed approach.

Family	Algorithm	Set Size	Time
yakes	U + G	682	25m
yakes	U + G + RUL	682	40m
yakes	HD + G	682	10m
yakes	HD + G + RUL	682	10m
yakes	HD + C + RUL	682	25m

features for each sample under analysis, hence the computation time, effectively decrease the number of false positives. Indeed, reuse existing rulesets allows to exploit the knowledge that experts applied to select significant piece of information which are strongly indicators of a malicious behaviors.

The next paragraphs provide a detailed comparison of the results for 3 malware families of different size.

Cryptowall Cryptowall is an example of a small malware family, with 47 samples. Table 4.20, Table 4.21, Table 4.22, and Table 4.23 illustrate the results in terms of false positives, true positives and number of generated rules for each algorithm configuration.

Table 4.20 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the unsupervised decision tree (UDT) algorithm to cluster the Cryptowall malware family.

Algorithm	FP	TP	Num rules
G	0	76	29
G + RUL	0	73	29
G + GOODWARE	0	75	31
C	0	76	30
C + RULE	0	72	29
C + RUL + GOODWARE	0	72	29

Cerber Cerber is an example of a medium size family, with 533 samples. Table 4.24, Table 4.25, Table 4.26, and Table 4.27 illustrate the results in terms of

Table 4.21 Comparison of false positives, true positives and number of rules for several algorithm configurations, using HDBSCAN to cluster the Cryptowall malware family.

Algorithm	FP	TP	Num rules
G	2	85	20
G + RUL	0	80	23
G + GOODWARE	0	83	26
C	8	119	23
C + RUL	0	80	23
C + RUL + GOODWARE	0	80	23

Table 4.22 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the yarGen approach to create signature coverage for the Cryptowall malware family.

Algorithm	FP	TP	Num rules
RUL (DEFAULT MINSCORE)	15	122	36
RUL + EXCLUDEGOOD	15	122	36
RUL + OPCODES	0	45	36
RUL Z0	1	130	53
RUL Z0 + EXCLUDEGOOD	0	79	51
RUL Z0 + OPCODES	0	86	53

Table 4.23 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the YaBin approach to create signature coverage for the Cryptowall malware family.

Algorithm	FP	TP	Num rules
Yara (-y)	0	76	36
YaraHunt (-yh)	10	194	36

false positives, true positives and number of generated rules for each algorithm configuration.

Teslacrypt Teslacrypt is an example of a large family, with 2478 samples. Table 4.28, Table 4.29, Table 4.30, and Table 4.31 illustrate the results in terms of false positives, true positives and number of generated rules for each algorithm configuration.

Table 4.24 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the unsupervised decision tree (UDT) algorithm to cluster the Cerber malware family.

Algorithm	FP	TP	Num rules
G	2	854	65
G + RUL	2	896	64
G + GOODWARE	0	936	61
C	2	838	69
C + RUL	2	877	62
C + RUL + GOODWARE	0	882	68

Table 4.25 Comparison of false positives, true positives and number of rules for several algorithm configurations, using HDBSCAN to cluster the Cerber malware family.

Algorithm	FP	TP	Num rules
G	0	773	135
G + RUL	0	768	137
G + GOODWARE	0	773	135
C	0	773	136
C + RUL	0	768	137
C + RUL + GOODWARE	0	768	138

Table 4.26 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the yarGen approach to create signature coverage for the Cerber malware family.

Algorithm	FP	TP	Num rules
RUL (DEFAULT MINSCORE)	19	693	174
RUL + EXCLUDEGOOD	19	693	174
RUL + OPCODES	7	474	174
RUL Z0	7	705	328
RUL Z0 + GOODWARE	9	769	328
RUL Z0 + EXCLUDEGOOD	4	687	321

Table 4.27 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the YaBin approach to create signature coverage for the Cerber malware family.

Algorithm	FP	TP	Num rules
Yara (-y)	0	737	157
YaraHunt (-yh)	16	937	157

Table 4.28 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the unsupervised decision tree (UDT) algorithm to cluster the Teslacrypt malware family.

Algorithm	FP	TP	Num rules
G	0	3349	497
G + RUL	0	3373	493
G + GOODWARE	0	3376	497
C	0	3356	491
C + RUL	0	3386	478
C + RUL + GOODWARE	0	3403	479

Table 4.29 Comparison of false positives, true positives and number of rules for several algorithm configurations, using HDBSCAN to cluster the Teslacrypt malware family.

Algorithm	FP	TP	Num rules
G	0	3237	837
G + RUL	0	3179	850
G + GOODWARE	0	3237	837
C	0	3236	839
C + RUL	0	3180	850
C + RUL + GOODWARE	0	3180	850

Table 4.30 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the yarGen approach to create signature coverage for the Teslacrypt malware family.

Algorithm	FP	TP	Num rules
RUL (DEFAULT MINSCORE)	0	2306	1796
RUL + GOODWARE	3	2595	1796
RUL + OPCODES	0	2175	1796
RUL Z0	2	3367	2782
RUL Z0 + GOODWARE	15	3349	2782
RUL Z0 + EXCLUDEGOOD	0	3226	2760

Table 4.31 Comparison of false positives, true positives and number of rules for several algorithm configurations, using the YaBin approach to create signature coverage for the Teslacrypt malware family.

Algorithm	FP	TP	Num rules
Yara (-y)	0	3172	1166
YaraHunt (-yh)	68	4027	1166

Chapter 5

Case of Study: Android Banking Trojans

Part of the work described in this chapter has been previously published in *"The Rise of Android Banking Trojan", IEEE Potentials, 2019 [88].*

5.1 Introduction

As banks started developing mobile applications, or simply “apps”, to enable users to perform financial activities on-line, cybercriminals introduced new ways to penetrate this channel of communication and execute illicit transactions. As a matter of fact, for cybercriminals it is easier to exploit the scarce end-user security awareness and attack individual clients devices, while directly targeting banks portals.

Malicious programs that hide their intention under an apparently legitimate behavior are generically called “Trojans”. *Banking Trojans* are written with the specific purpose of stealing confidential information from victims bank accounts and on-line payment services. They are so common, that most sources just refer to them as “bankers”. Attackers commonly exploit social engineering techniques, inducing users to visit hostile websites and install malicious applications; alternatively, the malware is spread through official (i.e., Google Play) and unofficial app-stores.

In December 2016, the source code of *Bankbot*, one of the most well known Android banker, was released in the wild: since then, the number of its variants exploded, and, as of 2017, they represent the most prominent threat in the Android ecosystem. Trojans of the Bankbot variety, “Bankbots” for short, steal users credentials by accessing other apps private data, or displaying a fraudulent log-in page on top of legitimate banking apps. Moreover, some variants known as “SMS Trojans”, activate paid services by sending SMS to premium-rate numbers without the user knowledge. Finally, the gathered credentials are uploaded to a *Command&Control* server, which operates as an administrative panel, and it is used to constantly update the malware behavior.

This chapter surveys the Android banking Trojans evolution, their mode of operation. Finally, we highlight how the proposed methodologies can be applied to stop their diffusion.

5.2 History

The ancestor of banking Trojans is *Zeus*, a PC malware created in 2006, which managed to compromise over 3.5 million devices in US and created one of the history largest internet-connected network of infected devices. *Zitmo*, an abbreviation for “Zeus in the mobile”, was the first banker for Android: it emerged later, in 2010, and was devised to work in symbiosis with the desktop version and intercept two-factor authentication messages.

In 2011 Alexander Panin created *SpyEye*, a banker that attacks popular web browsers, targeting users of both Mac OS and Microsoft Windows operating systems. The author sold at least 150 copies of the code on the black market, at a price starting from $1,000, up to $8,500. When the FBI tracked down one of the buyers, the Russian cybergang *Soldier*, it claimed that over $3.2 million were stolen in just six months.

The 2012 saw the rise of the *Carberp* family, when a few samples were released through the official Google Play Store masqueraded as mobile applications from major Russian banks, such as Sberbank and Alfa Bank. The Trojans were devised to steal SMS messages and upload them to a remote server. The framework, composed of mobile and desktop versions, was offered on a monthly subscription-based model, with prices ranging between $2,000 and $10,000, depending on the number of additional modules. In June 2013, the source code was leaked, and new variants of the malware targeted U.S., Europe, and Latin America.

In 2013, the new banker *Hesperbot* appeared: it targeted users in Turkey, Czech Republic, Portugal and the United Kingdom, stealing personal data and SMS messages. The Android application uses an *activation code* to match the desktop version of the virus, making the attacker aware of new possible victims. Moreover, whenever the user tries to uninstall the application, the malware locks the screen using a run-time generated password, making the smart phone useless.

The forefather of a new generation of Android banking Trojans was *GM Bot*, possibly derived from *SimpleLocker*, the first known file-encrypting ransomware for Android. It first appeared in 2014 in Russian-speaking forums of the cybercrime underground, and, beyond intercepting SMS messages, it was the first to exploit the *overlay attack*, tricking users into entering their access credentials into a fraudulent window, and providing the attackers enough information for illicit money transfers out of their accounts. *GM Bot* was mostly distributed on third party app stores, without strict security checks, as an adult-content app or a plug-in app, like Flash.

New variants, with new and different capabilities were developed soon after, such as *MazarBot*, *SlemBunk*, *Bankosy*, *Acecard*, and *Slempo*. In the meantime, its original creator, *GanjaMan*, developed a second version which included three new Android exploits, and eventually sold it for $15,000.

Later, in June 2015, a new Trojan was discovered in Russia: *Bankbot.65.Origin* was disguised as the patched version of the official Sberbank online app, able to offer a wider range of mobile-banking features. The attack reached 100,000 Sberbank users, who in July of the same year reported losses for over 2 billion rubles (about $35 million).

Through 2016 and 2017, probably the most widespread family was *Bankbot*. Criminals managed to upload several variants on the official Google Play Store, that were downloaded thousands of times before being eventually removed. In December 2016, similarly to previous cases, the original source code was released in a forum, this time by a user named *Maza-in*, and it showed similarities with other well-known malware, such *MazarBot* and *RedAlert*.

Later variants ascribable to the family include *Faketoken*, which is capable of running overlay attacks for about 2,000 financial applications, monitoring active apps and showing a fraudulent screen as soon as one of the target is open, or *Tordow*, which targeted Russian users through popular apps, including *Telegram*, the Russian-speaking social-network *Vkontakte*, and games such as *Pokemon Go*, and *Subway Surfers*.

At the same time, malware anti-detection techniques are growing in sophistication and effectiveness too. For example, *Loapi* moves the malicious code outside the application, in a code module downloaded and executed at runtime, or delay the execution waiting for commands from a remote server, thus eluding detection tools. Other anti-malware detection techniques involve identification of the running system and application, for example checking the functionality of the execution environment and the file name characteristics, which are pretty standard in many sandbox settings (e.g., detection programs often creates hashes for file names they analyze, resulting in long and detectable file name characteristics). If malware gets aware of being in a sandbox, it will frustrate the analysis by merely not showing malicious behavior.

These adapting anti-detection tactics highlight the capability to circumvent fresh security countermeasures and call for the development of promptly evolving security strategies.

5.3 Modus Operandi

Hundreds of variants exist, and new ones are developed on a daily basis. However, all banker Trojans share common traits aiming to get user credentials and steal money.

Phase I: Infection

Banking Trojans exploit several vectors to infect a device. However, the most common ones are malicious web pages and legit app store markets. In the former, social engineering attacks are used to cheat users to visit hostile websites, and then a malicious JavaScript code infects the device downloading the malware. The power of such attacks is further increased if exploits unknown vulnerabilities in the mobile web browser, making the need for user intervention to a minimum. In the latter case, users commonly percept Android app stores, both official and third parties, as a trusted source, despite most of them offer an “open market model” where applications are distributed without any preliminary check. Among the others, the Google Play Store is of particular interest for the attackers: since it is the default source to download new apps, an infection would easily obtain widespread diffusion.

In the past, several banking malware campaigns, like *Acecard* and *Marcher*, tried to compromise users with malicious apps distributed via Google Play. Even though these apps are often removed within days after having been reported, they still manage to infect thousands of users. As a countermeasure, since February 2012 Google Play adopts *Bouncer*, an automated tool designed to discover and remove malicious applications from the official store. Furthermore, in May 2017, Google presented Play Protect, comprehensive security services for Android which also include an anti-virus embedded in the Google Play Services installed on each device.

On the other hand, a direct app analysis can be no longer sufficient: attackers hide their malware using *Droppers*, that is applications designed to remotely download and install others apps, mostly without any user intervention. Such type of attack, known as *drive-by-download*, or *multi-stage*, is particularly challenging to detect, as the same pattern is commonly used by goodware in some circumstances, such as during the software updates.

As the Android security model relies on the least privilege principle, each installed application is provided with the minimum capabilities to guarantee its functionalities. The access to additional sensitive resources, from the battery level to the Wi-Fi state, is limited by the grant of specific permissions. To help end-user choice, the operating system classifies permissions into four distinct groups according to their danger, but ultimately their acceptance relies on the user perception of risk [89]. So, social engineering techniques and targeted vulnerabilities could be used to trick users for unintentional permission granting, thus trespassing the Android security model.

Phase II: Persistence

Once installed, recent Trojans (like *Bankbot*) try to hide both from users and anti-virus and achieve persistence on the target device to operate seamlessly. As persistence is a requirement for any Trojan, a variety of solutions are applied.

A standard technique is to hide the application icon from the list of installed apps. Alternatively, if the application obtains administrative privileges (i.e., the most privileged state of the operating system), it tries to be installed as a system app, making its removal more complex, e.g. requiring administrative privileges as well.

In order to escape from antivirus (AV) detection, applications commonly use anti-analysis tricks. Among the others, *triggers* are very practical: the app shows a non-suspicious behavior until a specific condition, known as *trigger*, exhibits. Triggers are mainly used to mislead the detection during the AV analysis, which is usually limited to few minutes of run-time monitoring. Typically, both “time bombs” (i.e., a timer) and device reboots are used as triggers.

As well as to escape AV detection, *drive-by-download* is commonly applied to dynamically adapt the malware behavior to the target device and user: initial versions embedded the list of target banks in plain text within the source code of the malware, despite could be easily detectable. On the contrary, in more recent malware the target information is either obfuscated or dynamically collected from a remote server, exploiting several file formats, and possibly encrypting the communication. For instance, in October 2017, malware researchers discovered the *Tornado FlashLight*, a Bankbot that uses the hash of the package names to identify the target banking apps.

In the same way, malware authors progressively moved the payload (i.e., the malicious behavior) outside the Trojan, either using Dropper to download a “second-stage” malware, or directly moving the malware code to native C/C++ libraries, like in the *Corkow* family.

Command&Control

In order to control their Trojans, attackers setup complex *Command&Control* (C&C) infrastructures, that is malicious networks where infected devices receive instructions (i.e. the *command*), from a central entity (i.e. the *control*).

Initially, communications were done using simple HTTP requests in plain text, while exploiting cheap botnets as C&C. Later, malware adopted increasingly complex encryption routines, obfuscating the password within the application package. Recent malware campaigns show that hiding such communications is vital for the malware, and the most disparate techniques are used. For instance, the banker family *Twitoor* exploits tweets published by a Twitter account, while other samples from the *Charger* family misuse the Firebase Cloud Messages.

Being able to communicate with the malware, attackers can change the targets of the attacks, as well as keep up to date the fraudulent log-in screens according to the new graphical releases of the legit banking apps. Moreover, applications send log information, credentials and credit card details to the C&C infrastructure, which acts as an administrative panel for the threat actors.

Phase III: the Attack

Bankbots could achieve their malicious behavior in different ways, mainly depending on the availability of superuser access rights.

If rooting capabilities are granted, the malware could subvert the Android security model and steal confidential data directly from other applications key store, like log-in passwords and bank card details. Furthermore, it could maliciously tamper the web traffic and alter web pages content to redirect users to fraudulent websites.

Otherwise, if superuser access rights are not granted, overlay attacks and screen recording could be applied. In the former case, the malware displays log-in forms seemingly coming from legitimate banking apps and trick users to insert their credentials into the fraudulent forms. In the latter, bankers could use screen recording or screen-shotting as a graphical keylogger to steal sensitive information manually inserted from the users.

Furthermore, malicious bankers commonly have the capability of intercepting text messages to bypass SMS-based 2-factor authentication, and send SMS to activate paid services.

Finally, some malware also provide locking capabilities, allowing attackers to remotely lock devices using a fake update screen to hide fraudulent activity, and ensuring that user cannot interfere.

Rooting

Obtaining superuser access rights, a procedure known as *rooting*, allows users to take full control over their devices. For some applications, including popular ones from Google Play Store like *GMD Gesture* and *FolderMount*, this is a requirement to function properly. Although rooting is commonly seen as a way to extend the capabilities of an Android device (e.g., game hacking, delete preinstalled apps, CPU overclocking), it has substantial security implications, since it overcomes the Android basic security principle. Normally, applications are executed in isolated environments without access to other apps or private system files. Rooting removes such limitation, eventually allowing free access to all the system resources and data.

The dark side of rooting is that malware could silently obtain administrative rights without the user agreement by exploiting a vulnerability in the operating system: Android is a Linux-based platform, and it shares most of its underlying code with the desktop operating system, hence most kernel exploits work roughly the same way.

Some rooting techniques are pretty naive, like the one presented in DRAMMER [90] which exploits an issue with new generation DRAM chips: repeatedly accessing a row of memory can cause a "bit flipping" in an adjacent row, allowing anyone to change the value of contents stored in the memory.

Although the original version of Bankbot did not exploit unauthorized rooting capabilities, most of the samples include checks to verify if the device is rooted, and thus directly steal user credentials in other apps key store or tamper web traffic.

On February 2016 was discovered *Tordow*, a malware which used a popular exploit pack to have unlimited access to the key store of the default Android browser and Google Chrome, being able to steal log-in details, passwords, and saved bank cards. Later, on March 2016, researchers identified a new Trojan named *Triada*, which gains unauthorized superuser privileges too. The application behaves like a Dropper, using a remote server to get the list of new applications to download, eventually installing new Trojans. One of them, *Triada.p/o/q* tampers URLs loaded in a browser, especially targeting online banking platforms.

The Overlay Attack

The *overlay* attack is a common strategy adopted by malware authors to draw fraudulent screens on top of a target application. Overlay was initially introduced in the Android platform as a way to increase the user experience. For example, Facebook Messenger uses overlays to pop-up "Chat Heads" alerts indicating that a new message has been received.

Commonly Bankbots monitor which apps are installed on the infected device; then, if sensitive applications are detected (e.g., Online Banking, VPN, Social

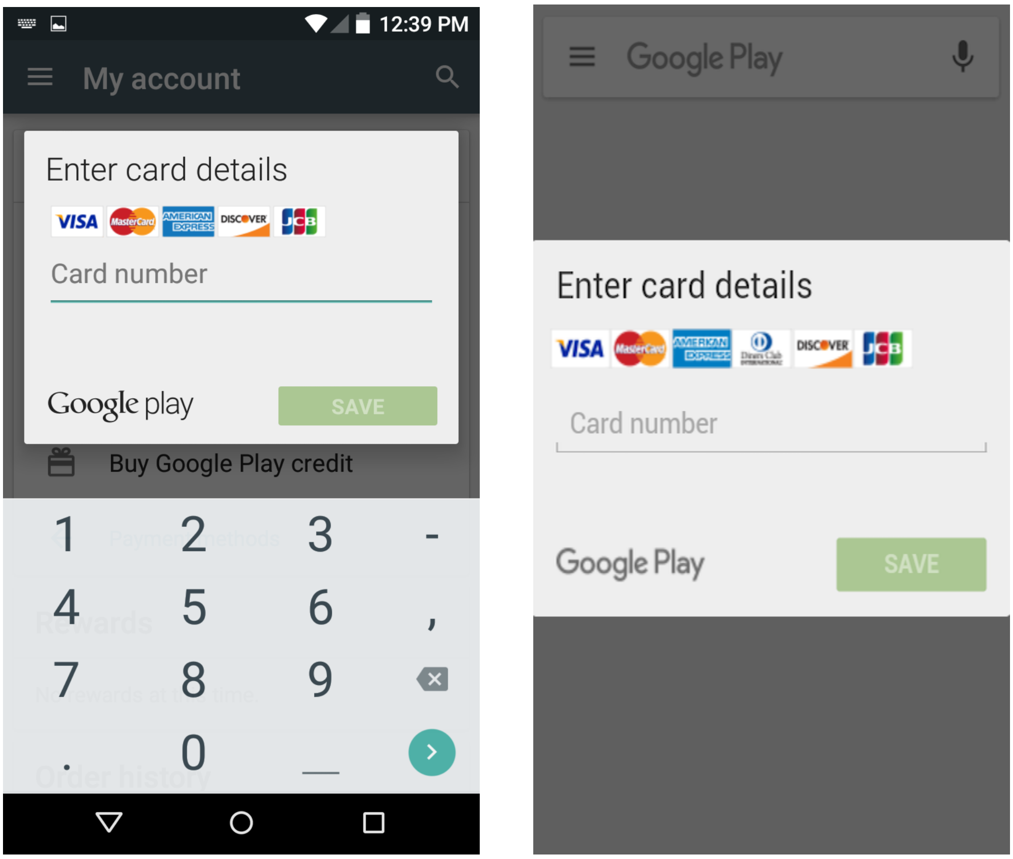

Fig. 5.1 Overlay attack on the Google Play Store. Picture on the left shows the original user interface, while the one on the right shows the fraudulent pop-up displayed during an overlay attack. In this case the attackers were able to replicate the same look and feel of the original application.

Networks) the malware redraw a similar log-in screen on top of the legitimate ones. In most of the cases, the user is not able to distinguish the fraudulent version from the original one and will insert its credentials which will uploaded on a C&C server. Figure 5.1 shows an example of overlay attack at the Google Play Store app: a malicious pop-up is displayed over the app, asking the credit card number.

Overlay attacks come in two variants, and both allow rogue developers to create apps that include the overlay functionality without user awareness. In the former, applications require a permission called *SYSTEM_ALERT_WINDOW*. Due to a peculiarity in the Android ecosystem, apps directly downloaded from the Google Play Store do not require users to grant it explicitly. In the latter, malware achieve a similar result abusing the *Toast* component, which is typically used to display a quick message, for instance indicating that an email has been removed. Due to a vulnerability in the Android operating system, its usage does not require any specific permission in all the Android versions prior the *7.1*.

The efficacy of overlay attack may increase if applied in combination with the *BIND_ACCESSIBILITY_SERVICE* permission which grants an app the ability to discover UI widgets displayed on the screen, query their content, and interact with them programmatically, introduced as a means to make Android devices more accessible to users with disabilities.

Moreover, if social engineering techniques and misleading overlays are applied together, attackers could even trick users to enable the Accessibility Services, as well as any other permission, without their explicit consensus. Similarly, Droppers may enable the "Unknown Sources" option in the Settings menu, and silently install new malicious applications without the user knowledge, overtaking the operating system security measure that blocks the installation of apps from untrusted sources.

Overlay issues are known since 2011, but the attack was lead to fame in 2017, when the *Cloak&Dagger* [91] research showed how granting only two permissions (i.e., System Alert Window and Accessibility Services) could result in a very effective technique to steal user credentials. When researchers tested a simulated Cloak&Dagger attack on 20 Android users, none of them was aware its device was being hacked.

As a remedy, in the *Mashmellow* (*v6.0*) release Android changed the default behavior, requiring users to manually approve the access to the overlay capabilities for all the apps, including those downloaded from the Google Play Store. Furthermore, Android *Oreo* (*v8.0*) includes a visual notification whenever an overlay is displayed, allowing the user to easily dispose it. However, since the attack works seamlessly in all the Android versions prior to *Oreo*, the scope of devices covered is still huge.

The ability to replicate high-quality user interfaces is essential to guarantee the phishing attack. For instance, Figure 5.2 compares two log-in pages for the Skype application; although graphically different, they have the same look and feel of the original Skype, and could easily mislead the user. Even for the most careful one, recognizing the legitimate version is hard due to the frequent graphical updates and variants that applications show.

[1] https://github.com/geeksonsecurity/android-overlay-malware-example

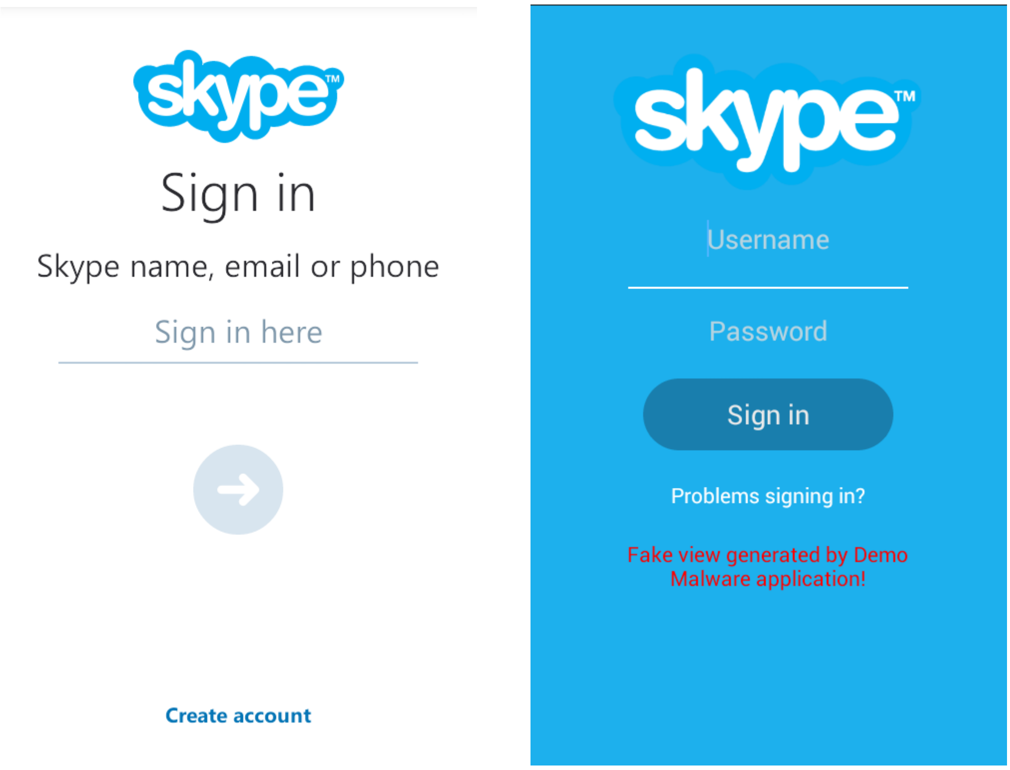

Fig. 5.2 Comparison between the original Skype log-in page (on the left), and the one prompted during an overlay attack (on the right). Pictures are generated using the android-overlay-malware-example[1].

Initially, only a few Russian Banks were included, but soon the number of targets was expanded. In order to operate silently to avoid AV detection, and keep up-to-date the targets, custom log-in screens are downloaded at run-time in HTML format from the C&C server, while other times a common template is used, downloading and updating only bank icons and writings.

Recent malware discovered with overlay capabilities include *Charger*, a banking Trojan masquerade as a flashlight app removed from the Google Play Store after 5,000 downloads: the targets were generic, covering from Westpac Mobile Banking, to Facebook and WhatsApp. Then, on November 2017, researchers found *Crypto Monitor*, used to track the cryptocurrency market, and *StorySaver* a third-party tool to save Instagram stories, both designed to target fourteen different Polish banks. In the same month, was found on Google Play *Smart AppLocker*, posed as a legit app that secure device applications using a PIN code, but it was the first known malware to exploit the Toast overlay attack: at the time of removal count between 100,000 and 500,000 installations.

SMS Spoofing

As online banking increased popularity, bank institutes adopted new countermeasures to ensure the security of online operations and stop banking frauds. Since the adoption of the two-factor authentication (2FA), a one-time password (OTP) uniquely generated for each new operation, attackers deployed new features to steal it.

Since the first versions of Bankbot, Trojans include SMS spoofing capabilities, being able to receive SMS and hide their notification to the users, finally sending the OTP to the C&C server.

Furthermore, if the permission of sending SMS is granted, either with the user consensus or not, malware can activate paid premium services and ensure an immediate reward to malicious actors. Even if this type of attack is inherently location dependent, as each premium phone number is bound to a specific area, the C&C infrastructure can enlarge its scope, providing an updated list of numbers according to the user position.

The Social Engineering Role

The term "social engineering" refers to the psychological manipulation of users, tricking their confidence with nefarious purposes, for example the illegal gathering of sensitive information. Bankbots regularly apply social engineering to circumvent the technical security measures imposed by the operating system, and more in general by the Android ecosystem: malware authors commonly deceive users to visit harmful web pages, install unwanted applications, grant risky permissions, and insert sensitive information in fraudulent log-in or payment forms.

By sending counterfeit SMS, and displaying intriguing advertisements that impersonate well-known and trusted parties, attackers encourage their victims to download applications from untrusted sources. Moreover, attackers usually exploit malicious repackaging: well-known apps are decompiled, modified inserting some malicious behavior, and finally redistributed. For example, the malware *Tordow* used popular app names like *Skype* and *MX Player*, while the app *SexDrugVokrug*, from the

Twitoor family, aims to mislead the users of the Russian Social Network “Drug Vokrug”.

Then, in order to maximize the efficacy of the attack, typical techniques include creating mirror icons of well-known apps (e.g., Gmail, Google Play and Chrome) and displaying notifications linking to a fraudulent “sign-in to your account” page, despite users could become suspicious from such a behavior, and the malware could be easily identified.

Finally, the family of overlay attacks is another example of social engineering, as it exploits the user confidence in well-known log-in and payment forms to steal user credential and credit card information.

Chapter 6

Conclusions

In this thesis, we introduced a set of semi-supervised techniques with the ultimate goal of assisting human experts in the generation of malware family signatures. As a result, we developed a scalable framework able to dig into real-world antivirus datasets of applications, with the main purpose of identifying new malware samples, and reducing false positive detections.

Our study shows that best results can be achieved combining the scalability of the automatic techniques with the inherent flexibility of the manual analysis. Eventually, the proposed approach introduces two essential automation improvements to assist the signatures generation, the standard detection mechanism in the antivirus industry.

An iterative clustering algorithm allows an easy identification of hard to find potential threats, reducing the human intervention from the manual analysis of thousands of applications to the validation of a much smaller number of clusters, where the applications reflect a similar class of behavior.

Subsequently an automated procedure named *clot*, which exploits a heuristic optimization strategy, generates a set of signatures to cover newly identified malware with an acceptable generalization capability, yet minimizing false positives. Moreover, an evolutionary strategy, based on the *Selfish Gene* algorithm, has been devised to further increase the detection of new malware variants, while minimally affecting the number of unwanted detections.

The proposed approach has been validated with both Android and Windows malware, and it has been implemented in two frameworks, *YaYaGen* and *YaYaGenPE*, which tackle the specific needs of the automatic signature generation for each platform.

Experimental results on 1.5 million unique Android applications confirm the effectiveness of the proposed methodology, in both the identification of new malware samples, and in the generation of new family signatures in the form of YARA rules. More in details, the clustering identified 21,450 applications ready for the signature coverage without requiring human validation, and other 20,464 potential new threats, subdivided in 500 tight malware families. Moreover, since January 2018 YaYaGen and the clustering system is in use on Koodous, the mobile antivirus platform developed by Hispasec. The evaluation for Windows malware was conducted on a dataset of ten thousands applications, and experimental results show that it successfully fulfilled all the defined requirements, which include the cluster quality, low true positives and false positives rates, the overall malware coverage, and the resistance to the most common open source packing software.

Similarly to others machine learning based approaches, the proposed system is vulnerable to noise-injection and other adversarial attacks, however the semi-supervised system is designed to suggest malware families to the experts, rather than completely replacing their role. On the other hand, application packing and obfuscation represent a limitation of the approach, since the ability of generating effective rules requires the analysis of the applications to be as precise as possible.

The presented work can be extended in a number of ways. For the clustering, fuzzy hashing techniques should be explored as a more efficient way to find clusters of similar applications. For instance, Locality-sensitive hashing (LSH) algorithms, such as SimHash or MinHash, can handle million of input vectors in an efficient way, and approximate nearest neighbors (ANN) techniques can be used for a fast and approximate clustering, while a more refined one can be used as a second step. For the rule generation, other approaches, like the the ones based on the graph isomorphism problem should be explored too.

The thesis concludes with the study of the most prominent recent malware threat in the Android ecosystem: the Android banking trojans, that is applications written with the specific purpose of stealing confidential information from victims bank accounts through online payment services. The chapter surveys the banking trojans history, their evolution, and mode of operation.

References

[1] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection*, pages 98–115. Springer, 2008.

[2] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[3] Saba Arshad, Munam Ali Shah, Abid Khan, and Mansoor Ahmed. Android malware detection & protection: a survey. *Int. J. Adv. Comput. Sci. Appl*, 7(2):463–475, 2016.

[4] Google_android_security_2016_report_final.pdf. https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf, Mar 2017. (Accessed on 11/12/2017).

[5] The judy malware: Possibly the largest malware campaign found on google play - check point software. https://blog.checkpoint.com/2017/05/25/judy-malware-possibly-largest-malware-campaign-found-google-play/. (Accessed on 07/21/2019).

[6] Martin Apel, Christian Bockermann, and Michael Meier. Measuring similarity of malware behavior. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 891–898. IEEE, 2009.

[7] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[8] Alejandro Calleja, Juan Tapiador, and Juan Caballero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 2018.

[9] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[10] R Winsniewski. Android–apktool: A tool for reverse engineering Android apk files, 2012.

[11] Jesus Freke. Smali, an assembler/disassembler for android's dex format. *Google Project Hosting [online] http://code. google. com/p/smali*, 2013.

[12] Anthony Desnos. Androguard: Reverse engineering, malware and goodware analysis of Android applications... and more (ninja!). *Retrieved June*, 10:2014, 2011.

[13] Anthony Desnos and Patrik Lantz. Droidbox: An android application sandbox for dynamic analysis, 2011.

[14] David Korczynski. Clusthedroid: Clustering Android malware. 2015.

[15] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[16] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, volume 10, page 14, 2010.

[17] Charles T Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on computers*, 100(1):68–86, 1971.

[18] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. Data mining cluster analysis: basic concepts and algorithms. *Introduction to data mining*, 2013.

[19] 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*, volume 96, pages 226–231, 1996.

[20] Ricardo JGB Campello, Davoud Moulavi, and Joerg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 160–172. Springer, 2013.

[21] How HDBSCAN works — hdbscan 0.8.1 documentation. https://hdbscan.readthedocs.io/en/latest/how_hdbscan_works.html. (Accessed on 03/27/2017).

[22] Ricardo JGB Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(1):5, 2015.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[24] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.

[25] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.

[26] Roberto Perdisci et al. Vamo: towards a fully automated malware clustering validity analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 329–338. ACM, 2012.

[27] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[28] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[29] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[30] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[31] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[32] Tony Lee and Jigar J Mody. Behavioral classification. In *EICAR Conference*, pages 1–17, 2006.

[33] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[34] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[35] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. Anubis: Analyzing unknown binaries, 2009.

[36] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[37] Maria Halkidi, Yannis Batistakis, and Michalis Vazirgiannis. On clustering validation techniques. *Journal of intelligent information systems*, 17(2-3):107–145, 2001.

[38] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198, 2013.

[39] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *NDSS*, 2015.

[40] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[41] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, 2015.

[42] Martin Pelikan, Mark W Hauschild, and Fernando G Lobo. Estimation of distribution algorithms. In *Springer Handbook of Computational Intelligence*, pages 899–928. Springer, 2015.

[43] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. The selfish gene algorithm: a new evolutionary optimization strategy. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 349–355. ACM, 1998.

[44] Richard Dawkins. The selfish gene. *New York*, 1976.

[45] Ari Juels, Shumeet Baluja, and Alistair Sinclair. The equilibrium genetic algorithm and the role of crossover. *Unpublished manuscript*, 1993.

[46] Shumeet Baluja. Population-based incremental learning. A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, 1994.

[47] Heinz Mühlenbein and Gerhard Paass. From recombination of genes to the estimation of distributions i. binary parameters. In *International conference on parallel problem solving from nature*, pages 178–187. Springer, 1996.

[48] Georges R Harik, Fernando G Lobo, and David E Goldberg. The compact genetic algorithm. *IEEE transactions on evolutionary computation*, 3(4):287–297, 1999.

[49] Junwu Zhang, Michael L Bushnell, and Vishwani D Agrawal. On random pattern generation with the selfish gene algorithm for testing digital sequential circuits. In *Test Conference, 2004. Proceedings. ITC 2004. International*, pages 617–626. IEEE, 2004.

[50] Rui Tavares, António Teófilo, Paulo Silva, and Agostinho C Rosa. Infected genes evolutionary algorithm. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 333–338. ACM, 1999.

[51] Noor Elaiza Abdul Khalid, Norharyati Md Ariff, Saadiah Yahya, and Noorhayati Mohamed Noor. A review of bio-inspired algorithms as image processing techniques. In *International Conference on Software Engineering and Computer Systems*, pages 660–673. Springer, 2011.

[52] Fulvio Corno, M. Sonza Reorda, and Giovanni Squillero. Optimizing deceptive functions with the SG-clans algorithm. In *Congress on Evolutionary Computation*, pages 2190–2195. IEEE, 1999.

[53] Virus Bulletin :: Rule-driven malware identification and classification. https://www.virusbulletin.com/virusbulletin/2008/01/rule-driven-malware-identification-and-classification, January 2008. (Accessed on 04/03/2017).

[54] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM computer communication review*, 34(1):51–56, 2004.

[55] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[56] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, volume 4, pages 4–4, 2004.

[57] Ke Wang, Gabriela Cretu, and Salvatore J Stolfo. Anomalous payload-based worm detection and signature generation. In *International Workshop on Recent Advances in Intrusion Detection*, pages 227–246. Springer, 2005.

[58] Vinod Yegneswaran, Jonathon T Giffin, Paul Barford, and Somesh Jha. An architecture for generating semantic aware signatures. In *USENIX Security Symposium*, pages 97–112, 2005.

[59] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[60] Konrad Rieck, Guido Schwenk, Tobias Limmer, Thorsten Holz, and Pavel Laskov. Botzilla: Detecting the phoning home of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1978–1984. ACM, 2010.

[61] Christian Rossow and Christian J Dietrich. Provex: Detecting botnets with encrypted command and control channels. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2013.

[62] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[63] Weidong Cui, Marcus Peinado, Helen J Wang, and Michael E Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 252–266. IEEE, 2007.

[64] Peter Wurzinger, Leyla Bilge, Thorsten Holz, Jan Goebel, Christopher Kruegel, and Engin Kirda. Automatically generating models for botnet detection. In *European symposium on research in computer security*, pages 232–249. Springer, 2009.

[65] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Security and Privacy, 2005 IEEE Symposium on*, pages 226–241. IEEE, 2005.

[66] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. Androsimilar: robust statistical feature signature for Android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 152–159. ACM, 2013.

[67] Min Zheng, Mingshen Sun, and John CS Lui. Droid analytics: a signature based analytic system to collect, extract, analyze and associate Android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 163–171. IEEE, 2013.

[68] Bass. https://github.com/Cisco-Talos/BASS.

[69] Andrea Atzeni, Fernando Díaz, Andrea Marcelli, Antonio Sánchez, Giovanni Squillero, and Alberto Tonda. Countering android malware: A scalable semi-supervised approach for family-signature generation. *IEEE Access*, 6:59540–59556, 2018.

[70] YARA — The pattern matching swiss knife for malware researchers. https://virustotal.github.io/yara/, November 2013. (Accessed on 03/27/2017).

[71] Leland McInnes and John Healy. Accelerated hierarchical density clustering. *arXiv preprint arXiv:1705.07321*, 2017.

[72] Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[73] Eliana Giovannitti, Luca Mannella, Andrea Marcelli, and Giovanni Squillero. Evolutionary antivirus signature optimization. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, 2019.

[74] Ling Wang, Chen Fang, Chun-Di Mu, and Min Liu. A pareto-archived estimation-of-distribution algorithm for multiobjective resource-constrained project scheduling problem. *IEEE Transactions on Engineering Management*, 60(3):617–626, 2013.

[75] Tracking malware with import hashing. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

[76] Things you didn't known about portable executable file format (pecoff). https://media.blackhat.com/bh-us-11/Vuksan/BH_US_11_VuksanPericin_PECOFF_Slides.pdf.

[77] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. Kharon dataset: Android malware under a microscope. In *The Learning from Authoritative Security Experiment Results (LASER) workshop*. The USENIX Association, 2016.

[78] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[79] James Newsome, Brad Karp, and Dawn Song. Paragraph: Thwarting signature learning by training maliciously. In *International Workshop on Recent Advances in Intrusion Detection*, pages 81–105. Springer, 2006.

[80] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[81] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. Poisoning behavioral malware clustering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 27–36. ACM, 2014.

[82] Jonathan Crussell and Philip Kegelmeyer. Attacking dbscan for fun and profit. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 235–243. SIAM, 2015.

[83] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), mar 2017.

[84] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external evaluation measure. In *EMNLP-CoNLL*, volume 7, pages 410–420, 2007.

[85] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[86] malicialab/avclass: AVClass malware labeling tool. https://github.com/malicialab/avclass, July 2016. (Accessed on 03/27/2017).

[87] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. Experimental study of fuzzy hashing in malware clustering analysis. In *8th workshop on cyber security experimentation and test (cset 15)*, volume 5, page 52. USENIX Association, 2015.

[88] Andrea Atzeni, Fernando Díaz, Francisco Lǿpez, Andrea Marcelli, Antonio Sánchez, and Giovanni Squillero. The rise of android banking trojans. *IEEE Potentials*, 2019.

[89] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012.

[90] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1675–1689. ACM, 2016.

[91] Yanick Fratantonio, Chenxiong Qian, Simon Chung, and Wenke Lee. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.