PhD-FSTC-2019-49

The Faculty of Sciences, Technology and Communication

DISSERTATION

Presented on 01/07/2019 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Médéric HURIER

Born on the 25th of November 1988 in Reims (France)

CREATING BETTER GROUND TRUTH TO FURTHER UNDERSTAND ANDROID MALWARE: A LARGE SCALE MINING APPROACH BASED ON ANTIVIRUS LABELS AND MALICIOUS ARTIFACTS

Abstract

Mobile applications are essential for interacting with technology and other people. With more than 2 billion devices deployed all over the world, Android offers a thriving ecosystem by making accessible the work of thousands of developers on digital marketplaces such as Google Play. Nevertheless, the success of Android also exposes millions of users to malware authors who seek to siphon private information and hijack mobile devices for their benefits.

To fight against the proliferation of Android malware, the security community embraced machine learning, a branch of artificial intelligence that powers a new generation of detection systems. Machine learning algorithms, however, require a substantial number of qualified samples to learn the classification rules enforced by security experts. Unfortunately, malware ground truths are notoriously hard to construct due to the inherent complexity of Android applications and the global lack of public information about malware. In a context where both information and human resources are limited, the security community is in demand for new approaches to aid practitioners to accurately define Android malware, automate classification decisions, and improve the comprehension of Android malware.

This dissertation proposes three solutions to assist with the creation of malware ground truths.

The first contribution is STASE, an analytical framework that qualifies the composition of malware ground truths. STASE reviews the information shared by antivirus products with nine metrics in order to support the reproducibility of research experiments and detect potential biases. This dissertation reports the results of STASE against three typical settings and suggests additional recommendations for designing experiments based on Android malware.

The second contribution is EUPHONY, a heuristic system built to unify family clusters belonging to malware ground truths. EUPHONY exploits the co-occurrence of malware labels obtained from antivirus reports to study the relationship between Android applications and proposes a single family name per sample for the sake of facilitating malware experiments. This dissertation evaluates EUPHONY on well-known malware ground truths to assess the precision of our approach and produce a large dataset of malware tags for the research community.

The third contribution is AP-GRAPH, a knowledge database for dissecting the characteristics of malware ground truths. AP-GRAPH leverages the results of EUPHONY and static analysis to index artifacts that are highly correlated with malware activities and recommend the inspection of the most suspicious components. This dissertation explores the set of artifacts retrieved by AP-GRAPH from popular malware families to track down their correlation and their evolution compared to other malware populations.

Acknowledgments

The pursuit of knowledge is a noble aspiration that led humanity through its most epic quests.

First and foremost, I want to express my gratitude to Pr. Yves Le Traon for allowing me to do my Ph.D. studies at the University of Luxembourg and to help him teach the Big Data course during four semesters. Yves was a supportive mentor and a wise counselor, even during the most challenging time of my studies.

I also want to say my sincere thanks to Dr. Jacques Klein and Dr. Tegawende Bissyande for their excellent advice and their collaboration on the field of Android security. I have learned a lot thanks to their experience and their insights on the world of academia.

My research experiments on Android malware would not have been possible without the work accomplished by the SerVal group on the Androzoo project. Thank you to SerVal, especially Dr. Kevin Allix, for maintaining this malware dataset, releasing it to the research community, and for our collaboration on extending the platform.

Besides, I thank the administrative team of the University of Luxembourg, and Solène Vincens in particular, for the organization of my thesis defense and their assistance during my Ph.D. studies. Thanks also to the team at the University of Luxembourg responsible for the High-Performance Computing (HPC) platform for letting me run my batch processing scripts.

On the 21st of March 2018, I had the opportunity to present my work to the team at Google responsible for the security of the Android platform. I am grateful to Pr. Yves Le Traon and Dr. Damien Octeau for giving me the chance of visiting Google headquarters in the beautiful city of San Francisco.

My Ph.D. studies were also a moment to meet and discuss with passionate colleagues and friends. Thank you to Dr. Matthieu Jimenez, Antonin Carette, Sankalp Ghatpande, Dr. Kevin Allix, Dr. Alexandre Bartel, Dr. Nicolas Sannier, Dr. François Fouquet, Dr. Assaad Moawad, Dr. Thomas Hartmann and Dr. Grégory Nain for the numerous discussions on the ultimate question of life, the universe, and the best programming language.

Furthermore, I want to thank the creators and contributors of the open source software that I used during my Ph.D. studies: Linux, Ubuntu, GNOME, Vim, Python, Clojure, Datomic, ElasticSearch, PostgreSQL, CouchDB, Jupyter, Celery, LaTeX, and all NumFOCUS projects.

Finally, I want to dedicate this dissertation to the loves of my life, my dear wife Alexandra Jacob, and our two daughters, Eglantine and Augustine Hurier Jacob. I also thank my sister, my parents, my great parents, and my parents in law for their interest in my research.

Table of Contents

List of Figures

List of Tables

Part I.

Introduction, background, and state of the art

Chapter 1.

Introduction

This chapter motivates the need of creating better malware ground truth.

The first section analyzes the state of mobile security for Android application.

The second section discusses challenges for the Android security community.

The third section summarizes the research contributions of this dissertation.

Table of Contents

1.1. Mobile security in the real world

1.1.1. Mobile security and innovation

1.1.1.1. Growths of mobile ecosystems

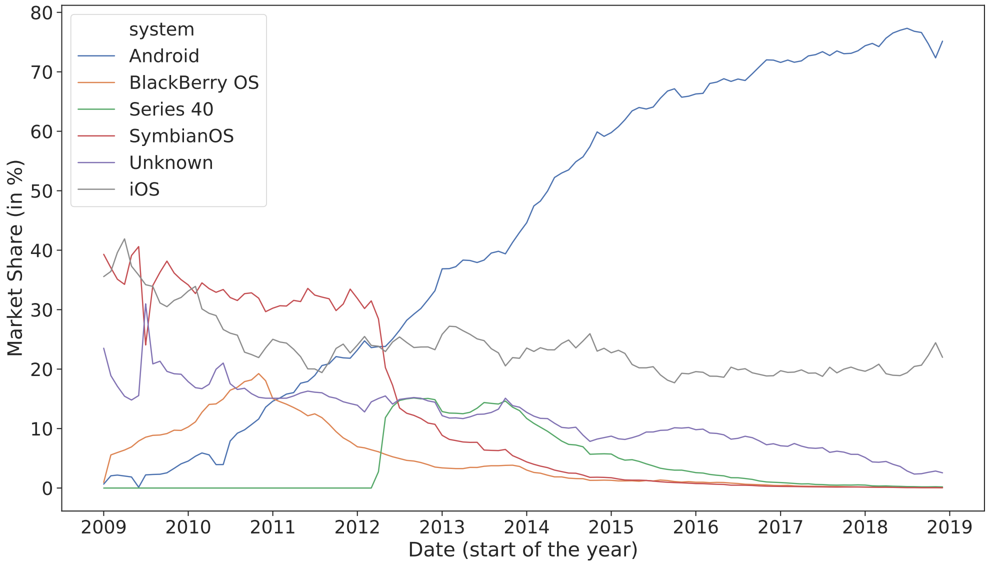


Figure 1.1.: Market shares of mobile operating systems, StatCounter [sta19a]

The mobile industry is a sector in constant evolution, thanks to the adoption of innovative technologies such as Android. In 2018, Google LLC identified at least 2 billion active devices powered by a version of Android [Goo18]. Moreover, Figure 1.1 shows that Android became the leading mobile operating system in 2012, with a global market share superior to 70% in 2019.

The success of Android can be explained by the development of the supply and demand for mobile services. On the one hand, mobile users adopt Android solutions for managing everything that relates to their personal and professional life, ranging from social networking to financial investments. On the other hand, Android developers meet the demand of mobile users with a growing catalog of applications distributed on digital marketplaces such as Google Play [Goo19a]. Figure 1.2 illustrates the growing number of applications released on Google Play from 2009 to 2018. We can observe that 3.5 million applications were available on the platform in 2018 [Sta19b]. In this context of social and economic growth, the task of protecting mobile devices has become more important than ever to support the expansion of mobile ecosystems.

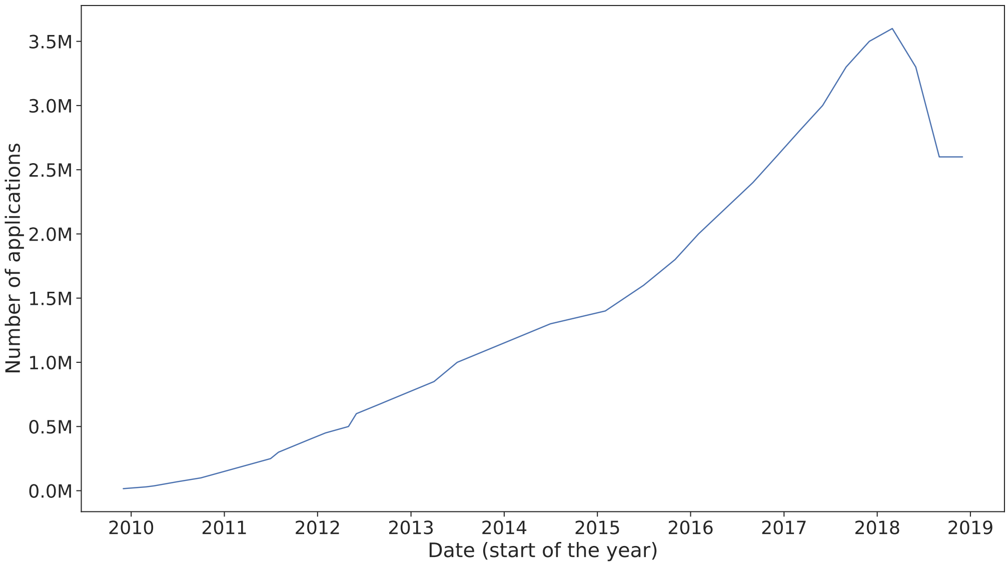


Figure 1.2.: Number of applications available on Google Play, Statista [Sta19b]

1.1.1.2. Implications for mobile security

We can define mobile security as the perception that nothing wrong can happen to mobile users, mobile hardware, or the information stored on mobile devices. It is a fundamental right that every mobile user should be entitled to, even for immaterial assets such as personal information.

However, not a single month goes by without a global security breach or a privacy issue that impacts thousands of users [Les18]. Figure 1.3 shows that the top 21 data breaches of 2018 impacted between 200,000 and 1 billion users each. We can conclude from these observations that the global lack of security continues to have real consequences on people life, which may deter mobile users from trusting digital marketplaces.

1.1.2. Mobile security as an arms race

1.1.2.1. Profiles of security attackers

Mobile security involves two main actors: an attacker whose goal is to harm the security of mobile users, and a defender that must guarantee an acceptable level of security in the presence of attackers. Attackers can be criminals, activists, state-affiliated employees, or other types of associations. Figure 1.4 shows the distribution of attackers' origin and motive identified in

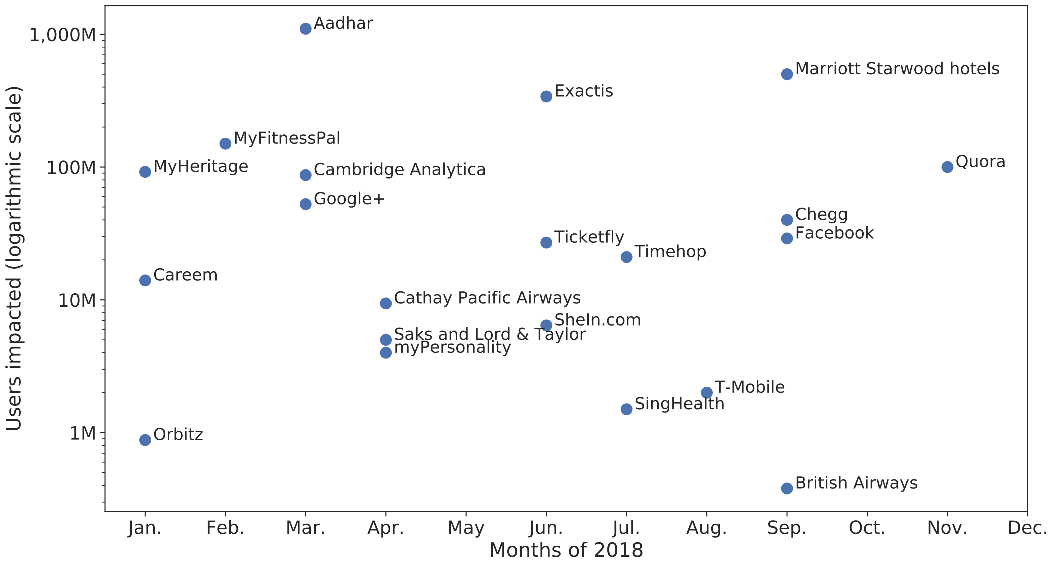


Figure 1.3.: The 21 scariest data breaches of 2018, Business Insider [Les18]

Verizon data break report [Ver19]. We can observe that while the origin of 63% attacker is unknown, their actions are motivated 43% of the time by financial gains.

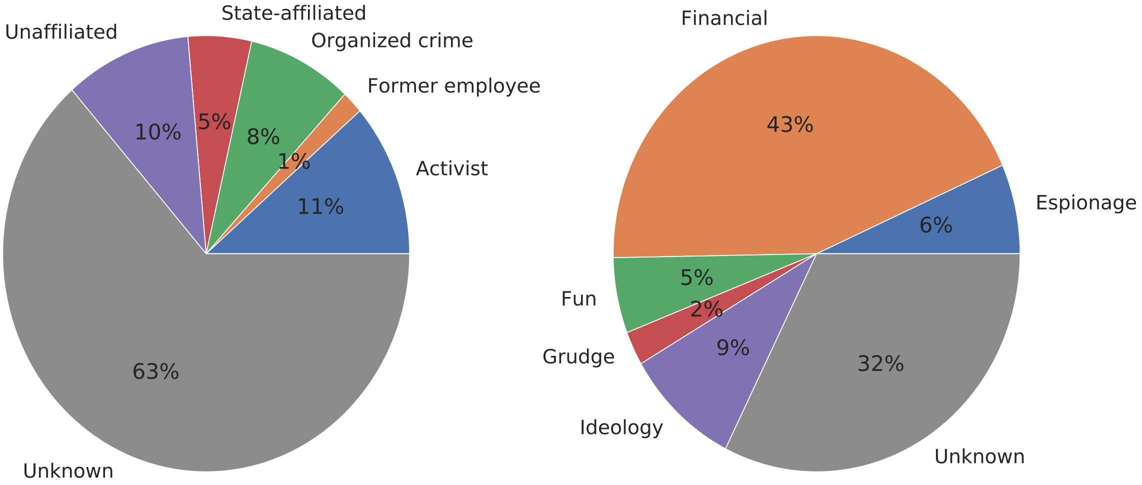


Figure 1.4.: Profile of security attackers by origin (left) and motive (right), Verizon VCDB [Ver19]

Furthermore, we can note that attackers tend to target infrastructures with the weakest security measures in place [Ver19]. This model is similar to a military arms race, where every technological edge can shift the balance of power between belligerents. Compared to other sectors, it is crucial to highlight that mobile security requires constant technological improvements to address the ever-changing risks associated with this landscape.

1.1.2.2. Efforts of the security community

To undertake the threats posed by attackers, the security community engaged tremendous effort in the development of better threat mitigation techniques. In 2018, Google announced that the annual probability for a user to download malware from Google Play was 0.02%, which is according to the report ‘less likely than the odds of an asteroid hitting the earth‘ [Goo18].

To contrast this figure, the Sophos threat report of 2019 mentions that the release of malware targeting mobile and IoT devices is not slowing down [Sop19]. The report points out unusual malicious campaigns, including crypto-miner code in games, advertising click fraud, and supply chain compromise that bypassed the security of Google Play. Since no trusted third party can firmly confirm or infirm the current state of mobile security, we must consider a conservative scenario where the mobile industry continues to face security challenges.

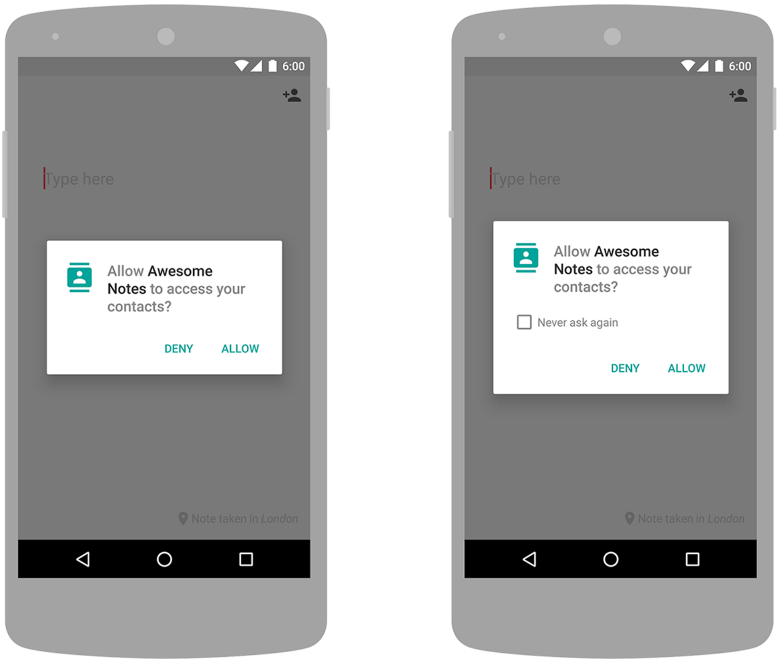
1.1.3. Mobile security for Android applications

1.1.3.1. The principle of least privilege

The security model of Android applications rests upon the well-known principle of least privilege [Goo19c]. This principle states that an application should always operate with the least amount of privilege to perform its task. For instance, a banking application might request Internet access to manage online accounts, but requesting access to contact information would be suspicious behavior for this category of application.

The decision to allow or deny a permission to an application currently relies on mobile users. This principle is then enforced by the Android operating system, which acts as a gatekeeper to access the personal information of mobile users. For instance, Figure 1.5 shows a permission dialog that requests access to contact information. If the user grants the permission, the application can keep access to contact information as long as the permission is not revoked. Otherwise, the application will keep asking for the same permission to enable one of its features. At any point during the authorization process, an Android user can block the application from asking the same request over and over again.

In theory, the security model of Android applications is supposed to provide much more security and flexibility than for regular desktop applications [Goo19c]. On the one hand, end users are always in control of the operations that an application can perform on their device. On the other hand, the operating system can take into account user inputs to better decide on how to handle application permissions.



Allow Awesome Notes to access your contacts?

Allow Awesome Notes to access your contacts?

□ Never ask again

Figure 1.5.: Permission dialog to let an application access user contacts

However, this model is not sufficient to deny the exploitation of user information in practice [Goo18]. First, the implementation of the principle of least privilege is not granular enough to protect against a specific type of misuse or scam (e.g., which information are sent over the Internet). Second, end users might not have enough information to make an informed decision about the level of permission to grant to an application. Finally, end users do not have the knowledge nor the expertise to understand the implication of their choices fully.

For these reasons, the security of Android systems cannot depend solely on the implementation of the principle of least privilege. Both human and system experts must evaluate the characteristics of Android applications to guide end users and remove lurking threats from Android marketplaces.

1.1.3.2. Example of an Android malware

To illustrate the threats that target the Android ecosystem, let us consider a malware analyzed by Upstream Systems in January 2019 [Sys19]. The company identified a weather forecast application developed by a Chinese company. At the time of the investigation, the application was pre-installed on Alcatel Android smartphones, such as the Pixi 4 and A3 Max models.

Upstream Systems identified that the weather application leaks user information such as the geographic locations, email addresses, smartphone identifiers (IMEIs), and send them back to a server in China [Sys19]. Moreover, the application uses a background task to subscribe users to unwanted paid services for an estimated earning of $ 1.5 million. The application was available on Google Play during the investigation by Upstream Systems, with more than 10 million installs and a user rating of 4.4 out of 5 when they disclosed their analysis. Figure 1.6 displays a screenshot of the application as it could be found on Google Play, under the name TCL Weather.

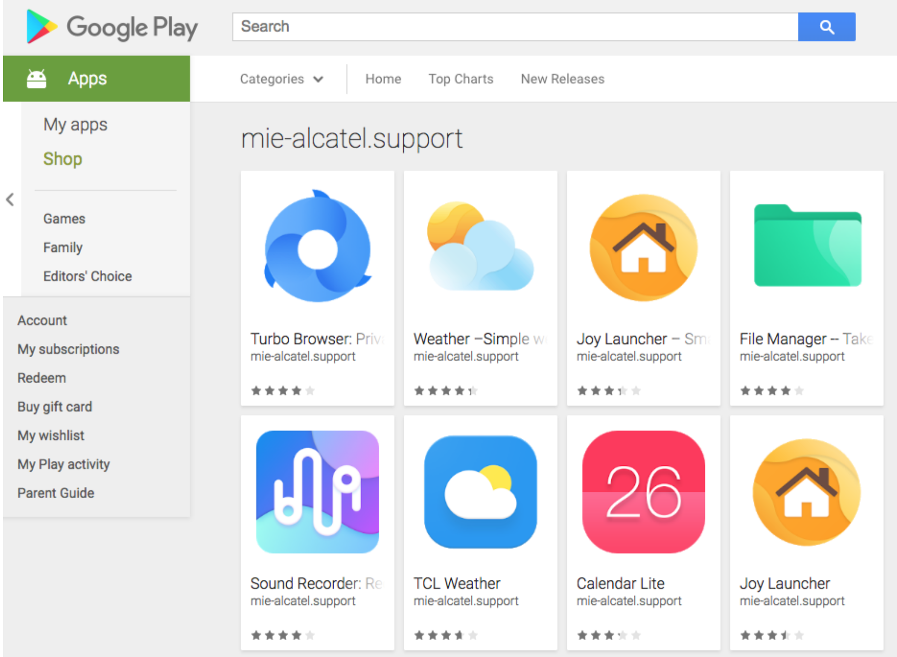


Figure 1.6.: Android applications of 'mie-alcatel.support' on Google Play

The investigation of Upstream Systems unveiled that the application was requesting invasive permissions. Table 1.1.3.2 enumerates the permissions and the personal information requested by the application. With these accesses, the application could click on ads banner without the consent of the device owner. Upstream Systems also found that the application could issue more than 250 transaction requests per month, leading to unwanted charges, additional network consumption, and device overheating.

Following the publication of Upstream Services findings, the application was removed from Google Play on the 5th of January 2019 [Sys19]. Nevertheless, the reaction of Google Play came too late, as more than 10 million users already installed the application on their device. This story illustrates that the lack of security on mobile devices is real and can impact a large population of users across the world.

Table 1.1.: Permissions requested by the application ’com.tct.weather’

Permission	What it allows
CHANGE_WIFI_STATE	Allows the application to change the state of the wi-fi network.
MOUNT_UNMOUNT_FILESYSTEMS	Allows mounting and unmounting of file systems (i.e., external SD cards). Android documentation says the permission is NOT intended for third-party applications
READ_PHONE_STATE	Allows read-only access to phone state, including the phone number of the device, current cellular network information, the status of any ongoing calls, and a list of any PhoneAccounts registered on the device. Documentation says that this permission is DANGEROUS
WRITE_EXTERNAL_STORAGE	Allows reading and writing of the external storage. This is dangerous (as the application can have access to other user files and third-party logs, system logs). This means that the application can read/write everything it wants and the read data might be sent to the server.
ACCESS_KEYGUARD_SECURE_STORAGE	This permission was removed in the OS since 4.4 (KitKat). However, for devices that are below that version, this permission can control a flaw in the OS to lock and unlock the device at any time (like pressing the power button and unlocking the phone)
READ_LOGS	Allows an application to read the low-level system log files. Not for use by third-party applications, because Log entries can contain the user’s private information.
SET_DEBUG_APP	Configure an application for debugging. Not for use by third-party applications.

1.2. Android security challenges

1.2.1. Definition of Android malware

1.2.1.1. Importance of malware definition

While we could state that a malware is a piece of software that causes harm to end users [Dic19], this definition on its own is not sufficient at a technical level to detect malicious applications.

On the one hand, maliciousness is a notion more relative than absolute. For instance, let us consider two Android applications that upload documents on a remote server. Depending on the appreciation of each user, one application could be considered malicious while the other would not. As Figure 1.7 illustrates, the decision to classify an application as malicious relies on an implicit contract between end users, developers, and digital marketplaces. If an end user is not informed about the intents of the developer, or if the platform imposes restrictions that are not met, then the contract is breached, and the application should be considered as potentially malicious by at least one of the parties involved. The goal of security experts in this context is to report which application components are unwanted to prevent their installation on mobile devices [Goo18].

On the other hand, the detection of malicious applications can be automated by a computer system if and only if the specification for this task is explicit and unambiguous. Since computers are not aware of the concept of harm and maliciousness, computer systems require a formal definition to return the security status of an application. In a landscape where Android malware is on the rise, the power of computer systems has to be leveraged to automate the analysis of Android applications and detect potential malware.

Hence, our ability to prevent the propagation of malware depends heavily on our capacity to characterize Android applications. Without a proper characterization, security experts lack the requirements to properly define malicious behaviors and automate their detection by computer systems in the large. In this regard, one of the key challenges for the security community is to provide a thorough and accurate report on the components contained in Android applications and explain how these components are associated with malicious behaviors.

1.2.1.2. Solving the lack of malware definition

In the absence of a formal definition, digital marketplaces must rely on human inputs to assess the dangerousness of Android applications. To illustrate this process, Figure 1.8 shows the analysis of Android applications from the viewpoint of a security organization. At the beginning of the process, samples are collected directly from the Internet or from online user submissions. Security experts can then analyze the application components and isolates which one are involved in the expression of malicious behaviors. The expert can later label the applications as either benign or malicious, and provide some generic rules to classify similar applications with the same tag. In this process, we notice that human expertise is essential to

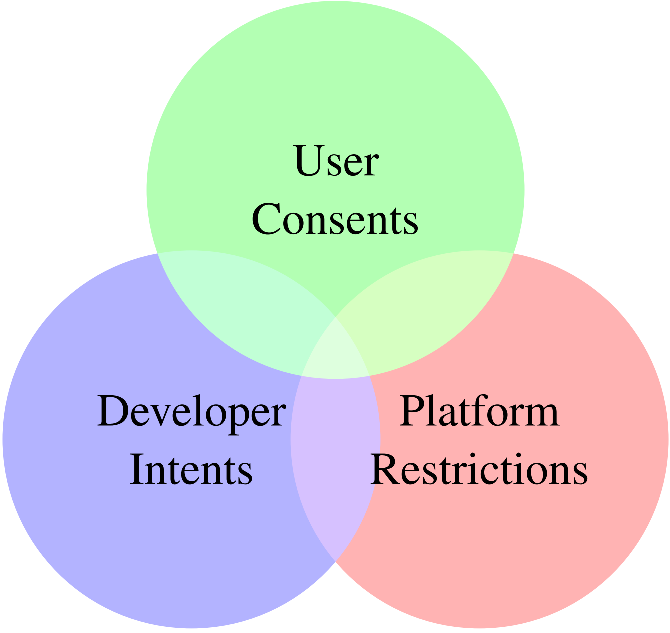


Figure 1.7.: The definition of malware at the intersection of 3 other notions

detect Android malware, as experts provide valuable results and knowledge that can be reused by computer systems and other analysts.

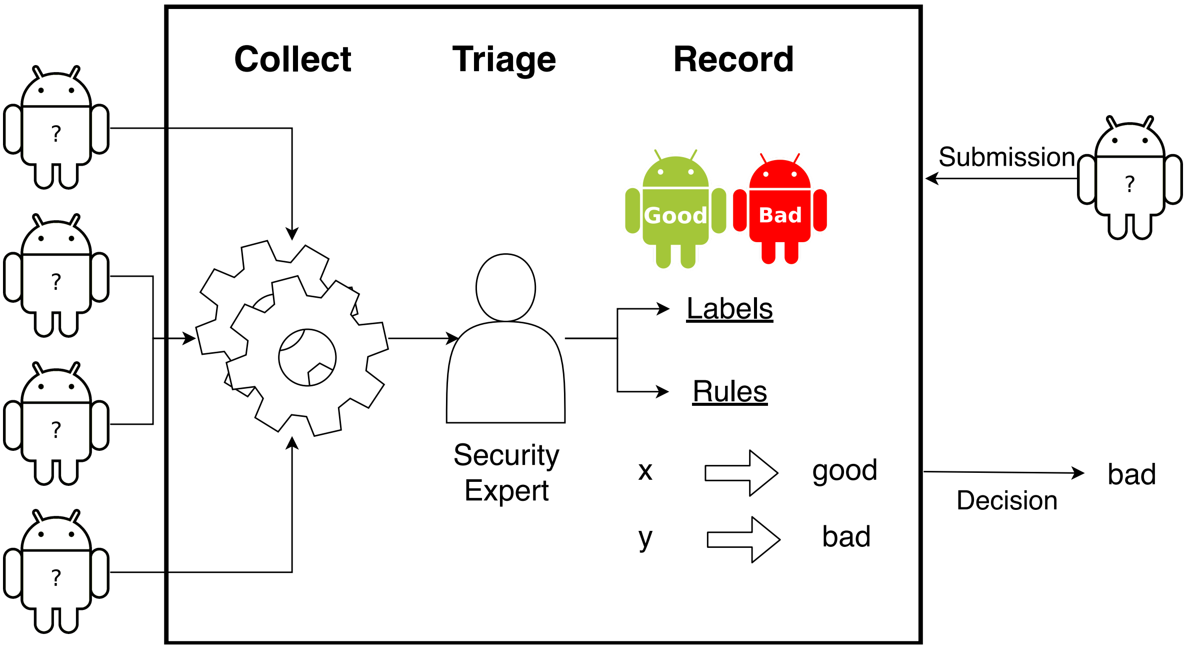
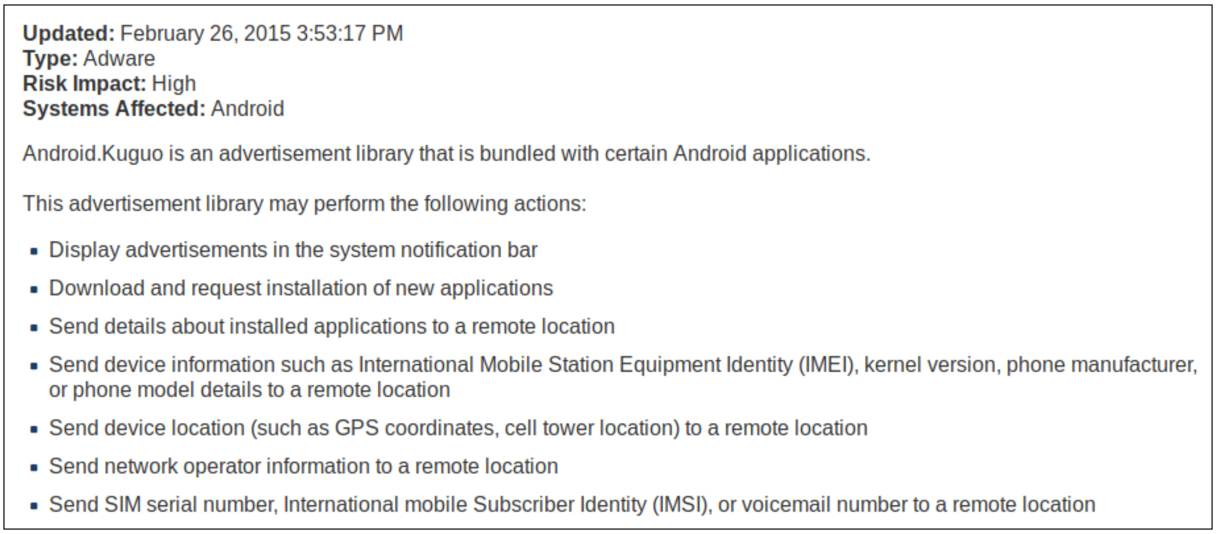


Figure 1.8.: Initial analysis and vetting process of Android applications

While most security organizations have structured themselves around this process, no single organization can cover the current demand for malware analysis on its own. Besides, few security organizations share the knowledge they gathered about malware publicly or with other organizations, as this knowledge gives them an edge over their competitors. Due to this lack of communication, the security community can only access small tokens of information from security organizations. This information is often sparse, unstructured, and lacks the level of details required to understand the relationship between application components and malicious behaviors. On Figure 1.9, we display a screenshot containing the information shared by Symantec about the malware family Android.Kuguo. While the article describes the malicious behaviors and the risks associated with this malware family, the company fails to mention both the malicious components involved and the steps to reproduce the analysis.



Updated: February 26, 2015 3:53:17 PM
Type: Adware
Risk Impact: High
Systems Affected: Android

Android.Kuguo is an advertisement library that is bundled with certain Android applications.

This advertisement library may perform the following actions:

- Display advertisements in the system notification bar
- Download and request installation of new applications
- Send details about installed applications to a remote location
- Send device information such as International Mobile Station Equipment Identity (IMEI), kernel version, phone manufacturer, or phone model details to a remote location
- Send device location (such as GPS coordinates, cell tower location) to a remote location
- Send network operator information to a remote location
- Send SIM serial number, International mobile Subscriber Identity (IMSI), or voicemail number to a remote location

Figure 1.9.: Information about 'Android.Kuguo' from Symantec website [Sym19]

To provide a definitive answer to the lack of malware definition, the security community must not only report the effects caused by malicious applications, but also the causes and the elements related to these effects. It is only at this condition that the security community can create a reliable infrastructure that supports the growth of the Android ecosystem.

1.2.2. Automation of security decisions

1.2.2.1. Shortage of security experts

Over the years, the security of information systems evolved to become an essential facet of our society. As more and more people and infrastructure are connected, the risks of security attacks and data leakages are now more important than ever [Les18]. Governments have strengthened their regulations to enforce a better level of security, while end users are increasingly sensitive to threats posed by unsecured systems. Unfortunately, the number of individuals capable of handling security events is lagging, leaving many demands for security experts unanswered.

In October 2018, $(ICS)^2$ estimated the future lack of security experts around the globe at nearly 3 million people [ICS18]. As we can see on Figure 1.10, the most impacted area will be Asia/Pacific with 2.14 million people, North American with nearly 500,000 people and Europe/Middle East/Africa with 142,000 people. $(ICS)^2$ also noted that 63% of participating organizations are suffering a shortage of security experts right now and that 60% of the respondent are at moderate or extreme risk of security attacks as a result of this shortage.

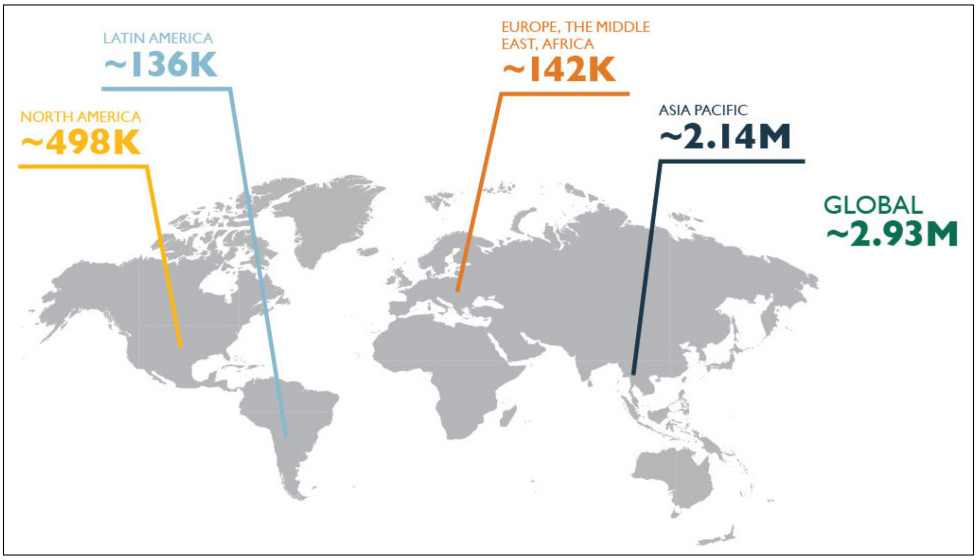


Figure 1.10.: Cybersecurity skills shortage around the globe, $(ISC)^2$ Cybersecurity Workforce Study [ICS18]

The use of automation techniques by security attackers also worsens the situation. Every day, AV-Test registers over 350,000 new malicious programs and potentially unwanted applications across all computer platforms [AT19]. On Figure 1.11, we track the development of new Android malware over the years. While only a few samples were registered in 2012, the number of malware accelerated and peaked in 2016 and 2017 with more than 4 million malware every year. In 2018, six new Android malware were created every minute. We can conclude from these figures that the sheer amount of malware created in the world must be generated automatically by computer systems.

To answer both the lack of security experts and the velocity of malware proliferation, organizations are actively seeking alternative solutions. In 2018, Forbes mentioned that new technology such as big data, artificial intelligence, and machine learning have the potential to address the lack of human resources [For18]. Furthermore, security solutions could also improve the security of our infrastructures, as computers are capable of handling billions of security events per hour while humans can only process a limited amount of events and during a short period. For these reasons, the creation of human-aided assistants is an crucial objective for the field of Android security and the security of information systems in general.

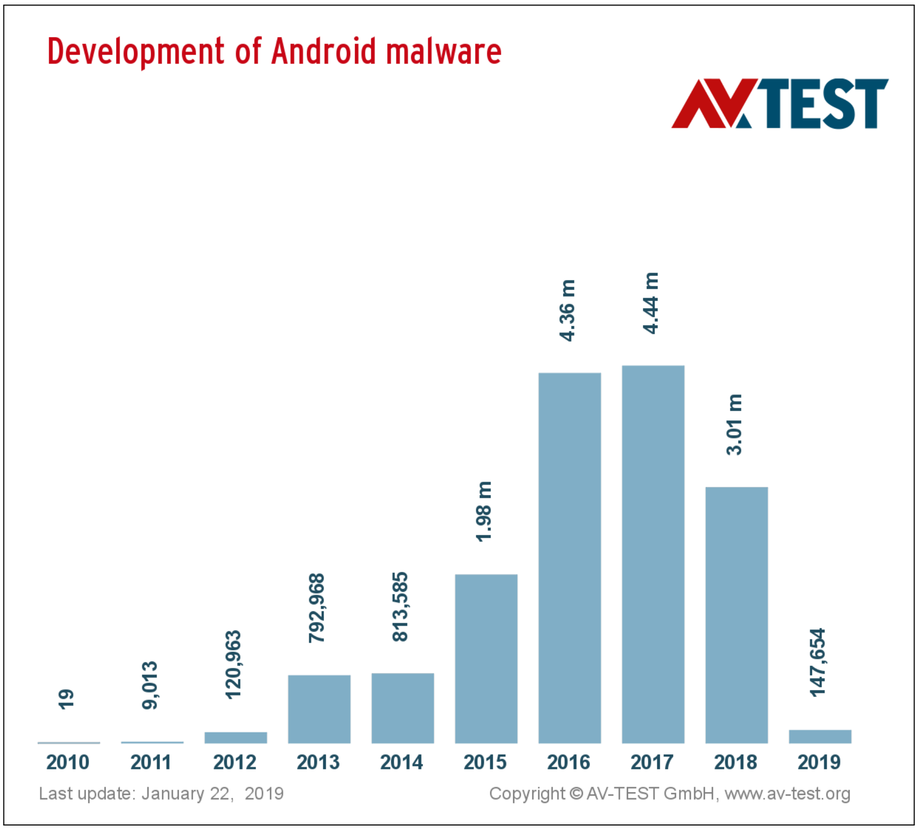


Figure 1.11.: Registration of new Android malware over the years, AV-Test [AT19]

1.2.2.2. Machine learning to the rescue

As a substitute for human resources, the security community has turned its attention to a branch of artificial intelligence called machine learning. In essence, machine learning is the application of algorithms and statistics to train computer systems in performing a specific task. For instance, machine learning is commonly used on Android devices to translate speech to text, recognize digital prints, or adjust energy consumption based on user preferences. Machine learning algorithms excel in situations where data is abundant and well-annotated. In this case, experts can rely on supervised learning, a subset of machine learning focused on training computer systems based on real-world examples.

Figure 1.12 illustrates the components involved during the training of supervised learning systems. From raw input data, the algorithm progressively tunes a statistical model based on a training set and an output selected by a human supervisor. For instance, if the task of the algorithm is to predict a type of fruit, then the model can rely on previous examples to make prediction for unknown cases. The learning process of the algorithm is similar to the sequence of steps followed by a human student. In both cases, the student and the algorithm take into account feedbacks that encourages them to change their decision when they make a mistake. On the contrary, the right decision will strengthen the current model and improve their confidence. To verify that the training is complete, model predictions are then compared against a testing set that was left out from the initial dataset. The training and testing sets constitute the ground truth and are used by machine learning practitioners during their experiments. It is important to note that human supervision remains crucial to train supervised learning algorithms, as the choice of inputs and outputs originates from a human operator.

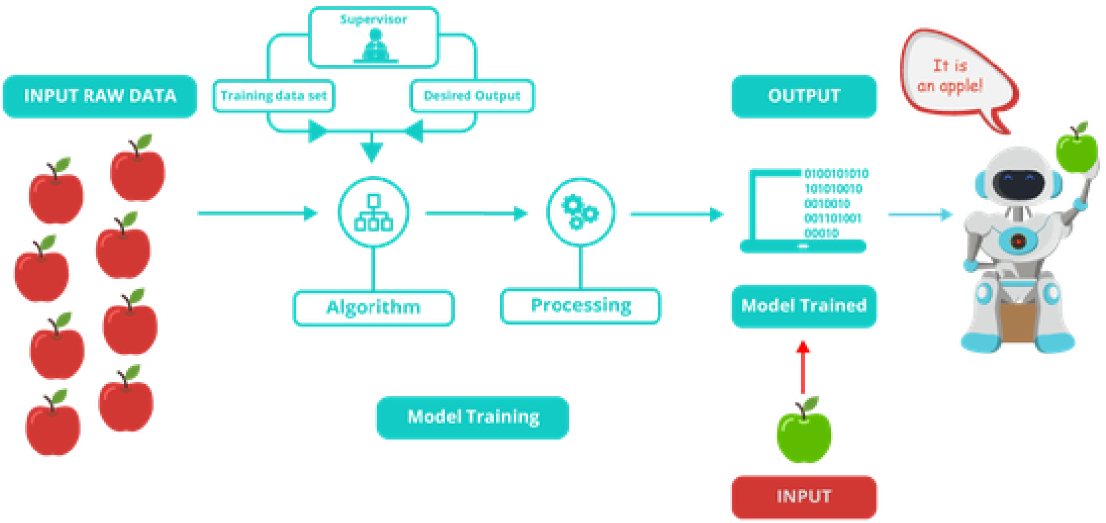


Figure 1.12.: Supervised learning is the process of learning from annotated examples [Vas18]

While machine learning algorithms have greatly improved our capacity to automate security decisions, their performance depends heavily on the quality of the ground truth chosen during the training and verification of statistical models. The problem of ground truth quality is recurrent, and often referred in computer science by the term ‘Garbage in, garbage out’, which implies that flawed input data produces flawed output data. In the absence of qualified ground truth datasets, machine learning techniques cannot be trusted to analyze Android malware in production, as their predictions might introduce both false positive and false negative detections.

1.2.3. Progression of human comprehension

1.2.3.1. The curse of dimensionality

Despite the recent use of machine learning, digital marketplaces are not in a position to fully automate the detection of Android malware. As we reviewed in the last section, machine learning algorithms require a large set of well-annotated malware samples that only human experts can provide. While unsupervised learning techniques can be applied to collect more annotation, the analysis of a single malware is a time-consuming task that incurs a bottleneck in the annotation process.

The situation faced by security experts is similar to a phenomenon observed in computer science called the curse of dimensionality [Bel13]. As the number of items to look at (a.k.a, dimensions) increases, the sparsity and the complexity of data inflates to a point where both the analysis and the prediction become much less efficient. Android applications are complex pieces of software that share the same property. Figure 1.13 shows the distribution of Android applications size collected by the Androzoo project [ABKT16], as a proxy measure to analyze their complexity. We notice that both plots follow a power law distribution, with an average source code size of 3.5 MB and an average application size of 10.4 MB. Csikszentmihalyi et al. [CL14] found that a human can only process 15 bytes per second. Thus, we can compute that it would take about 68 hours to process the information contained in the source code alone and 202 hours to look at the whole Android application.

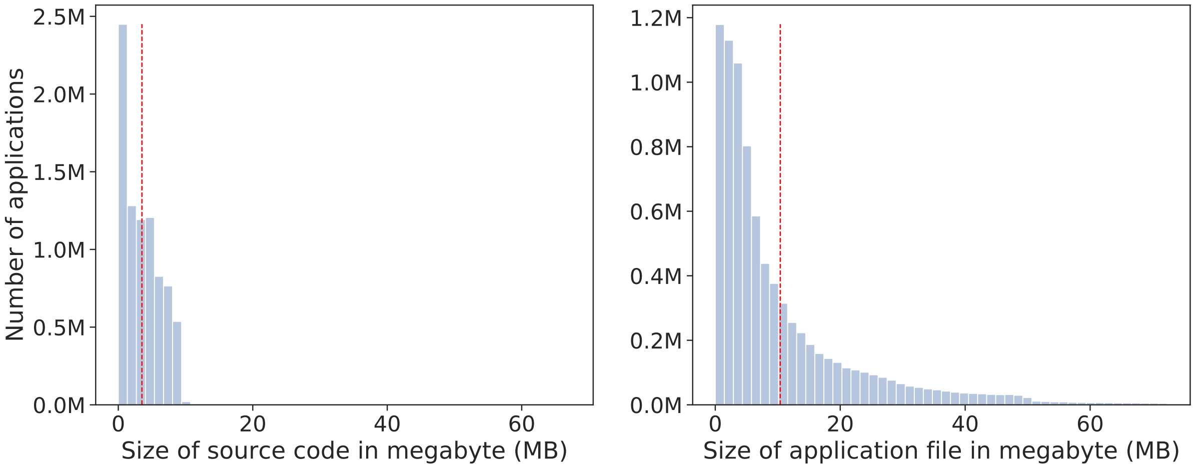


Figure 1.13.: Complexity metrics of Android applications found on Androzoo

As our protection solutions continue to mature, the security community must propose new ways to improve the processing capacity of human analysts. Since we can not increase the bandwidth of the human brain, our technology must compensate for our lack of analysis power by providing a smarter output and by focusing our attention on the details that matter. This condition is critical both to augment our protection mechanisms against malware and to become more efficient at detecting new malware threats.

1.2.3.2. Dissection of Android malware

To understand the behaviors of Android malware, security analysts have no choice but to inspect the components of Android applications one after the other. This operation, called reverse engineering, is similar for all intents and purposes to the dissection of a living organism. The goal of a reverse engineer is to start with a final product, and then retrieve the source code of the application. At the end of the analysis, human experts try to answer a few basic questions about the malware:

- What does the application do?

- Who is the author of the application?
- Which of its components are malicious?
- Where are malicious components located?
- Which other applications include its components?

The data required to answer these questions can take hours, and even days to gather, even for a small application. For instance, Figure 1.14 shows the elements scrutinized by security analysts to understand the behavior of Android applications. In addition to the application source code, security analysts must look at the files, assets, services, and messages contained in the application. Moreover, malware authors attempt to hide the true nature of their application by using obfuscation techniques. This scheme prevents security analysts from reading the resources directly, which slows down the analysis even more.

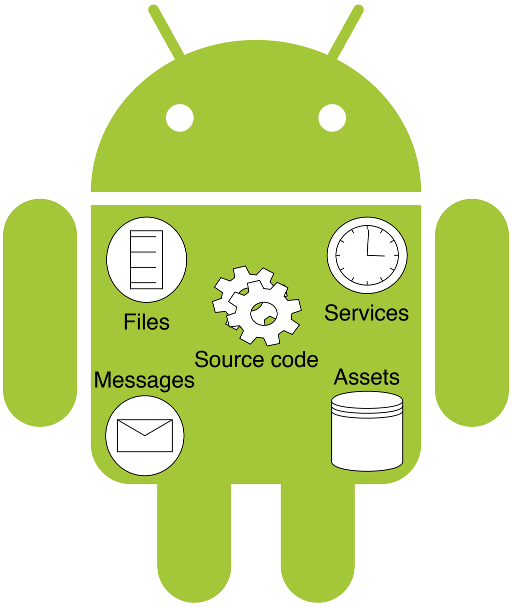


Figure 1.14.: Artifacts that can be extracted from an Android application

A computer system capable of answering these questions would be a valuable asset for a security analyst. Not only could the system speed up the analysis process, but it could also establish the foundation for a knowledge base about Android malware and serve the whole security community. The main challenge imposed on the dissection of Android malware is to provide an approach capable of handling a vast amount of data at scale.

1.3. Contributions to the realm of Android security

1.3.1. Qualification of malware datasets

1.3.1.1. Metrics to understand malware datasets

As we have seen in the previous section, the performance of machine learning based systems depends heavily on the quality of the samples chosen to support the learning process. This requirement means that the security community must have access to large sets of qualified Android applications to detect and classify various kinds of malware. However, the analysis of Android malware remains tedious and time consuming for security experts.

Different approaches have been proposed to create malware datasets based on samples found in the wild, each with their advantages, drawbacks, and biases. A standard method for machine learning practitioners is to rely on external labeling services such as VirusTotal [noa] to alleviate the need for analyzing Android application manually [ASH$^+$14]. Alternatively, other research groups worked with established malware datasets [ZJ12], as these sets are both well-studied and readily available to the community. In either case, the research community must consider the impact that these choices have on their datasets.

Questions:

1. What are the impacts of malware ground truth used in research experiments?
2. How do malware datasets compare with each other, and what are their desirable properties?
3. Do malware datasets incorporate biases due to their creation method or to their origin?

To answer these questions, Chapter 4 introduces STASE, a framework created to measure the characteristics of malware datasets. Based on a set of 9 metrics, STASE highlights the key properties of malware ground truth employed either to detect malware or classify applications in different threat groups. For instance, STASE can measure the degree of consensus across antivirus products or inform about the degree of confidence that a particular application is indeed malicious. The practical goal of STASE is to provide an overview of the features entrenched in malware ground truth before their use in research experiments. STASE metrics can also report information a posteriori to scrutinize biases in malware datasets and assess that the properties of ground truth are comparable to previous studies.

Contributions:

1. We define a set of nine metrics that quantifies the main characteristics of malware datasets created from the aggregation of antivirus results.
2. We provide a framework to encourage the development of heuristics that increases the confidence in malware dataset by removing potential biases.
3. We propose a method to compare malware datasets against each other and ensure that the properties of malware ground truth remain faithful over time.

1.3.1.2. Evaluation of common malware datasets

Over the past few years, the research community around Android security has adopted two popular malware datasets for their experiments. The first dataset was assembled under the umbrella of the Android Malware Genome Project by the North Carolina State University between 2010 and 2011 [ZJ12]. MalGenome includes 1,200 malicious applications and covers popular Android malware families found at the time of the study. The specificity of this dataset is that most samples were analyzed manually by a team of students. Thus, many aspects of the applications, including the installation method, the activation mechanisms, and the nature of the malware payload, have been inspected and studied by the Android community. The other popular malware dataset was created by the Technische Universität Braunschweig between 2010 and 2012 under the name Drebin. Compared to MalGenome, this dataset was analyzed with a mix of both manual and automated techniques. The authors of Drebin mention that their dataset contains 5,560 applications divided into 179 malware families. With various datasets at their disposal, the security community has to design experiments with the most representative samples that represent the whole population of Android malware.

Questions:

1. How are the properties of common malware datasets influenced by the methodology used to construct these sets?
2. Do the degree of confidence and the degree of consensus in the dataset vary according to the choice of malware construction techniques?
3. What are the distinctions between well-known malware datasets and recent malware samples collected directly from the Internet?

With the help of STASE, Chapter 4 presents an empirical study on the most common techniques used to build ground truths of Android malware. In the first part, our study reviews the capacity of antivirus systems to decide if an Android application is malicious or not. In the second part, our study analyzes the quality of the information mentioned in the label returned by antivirus systems. Our evaluation reveals that the degree of consensus between antivirus is often low and that some antiviruses provide fairly generic labels. Moreover, the selection of antivirus systems has a significant influence on the characteristics of the dataset, such as the ability to divide malware into unambiguous threat groups. Thus, machine learning practitioners are encouraged to pick the subset of antivirus systems that provides either the highest degree of confidence or the largest amount of information depending on their use cases.

Contributions:

1. We review the difference between popular malware datasets according to STASE in order to quantify the disparity between common dataset construction techniques.
2. We provide insights on the practice of building ground truth datasets based on VirusTotal labeling service and a large dataset of thousands of Android applications.
3. We extensively overview the lack of consensus between both antivirus decisions and antivirus labels to call for new approaches in building authoritative malware ground truth.

1.3.2. Unification of malware information

1.3.2.1. Extraction from malware labels

To conduct experiments on Android malware, we mentioned that the security community depends either on well-studied malware sets or Android applications amassed directly from the Internet. While the former method has the benefit of leveraging knowledge from previous works, the latter method provides larger datasets and a more up to date view of the current malware landscape. To illustrate this difference, let us consider the AndroZoo project, which provides Android applications downloaded from official and third-party markets to the research community [ABKT16]. As of January 2019, the project references more than 8 million applications compiled from 2008 to 2019. Compared to well-studied datasets such as Drebin [ASH$^+$14] or MalGenome [ZJ12], Androzoo contains 1,500 times more samples and over a period of time nine years larger. However, the idea of using samples collected in the wild has a major setback. Without proper information about the true nature of applications gathered with this method, experimenters are not in possession of a ground truth that enables them to decide which applications should be considered malicious.

To address the lack of information about Android malware, the security community must rely on external sources of information to review the content of Android applications. The solution currently favored by the security community is VirusTotal [noa], an online service that acts as a proxy to antivirus solutions available to the security industry. Once an Android application is submitted to VirusTotal, a set of antivirus analyzes the application and returns their decision in the form of a malware label (i.e., a string of characters that embedded several information about the applications). For example, Figure 1.15 shows a single malware label that contains four parts: the threat type (monitoring tool), the platform (Android), family name (AccuTrack) and the variant (Beta). However, while labels information is intelligible to human experts, their interpretation by computer systems is a challenging task. On the one hand, the syntax of labels (i.e., the position and separation of information tokens) varies across antivirus solutions and sometimes even inside a single antivirus product. On the other hand, the semantic (i.e., the vocabulary and lexicon) is not stable as different labels can be assigned to the same underlying threat.

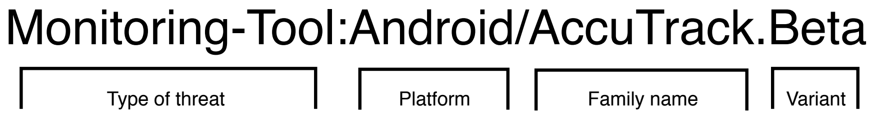


Figure 1.15.: Example of an antivirus label reported by VirusTotal (with information annotated)

Questions:

1. How to accurately extract the information contained in antivirus labels?
2. Can the process of extracting information from antivirus label be automated?
3. What is the minimum amount of knowledge required to extract label information?

To address these challenges, we introduce EUPHONY in Chapter 5. EUPHONY is a heuristics and statistics based system that extracts information from malware labels. Compared to previous solutions, the distinctive feature of EUPHONY is its ability to improve its extraction performance over time, starting from a minimal vocabulary up to much larger lexicons. This feature is critical for the discovery of new threats, as more and more types of malicious techniques are uncovered and identified over time. In practice, we expect EUPHONY to be used both by industrial and academic actors willing to automate the recovery of information from malware labels. Finally, the performance of EUPHONY has been successfully tested both on well-known malware datasets and on the Androzoo project to propose a larger ground truth of Android applications.

Contributions:

1. We present the design and implementation of EUPHONY, an approach designed to mine the information contained in antivirus labels gathered from malware samples found in the wild.
2. We report the evaluation of EUPHONY on MalGenome [ZJ12] and Drebin [ASH$^+$14] datasets, where EUPHONY achieves relatively high performance of 92.7% and 95.5% F-measure respectively.
3. We examine the distinctive features of EUPHONY in contrast with previous work, such as the ability to automatically learn the structure and lexicon of antivirus labels and to iteratively improve the inference performance over time.

1.3.2.2. Aggregation of malware families

Out of the different label parts, the most valuable element is the family name associated with the malware, as this name identifies a particular type of threat that shares common attributes with other members of its family. For instance, the AccuTrack family is known to turn Android smartphones into GPS trackers and display unwanted ads to the device owner [Sol]. This knowledge could be leveraged to focus the attention of security experts on the application components responsible for user geolocation.

However, the extraction process of EUPHONY only addresses the surface of the problem, which is to remove the elements of syntax specific to each antivirus system. The lack of naming convention potentially implies that antivirus systems do not share the same semantic, and that different family names could reference the same applications. In this situation, security experts could not distinguish between a problem of consensus between antivirus systems and a case where two family names are aliases of each other. To prevent this issue, the security community must have a solution that proposes a single malware name for each sample present in a set of applications, even if they are classified by multiple antivirus systems that follow different conventions.

Questions:

1. How can we evaluate the similarity between antivirus labels?
2. At which conditions can malware names be unified into a single name?
3. How can we apply a unification scheme to malicious applications found in the wild?

To address these issues, EUPHONY provides a module that clusters family names into cohesive groups. The primary purpose of the clustering module is to aggregate malware families when their names appear jointly on the same set of applications. Based on these observations, EUPHONY can model the association between family names in the form of a weighted graph and find substructures that maximize the strength of their relationship. On the contrary, relationships between family names that are too weak are removed to create new family names. The clustering module allows malware practitioners to obtain a single family name per application, even if they rely on multiple antiviruses to improve their confidence in antivirus decisions. With this technique, EUPHONY can create malware ground truths that combine samples collected in the wild and antivirus reports gathered with VirusTotal.

Contributions:

1. We propose a clustering scheme as part of EUPHONY that infers the family names of malicious applications based on their co-occurrence in malware datasets.
2. We provide the security community with a new and larger dataset of Android malware constructed from the annotation inferred by EUPHONY for the Androzoo project [ABKT16].
3. We release EUPHONY as an open source application to support the creation of better malware ground truth based on antivirus reports gathered from VirusTotal.

1.3.3. Dissection of malicious components

1.3.3.1. Gathering knowledge on malware

We observed in the previous section that the protection mechanisms of our infrastructure are centered around two activities: the deployment of detection systems in production and the collection of knowledge to improve the performance of these systems. The compilation of knowledge is a fundamental aspect to the security community, as the pertinence of statistical models fades as soon as malware authors turn their attention to new pervasion techniques. Thus, the constant pace of modification required to update systems in production depends on our capacity to mine Android malware as fast as malicious applications are released to the world. However, the dissection of malicious applications is not in a satisfying state of automation. Firstly, malware dissection is performed by human experts and requires hours and sometimes days to reverse even a single malicious application. Secondly, navigating the vast amount of components found in Android applications is a slow process, as humans are not accustomed to work at this level of complexity.

To improve the dissection of Android applications, security experts must have access to a database dedicated to the collection of knowledge about malware. This knowledge base would be a valuable asset, as it could be queried to reveal the relationship between Android applications and the internal components linked to the expression of malicious behaviors. The database could also work as a hub that stores various kind of metadata gathered from Android malware analysis. With this system in place, practitioners could work at a higher level of understanding and craft new features with a more comprehensive process. However, as the number of components found in Android applications is likely huge, the security community must deal with several challenges.

Questions:

1. How to represent the association between Android applications and malicious artifacts?
2. What kind of information is essential to analyze the behavior of Android applications?
3. Which techniques can be applied to limit the lack of scalability of knowledge indexing?

In Chapter 6, we introduce AP-GRAPH as a knowledge base designed to store and extract information from large sets of Android applications. The model of AP-GRAPH is focused on exploring the relationship between applications and their internal components, also called artifacts. Both applications and artifacts are represented as nodes in a graph, linked together by edges to express the relationship between them. The main benefit of AP-GRAPH is to provide a database that supports a wide range of query, opening to a new way of looking at Android applications. AP-GRAPH can be used by security experts to find applications impacted by particular malicious artifacts or to uncovered potentially harmful artifacts based on analytic queries.

Contributions:

1. We propose a representation model for exploring the relationship between Android applications and artifacts that are potentially harmful.
2. We postulate that the knowledge provided by AP-GRAPH can improve our understanding of malicious applications by providing a graph structure focused on the relationship between applications and artifacts.
3. We publish an online service named APKSEARCH to let other researchers query the associations uncovered by AP-GRAPH and let them create more relevant training sets for their experiments.

1.3.3.2. Locating potentially malicious artifacts

To automate the dissection of Android malware, we postulate that malicious behaviors associated with malware are linked to at least one internal component of the application. For instance, a fake banking application that steals personal information cloud include a service that collects data in the background or an activity that gives the impression that the application is benign. Even if malicious components could be first downloaded from the Internet, there must be a part of the application responsible for downloading and executing the remote code. Hence, the presence of artifacts within Android applications could be exploited by computer systems to draw associations between malware files and malicious behaviors.

With the information produced by both EUPHONY and AP-GRAPH, we have access respectively to information about malicious behaviors linked to family names and artifacts related to Android applications. Thus, large sets of Android malware gathered by the security community could be mined to locate suspicious elements in place of security experts. The gain for the security community would be enormous, as security experts allocate a considerable amount of time on the location of relevant pieces of information within Android applications. The feasibility of this approach would depend on several criteria:

Questions:

1. How can we determine that an artifact is related to the malicious behavior of an application?
2. At which point can we consider that a malicious artifact is tied precisely to a malware family?
3. Which method can be used to find the location a malicious artifact once revealed by an automated approach?

Chapter 6 presents how AP-GRAPH database can be applied to locate potentially malicious artifacts. Using the occurrence of artifacts within malware families, AP-GRAPH can filter the most relevant artifacts and report components specific to a malware family. Moreover, AP-GRAPH could be used to maintain both a white list of artifacts found in benign applications and a black list of artifacts that should trigger a security alarm. Thus, the approach we propose could assist security experts in finding both known and unknown malicious artifacts. Applied to a large set of malware such as Androzoo [ABKT16], AP-GRAPH could support the creation of a malware a ground truth composed of artifacts strongly correlated with malicious behaviors.

Contributions:

1. We propose a scheme to automatically locate potentially malicious artifacts that are associated with malware families extracted from antivirus solutions.
2. We perform a systematic study on the most important Android malware families collected by the research community. In particular, we explain that thousands of malware artifacts are directly linked to specific malware families.
3. We release a public dataset of malware descriptions mined by AP-GRAPH to support the continuous effort of the research community in preventing the propagation of malicious applications.

Chapter 2.

Technical Background

This chapter presents the concepts related to the creation of malware ground truth.

This first section introduces the development process of Android applications and the techniques used to guarantee the security and the privacy of Android users.

The second section explains the components of malware ground truth and the information gathered by the security community to support their creation.

The third section gives an overview of the construction of machine learning systems built to detect malicious applications from malware ground truth.

Table of Contents

2.1. Android ecosystem

2.1.1. Overview

The first stable version of Android was released on the 23rd of September 2008 by Google LLC and members of the Open Handset Alliance [Dev08]. The core technologies of the system are written mainly in C and C++ while the graphical interface and the software development kit are developed with Java [Hub19]. Android is based on a modified version of the Linux kernel and licensed under the Apache License 2.0 [And19a]. As a consequence, components of the Android project are accessible under a permissive open source license that allows researchers and other experts to analyze the source code and understand the internal structure of the project.

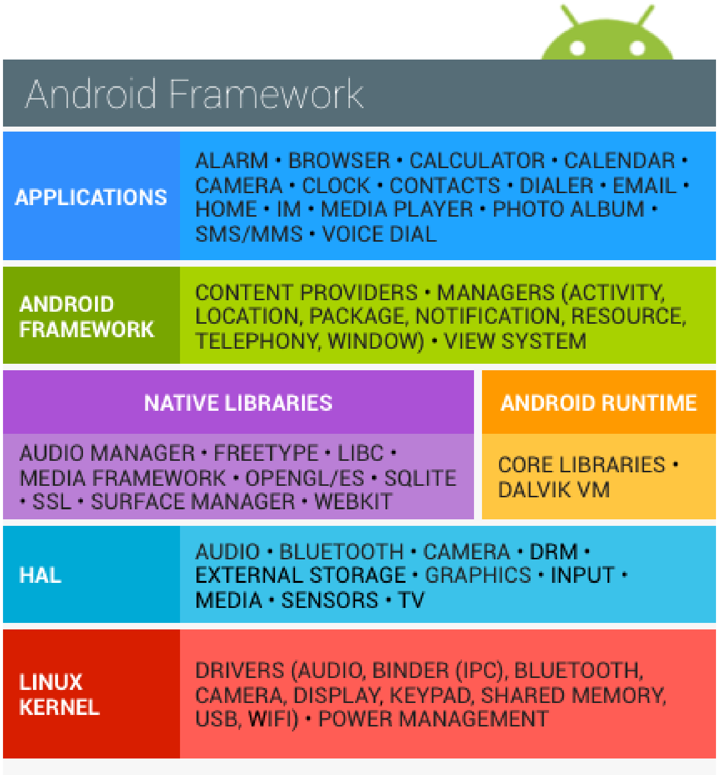

Android Framework
APPLICATIONS
ALARM • BROWSER • CALCULATOR • CALENDAR • CAMERA • CLOCK • CONTACTS • DIALER • EMAIL • HOME • IM • MEDIA PLAYER • PHOTO ALBUM • SMS/MMS • VOICE DIAL
ANDROID FRAMEWORK
CONTENT PROVIDERS • MANAGERS (ACTIVITY, LOCATION, PACKAGE, NOTIFICATION, RESOURCE, TELEPHONY, WINDOW) • VIEW SYSTEM
NATIVE LIBRARIES
AUDIO MANAGER • FREETYPE • LIBC • MEDIA FRAMEWORK • OPENGL/ES • SQLITE • SSL • SURFACE MANAGER • WEBKIT
ANDROID RUNTIME
CORE LIBRARIES • DALVIK VM
HAL
AUDIO • BLUETOOTH • CAMERA • DRM • EXTERNAL STORAGE • GRAPHICS • INPUT • MEDIA • SENSORS • TV
LINUX KERNEL
DRIVERS (AUDIO, BINDER (IPC), BLUETOOTH, CAMERA, DISPLAY, KEYPAD, SHARED MEMORY, USB, WIFI) • POWER MANAGEMENT

Figure 2.1.: Components of the Android framework, Android Source [Sou]

The software development kit of Android provides several services to mobile developers. Figure 2.1 enumerates the components that are part of the Android framework proposed by Google LLC. At the bottom of the diagram, we find the Linux Kernel and the HAL services that implement the interface between the hardware and the software stack. The middle part of the diagram lists the native libraries, the runtime and the components of the framework that are installed to support the creation of Android applications. Finally, the top of the diagram gives some examples of applications that are pre-installed on Android devices, such as Internet browsers, media players and calculators.

Official and third-party application stores support the distribution of Android applications. Google Play (previously known as Android Market) is the official application store created by Google LLC on the 22nd of October 2008 to download Android applications and other digital media such as music, books, and movies [Goo19a]. On the one hand, mobile developers can publish their applications on Google store and add promotional contents like videos, graphics or descriptions. On the other and, Android users can use Google Play to find more applications, install them on their devices and comment or rate the applications in order to share their experience with other users. Google Play also features a developer policy center [And19c] and application security systems [Goo19b] to guarantee the security and the privacy of Android users.

2.1.2. Applications

An Android application is a file that can be installed on Android devices to propose new services. For instance, Android users can install applications from Google Play [Goo19a] to read books, monitor their glucose levels or send short messages to their friends. While Android applications are isolated from each other thanks to the Android sandbox system [Dev19a], developers can leverage the inter-process communication mechanisms provided by intents [Dev19c] to communicate messages across applications installed on the same device. Thus, software components such as activities that display maps or sends rich content can be reused to improve the experience of Android users.

From a technical point of view, an Android application is a zip archive whose file name ends with the extension '.apk'. Similar to zip archives, the content of Android applications can be extracted with standard tools to retrieve their content without information loss. The following list shows the structure of most Android applications.

- **classes.dex**: A packaged and compiled version of the application source code.
- **resources.arsc**: A file that indexes simple values like string translation or layout configurations.
- **META-INF/**: A folder used to certify the origin of the application and the integrity of files in the archive.
- **lib/**: A folder that provides native libraries to access the physical components of the device directly from the application.
- **res/**: A folder that contains additional files to support the application, such as images or audio files.
- **assets/**: An alternative folder to provide additional files where resource identifiers are not linked at compile time.

- **AndroidManifest.xml**: A file that enumerates essential information about the application, such as the package name, the version number or the software and hardware features that the application requires.

Table 2.1.: History of Android versions [And19b]

Name	Version	Release	API
(no codename)	1.0	September 23, 2008	1
Petit Four	1.1	February 9, 2009	2
Cupcake	1.5	April 27, 2009	3
Donut	1.6	September 15, 2009	4
Eclair	2.0 - 2.1	October 26, 2009	5 - 7
Froyo	2.2 - 2.2.3	May 20, 2010	8
Gingerbread	2.3 - 2.3.7	December 6, 2010	9 - 10
Honeycomb	3.0 - 3.2.6	February 22, 2011	11 - 13
Ice Cream Sandwich	4.0 - 4.0.4	October 18, 2011	14 - 15
Jelly Bean	4.1 - 4.3.1	July 9, 2012	16 - 18
KitKat	4.4 - 4.4.4	October 31, 2013	19 - 20
Lollipop	5.0 - 5.1.1	November 12, 2014	21 - 22
Marshmallow	6.0 - 6.0.1	October 5, 2015	23
Nougat	7.0 - 7.1.2	August 22, 2016	24 - 25
Oreo	8.0 - 8.1	August 21, 2017	26 - 27
Pie	9.0	August 6, 2018	28
Android Q	10.0		29

As new versions of the Android project are released over time, external applications must maintain a level of backward compatibility to support the execution environments deployed on user devices. Table 2.1.2 details the versions of Android published by Google LLC starting from version 1.0. Each release is associated with a code name, a version number and an API level targeted by application developers to indicate the degree of compatibility that the application ensures. For instance, an application with a minimum API level of 23 will be compatible with Android devices that feature an API level greater or equals to 23. This information can be found in the ’AndroidManifest.xml’ file of the application.

2.1.3. Security model

First and foremost, the security of Android applications depends on the integrity of the layers that support their execution. Figure 2.2 shows the parts of the Android framework that are responsible for the security of Android applications as described on the Android developers’ website [Web19]. Similar to Figure 2.1, the hardware and the operating system layers powered by the Linux kernel are run in a trusted execution environment to ensure the security of the underlying system. Then, the Android framework implements several trusted services in

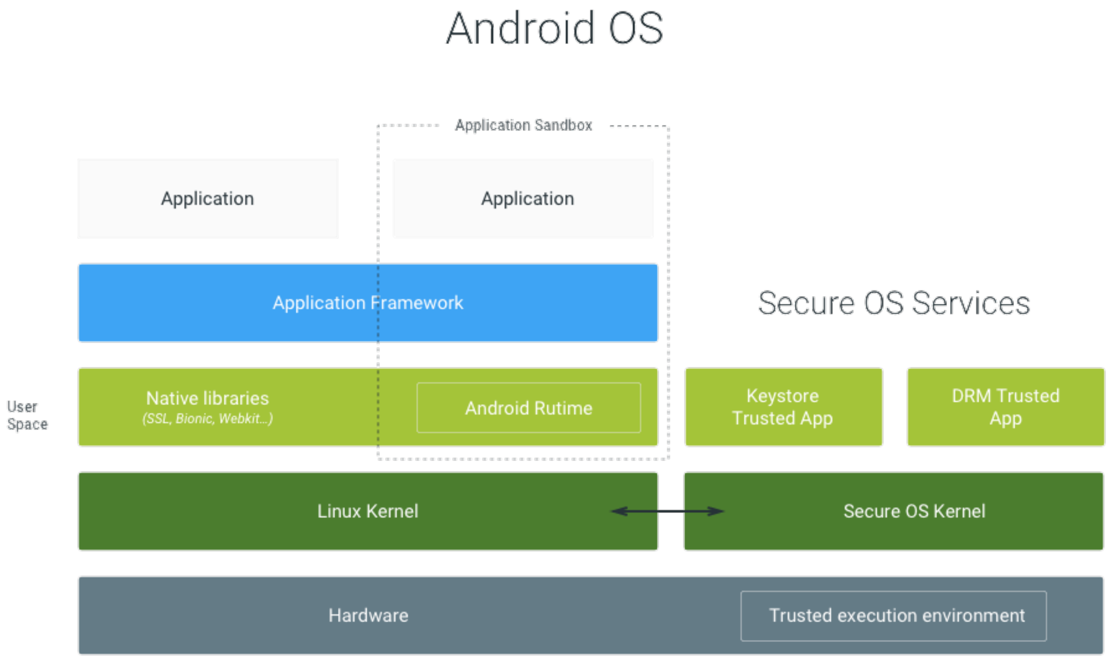


Figure 2.2.: Components of Android security, Android Website [Web19]

user space such as keystore and DRM to enable the creation of secure applications. Finally, applications are executed in a sandbox system that isolates their runtime context from the rest of the system, including access from other applications.

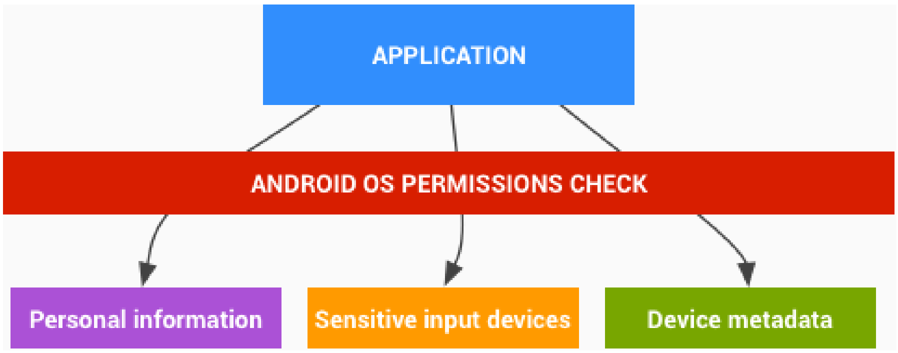


Figure 2.3.: Android relies on permission checks to protect private data, Android Website [Dev19b]

Several features are also implemented to secure Android at the application level [Dev19b]. By default, Android applications can not access the most critical system resources such as camera functions, network connections, and location data as a result of the application sandbox. Figure 2.3 illustrates that the Android permission model ensures that applications can access personal information or device metadata only with the consent of the device owner. This verification process extends to cost-sensitive resources such as billing services or SIM card accesses. Android applications must also be signed before their distribution on application stores to identify application authors and make them accountable for the potential misbehavior of their application. In addition to technical implementations, the Android developers' website proposes several education resources to encourage the best development practices [Dev19d].

In 2017, Google LLC announced the released of Google Play Protect [Tet18], a machine learning based system that detects malicious applications on user devices. Google Play Protect continuously scans applications to find harmful behaviors and submit them to security experts for an extensive review. Malicious applications are then grouped into families to uncover similar applications that are not yet recognized by the security system. The same year, the Google security team announced that their machine learning based system detected 60.3% of malware identified by Google Play Protect to further secure Android devices [Tet18].

2.2. Malware ground truth

2.2.1. Files

A ground truth of Android malware is a collection of Android applications that are recognized as either malicious or harmful for a security expert. The main goal of malware ground truths is to provide reference datasets that researchers and other security experts can use in their experiments to design better security solutions. These datasets are important for the scientific community, as they allow research groups to engage in reproducible experiments which guarantees the quality of their research output. Moreover, malware ground truths are crucial to the performance of machine learning algorithms, as the training and the results of these algorithms depends on the quality of the samples used as examples [RDG$^+$12, SP10].

To illustrate the definition of malware ground truths, we will consider the case of the Androzoo project created by the University of Luxembourg. Androzoo is a collection of Android applications gathered from a broad set of application stores. Table 2.2.1 shows the distribution of applications by Android markets. We observe that the official Android market is the main source of Android applications for the repository. Androzoo also includes Chinese application markets such as anzhi or appchina in addition to open source applications found on the F-Droid repository. In total, Androzoo contains more than 8 million applications as of January 2019.

One critical aspect of malware ground truths is to maintain an index that associates a unique identifier to each application in the collection. This index is useful for security experts to aggregate multiple sources of applications or check the presence of a single application within the set. To implement the name index, security experts rely on cryptographic hash functions that compute a fixed size string from an arbitrary size bit string, like the content of Android applications. For instance, The SHA-256 algorithm associates a 32 bit signature to a file that can not be easily forged or inverted by an attacker. Thus, the only feasible method to retrieve

Table 2.2.: Distribution of Android applications by markets on Androzoo [ABKT16]

Market	Count
play.google.com	7,048,293
anzhi	832,997
appchina	771,970
mi.com	113,583
1mobile	57,530
angeeks	55,818
slideme	52,467
fdroid	18,304
praguard	10,186
torrents	5,294
freewarelovers	4,145
proandroid	3,683
hiapk	2,512
genome	1,247
apk_bang	363
unknown	138

the file signature is to apply the same algorithm on the same file. The following value is an example of SHA-256 signature formatted as an hexadecimal string and present in the Androzoo repository: ’00001f58c32e40376f64cc88b70f8fad2fda054e0863abd5e41f4c6f18a65da2’.

2.2.2. Metadata

To complement the information contained in Android applications, security actors provide additional metadata to enrich the content of malware ground truth. For instance, the Androzoo project [ABKT16] proposes a metadata file that details several characteristics of Android applications:

- **SHA256, SHA1, MD5**: file signatures to identify the application
- **APK and DEX size**: size of the source code and the application
- **DEX date**: compilation date of the packaged source code
- **Market names**: markets where the application was found
- **Package name**: package name from the Android manifest
- **Version code**: version code from the Android manifest
- **VirusTotal detections**: number of positive detections
- **VirusTotal scan date**: date of the antivirus scan

This information is useful for security practitioners to search applications tailored toward a specific use case or to compute statistics about the files listed in malware ground truths.

Table 2.3.: Example of a developer certificate found in an Android application

Key	**Value**

Owner	CN=Eyvind Almqvist, OU=Mobile Visuals, O=Javsym,
	L=Kista, ST=Kista, C=SE

Issuer	CN=Eyvind Almqvist, OU=Mobile Visuals, O=Javsym, L=Kista, ST=Kista, C=SE
Serial number	4d53c582
Valid from	Thu Feb 10 06:01:22 EST 2011 until: Fri Jan 28 06:01:22 EST 2061
MD5	58:94:63:63:C1:ED:4C:02:CE:90:CE:64:DA:D7:4A:E4
SHA1	17:5C:44:E3:A6:1A:F2:4F:A5:78:6E:C7:F0:42:4C:AD:E6:F5:CA:DF
Algorithm name	SHA1withRSA Version: 3

As we mentioned in the previous section, Android applications downloaded from the official market must include a developer certificate that identifies the application authors to hold them accountable for potential misbehaviors. The developer certificate is a public key embedded in the 'CERT.RSA' file, located in the 'META-INF' folder of the application package. Standard cryptographic software can be used to extract this information as a human-readable text. For instance, Table 2.2.2 shows an example of a developer certificate found in an Android application. The certificate contains the owner and issuer, in addition to a serial number, an MD5, and SHA1 signature to guarantee the integrity of the certificate. This information is essential to trace the origin of Android applications and find software packages from the same authors.

Malware ground truth can also inform about the package name and the version code of Android applications. As multiple versions of the same application can be published, these two pieces of information help practitioners to track the lineage of Android packages over time. On the one hand, the package name remains the same for every version of the same application and serves as a common identifier between them. On the other hand, the version code must be different for every new version of the application to mark the evolution of the package. Both the package name and the version code can be found in the Android manifest file. As an example, the 'ul.edu.routes' package name can be associated both to a version '1' and a version '2' of the same application.

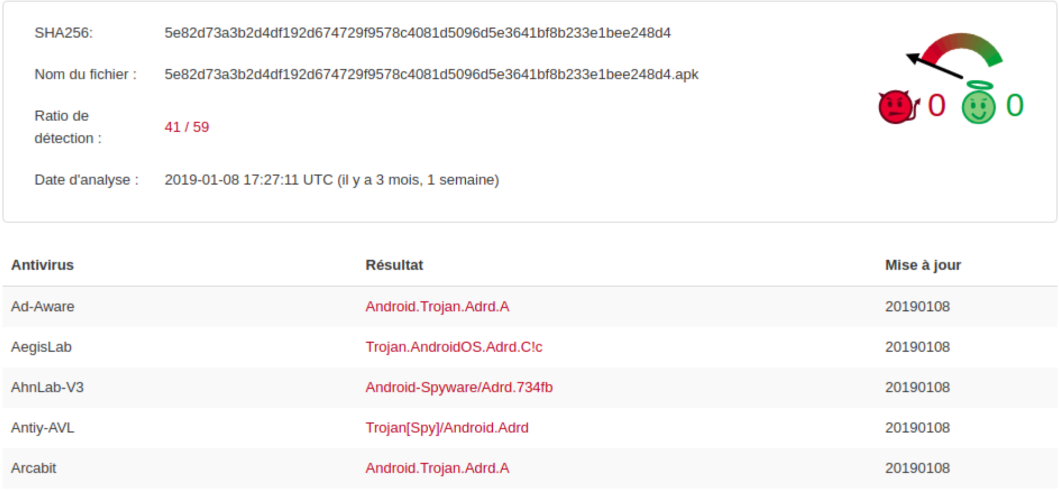
2.2.3. Classification

We saw that malware files and their metadata supply useful information about the content of Android applications, yet they do not inform security practitioners about the danger of malware samples. Compared to other classification problems, Android applications require more time and expertise from specialists to decide if an application should be considered malicious or not [ICS18]. This limitation affects the ability of the security community to leverage malware datasets, as security experts need access to qualified classification results before they can train and evaluate new solutions.

To address this problem, the research community relies on external source information to classify Android applications at a large scale. At least three types of techniques are available to isolate malware from the rest of Android applications:

- **Internal classification**: applications are analyzed manually by a security expert to extract suspicious artifacts and establish the ground truth. This technique was applied to build the Genome dataset [ZJ12].
- **Relative classification**: applications with similar properties are grouped into clusters where the notion of proximity depends on the definition given by the ground truth authors. This technique was used to create the Android Malware dataset [WLR$^+$17].
- **External classification**: applications are analyzed by a trusted third party actor to gather classification results. This technique was involved in the creation of the Androzoo dataset [ABKT16] and the Drebin dataset [ASH$^+$14].

Each classification method has its benefits and its drawbacks. An internal classification scheme provides an unambiguous analysis of malware samples but at the highest cost in terms of human resources. A relative classification scheme applies to large malware datasets, but the results of clustering algorithms are not thoroughly verified to ensure that applications are genuinely malicious. An external classification technique is possible at a large scale and yields accurate results as long as the expertise and the availability of the third party source can be trusted.



SHA256:	5e82d73a3b2d4df192d674729f9578c4081d5096d5e3641bf8b233e1bee248d4
Nom du fichier :	5e82d73a3b2d4df192d674729f9578c4081d5096d5e3641bf8b233e1bee248d4.apk
Ratio de détection :	41 / 59
Date d'analyse :	2019-01-08 17:27:11 UTC (il y a 3 mois, 1 semaine)

Antivirus	Résultat	Mise à jour
Ad-Aware	Android.Trojan.Adrd.A	20190108
AegisLab	Trojan.AndroidOS.Adrd.C!c	20190108
AhnLab-V3	Android-Spyware/Adrd.734fb	20190108
Antiy-AVL	Trojan[Spy]/Android.Adrd	20190108
Arcabit	Android.Trojan.Adrd.A	20190108

Figure 2.4.: VirusTotal report generated after the submission of an Android malware

In this dissertation, we focused our work on an external classification solution built upon VirusTotal to leverage the efforts of security actors in performing malware analysis. VirusTotal [noa] is an online service that aggregates the output of commercial antivirus products to let users decide if an application should be considered malicious. Figure 2.4 displays the result of a VirusTotal scan performed on an Android application. First, a ratio of detection indicates that 41 antivirus products out of 59 consider the file as malicious. Moreover, each antivirus that returned a positive detection supplies a label that contains several pieces of information about the malware, such as the platform, the type of threat, the name of the malware family and its variant. While antivirus labels cannot be used as if due to their syntactic and semantic discrepancies, the information they contain can support the creation of better malware ground truth.

2.3. Machine learning systems

2.3.1. Feature engineering

The first step toward the creation of machine learning based solutions is to gather information about the task that must be learned by the algorithm and performed by the system. For that purpose, security practitioners take advantage of existing malware ground truths as a source of examples to train and evaluate statistical models. As information about Android applications cannot be inferred directly by algorithms, experts must engineer machine learning features that describe the task at a higher level of abstraction. For the detection of Android malware, this step consists of analyzing Android applications to find characteristics that might support the decision of the model.

At least two types of analysis are available to mine knowledge from Android malware. On the one hand, experts can perform static analysis to extract information about the structure of Android applications. For instance, static analysis tools can look at the application source code, retrieve information about the resources included in the package and describe the file included in the archive. While static analysis covers most of the information present in the application, this technique does not inform the expert about the behavior of the application at runtime. To gather information from the execution of Android applications, practitioners turn to dynamic analysis tools for observing the actions of the application in a confined environment. This type of analysis can find which websites the application tries to contact, which data are read or written on the device and if the application triggers on specific events (e.g., the phone rebooted or connected to the Internet). Together, static and dynamic analysis provide the raw material for the creation of machine learning features and complement each other to cover the facets of the application.

Machine learning practitioners can then engineer three types of features depending on the level of abstraction they target. The first type is binary features that are the direct translation of the analysis results. If an analysis reported that the application uses a particular string or a file, then this information can be converted to a simple ’yes’ or ’no’ structure and inform the algorithm about the presence or the absence of a particular item. The second type is derived features that further abstract the results of the analysis. For instance, strings found in Android applications can be scrutinized to find URL scheme or file system paths and declare that the applications might perform some file or network operations. Finally, the third type is statistical features which are the results of a computation performed on raw data. Statistical features provide an overview of the application characteristics, such as the number of classes it contains or the number of permissions required per line of code. To illustrate each category, the following list provides some examples of popular machine learning features applied to the detection of Android malware:

- **Binary features**: requires access to the Internet, requires a maximum API level of 19, is in debug mode, informs about the owner of the developer certificate, is vulnerable to a known CVE.
- **Derived features**: contains obfuscated names, performs cryptographic operations, attempts to contact a botnet URL, loads additional code at runtime, communicates with native libraries.
- **Statistical features**: size of the application, number of classes, number of dangerous permissions per line of code, average instructions per method, entropy of the developer certificate.

2.3.2. Model training

With features engineered from ground truth datasets, machine learning algorithms can train detection models under the supervision of a security expert. The goal of this step is to draw a decision boundary between possible outcomes, such as the possibility that an application is benign or malicious. For instance, Figure 2.5 presents the result of four different machine algorithms in classifying a simple XOR pattern. The decision boundary created by machine learning algorithms takes the form of a purple curve that splits the dots by colors. During model training, the purple curves are adjusted by the algorithm to fit the data and avoid classification errors. We can observe that different machine learning algorithms generate decision boundaries of various shapes, as the mathematical foundations of these algorithms differ.

The training process of machine learning algorithms can be explained and formalized if we consider a simple case like linear regression. A linear regression algorithm finds a linear relationship between dependent variables and the target class. The regression can be formulated as follow:

$$y_i = \beta_0 1 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \varepsilon_i = \mathbf{x}_i^\mathsf{T} \beta + \varepsilon_i, \qquad i = 1, \ldots, n,$$

where $\{y_i, x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$ represents an input dataset and β the coefficients of the linear relationship. Finding the coefficients that best fit the data can be transformed into a minimization problem where the algorithm attempts to minimize the sum of squared residuals (i.e., the total difference between the actual and the predicted values):

$$\text{Find } \min_{\beta} Q(\beta), \quad \text{for } Q(\beta) = \sum_{i=1}^{n} \widehat{\varepsilon}_i^{\,2} = \sum_{i=1}^{n} (y_i - \beta x_i)^2$$

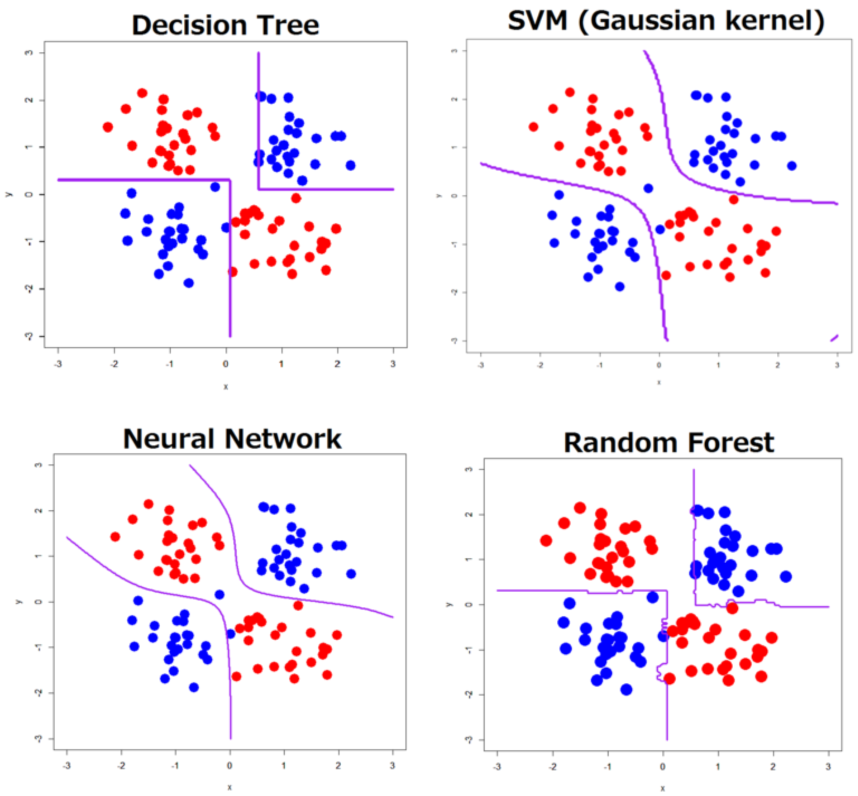


Figure 2.5.: Example of machine learning algorithms applied on a XOR pattern [OZA15]

Machine learning algorithms propose some hyper-parameters that can be tuned to limit the complexity of statistical models or prevent extreme coefficients that cause too much variance. For instance, a linear regression algorithm can include a regularization parameter λ that penalizes either the number of non zero coefficients (L1 norm, LASSO) or the imbalance between coefficient values (L2 norm, RIDGE). The two approaches can also be combined to optimize both cases with a technique called Elastic Net [ZH05].

2.3.3. Evaluation

As multiple choices of algorithms, coefficients, and hyper-parameters are available to train machine learning based systems, security experts must evaluate the performance of their models to explore the solutions at their disposal. A convenient solution for practitioners is to evaluate statistical models with a set of metrics that summarizes the performances of the model on a reference dataset. For detecting malicious Android applications, precision and recall are often used to measure the number of correct answers, depending on the type of errors. On the one hand, the precision measures the number of correct positive results divided by the total number

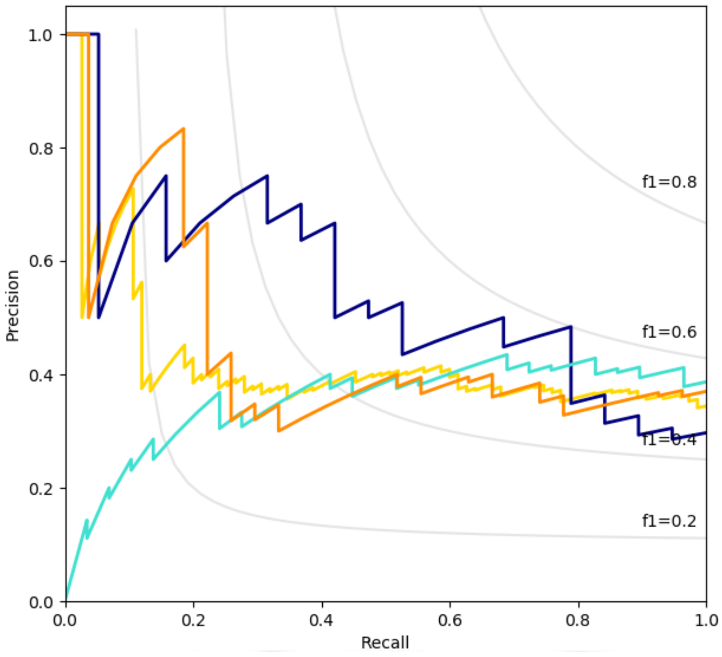


Figure 2.6.: Precision-Recall curve obtained from a classification problem

of positive results predicted by the algorithm:

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

On the other hand, the recall measures the number of correct positive results divided by the total number of true results in the dataset:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

The two metrics can be combined into an F1-score, which is the harmonic mean of the precision and recall that averages their values:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

In a real-world scenario, practitioners may also explore the trade-off of different models or parameters with a precision-recall curve such as Figure 2.6 to find a more suitable solution.

Another fundamental trade-off explored by machine learning practitioners is the ability of statistical models to store example details versus the ability to generalize on new examples. Figure 2.7 illustrates the problem with three curves that represent the decision boundary of statistical models. The model on the left is considered too simple (or under-fitted), as the shape of the decision boundary is linear while the data distribution is polynomial. The model on the right is too complex (or over-fitted), since the decision boundary is too specific regarding

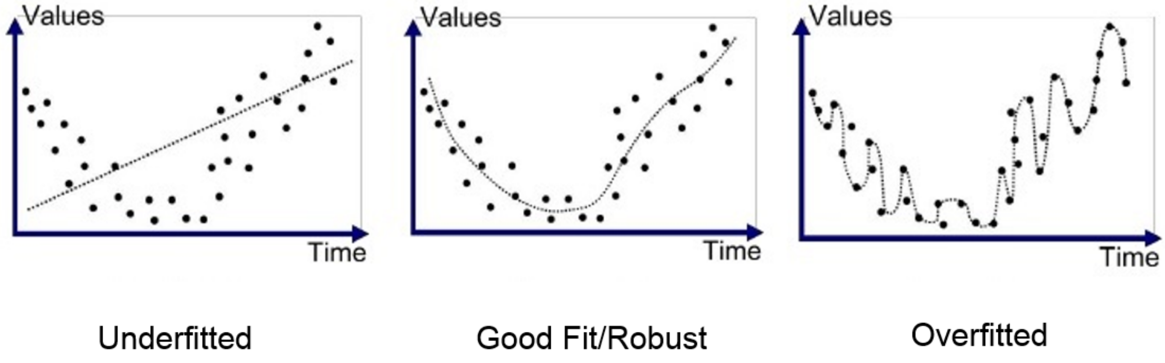


Figure 2.7.: Difference between under-fitting and over-fitting [Bha18]

the data provided to the algorithm. The model in the middle offers the right balance, as the general shape of the model corresponds to the process that might generate the data points. Best practices such as the use of cross-validation techniques and the inclusion of testing datasets can improve the robustness of machine learning experiments in practice.

In this section, we explored the use of machine learning to extract statistical patterns from a large corpus of data. However, the task conveyed to statistical models can only be learned with a great reference ground truth that supports the training and the evaluation of machine learning algorithms. In the absence of qualified ground truth, machine learning algorithms are at risk of proposing a model with suitable performance in the lab, but with unacceptable performance in real-world scenarios. Thus, our comprehension of malware ground truth is crucial to the development and the adoption of machine learning based systems. Under these circumstances, the main risk for the security community is to focus its attention only on quantitative assessments while the description of Android malware remains a vital requirement to justify the decision of automated systems.

Chapter 3.

State of the art

This chapter highlights the literature about the creation of malware ground truth.

The first section reviews the approaches developed to detect Android malware from the use of ground truth datasets and machine learning algorithms.

The second section enumerates the datasets currently used by the Android security community and the techniques applied to assist their construction.

The third section references the techniques developed to better understand Android malware with reverse engineering and data mining analysis.

Table of Contents

3.1. Detection of Android malware

The value of the Android ecosystem depends on the quality of the applications proposed to its users. As developers are pushing new applications every day, the task of detecting fraudulent applications continues to be an essential requirement that guarantees the security of application marketplaces.

The research community addressed these concerns by proposing specific approaches to vet the behaviors of Android applications. On the one hand, researchers created and adapted dissection techniques to extract valuable information from malware with either static or dynamic analysis. On the other hand, the research community developed detection programs to find malicious applications with the help of machine learning algorithms.

The first part of this section reviews the techniques currently used to analyze Android applications.

The second part of this section discusses the statistical models built to detect Android malware.

3.1.1. Malware analysis

3.1.1.1. Static analysis

As the privacy model of Android revolves around user consent, various research groups have studied the use and abuse of system permissions to leak data from Android applications. Felt et al. [FCH$^+$11] analyzed the API calls of Android applications to determine if Android developers implement the principle of least privilege. Wang et al. [WJ12] developed DroidRanger to create a behavioral footprint of Android malware families based on their permissions. Barrera et al. [BKvOS10] presented a methodology to analyze permission-based models such as Android to suggest security improvements.

Other authors analyzed the byte code of Android applications to create more abstract features. Enck et al. [EOMC11] introduced the decompiler ded to reverse the bytecode of Android applications and study the use of Android APIs. Bartel et al. [BKLTM12] created Dexpler that converts Android bytecode into an uncomplicated representation that is more accessible to security analysts. Suarez-Tangil et al. [STTPLB14] studied the used of text mining techniques adapted from vector space modelization to cluster applications based on their similarity.

Another specificity of Android systems is the reliance on inter-program communication to create services reusable from other applications. Octeau et al. [OMJ$^+$13] proposed Epicc as a sound static analysis technique able to improve the discovery of inter-component communication and detect their exploitation. Feng et al. [FADA14] developed Apposcopy to suggest a semantic signature from taint analysis and inter-component call graphs. Federrath et al. [LBB$^+$15] presented a tool called ApkCombiner to reduce the problem of communication between multiple applications as if there were a single application. Kutylowski et al. [YXG+14] developed DroidMiner to mine logic conditions from Android applications and detect potential malware. Barrera et al. [FBR+16] designed a system called TriggerScope that detects logic bombs that trigger malicious behaviors hidden inside Android applications.

Furthermore, Li et al. [LBP+17] performed a systematic literature review on 124 research papers related to the application of static analysis on Android applications.

3.1.1.2. Dynamic analysis

To complement information obtained from static analysis, research groups developed dynamic analysis techniques to gather runtime information from the controlled execution of Android applications. Rastogi et al. [RCE13] created a framework called AppsPlayground to perform various dynamic analysis such as taint tracing, APIS and kernel level monitoring on Android applications. Yan et al. [YY12] proposed DroidScope to reconstruct the semantic of Android applications based on the trace left on the hardware, the operating system, and the Android virtual machine. Tam et al. [TKFC15] presented CopperDroid, a tool that performs dynamic behavior analysis from system calls created by the execution of Android applications. Jang et al. [JYM+16] introduced Andro-profiler to mine the information contained in system logs and generate human-readable behavior profiles. Spreitzer et al. [SKGM18] developed ProcHarvester to mine the information contained in Linux procfs systems and detect information leaks. Rasthofer et al. [RAMB16] released Harvester, an approach that extracts runtime values from highly obfuscated Android applications and covers reflection and dynamic code loading mechanisms.

Static analysis and dynamic analysis have also been applied simultaneously to build on the strengths of both approaches Spreitzenbarth et al. [SFE+13] created Mobile-Sandbox, a framework that relies on static analysis to the execution of dynamic analysis and find native codes in Android applications. Lindorfer et al. [LNW+14] released Andrubis as a public online service that combines static and dynamic analysis to gather information from a dataset of 1,000,000 Android applications.

Moreover, Tam et al. [TFA+17] performed a systematic literature review on the evolution of Android analysis techniques to discuss future research directions on this topic.

3.1.2. Malware classification

3.1.2.1. Malware families

The evolution of unsupervised machine learning algorithms led to the development of new analysis techniques that can group similar malware samples into malware families. Bayer et al. [BCH+09] proposed an automated clustering technique that finds malware families at scale, as their approach was able to process 75,000 samples in less than three hours. Ye et al. [YLCJ10] developed a malware categorization system that combines hierarchical clustering and k-medoids algorithms to create malware family signatures. Wang et al. [WLC15] performed a state of the art classification of malware running on the Microsoft operating system with intensive feature engineering and gradient tree boosting (XGboost).

Other authors refined machine learning based approaches to improve the performance of clustering algorithms. Jang et al. [JBV11] created BitShed to boost the performance of malware clustering algorithms by hashing feature with a distributed execution framework. Hutchison et al. [YBK13] explored a discriminatory feature approach to find the subset of features that best support a classification decision.

3.1.2.2. Malware classifiers

Over the years, the security community has built upon existing classification techniques and adapted their approach to the Android ecosystem. Zia et al. [ADY13] created DroidAPIMiner to mine API level features and detect Android malware with a k-NN classifier. Dash et al. [DSTK+16] released DroidScribe, a system that inspects the state of a virtual machine running Android application and reconstructs inter-process communication to predict a malware family with support vector machine algorithms. Yuan et al. [YLX16] combined features from static analysis and dynamic analysis with deep learning to build DroidDetector and classify Android applications with high accuracy. Mariconti et al. [MOA+17] made MaMaDroid, a tool that incorporates application behaviors from a sequence of abstracted API calls into a Markov chain to detect Android malware.

The continuous growth of malicious Android applications also contributed to the creation of ranking systems able to prioritize the classification and the analysis of malware. Chakradeo et al. [CRTE13] proposed MAST to triage Android applications before their inspection by security analysts and other resource intensive analysis systems. Lindofer et al. [LNP15] designed a system called MARVIN that combines static and dynamic analysis to compute a malice score that indicates the risk associated with an Android application.

In response to the use of obfuscation techniques by malware authors, research groups continued to develop more robust and accurate classifiers to prevent the propagation of malware. Gascon et al. [GYAR13] explored the recent use of machine learning classification to detect Android malware based on an efficient embedding of function call graphs. Kutylowsi et al. [YXG+14] released DroidMiner to abstract program logic in threat modalities and suggest malicious behavioral patterns associated with malware families. Suarez-Tangil et al. [STDA+17] worked on the problem of obfuscated Android malware and created DroidSieve, a system that can detect malicious applications and classify them into malware families with obfuscation invariant features. Nix et al. [NZ17] investigated the performance of recurrent neural network and system call sequences to classify Android applications into malware families. Garcia et al. [GHM18] built RevealDroid to classify Android malware based on their API usage and from the features found in native binaries inside Android applications.

As an alternative to market-based solutions, other authors proposed to execute detection systems directly on user devices. Arp et al. [ASH$^+$14] released DREBIN, a state of the art detection and classification approach that combines a lightweight method for detecting malicious applications directly on the user smartphone and an interpretation of the decision suggested by the classifier. Saracino et al. [SSDM16] created MADAM, a host-based malware detection system that compiles information about kernels, applications, users and packages on the device to detect malicious applications.

3.2. Creation of malware ground truth

Ground truth datasets are essential to detect malicious patterns and analyze malware samples. However, despite their importance, few malware sets are thoroughly qualified by the research community as research groups do not have access to a public source of information to understand the classification of Android malware. Moreover, the research community also suffers from the lack of human resources as research groups can not investigate malicious behaviors from scratch either on large sets of applications and over a long period. In this context, antivirus products appear to be the only source of truth that can be leveraged to perform experiments on Android malware.

In the first part of this section, we review studies of antivirus decisions and the implication of using these antivirus systems in research experiments.

In the second part of this section, we list malware datasets currently used by the research community to design experiments based on Android applications.

3.2.1. Study of antivirus results

3.2.1.1. Antivirus decisions

A handful of research authors studied the challenge of integrating antivirus results in research experiments. Bureau et al. [BH08] discussed that the exponential growth of malware samples impairs both our ability to cross-reference malicious behaviors and to design tailored solutions. Kelchner et al. [Kel10] reviewed the problem of consistent naming in antivirus engines and suggested that generic detection based on malware behavior will become the norm in the future. Gashi et al. [GSM$^+$13] studied the relationship between antivirus regressions and label changes to highlight the difference between antivirus systems. Kantchelian et al. [KTA$^+$15] also explained that the rapid development of malware variants forces the community to collect malware samples through generic techniques that do not thoroughly validate malicious behaviors they exposed.

Other studies empathized the need to design better research experiments based on antivirus results. Li et al. [LLGR10] argued that ground truth datasets obtained from a single antivirus could implicitly remove the most difficult cases for malware classifiers. Perdisci et al. [PU12] created VAMO as both an alternative to majority-based voting and as a way for malware analysts to measure the quality of malware clustering results. Mohaisen & Alrawi [MA14] proposed four metrics to assess the performance of antivirus scanners. The authors also recommended combining multiple antivirus engines to obtain detections of malicious applications that are both complete and correct. Allix et al. [AK14] studied the importance of time in the construction of malware datasets to avoid data leakage in research experiments and better recognize malware lineages.

With STASE, we proposed a complementary solution to build better malware ground truth. STASE metrics quantify essentials properties of malware sets and are independent of the number of samples included in malware datasets. Thus, STASE can be used to compare ground truth datasets at a coarse grain level and spot potential biases introduced by label inconsistencies or generic decisions in antivirus results.

3.2.1.2. Antivirus labels

Research groups have discussed the importance of creating a standard malware labeling scheme to consolidate the output of antivirus systems. Bontchev et al. [Bon05] engaged in the maintenance of CARO [SSB], a malware naming scheme developed in 1990 to report the result of antivirus products. Harley et al. [Har09] spoke about the limits of malware naming as the complexity of malicious applications makes precise identification a challenge for the antivirus industry. Maggi et al. [MBSZ11] developed a graph-based approach to quantify the degree of inconsistency between antivirus vendors and reported that current naming models are both syntactically and semantically incoherent. Gregio et al. [GAF$^+$15] surveyed existing malware naming schemes and introduced a new convention to provide supplementary information to complement existing approaches.

Other research groups addressed the problem of label inconsistencies with external solutions. Wang et al. [WMGH14] created Latin, a first attempt to reconcile both syntactic and semantic naming discrepancies at a large scale based on malware encyclopedia and antivirus reports. Sebastián et al. [SRKC16] proposed AVCLASS as an improvement over Latin to cluster malware labels based on existing malware ground truth such as malware family names and vendor-specific rules.

Similar to Latin and AVCLASS, the goal of our approach with EUPHONY is to assist practitioners in the creation of reference datasets from antivirus results. However, EUPHONY does not require a ground truth list of malware families to distinguish between family names from generic tokens. Moreover, EUPHONY does not contain vendor-specific rules similar to AVCLASS that remove label suffixes. As the accuracy of AVCLASS and EUPHONY are on the same order of magnitude, we think that the learning mechanisms of EUPHONY are more sustainable in practice to both bootstrap and generalize the unification of malware labels over a broad set of unknown samples.

3.2.2. Datasets of Android malware

Ground truth datasets are essential for developing new approaches against Android malware. On the one hand, malware datasets provide a source of samples to support the creation of detection patterns. On the other hand, a malware ground truth is mandatory to validate the methodology proposed by other researchers. Moreover, ground truth datasets are one of the first resources that security practitioners must acquire to run experiments on Android malware.

This section enumerates ground truth datasets used in the research literature.

The first part of this section reports the datasets created by academic actors.

The second part of this section presents solutions proposed by industrial actors.

3.2.2.1. Research projects

The two most cited work on Android malware relates to the creation of malware datasets. Zhou et al. [ZJ12] created the Genome dataset with malware samples gathered from August 2010 to October 2011. The Genome project contains 1,260 malware samples divided into 49 families that were analyzed manually by the authors. The paper reports the installation method, the activation mechanism in addition to the actions that the malware performs and the permissions that the application requires. Arp et al. [ASH+14] extended the Genome project to create Drebin, a detection system created from a dataset of 5,560 malware samples. Compared to Genome, malware families were inferred with machine learning algorithms and divided into 178 malware families.

As Genome and Drebin projects became more and more obsolete over the years [WLR+17], other research groups proposed new datasets with up to date samples and additional features. Allix et al. [ABKT16] developed Androzoo, a collection of 8 million Android applications collected from Google Play and alternative markets to engage the research community in reproducible experiments. Kiss et al. [KLLVTT16] started the Kharon dataset to further document Android malware with manual execution traces about files, processes, and network sockets. Wei et al. [WLR+17] released the Android Malware Dataset (AMD) from samples gathered between 2010 and 2016. AMD contains 24,650 samples divided into 71 families that were analyzed manually at a small scale. The authors of AMD then used the knowledge gathered from manual inspection to find similar samples in their set thanks to clustering algorithms.

3.2.2.2. Industry projects

Industrial actors started their initiative to share information about malware threats that target the Android ecosystem. Contagio [Par] is a public dataset of Android malware created in 2011 that allows its contributors to download and upload suspicious samples with a simple service.

Koodous [RLVS] is a collaborative effort to vet Android malware at scale based on a rich threat exchange platform created by a handful of engineers.

Other industrial actors contributed to the development of malware databases outside the Android community. Malpedia [Plo] is an online service that provides a fast and transparent solution to comment on malware families. MISP [CIR] is an open source threat intelligence sharing platform that provides indicators of compromises in a structured and automated manner.

3.3. Explanation of black box systems

The use of machine learning algorithms for automating the classification of malicious applications helped the security community in its arms race against Android malware. However, the predictions returned by complex statistical models cannot be explained to human operators, as these models do not provide a high-level representation of their decision boundary. Therefore, machine learning models cannot be leveraged in their current state to explain the ground truth behavior of Android malware. To appreciate the impact that machine learning algorithms have on Android security, we must review the approaches developed by the artificial intelligence community to bridge the gap between model accuracy and model interpretability.

Similarly, Android applications can also be considered as black box systems by security analysts when operators do not have access to the source code. Obfuscation techniques deployed by malware authors can also limit our ability to inspect software artifacts with static and dynamic analysis techniques altogether. As the comprehension of malicious code is critical to understand what a malicious application can do, we must review the motivation and the techniques for describing Android families based on large sets of malware.

In the first part of this section, we present the problem of interpreting statistical models and the solutions developed to explain the output of machine learning algorithms.

In the second part of this section, we focus our attention on explaining the behavior of Android malware and on the approaches proposed by the research community to dissect malware families.

3.3.1. Machine learning models

3.3.1.1. Model interpretation

The first step towards model interpretation was to explain the tension that happens between model accuracy and model interpretability. Lipton et al. [Lip18] discussed the desirability and the feasibility of model interpretation regarding existing machine learning algorithms. Doshi-Velez et al. [DVK17] worked to rigorously define the notion of model interpretability and describe in which circumstances this property is desirable. Murdoch et al. [MSK+19] created a framework called Predictive, Descriptive, Relevant (PDR) to evaluate and understand model interpretation proposed by other authors.

As the demand for better model explainability strives, other research groups worked on a new topic called Explainable Artificial Intelligence (XAI). Gunning et al. [Gun17] introduced an explainable artificial intelligence program at DARPA to produce more transparent models and enable human experts to understand their output. Adadi et al. [AB18] and Dosilovic et al. [DBH18] both presented a survey of the field of explainable artificial intelligence to review existing approaches and discuss future research directions.

3.3.1.2. Model exploitation

Conscious of the danger posed by black box statistical models, authors attempted to exploit the weaknesses of machine learning algorithms and yield incorrect predictions. Papernot et al. [PMG$^+$16] created a practical demonstration where the authors took control of a remotely hosted neural network by implementing a local model that substitutes the output classes of the target network. Tramer et al. [TKP$^+$17] crafted perturbations with fast single step methods to both validate their influence on black box exploits and craft models that more robust to this kind of attack.

To improve the trust in machine learning algorithms, researchers proposed to explain the output of complex statistical models with approximation techniques. Ribeiro et al. [RSG16] released LIME, a tool that creates a local approximation around a given prediction for any machine learning model. Lundberg et al. [LL17] proposed SHAP, a framework that interprets statistical models by assigning an importance value to the features involved in a particular prediction. Shrikumar et al. [SGK17] presented DeepLIFT, a method for decomposing the prediction of neural networks by back propagating the importance of each neuron to the input layer.

3.3.2. Malicious Android applications

3.3.2.1. Malware comprehension

Some research groups pointed out that the comprehension of malware is crucial to the adoption of automated approaches out of the lab. Sommer & Paxson [SP10] observed that machine learning algorithms are rarely used in a real-world scenario, as the task of finding security attacks is fundamentally different from other application domains. Rossow et al. [RDG$^+$12] analyzed the methodology of 36 research papers related to malware research and identified several shortcomings in malware description and experimental settings. Allix et al. [ABJ$^+$16] considered the results of machine learning classifiers applied in the lab against the performance of the same classifiers in the wild and observed a significant drop of F-measures in the latter setting. Canto et al. [CSD+17] mentioned that access to an unbiased and representative source of malicious samples is crucial to ensure the accuracy and the realism of security protections.

Other authors aimed at monitoring the evolution of malware artifacts over time to observe trends in malware developments and tailor their protection systems. Lindorfer et al.[LDFM+12] created BEAGLE, a tool that associates and tracks dynamic behaviors from malicious codes over time to monitor the evolution of 16 malware families. Suarez-Tangil et al. [STS18] studied the evolution of 1.2 million Android malware over eight years to study the behavior of repackaged applications.

3.3.2.2. Malware dissection

Recent advancements in machine learning based approaches enabled the research community to create meaningful descriptions that support the predictions of their statistical models. Arp et al. [ASH+14] pioneered this approach on the Android ecosystem with Drebin, which proposed to weight the features that contribute the most to the prediction of a linear SVM model. Jang et al. [JYM+16] introduced a different approach with Andro-profile, a hybrid profiling engine that extracts system logs information and generates human-readable descriptions.

The work on benign applications repackaged with malicious components (i.e., piggybacked malware) revealed that the construction of malware could be exploited to reverse their creation process. Zhou et al. [ZZG+13] developed a fingerprinting technique from semantic features to reveal piggybacked applications found on application markets in linearithmic time complexity. Allix et al. [AJB+14] observed noteworthy patterns in the criminal industry that lead to artifact leakages thanks to a weak comprehension of Android security measures by malware authors. Li et al. [LLB+17] investigated the use of Android packaging models by malware writers to mass produce Android malware with simple automated techniques that rely on code library. Fan et al. [FLW+17] created DAPASA, an approach that detects piggybacked malware with sensitive sub-graph analysis to profile the most suspicious components of an application. Wermke et al. [WHA+18] investigated the use of obfuscation and found that only 25% applications out of 1.7 million applications were obfuscated, preventing the protection of Android applications against piggybacking.

With AP-GRAPH, we propose to complement machine learning based approaches by introducing a data mining method that identifies the most specific artifacts of malware families. Contrary to statistical models, AP-GRAPH focuses on the most discriminative artifacts to ensure that one and only one artifact informs about a malware family during the ranking process. The main benefit that AP-GRAPH provides to the security community is to propose a list of critical features that can be tracked over time or be used as entry points to bootstrap a security analysis. Moreover, our approach supports the creation of descriptive malware ground truth from large sample sets, as AP-GRAPH can select artifacts highly correlated with the presence of malware families. We think that this work is the first step towards a clear and unambiguous answer to what are malicious applications and which artifacts are the root cause of their malicious behaviors.

Part II.

The creation of better malware ground truth

Chapter 4.

STASE: statistics for malware datasets

There is generally a lack of consensus in antivirus engines' decisions on a given malware sample. This problem challenges the building of authoritative ground-truth datasets. Instead, researchers and practitioners may rely on unvalidated approaches to build their ground truth, e.g., by considering decisions from a selected set of antivirus vendors or by setting up a threshold number of positive detections before classifying a sample. Both approaches are biased, as they implicitly decide either on ranking antivirus products or on considering that all antivirus decisions have equal weights.

In this chapter, we extensively investigate the lack of agreement among antivirus engines. To that end, we propose a set of metrics that quantitatively describe the different dimensions of this lack of consensus. We show how our metrics can bring important insights by using the detection results of 66 AV products on 2 million Android apps as a case study. Our analysis focuses not only on antivirus binary decision but also on the notoriously hard problem of labels that antivirus associate with suspicious files, and allows to highlight biases hidden in the collection of a malware ground truth, a foundation stone of any machine learning-based malware detection approach.

On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights
on Building Ground Truths of Android Malware with VirusTotal

Médéric Hurier, Kevin Allix,
Tegawendé F. Bissyandé, Jacques Klein, Yves le Traon

13th Conference on Detection of Intrusions and
Malware & Vulnerability Assessment (DIMVA)
July 6-8, 2016. San Sebastián, Spain

Source code: https://github.com/fmind/stase

Table of Contents

To build ground truth datasets, antivirus engines appear to be the most affordable means today. In particular, their application in research studies became more accessible thanks to online free services such as VirusTotal [noa] that accepts the submission of any file for which it reports back the antivirus decisions from several vendors. Unfortunately, antivirus engines disagree regularly on vetting malicious samples. Their lack of consensus is observed in two dimensions: their binary decisions on the maliciousness of a sample are often conflicting and their labels are challenging to compare because of the lack of a standard for naming malware.

To consolidate datasets as ground truth based on antivirus decisions, researchers often opt to use heuristics that they claim to be reasonable. For example, in the assessment of a state of the art machine learning based malware detection for Android [ASH$^+$14], the authors have considered the reports from only ten antivirus engines, selected based on their popularity, dismissing all other reports. They further consider a sample to be malicious once two antivirus engines agree to say so. They claim that:

> This procedure ensures that [their] data is (almost) correctly split into benign and malicious samples—even if one of the ten scanners falsely labels a benign application as malicious [ASH$^+$14, p.7]

To gain some insights on the impact of such heuristics, we have built a dataset following these heuristics and another dataset following another typical process in the literature [YXG$^+$14], which considers all antivirus reports from VirusTotal and accepts a sample as malicious as long as any of the antivirus flags it as such. Furthermore, we propose a set of metrics for quantifying various dimensions of comparison for antivirus decisions and labels. These metrics typically investigate to what extent the decisions of a given antivirus are exclusive with respect to other antivirus, or the degree of genericity at which antivirus vendors assign malware labels.

Our in-depth study of different heuristics parameters reveals discrepancies in the construction of ground truth datasets, and thus further question any comparison of detectors performance. Similarly, the lack of consensus we observed in label naming prevents a proper assessment of the performance of detectors across malware families.

4.1. Studying the impact of malware datasets

4.1.1. Dataset of Android applications and antivirus

Our study leverages a large dataset of 2 117 825 Android applications and their analysis reports by 66 antivirus engines hosted by VirusTotal [noa].

Application dataset: We obtained our application samples by crawling popular application stores, including Google Play (70.33% of the dataset), Anzhi (17.35%) and AppChina (8.44%), as well as via direct downloads (e.g., Genome - 0.06%) [ABKT16]. Table 4.1 shows the distribution of Android applications across marketplaces.

Table 4.1.: Distribution of applications by markets in our study

Marketplace	**# of Android applications**	**Percentage**
`Google Play`	1 489 572	70.33%
`Anzhi`	367 534	17.35%
`AppChina`	178 648	8.44%
`1mobile`	57 506	2.72%
`AnGeeks`	55 481	2.62%
`Slideme`	31 681	1.50%
`torrents`	5 294	0.25%
`freewarelovers`	4 145	0.20%
`proandroid`	3 683	0.17%
`HiApk`	2 453	0.12%
`fdroid`	2 023	0.10%
`genome`	1 247	0.06%
`apk_bang`	363	0.02%
Total	**2 117 825**	

Antivirus reports: We collected antivirus reports associated with our malware sets from VirusTotal [noa], an online platform that can test files against commercial antivirus engines. For each application package file (APK) sent to VirusTotal, the platform returns, among other information, two pieces of information for each antivirus:

- A binary flag (`True` = positive detection, `False` = negative detection)
- A string label to identify the threat (e.g., `Trojan:AndroidOS/GingerMaster.A`)

Overall, we managed to obtain antivirus reports for 2 063 674 Android applications[1]. In this study, we explore those reports and define metrics to quantify the characteristics of several *tentative ground truths*.

4.1.2. Variations in experimental settings

When experimenting with machine learning based malware detector, as it is nowadays common among security researchers, one of the very first steps is to build a ground truth, for training and also assessing the detector. The question is then how to derive a ground truth

[1] we could not obtain the results for 54 151 (2.56%) applications because of a file size limit by VirusTotal

based on antivirus reports of the millions of applications in existence. In particular, we focus on which samples are considered as malicious and included in the malware set of the ground truth. Based on methods seen in the literature, we consider the following three settings for building a ground truth:

- **Baseline settings**: In these settings, we consider a straightforward process often used [ABJ+16, YXG+14] where a sample is malicious as long as any antivirus reports it with a positive detection. Thus, our ground truth with the Baseline settings and based on our 2 million applications, contains 689 209 malware applications. These samples are reported by antivirus with 119 156 distinct labels.
- **Genome settings**: In a few papers of the literature, researchers use for ground truth smaller datasets constituted of manually compiled and verified malicious samples. We consider such a case and propose such settings where the malware set of the ground truth is the Genome [ZJ12] dataset containing 1 248 applications. Antivirus reports on these applications have yielded 7 101 distinct labels.
- **Filtered settings**: Finally we consider a refined process in the literature where authors attempt to produce a clean ground truth dataset using heuristics. We follow the process used in a recent state-of-the-art work [ASH+14]:
  1. Use a set of popular antivirus scanners[2].
  2. Select applications detected by at least two antiviruses in this set.
  3. Remove applications whose label from any antivirus includes the keyword adware.

  With these settings, the malware set of the ground truth includes 44 615 applications associated with 20 308 distinct labels.

In the remainder of this chapter, we use $\mathscr{D}_{genome}$, $\mathscr{D}_{base}$, and $\mathscr{D}_{filtered}$ to refer to the three ground truth datasets. The property of each dataset are summarized in Table 4.1.2, which includes the selection criteria and the final number of applications for each set. We did not performed supplementary preprocessing besides the heuristics we mentioned in the previous paragraph to avoid potential biases in our study.

Table 4.2.: Experimental ground-truth settings studied with STASE

	Initial Dataset	Genome dataset	Baseline dataset	Filtered dataset
Notation	$\mathscr{D}^I$	$\mathscr{D}^G$	$\mathscr{D}^B$	$\mathscr{D}^D$
Antivirus	66	66	66	10
Applications	2 117 825	1 248	689 209	44 615
Distinct labels	119 156	7 101	119 156	20 308
Discard adware	No	No	No	Yes
Threshold (τ)	≥ 0	≥ 0	≥ 1	≥ 2

[2] antivirus considered in [ASH+14]: AntiVir, AVG, Bit-Defender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos

4.1.3. Notations and definitions

Given a set of n antivirus engines $\mathscr{A} = \{a_1, a_2, \cdots, a_n\}$ and a set of m applications $\mathscr{P} = \{p_1, p_2, \cdots, p_m\}$, we collect the binary decisions and string labels in two $n \times m$ matrices denoted $\mathscr{B}$ and $\mathscr{L}$ respectively:

$$\mathscr{B} = \begin{array}{c c} & \begin{array}{cccc} a_1 & a_2 & \dots & a_n \end{array} \\ \begin{array}{c} p_1 \\ p_2 \\ \vdots \\ p_m \end{array} & \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \dots & b_{m,n} \end{pmatrix} \end{array} \mathscr{L} = \begin{array}{c c} & \begin{array}{cccc} a_1 & a_2 & \dots & a_n \end{array} \\ \begin{array}{c} p_1 \\ p_2 \\ \vdots \\ p_m \end{array} & \begin{pmatrix} l_{1,1} & l_{1,2} & \dots & l_{1,n} \\ l_{2,1} & l_{2,2} & \dots & l_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ l_{m,1} & l_{m,2} & \dots & l_{m,n} \end{pmatrix} \end{array}$$

Where entry $b_{i,j}$ corresponds to the binary flag assigned by antivirus a_j to application p_i and entry $l_{i,j}$ corresponds to the string label assigned by antivirus a_j to application p_i. String label $l_{i,j}$ is $\emptyset$ (null or empty string) if the application p_i is not flagged by antivirus a_j. For any settings under study, a ground truth $\mathscr{D}$ will be characterized by both $\mathscr{B}$ and $\mathscr{L}$.

Let note $R_i = \{m_{i,1}, m_{i,2}, \cdots, m_{i,n}\}$ the i^{th} row vector of a matrix M, and $C_j = \{m_{1,j}, m_{2,j}, \cdots, m_{m,j}\}$ the j^{th} column. The label matrix $\mathscr{L}$ can also be vectorized as a column vector $\mathscr{L}' = (l_1, l_2, \cdots, l_k)$ which includes all distinct labels from matrix $\mathscr{L}$, excluding null values ($\emptyset$).

We also define six specific functions that are part of the formula defined in this chapter:

- Let *positives* be the function that returns the number of positive detections from matrix $\mathscr{B}$, or the number of not null labels from matrix $\mathscr{L}$.
- Let *exclusives* be the function that returns the number of samples detected by only one antivirus in matrix $\mathscr{B}$.
- Let *distincts* be the function that returns the number of distinct labels (excluding $\emptyset$) in matrix $\mathscr{L}$.
- Let *freqmax* be the function that returns the number of occurrences of the most frequent label (excluding $\emptyset$) from matrix $\mathscr{L}$.
- Let *clusters* be the function that returns the number of applications that received a given label l_o with $l_o \in L'$.
- Let *Ouroboros* be the function that returns the minimum proportion of groups including 50% elements of the dataset, normalized between 0 and 1 [Hur]. This function is used to quantify the uniformity of a list of frequencies, independently of the size of the list.

4.2. Analysis of antivirus detection

The primary role of an antivirus engine is to decide whether a given sample is malicious [BH08]. These decisions have significant consequences in production environments since a positive detection will probably trigger an alert and an investigation to mitigate a potential threat. False positives would thus lead to a waste of resources, while False negatives can have dire consequences such as substantial losses. Antivirus engines must then select an adequate trade-off between a deterring high number of false positives and a damagingly high number of false negatives.

In this section, we analyze the characteristics of antivirus decisions and their discrepancies between each other.

4.2.1. Equiponderance

The first concern in using a set of antivirus engines is to quantify their detection accuracies. If there are extreme differences, the collected ground truth may contain decisions from a few engines. In the absence of a significant golden set to compute accuracies, one can estimate, to some extent, the differences among antivirus by quantifying their detection rates (i.e., number of positive decisions).

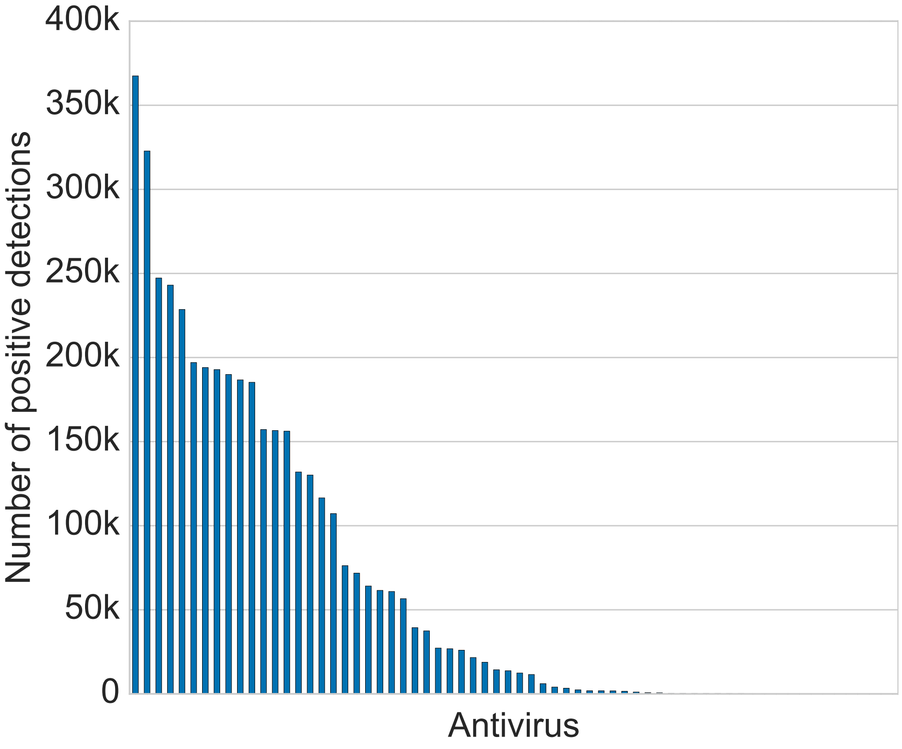


Figure 4.1.: Positive detections by antivirus in $\mathscr{D}_{base}$

Figure 4.1 highlights the uneven distribution of positive detections per antivirus in the $\mathscr{D}_{base}$ baseline ground truth. The number of detected applications indeed ranges from 0 to 367 435. This problem raises the question of the confidence in a ground truth when only antiviruses from the head and tail of the distribution contribute to the decision process. Indeed, although we cannot assume that antivirus engines with high (or low) detection rates have better performances, because of their potential false positives (or false negatives), it is essential to consider the detection rates of antivirus for a given dataset to allow comparisons on common ground. A corollary concern is then to characterize the ground truth to allow comparisons. To generalize and quantify this characteristic of ground truth datasets, we consider the following research question:

Research question 1: Given a set of antivirus and the ground truth that they produce together, Is the resulting ground truth dominated by only a few antiviruses, or do all antivirus contribute the same amount of information?

We answer this research question with a single metric, *Equiponderance*, which measures how balanced or how imbalanced are the contributions of each antivirus. Considering our baseline settings with all antivirus engines, we infer that 9, i.e., 13.5%, antivirus provided as many positive detections as all the other antivirus combined. The *Equiponderance* aims to capture this percentage in its output. Because the maximum value for this percentage is 50%[3], we weigh this percentage, by multiplying it by 2, to yield a metric between 0 and 1. We define the function *Ouroboros* [Hur] which computes this value and also returns the corresponding number of antiviruses, which we refer to as the Index of the *Equiponderance*.

$$Equiponderance(\mathscr{B}) = Ouroboros(X) \text{ with } X = \{positives(C_j) : C_j \in \mathscr{B}, 1 \leq j \leq n\}$$

- **Interpretation** – the minimal proportion of antivirus that detected at least 50% applications in the dataset. The metric value is weighted.
- **Minimum**: 0 – when a single antivirus made all the positive detections
- **Maximum**: 1 – when the distribution of detection rates is perfectly even

When the *Equiponderance* is close to zero, the ground truth analyzed is dominated by extreme cases: a large number of antivirus engines provide only a few positive detections, while only a few antivirus engines provide most positive detections. In comparison with $\mathscr{D}_{base}$'s *Equiponderance* value of 0.27, $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ present *Equiponderance* values of 0.48 and 0.59 respectively.

[3] If one set of antivirus leads to a percentage x over 50%, then the other set relevant value is 100-x% < 50%.

4.2.2. Exclusivity

Even in the case where several antiviruses would have the same number of detections, it does not imply any agreement of antivirus. It is thus crucial to also quantify to what extent each antivirus tends to detect samples that no other antivirus detects.

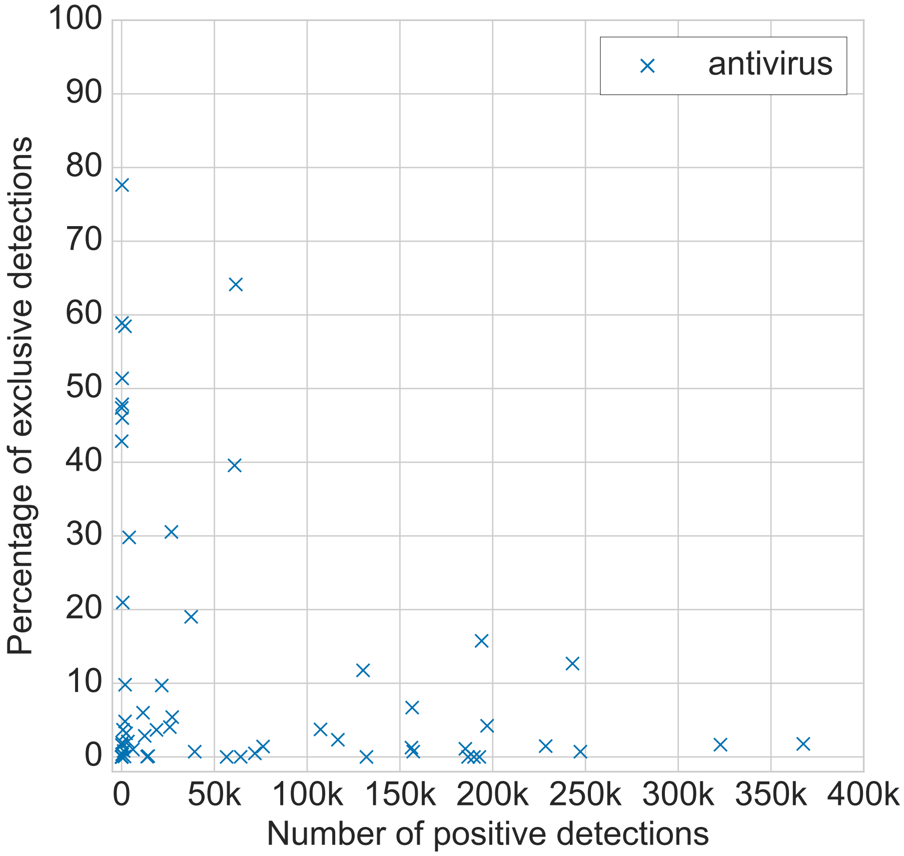


Figure 4.2.: Relation between positive and exclusive detections in $\mathscr{D}_{base}$

Figure 4.2 plots, for every antivirus product, the proportion of exclusive detections (i.e., samples no other antivirus detects) over the total number of positive detection of this antivirus. Five antiviruses provide a majority of exclusive detections while a large part of other antivirus (45) provides less than 10% such detections. For the 21 antiviruses that made the most positive detections, the proportion of exclusive detections remains below 16%, while the highest ratios of exclusive detections are associated with an antivirus that made a (relatively) small number of positive detections. Figure 4.2 provides a important insight into Android malware detection by antivirus: A very high absolute number of detections comes from adding more non-exclusive detections, not from detecting applications no other antivirus detects as could have been intuitively expected. The following research question aims at formally characterizing this bias in datasets:

Research question 2: Given a set of antivirus and the ground truth that they produce together, what is the proportion of samples that were included only due to one antivirus engine?

To answer this research question, we propose the *Exclusivity* metric, which measures the proportion of a tentative ground truth that is specific to a single detector.

$$Exclusivity(\mathscr{B}) = \frac{exclusives(\mathscr{B})}{m}$$

- **Interpretation** – the proportion of applications detected by only one antivirus
- **Minimum**: 0 – when every sample has been detected by more than one antivirus
- **Maximum**: 1 – when every sample has been detected by only one antivirus

In $\mathscr{D}_{base}$, 31% of applications were detected exclusively by only one antivirus, leading to an *Exclusivity* value of 0.31. On the contrary, both $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ do not include applications detected by only one antivirus and have an *Exclusivity* of 0.

4.2.3. Recognition

Because *Equiponderance* and *Exclusivity* alone are not sufficient to describe how experimental ground truth datasets are built, we investigate the impact of the threshold parameter often used in the literature about malware detection to consolidate the value of positive detections [ASH$^+$14]. A threshold τ indicates that a sample is considered as malware in the ground truth if and only if at least τ antivirus engines have reported positive detections on it. Unfortunately, to the best of our knowledge, there is no theory or golden rule behind the selection of τ. On the one hand, it should be noted that samples rejected because of a threshold requirement may be either (a) new malware samples not yet recognized by all industry players, or (b) difficult cases of malware whose patterns are not easily spotted [KTA$^+$15]. On the other hand, when a sample detected by λ or γ antivirus (where λ is close to τ and γ is much bigger than τ), the confidence of including the application in the malware set is not equivalent for both cases.

Figure 4.3 explores the variations in the numbers of applications included in the ground truth dataset $\mathscr{D}_{base}$ as malware when the threshold value for detection rates (i.e., threshold number τ of antivirus assigning a positive detection a sample) changes. We also provide the number of applications detected by more than τ antivirus for the different values of τ.

Both bar plots appear to be right skewed, with far more samples detected by a small number of antiviruses than by the majority of them. Thus, any threshold value applied to this dataset

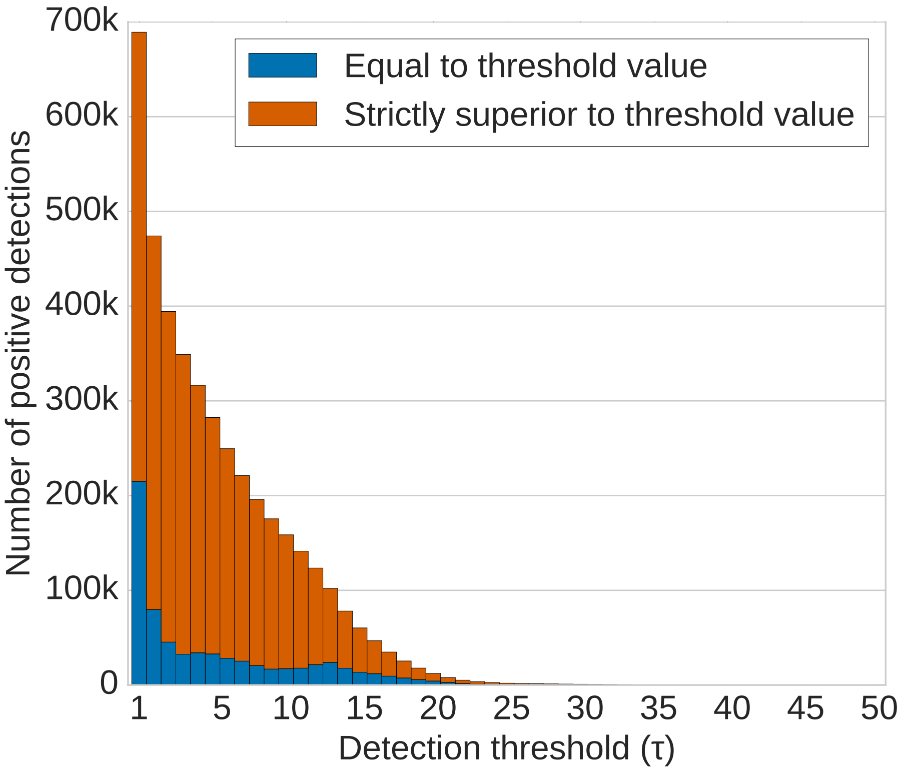


Figure 4.3.: Distribution of applications flagged by τ antivirus in $\mathscr{D}_{base}$

would remove a large portion of the potential malware set (and, in some settings, shift them into the benign set). To quantify this property of ground truth datasets, we investigate the following research question:

> *Research question 3: Given the result of antivirus scans on the ground truth dataset, have applications been marginally or widely recognized to be malicious?*

We answer this Research question with a single metric, *Recognition*, which computes the average number of positive detections assigned to a sample. In other words, it estimates the number of antiviruses agreeing on a given app.

$$Recognition(\mathscr{B}) = \frac{\sum_{i=1}^{m} X_i}{n \times m} \text{ with } X = \{positives(R_i) : R_i \in \mathscr{B}, 1 \leq i \leq m\}$$

- **Interpretation** – the proportion of antivirus which provided a positive detection to an application, averaging on the entire dataset
- **Minimum**: 0 – when no detections were provided at all
- **Maximum**: 1 – when each antivirus have agreed to flag all applications

When a threshold is applied to an experimental dataset, the desired objective is often to increase the confidence by ensuring that malware samples are widely recognized to be malicious by existing antivirus engines. Although researchers often report the effect on the dataset size, they do not measure the level of confidence that was reached. As an example, the *Recognition* of $\mathscr{D}_{base}$ is 0.09: on average, 6 (9%) antivirus engines provided positive detections per sample, suggesting a marginal recognition by antivirus. The *Recognition* values for $\mathscr{D}_{filtered}$ and $\mathscr{D}_{genome}$ amounts to 0.36 and 0.48 respectively. These values characterize the datasets by estimating the extent to which antivirus agree more to recognize samples from $\mathscr{D}_{filtered}$ as positive detections more widely than in $\mathscr{D}_{base}$. Antivirus recognize samples from $\mathscr{D}_{genome}$ even more widely.

4.2.4. Synchronicity

In complement to *Recognition* and *Exclusivity*, we investigate the scenarios where pairs of antivirus engines conflict in their detection decisions. Let us consider two antivirus engines U and V and the result of their detections on a fixed set of samples. For each sample, we can expect 4 cases:

	Detected by U	Not detected by U
Detected By V	`(True, True)`	`(True, False)`
Not detected by V	`(False, True)`	`(False, False)`

Even if the *Equiponderance* value of the dataset produced by antivirus U and V amounts to 1, one cannot conclude on the distribution of those cases. The most extreme scenarios could be 50% (True, True) and 50% (False, False) or 50% (True, False) and 50% (False, True). For the first one, both antiviruses are in perfect synchrony while they are in perfect asynchrony in the second one.

Figure 4.4 is a heat map representation of the pairwise agreement among the 66 antivirus engines on our dataset. For simplicity, we have ordered the antivirus engines by their number of positive detections (the top row left to right and the left column top to bottom correspond to the same antivirus). For each of the $\binom{66}{2}$ entries, we compute the *overlap* function [PLP09]: $overlap(X,Y) = |X \cap Y| / \min(|X|, |Y|)$. This function normalizes the pairwise comparison with the case of the antivirus presenting the smallest number of positive detections. From the heat map, we can observe two patterns: (a) The number of cells where a full similarity is achieved is relatively small with respect to the number of entries. Only 12% of pairs of antivirus achieved a pairwise similarity superior to 0.8, and only 1% of pairs presented a perfect similarity. (b) There is no continuity from right to left (nor from top to bottom) of the map. These observations indicate that antiviruses with an equal number of positive detections do not necessarily detect the same samples. We aim to quantify this level of agreement through the following research question:

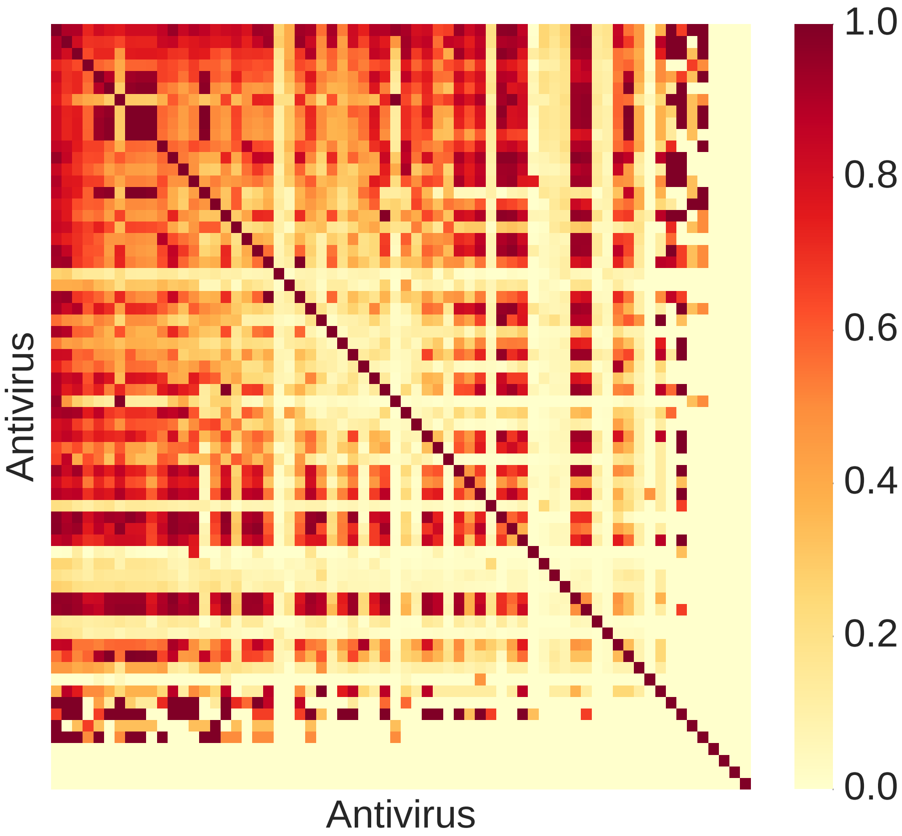


Figure 4.4.: Overlap between pairs of antivirus in $\mathscr{D}_{base}$

Research question 4: Given a dataset of samples and a set of antivirus, what is the likelihood for any pair of distinct antivirus engines to agree on a given sample?

We answer this research question with the *Synchronicity* metric which measures the tendency of a set of antivirus to provide positive detections at the same time as other antivirus in the set:

$$Synchronicity(\mathscr{B}) = \frac{\sum_{j=1}^{n}\sum_{j'=1}^{n} PairwiseSimilarity(C_j, C_{j'})}{n(n-1)} \text{ with } j \neq j', C_j \in \mathscr{B}, C_{j'} \in \mathscr{B}$$

- **Interpretation** – average pairwise similarity between pairs of antivirus
- **Minimum**: 0 – when no sample is detected at the same time by more than one antivirus
- **Maximum**: 1 – when each sample is detected by every antivirus
- **Parameters**
  - *PairwiseSimilarity*: a binary distance function [PLP09]
    - Overlap: based on positive detections and normalized (default)
    - Jaccard: based on positive detections, but not normalized
    - Rand: based on positive and negative detections

High values of *Synchronicity* should be expected for datasets where no uncertainty remains to recognize applications as either malicious or not malicious. $\mathscr{D}_{base}$ presents a *Synchronicity* of 0.32, which is lower than values for $\mathscr{D}_{genome}$ (0.41), and $\mathscr{D}_{filtered}$ (0.75). The gap between values for $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ suggests the impact that a selection of Antivirus can have on artificially increasing the *Synchronicity* of the dataset.

4.3. Analysis of antivirus labeling

Besides binary decisions on detection of maliciousness in a sample, antivirus engines also provide, in case of a positive detection, a string label that indicates the type/family/behavior of the malware and identifies the malicious traits. These labels are thus expected to specify the threat appropriately and in a meaningful and consistent way. Nevertheless, previous works have found that the disagreement of many antiviruses on labeling a sample malware challenges their practical use [BOA$^+$07, CSD$^+$17, MBSZ11, MA14].

In this section, we further investigate the inconsistencies of malware labels and quantify different dimensions of disagreements in ground truth settings.

4.3.1. Uniformity

Figure 4.5 represents the distribution of the most frequently used labels on our $\mathscr{D}_{base}$ dataset. In total, the 689 209 samples detected by at least one antivirus were labeled with 119 156 distinct labels.

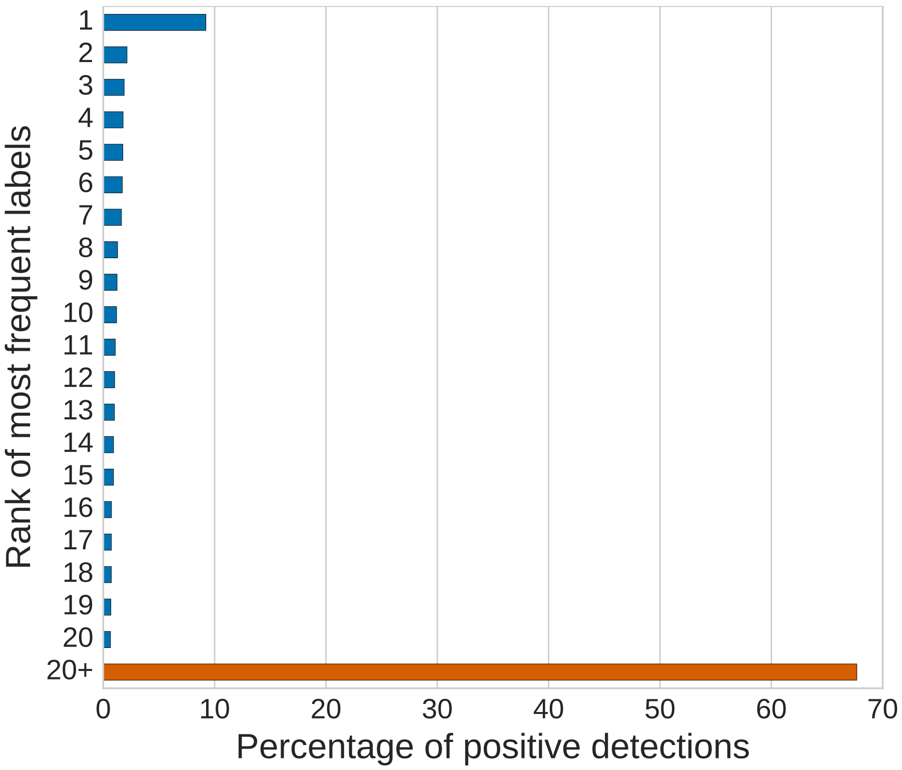


Figure 4.5.: Distribution of malware labels in $\mathscr{D}_{base}$

68% of positive detections were associated with the most infrequent labels, i.e., outside the top 20 labels (grouped under the OTHERS label). The most frequent label, `Android.Adware.Dowgin.I`, is associated with 9% of the positive detections. In a ground truth dataset, it is essential to estimate the balance between different malicious traits, to ensure that the reported performance of an automated detector can generalize. We assess this property of ground truth by answering the following research question:

Research question 5: Given a ground truth derived by leveraging a set of antivirus, are the labels associated with samples evenly distributed?

We answer this research question with a single metric, *Uniformity*, which measures how balanced or how imbalanced are the clusters of samples associated with the different labels.

$$Uniformity(\mathscr{L}') = Ouroboros(X) \text{ with } X = \{clusters(l_k) : l_k \in \mathscr{L}', 1 \leq k \leq o\}$$

- **Interpretation** – minimal proportion of labels assigned to at least 50% of the total number of detected samples. The metric value is weighted
- **Minimum**: 0 – when each sample is assigned a unique label by each antivirus
- **Maximum**: 1 – when the same label is assigned to every sample by all antivirus

The *Uniformity* metric is important as it may hint on whether some malware families are undersampled with respect to others in the ground truth. In can thus help, to some extent, to quantify potential biases due to malware family imbalance. $\mathscr{D}_{base}$ exhibits a *Uniformity* value close to 0 (12×10^{-4}) with an index of 75: 75 labels occur as often in the distribution than the rest of labels (119 081), leading to uneven distribution. We also found extreme values for both Filtered and Genome settings with *Uniformity* of 0.01 and 0.04 respectively. These values raise the question of malware families imbalance in most ground truth datasets. However, it is possible that some labels, although distinct, because of the lack of naming standard, actually represent the same malware type. We thus propose to examine labels on other dimensions further.

4.3.2. Genericity

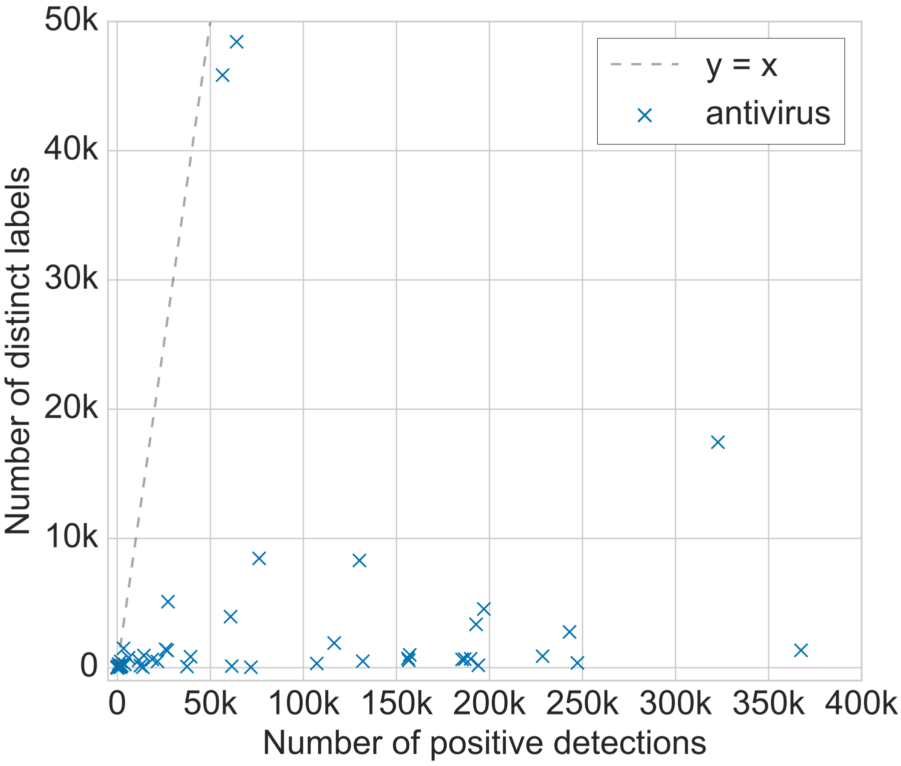


Figure 4.6.: Relation between distinct labels and positive detections per antivirus in $\mathscr{D}_{base}$

Once the distribution of labels has been extracted from the dataset, we can also measure how often labels are reused by antivirus. This property is an unusual behavior that Bureau & Harley highlighted [BH08]. If we consider the two extreme cases, an antivirus could either assign a different label to every sample (e.g., hash value), or a unique label to all samples. In both scenarios, labels would be of no value to group malware together [BOA+07].

In Figure 4.6, we plot the number of detections against the number of distinct labels for each antivirus. While two antiviruses assign almost a different label for each detected sample (points close to the $y = x$ line), the majority of antivirus have much fewer distinct labels than detected samples: they reuse labels amongst several samples. Different levels of labels genericity might explain these two different behaviors For example, using exact labels would make the sharing of labels among samples harder than in the case of generic labels that could each be shared by several samples.

To quantify this characteristic of labels produced by a set of antivirus contributing to define a ground truth, we raise the following research question:

> *Research question 6: Given a ground truth derived by leveraging a set of antivirus, what is, on average for an antivirus, the degree of reuse of a label to characterize several samples?*

We propose the *genericity* metric to quantify this information:

$$Genericity(\mathscr{L}) = 1 - \frac{o - 1}{positives(\mathscr{L}) - 1} \text{with } o \leftarrow \text{number of distinct labels}$$

- **Interpretation** – the ratio between the number of distinct labels and the number of positive detections
- **Minimum**: 0 – when every assigned label is unique
- **Maximum**: 1 – when all labels are identical

Genericity assesses whether an antivirus assigns precise labels or generic ones to samples. Although detectors with low *Genericity* would appear to be more precise in their naming, Bureau & Harley [BH08] support that such engines may not be the most appropriate concerning the exponential growth of malware variants.

The *Genericity* $\mathscr{D}_{base}$ is 0.97, in line with our visual observation that there are far less distinct labels than positive detections. The *Genericity* values of $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ are equal to 0.82 and 0.87 respectively.

4.3.3. Divergence

While *Uniformity* and *Genericity* can evaluate the overall distribution of labels that were assigned by antivirus, they do not consider the question of agreement of antivirus on each sample. Ideally, antiviruses should be consistent and provide labels similar to that of their peers. Even if this ideal case cannot be achieved, the number of distinct labels per application should remain limited to the number of antiviruses agreeing to detect it.

For $\mathscr{D}_{base}$, Figure 4.7 plots the relationship between the number of positive detections of a sample and the average number of distinct labels associated with it. As a confidence margin,

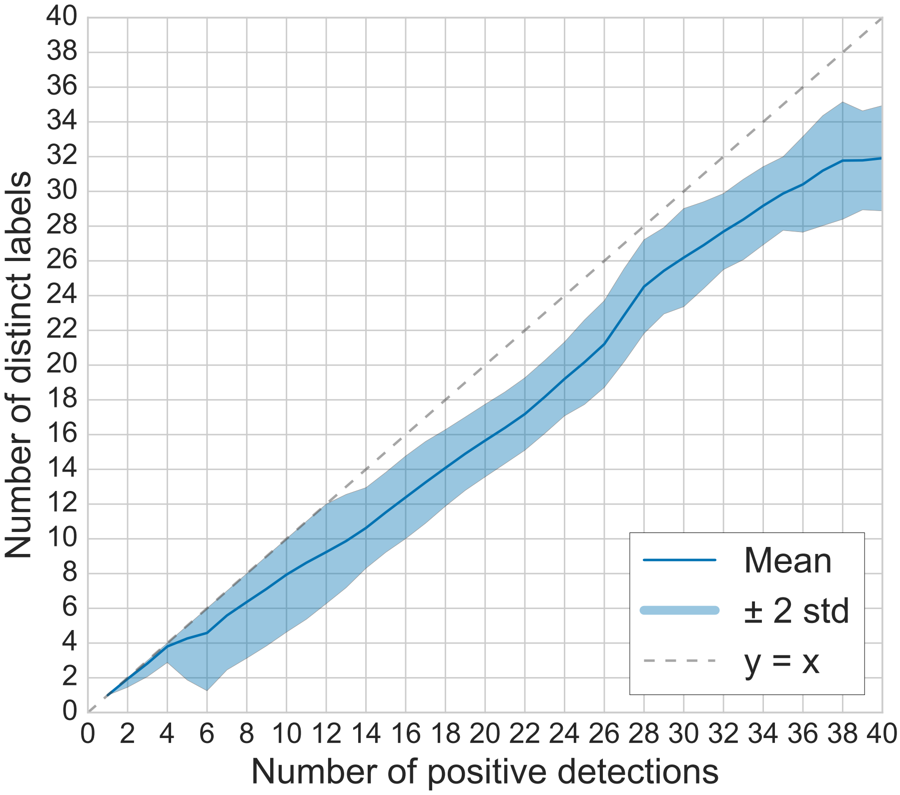


Figure 4.7.: Relation between distinct labels and positive detections per application in $\mathscr{D}_{base}$

we also draw an area of two standard deviations centered on the mean. We note that the mean value for the number of labels grows steadily with the number of detection, close to the maximum possible values represented by the dotted line. The Pearson correlation coefficient ρ between these variables evaluates to 0.98, indicating a strong correlation. Overall, the results suggest not only that there is a high number of different labels per application on our dataset, but also that this behavior is real for both small and high values of positive detections. The following research question investigates this characteristic of ground truth datasets:

Research question 7: Given a set of antivirus and the ground truth that they produce, to what extent do antivirus provide for each sample a label that is inconsistent regarding other antivirus labels

We can quantify this factor with the following metric that measures the capacity of a set of antivirus to assign a high number of different labels per application.

$$Divergence(\mathscr{L}) = \frac{(\sum_{i=1}^{m} X_i) - n}{positives(\mathscr{L}) - n} \text{ with } X = \{distincts(R_i) : R_i \in \mathscr{L}, 1 \leq i \leq m\}$$

- **Interpretation**: – the average proportion of distinct labels per application with respect to the number of antiviruses providing positive detection flags
- **Minimum**: 0 – when antivirus assign a single label to each application
- **Maximum**: 1 – when each antivirus assigns its label to each application

Two conditions must be met in a ground truth dataset to reach a low *Divergence*: antivirus must apply the same syntax consistently for each label, and they should refer to a common semantics when mapping labels with malicious behaviors/types. If label syntax is not consistent within the dataset, then the semantics cannot be assessed via the *Divergence* metric.

The *Divergence* values of $\mathscr{D}_{base}$, $\mathscr{D}_{filtered}$ and $\mathscr{D}_{genome}$ are 0.77, 0.87 and 0.95 respectively. These results are counter-intuitive since they suggest that more constrained settings create more disagreement among antivirus in terms of labeling.

4.3.4. Consensuality

To complement the property highlighted by *Divergence*, we can look at the most frequent label assigned per application. Indeed, while the previous metric describes the number of distinct labels assigned per application, it does not measure the weight of each label, notably that of the most used label. To some extent, this label could be used to infer the family and the version of the malware, e.g., if it used by a significant portion of antivirus to characterize a sample.

To visualize this information, still for $\mathscr{D}_{base}$, we create in Figure 4.8 a plot similar to that of Figure 4.7, looking now at the average number of occurrence of the most frequent label against the number of positive detections per application.

The correlation coefficient ρ between the two variables is 0.76, indicative of a correlation. Nevertheless, the relation is close to the potential minimum (x-axis). This result is in line with our previous observations on $\mathscr{D}_{base}$ that the number of distinct labels per application was high. The plot further highlights that the most frequent label for an application is assigned simultaneously by one to six antivirus (out of 66) on average. This finding suggests that, at least in $\mathscr{D}_{base}$, using the most frequent label to characterize the malicious sample is not a sound approximation. The following research question generalizes the dimension of disagreement that we investigate:

> *Research question 8: Given a set antivirus and the ground truth that they produce, to what extent can we rely on the most frequently assigned label for each detected sample as an authoritative label?*

We answer this research question with the *Consensuality* metric:

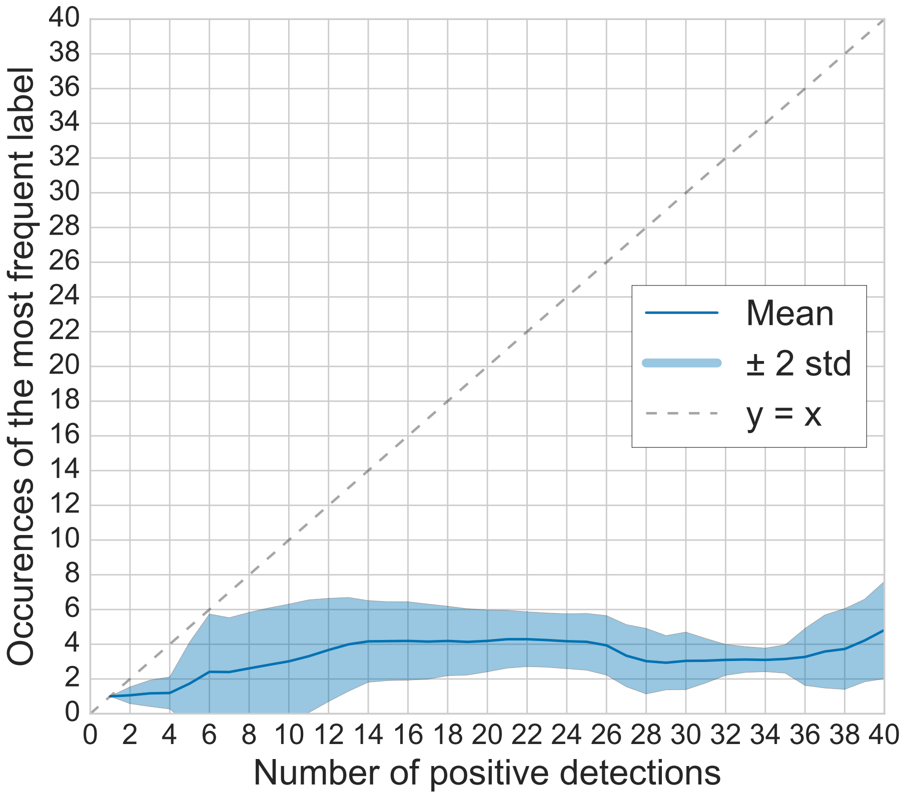


Figure 4.8.: Relation between the most frequent label/τ and positive detections per application in $\mathscr{D}_{base}$

$$Consensuality(\mathscr{L}) = \frac{(\sum_{i=1}^{m} X_i) - n}{positives(\mathscr{L}) - n} \text{ with } X = \{freqmax(R_i) : R_i \in \mathscr{L}, 1 \leq i \leq m\}$$

- **Interpretation** – the average proportion of antivirus that agrees to assign the most frequent label. The frequency is computed per sample.
- **Minimum**: 0 – when each antivirus assigns to each detected sample its own label (i.e., unused by others on this sample)
- **Maximum**: 1 - when all antivirus assign the same label to each sample. Different samples can have different labels however

A high *Consensuality* value highlights that the antiviruses agree on most applications to assign a most frequent label. This metric is essential for validating, to some extent, the opportunity to summarize multiple labels into a single one. In the $\mathscr{D}_{base}$ set, 79% detection reports by antivirus do not come with a label that, for each sample, corresponds to the most frequent label on the sample.

The *Consensuality* value of the set evaluates to 0.21. In comparison, the *Consensuality* values for $\mathscr{D}_{filtered}$ and $\mathscr{D}_{genome}$ are 0.05 and 0.06 respectively.

4.3.5. Resemblance

Divergence and *Consensuality* values on $\mathscr{D}_{base}$ suggest that labels assigned to samples cannot be used directly to represent malware families. Indeed, the number of distinct labels per application is high (high *Divergence*), and the most frequent label per application does not often occur (low *Consensuality*).

We further investigate these disagreements in labels to verify whether the differences between label strings are small or large across antivirus. Indeed, in the previous comparison, given the lack of standard naming, we have chosen to compute exact matching. Thus, minor variations in label strings may have widely influenced our metric values. We thus compute the similarity between label strings for each application and present the summary in Figure 4.9.

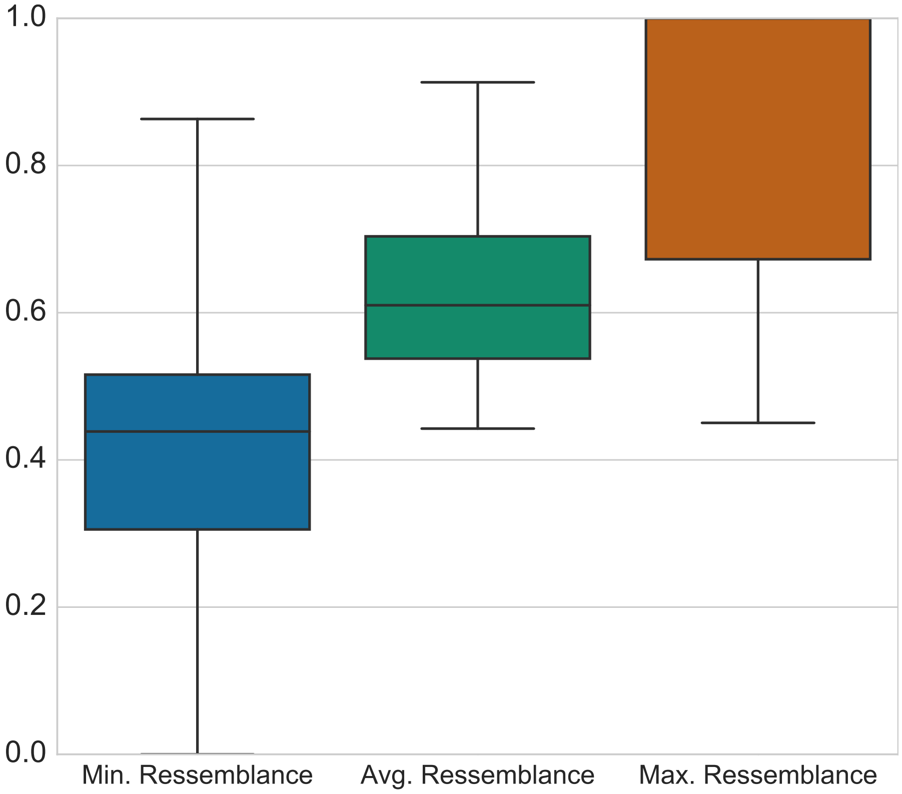


Figure 4.9.: String similarity between labels per application in $\mathscr{D}_{base}$

For each detected sample, we computed the Jaro-Winkler [CRF03] similarity between pairwise combinations of labels provided by antivirus. This distance metric builds on the same intuition as the edit distance (i.e., Levenshtein distance), but is directly normalized between 0 and 1. A similarity value of 1 implies the identically of strings while a value of 0 is indicative of high difference. We consider the minimum, mean and maximum of these similarity values and represent their distributions across all applications. The median of mean similarity values is around 0.6: on average labels only slightly resemble each other. The following research question highlights the consensus that we attempt to measure:

> *Research question 9: Given a set antivirus and the ground truth that they produce, how resembling are the labels assigned by antivirus for each detected sample?*

We answer this metric with the *Resemblance* metric which measures the average similarity between labels assigned by a set of antivirus to a given detected sample.

$$Ressemblance(\mathscr{L}) = \frac{1}{m}\sum_{i=1}^{m}\frac{\sum_{j=1}^{n'_i}\sum_{j'=1}^{n'_i} Jaro-Winkler(l_{i,j}, l_{i,j'})}{n'_i(n'_i-1)}$$

$$\text{with } j \neq j', l_{i,j} \neq \emptyset, l_{i,j'} \neq \emptyset, l_{i,j} \in \mathscr{B}, l_{i,j'} \in \mathscr{B} \text{ and } n'_i = positives(R_i), 2 \leq n'_i \leq n$$

- **Interpretation** estimation of the global resemblance between labels for each app
- **Minimum** 0 when there is no similitude between labels of an application
- **Maximum** 1 when labels are identical per application

Resemblance assesses how labels assigned to a given application would be similar across the considered antivirus. This metric, which is necessary when *Divergence* is high and *Consensuality* is low, can evaluate if the differences between label strings per application are small or large. $\mathscr{D}_{base}$, $\mathscr{D}_{filtered}$ and $\mathscr{D}_{genome}$ present *Resemblance* values of 0.63, 0.57 and 0.60 respectively. Combined with the *Divergence* metric values, we note that reducing the set of antivirus has not yielded datasets where antivirus agree more on the labels.

4.4. Observations on malware datasets

Table 4.3 summarizes the metric values for the three settings described that researchers might use to build ground truth datasets.

Table 4.3.: Summary of STASE Metrics for three common ground-truth settings

	Equiponderance	Exclusivity	Recognition	Synchronicity	Uniformity	Genericity	Divergence	Consensuality	Resemblance
$\mathscr{D}_{base}$	0.27	0.31	0.09	0.32	0.001	0.97	0.77	0.21	0.63
$\mathscr{D}_{filtered}$	0.59	0	0.36	0.75	0.01	0.87	0.95	0.05	0.57
$\mathscr{D}_{genome}$	0.48	0	0.48	0.41	0.04	0.82	0.87	0.06	0.60

The higher values of *Recognition* and *Synchronicity* for $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ in comparison with $\mathscr{D}_{base}$ suggest that these datasets were built with samples that are well known to be malicious in the industry. If we consider that higher *Recognition* and *Synchronicity* values provide guarantees for more reliable ground truth, then $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ are better ground truth candidates than $\mathscr{D}_{base}$. Their lower value of *Genericity* also suggests that antivirus labels provided are more precise than those in $\mathscr{D}_{base}$. At the same time, higher values of *Equiponderance* and *Uniformity* imply that both antivirus detections and labels are more balanced across antivirus.

Divergence and *Consensuality* values, however, suggest that the general agreement on antivirus labels has diminished in $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ in comparison with $\mathscr{D}_{base}$. The *Exclusivity* value of 0 for $\mathscr{D}_{genome}$ and $\mathscr{D}_{filtered}$ further highlights that the constraints put on building those datasets may have eliminated corner cases of malware that only a few, if not 1, antivirus could have been able to spot.

We also note that $\mathscr{D}_{filtered}$ has a higher *Synchronicity* value than $\mathscr{D}_{genome}$, indicating that its settings lead to a selection of antivirus which was more in agreement on their decision. In contrast, the *Divergence* values indicate that the proportion of distinct labels for each sample was higher in $\mathscr{D}_{filtered}$ than in $\mathscr{D}_{genome}$, suggesting that decisions in $\mathscr{D}_{genome}$ are more comfortable to interpret for each sample. Nevertheless, the classification of samples in malware families would be more difficult because of the higher proportion of distinct labels to take into consideration.

4.5. Recommendations for experiments

In this work, we have investigated the output of antivirus systems for Android applications. Based on different metrics, we assessed the discrepancies between three ground truth datasets, independently of their size, and question their reliability for evaluating the performance of malware experiments. The main objective of our work is to provide means for researchers to qualify their ground truth datasets, with respect to antivirus and their heuristics, to increase confidence in research assessments. We also believe that our work can improve the reproducibility of experimental settings, given the limited sharing of security data such as malware samples.

Besides, our analysis of antivirus reports has exposed a global lack of consensus previously highlighted by other authors on other computing platforms [BOA$^+$07, BH08, Har09, MBSZ11]. Although our work cannot solve the challenge of naming inconsistencies directly, the metrics we propose can evaluate ground truth datasets prior and posterior to their transformation by other techniques [PU12, WMGH14, KTA$^+$15]. To uncover the information contained in antivirus reports, we propose that new approaches should be developed to understand the structure of malware labels and unify their output into a standard naming scheme.

Chapter 5.

EUPHONY: unification of malware labels

Android malware is now pervasive and evolving rapidly. Thousands of malware samples are discovered every day with new models of attacks. The growth of these threats has come hand in hand with the proliferation of collective repositories sharing the latest specimens. Having access to a large number of samples opens new research directions aiming at efficiently vetting apps. However, automatically inferring a reference dataset from those repositories is not straightforward and can inadvertently lead to unforeseen misconceptions. On the one hand, samples are often mislabeled as different parties use distinct naming schemes for the same sample. On the other hand, samples are frequently misclassified due to conceptual errors made during labeling processes.

In this chapter, we mine antivirus labels and analyze the associations between all labels given by different vendors to systematically unify common samples into family groups. The key novelty of our approach, named EUPHONY, is that no apriori knowledge on malware families is needed. We evaluate EUPHONY using reference datasets and more than 400 thousand additional samples outside of these datasets. Results show that EUPHONY can accurately label malware with a fine-grained clustering of families while providing competitive performance against the state-of-the-art.

Euphony: Harmonious Unification of Cacophonous
Anti-Virus Vendor Labels for Android Malware

Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash,
Tegawendé F. Bissyande, Yves le Traon, Jacques Klein, Lorenzo Cavallaro

The 14th International Conference on Mining Software Repositories (MSR)
May 20-21, 2017. Buenos Aires, Argentina

Source code: https://github.com/fmind/euphony

Dataset: https://androzoo.uni.lu/labels

Table of Contents

Machine learning based approaches rely on ground truth datasets to training and evaluate statistical models. Unfortunately, as Rossow et al. [RDG+12] pointed out, the literature exhibits several shortcomings, including a lack of correctness, transparency, and realism in the handling of malware datasets. In particular, reliable malware labels are a necessary input to guarantee the quality of both malware detection and classification models.

Malware labeling, however, is not a trivial task. Manual labeling, where a human analyst inspects the actions of the malware in a bid to classify them, is prohibitively expensive, given the number of malware samples discovered every day. In such a setting, it is reasonable to rely on the collective judgment of antivirus vendors who specialize in malware labeling. However, deriving a unified label from labels attached to samples by antivirus vendors is difficult. Inconsistencies in antivirus labels are indeed typical. As we reviewed in the previous chapter, several inconsistencies were found in malware labels such as a global lack of consensus and a high degree of divergence. These inconsistencies are due to both naming disagreements [KTA+15] across vendors, and also a lack of adopted standards[1] for naming malware.

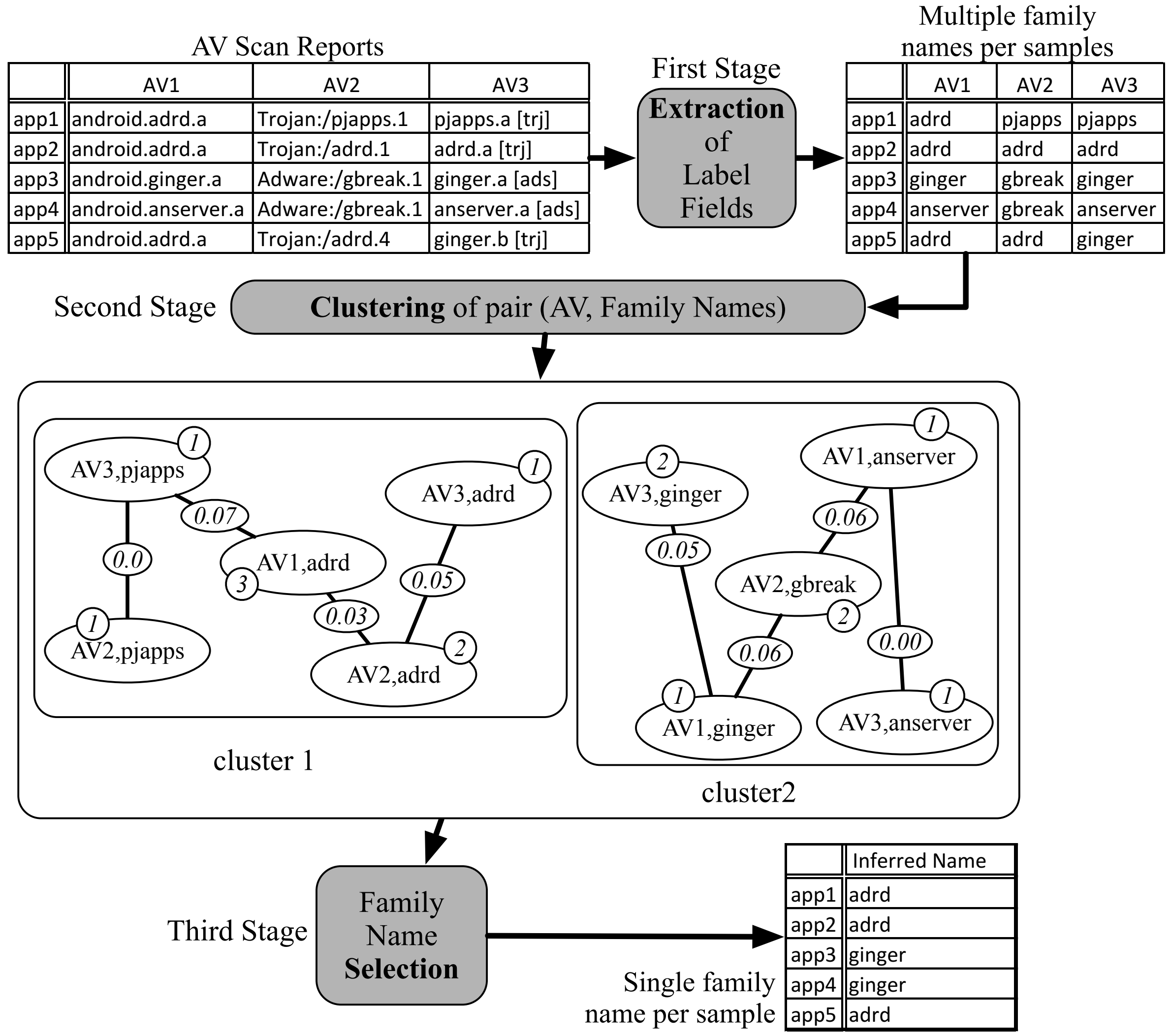
Previous works have relied on simple heuristics to come up with unified labels based on assessment reports of antivirus vendors. For the case of labeling malware as benign or malicious, techniques have labeled a sample as malicious if at least one antivirus vendor flags it as malicious or at least the majority of antivirus vendors have flagged it as malicious [ASH+14, LNP15].

While such heuristics work for flagging samples as malicious or benign, labeling samples with the specific class they belong to is fraught with difficulties. Antivirus vendors can choose different norms to name classes, prefixing qualifying attributes such as attack type (e.g., *Trojan*) or platform (e.g., *Android*) to the label. What further complicates things is that it is not uncommon for typographic and orthographic inconsistencies to creep into the labeling process not just across vendors but sometimes even for the same vendor. Consequently, a sample's full antivirus label is a poor indicator of its generic family name. For example, the family name *Adrd* is "lost" in the full antivirus label *Android.Trojan.Adrd.A (B)*.

In this work, we present EUPHONY, a tunable antivirus labeling system that can systematically extract information from antivirus labels and learns their patterns and vocabulary over time. EUPHONY is an inference based system, which allows for end to end automation, relieving practitioners from the need to collect, aggregate and verify malware families manually. EUPHONY label unification scheme is also vendor agnostic. No specific rules about antivirus engines (e.g., which label parts are suffixes to be removed) are encoded in the process as these rules are inferred from the available antivirus labels data.

Figure 5.1 illustrates the high-level overview of the architecture of EUPHONY. As input, the tool takes a collection of antivirus scanning reports. Such reports can be readily obtained from online services such as *VirusTotal* [noa], which gathers shared intelligence from several antivirus engines. Then for each sample, EUPHONY performs the following tasks:

[1] CARO and CME conventions are not widely used by antivirus vendors.



AV Scan Reports

	AV1	AV2	AV3
app1	android.adrd.a	Trojan:/pjapps.1	pjapps.a [trj]
app2	android.adrd.a	Trojan:/adrd.1	adrd.a [trj]
app3	android.ginger.a	Adware:/gbreak.1	ginger.a [ads]
app4	android.anserver.a	Adware:/gbreak.1	anserver.a [ads]
app5	android.adrd.a	Trojan:/adrd.4	ginger.b [trj]

Multiple family names per samples

	AV1	AV2	AV3
app1	adrd	pjapps	pjapps
app2	adrd	adrd	adrd
app3	ginger	gbreak	ginger
app4	anserver	gbreak	anserver
app5	adrd	adrd	ginger

	Inferred Name
app1	adrd
app2	adrd
app3	ginger
app4	ginger
app5	adrd

Figure 5.1.: Overview of EUPHONY architecture

- **First stage:** antivirus labels are pre-processed to derive the family name assigned by each vendor to a given sample. This task allows EUPHONY to deal with different unstructured naming patterns used by the antivirus and the lack of convention motivates it.

- **Second stage:** This task aims at structuring the relationship between different family names and provides the most appropriate associations between them. EUPHONY analyses both the correlation and the overlap between all family names to understand (i) *mislabeled* and (ii) *misclassified* samples. While mislabeling a sample generally happens when many antiviruses use a different naming scheme for the same family (e.g.: *DroidKungFu* vs. *DrdKngFu*), misclassifying a sample is usually associated with a conceptual error made by an antivirus—with respect to the others (e.g., a vendor labels a sample as *GingerMaster* while the others decide that belongs to *DroidKungFu*).

- **Third stage:** This task aims at bringing consensus between the different vendors and outputs the most appropriate family name for a given sample. Although our framework can also output a set of family names for a sample (i.e., synonyms), for the sake of simplicity, we only report the most prevalent one.

We next describe the details of each of the components in EUPHONY as well as the choices made during their design and implementation.

5.1. Definition of labeling process

5.1.1. Antivirus labels

Definition 1 (Antivirus label)**.** *An antivirus label l is a sequence of words (i.e., alphanumerical tokens) w_i divided by separators (i.e., blanks and punctuation signs) u_j. Formally, $l = (w_1, u_1, \dots, u_n, w_{n+1})$.*

`Android.Trojan.Adrd.A (B)` is a concrete example of such a label, where '.' (*dot*), '(', ')' (*parentheses*), and '␣' (*space*) are the separators and *Android*, *Trojan*, *Adrd*, *A*, and *B* are the words.

Definition 2 (Antivirus label field)**.** *A label field f represents the category of a given word w_i.*

The word *Android* in `Android.Trojan.Adrd.A (B)` indicates the target platform of the malware, while *Trojan* and *Adrd* indicate its type and family respectively. Overall, we define 4 fields that match details required in the CARO naming convention [SSB]: **type** (the kind of threat, i.e., trojan, worm, etc.), **platform** (the OS that the threat is designed to work on, i.e., Windows, Android, etc.), **family** (the group of threats it is associated with in terms of behavior), **information** (extra description of this threat, including its variant).

Grammars described in Table 5.1 and Table 5.2 below provide the lexing rules used by EUPHONY to tokenize antivirus labels.

Table 5.1.: Lexing rules of EUPHONY

<*family*>	::=	[:alpha:]{3,}
<*type*>	::=	[:alpha:]{2,}
<*info*>	::=	[:alnum:]+
<*plat*>	::=	[:alnum:]{2,}
<*sep*>	::=	([:punct:] | [:blank:])+

Definition 3 (Antivirus labeling pattern)**.** *Given an antivirus* av*, its corresponding antivirus labeling Pattern, noted p_{av} represents the syntax of its labels, i.e., how the different fields are combined to form its labels.*

Table 5.2.: Parsing rules of EUPHONY

<*word*>	::=	<family> | <type> | <plat> | <info>
<*label*>	::=	<word> | <sep> | <label>

We provide in Table 5.3 some illustrative examples of antivirus labels, their fields, and their associated labeling patterns.

Table 5.3.: Examples of antivirus labeling patterns

Label	**AV Pattern**	**Family**	**Type**	**Plat.**	**Info**
Android.Trojan.Adrd	<plat>.<type>.<name>	adrd	trojan	android	–
Trojan:/Adrd.b	<type>:/<name>.<info>	adrd	trojan	–	b
Android:PjApps [Trj]	<plat>:<name> [<type>]	pjapps	trj	android	–
Troj.PjApps (kcloud)	<type>.<name> (<info>)	pjapps	troj	–	kcloud
Android/Adrd.5e2f	<plat>/<name>.<info>	adrd	–	android	5e2f

5.1.2. Sample sets

We define a labeling function $label\_of$ which associates a label to a pair (av, app) of antivirus and application from a dataset:

Definition 4 (Labeling function)**.** *Let APP be a set of applications, antivirus a set of antivirus, and $\mathscr{L}$ the set of associated labels.*

The function $label\_of : AV \times APP \rightarrow \mathscr{L}$ maps a pair of antivirus and application to a label.

From a given label, we further define a family function $family\_of$ which extracts the family field value.

Definition 5 (Family function)**.** *Let AV be a set of antivirus, APP a set of applications. Let be $\mathscr{L}$ the set of labels such as $\mathscr{L} = label\_of(AV, APP)$, and $\mathscr{F}$ a set of associated family names.*

The function $family\_of : AV \times \mathscr{L} \rightarrow \mathscr{F}$ maps a pair of antivirus and label to a family name.

For a given AV, we put together apps with the same family name in a set that we call *Sample Set*. More formally:

Definition 6 (Sample set)**.** *Let $APP = (app_1, app_2, ..., app_n)$ be a set of app samples, and $\mathscr{F} = (f_1, f_2, ..., f_k)$ a set of associated families. For a given antivirus* av*, the sample set S_{av,f_j} defines the set of apps with the same family name f_j. More formally,*

$$\forall j \in (1, ..., k), S_{av,f_j} = \{app_x \in APP | family\_of(av, app_x) = f_j\}$$

5.1.3. Metrics

Sample sets can be disjoint or overlapping, and may be imbalanced. For instance, the sample sets represented in the left of Figure 5.2 are imbalanced as the number of samples associated with the family name f_i by av_a is much smaller than the number of samples associated with family name f_j by av_b. Understanding when a sample set is imbalanced is important when weighting the relevance of a label over another in a dataset.

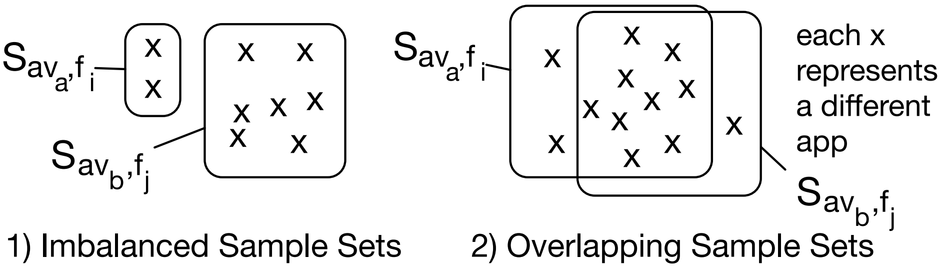

S_{av_a,f_i}
S_{av_b,f_j}
1) Imbalanced Sample Sets
S_{av_a,f_i}
each x represents a different app
S_{av_b,f_j}
2) Overlapping Sample Sets

Figure 5.2.: Examples of application sample sets

Definition 7 (Imbalance metric). *Given two sample sets S_{av_a,f_i} and S_{av_b,f_j}, we define the* Imbalance *metric as the complement between the minimum and maximum cardinality of the two sample sets, formalized in Equation 5.1.*

$$Im(S_{av_a,f_i}, S_{av_b,f_j}) = 1 - \frac{\min(|S_{av_a,f_i}|, |S_{av_b,f_j}|)}{\max(|S_{av_a,f_i}|, |S_{av_b,f_j}|)} \tag{5.1}$$

When both sets S_{av_a,f_i} and S_{av_b,f_j} have the same cardinality, there is no imbalance, and *Im* is equal to 0. In contrast, imbalance *Im* gets close to 1 as S_{av_a,f_i} contains fewer apps and S_{av_b,f_j} contains a lot more.

The right part of Figure 5.2 depicts overlapping sample sets, i.e., a scenario where several apps have been associated at the same time with f_i and f_j by av_a and av_b respectively. This setting suggests that, despite syntactic differences between both family names f_i and f_j, the family names may characterize the same information (e.g., they point to the same malware). The notion of overlapping is essential to assess whether family names should be merged. Thus, we define the exclusion metric which quantifies the degree of overlapping, or lack thereof.

Definition 8 (Exclusion metric). *Given two sample sets S_{av_a,f_i} and S_{av_b,f_j}, we define the* Exclusion Metric *as the complement of the ratio between the intersection cardinality of two sample sets S_{av_a,f_i} and S_{av_b,f_j} the cardinality of the smallest sample set. This metric is formalized by Equation 5.2.*

$$Ex(S_{av_a,f_i}, S_{av_b,f_j}) = 1 - \frac{|(S_{av_a,f_i} \cap S_{av_b,f_j})|}{\min(|S_{av_a,f_i}|, |S_{av_b,f_j}|)} \tag{5.2}$$

When the sets S_{av_a,f_i} and S_{av_b,f_j} share no app, there is no overlapping as their intersection is empty, and *Ex* is at its maximum at 1. In contrast, as the overlapping gets higher, *Ex* gets close to 0.

Given that family names may contain small syntactic differences, we consider a distance function to relax the equality constraint on strings. To that end, we define our String distance based on the Sørensen–Dice index [Dic45].

Definition 9 (Distance metric of family names)**.** *The distance metric of family names is computed as the string distance between two family names f_i and f_j:*

$$D(f_i, f_j) = 1 - dice(f_i, f_j) \tag{5.3}$$

Finally, we measure how two family names f_i and f_j, given by the antivirus av_a and av_b respectively, are far to designate the same malware family. More specifically, we compute the distance between two samples sets S_{av_a,f_i} and S_{av_b,f_j} by combining the imbalance, exclusion as well as the string distance metrics that we introduced. The following equation provides the formula that we use:

Definition 10 (Sample set distance metric)**.** *Given two sample sets S_{av_a,f_i} and S_{av_b,f_j}, we define the* Sample Set Distance Metric *as follows:*

$$W(S_{av_a,f_i}, S_{av_b,f_j}) = \alpha \times Ex(S_{av_a,f_i}, S_{av_b,f_j}) + \beta \times Im(S_{av_a,f_i}, S_{av_b,f_j}) + \gamma \times D(f_i, f_j) \tag{5.4}$$

Where α, β and γ are weight coefficients for adjusting the importance of the different metrics leveraged to compute the sample set distance. First and foremost, we consider that two family names are close to each other only if there is a strong overlap (i.e., low exclusion) between their associated sample sets. Thus, for example, it is not opportune to consider two family names as similar if they do not occur concurrently for the same samples. Consequently, the value of α will reflect the importance of the *Ex* metric. Second, the imbalance of sample sets is considered to account for the degree of granularity within malware families. For example, an antivirus might assign two family names to a sample set (e.g., *ADRD*, *Pjapps*) while another antivirus might use only one (e.g., *Pjapps*) family name for all samples in the set. Finally, the impact of typos, which may increase distances between sample sets, requires the string distance to be the least weighted. We have empirically found that a difference of an order of magnitude captures the best relative importance among the coefficients. Thus, in EUPHONY, we set α, β and γ to 1, $\frac{1}{10}$ and $\frac{1}{100}$ respectively.

5.2. Extraction of label information

An antivirus label is an informally structured string concatenating various pieces of information for describing the malware. In the previous section, we have identified four recurrent fields in antivirus labels which are identifiable in labels: family, type, platform, and extra information. Each vendor generally adopts a specific naming convention to represent and combine these fields in a string. For example, while some vendors start with the platform first, followed by the type and the name; other vendors opt to put the type first or enclose it between square brackets at the end of the string. We further note that antiviruses change their convention over time, varying field ordering and the punctuation signs that separate fields. In this context, the normalization of their syntax cannot be achieved consistently via fixed rules such as regular expressions.

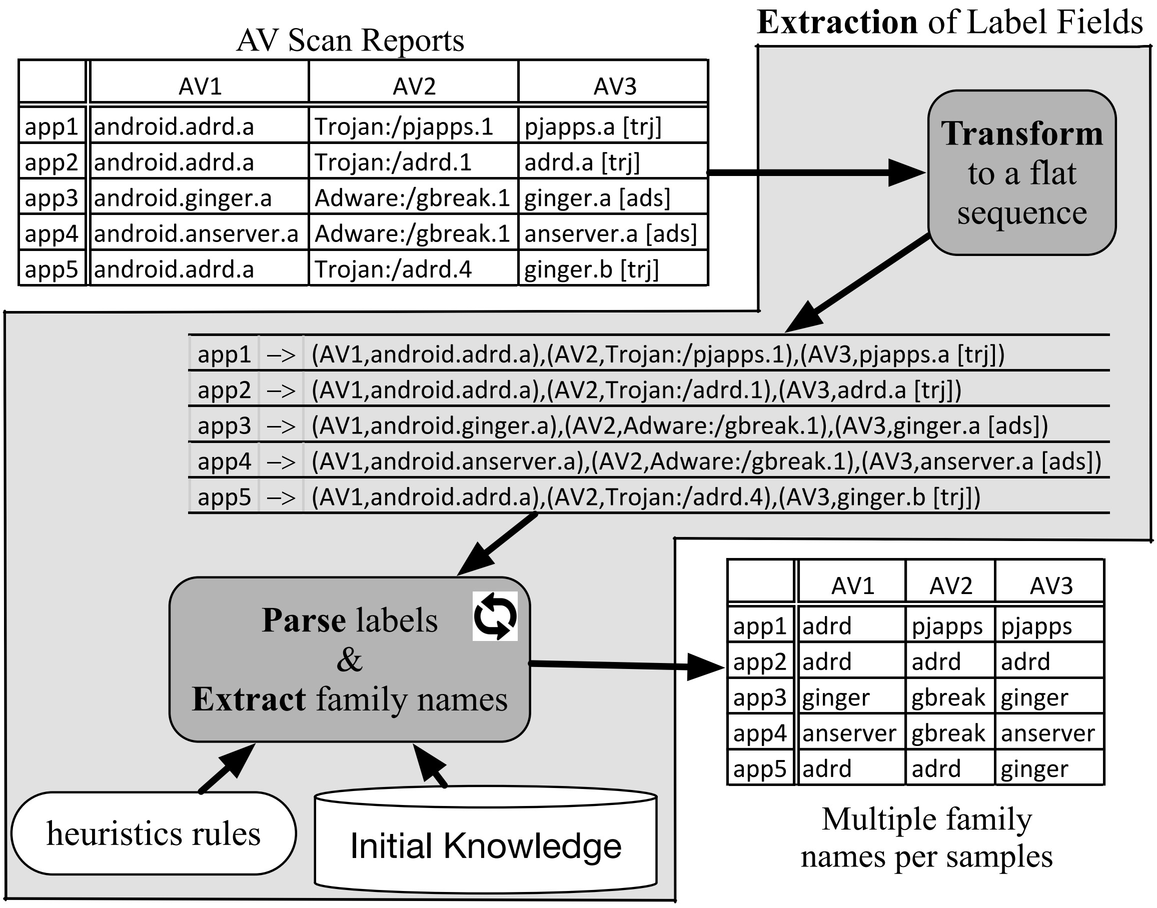
Another crucial constraint in parsing antivirus labels is the lack of a complete, universal and up-to-date lexicon. Indeed, new types, platforms, and family names are continuously added by antivirus vendors to describe emergent threats, and refine the description of old threats. Malware family names, in particular, are highly dynamic as new malicious behaviors appear regularly.

With these limitations in mind, we propose several heuristics for mapping antivirus label tokens to a lexical field. The overall family name extraction process is described in Figure 5.3. Given a collection of antivirus labels, the system can infer the most apparent fields and then iteratively move to the most challenging cases as its knowledge grows. To bootstrap the process, EUPHONY builds on heuristics based rules, as well as a bare amount of vocabulary on some platform names (e.g., *Android*) and types (e.g., *trojan*). The final output which is the family names given by each antivirus to the samples is obtained by inferring it from the sample's label for each vendor.

5.2.1. Parsing algorithm

The parsing algorithm is at the core of the labeling process and its steps are described in Algorithm 1. The process takes as input a set of labels, some defined heuristics and an initial knowledge database on malware lexicon. First, the algorithm tokenizes each antivirus label and initializes the mappings between the tokens and the different label fields. At this stage, a given token can be associated with all fields (name, type, platform or information). To decide the unique field to which it should be assigned, the algorithm proceeds by iteratively eliminating improper assignment, starting with the easiest cases: the order of processing is conveyed by a priority queue (line 4), where mappings with the least amount of unknown fields are pushed at the head of the queue, while mappings with the most amount of unknown fields are pushed back at the end of the queue. At each step, the algorithm takes the first mapping of the queue (line 6) and applies the heuristics based rules to collect more information about the mapping (line 8). Then, it merges this information to create a new mapping. In case of conflicts, the merge operation will always keep the oldest knowledge at its disposal.

If the mapping is complete at the end of this operation (line 11), i.e., if each token is associated with a single field, this mapping is removed from the queue and its information pieces are extracted to enrich the knowledge database (line 12). Otherwise, the mapping is pushed back in the queue to be processed at a later iteration (line 14). Once the queue is empty, the mapping


Extraction of Label Fields
AV Scan Reports
Transform to a flat sequence
Parse labels & Extract family names
heuristics rules
Initial Knowledge
Multiple family names per samples

	AV1	AV2	AV3
app1	android.adrd.a	Trojan:/pjapps.1	pjapps.a [trj]
app2	android.adrd.a	Trojan:/adrd.1	adrd.a [trj]
app3	android.ginger.a	Adware:/gbreak.1	ginger.a [ads]
app4	android.anserver.a	Adware:/gbreak.1	anserver.a [ads]
app5	android.adrd.a	Trojan:/adrd.4	ginger.b [trj]

app1	–>	(AV1,android.adrd.a),(AV2,Trojan:/pjapps.1),(AV3,pjapps.a [trj])
app2	–>	(AV1,android.adrd.a),(AV2,Trojan:/adrd.1),(AV3,adrd.a [trj])
app3	–>	(AV1,android.ginger.a),(AV2,Adware:/gbreak.1),(AV3,ginger.a [ads])
app4	–>	(AV1,android.anserver.a),(AV2,Adware:/gbreak.1),(AV3,anserver.a [ads])
app5	–>	(AV1,android.adrd.a),(AV2,Trojan:/adrd.4),(AV3,ginger.b [trj])

	AV1	AV2	AV3
app1	adrd	pjapps	pjapps
app2	adrd	adrd	adrd
app3	ginger	gbreak	ginger
app4	anserver	gbreak	anserver
app5	adrd	adrd	ginger

Figure 5.3.: First stage - extraction of label fields from malware reports

Algorithm 1 Incremental Parsing by EUPHONY

```
 1: $Mapping \leftarrow [name, type, platform, information]$
 2: function PARSE($knowledge_{db}$, $heuristics$, $labels$)
 3:     $mappings \leftarrow$ MAP($Mapping, labels$)
 4:     $pqueue \leftarrow$ PRIO-QUEUE($mappings$)
 5:     while NOT-EMPTY?($pqueue$) do
 6:         $m \leftarrow$ PEEK($pqueue$)
 7:         for $H$ in $heuristics$ do
 8:             $findings \leftarrow$ H($knowledge_{db}, m$)
 9:             MERGE($m, findings$)
10:         end for
11:         if COMPLETE?($m$) then
12:             ENRICH($knowledge_{db}, m$)
13:         else
14:             PUSH($pqueue, m$)
15:         end if
16:     end while
17:     return $mappings$
18: end function
```

list is returned with the complete list of associations (line 17). To force early termination, EUPHONY provides a parameter for setting a maximum number of iterations performed by the algorithm.

5.2.2. Heuristics rules

We now provide details on the parameters of the algorithm. In our current implementation, we rely on ten heuristic rules to find associations between words and fields. These rules are listed in Table 5.4.

Let us consider two of the rules to illustrate the associated action. Rule 1 is the most straightforward heuristic. During its execution, EUPHONY accesses the database to check if the word is already associated with a particular field. In particular, the word *Android* is commonly known to match with the field `platform`. Thus, Rule 1 can leverage existing knowledge to identify obvious fields. Rule 9, on the other hand, is tuned to create more knowledge by inferring the field of an unknown token. For example, given the antivirus label 'ransom.android.pjapps' and its incomplete mapping [ransom: `?`, android: `platform`, pjapps: `family`], the algorithm can know at this stage that *ransom* is likely a `type`, and yield the following antivirus labeling pattern: '`<type>.<platform>.<name>`'. This inference is validated by correlating with mappings for all samples of the same antivirus. Once the mapping is complete, the inferred information will be added to the database and support the identification of more tokens.

	Property	**Action**
1	Word is associated to a known field in the database	associate the same field
2	Word is suffixed by -ware	word is a type
3	Word is between parenthesis	word is an info
4	Word is between square brackets	word is a type or info
5	Only one family, type and platform per label	enforce when field is found
6	Word is a synonym of a type or platform in the database (e.g. troj, trojan)	associate the same field
7	Word is the last token not associated to a field	word is a family
8	Words are part of common word sentence	words are info
9	Label is compatible with a pattern of the same AV	associate fields based on pattern
10	Given two remaining tokens, one is a common word and the other is not	common word: information, other word: family

Table 5.4.: Heuristics for mapping label words to fields

5.2.3. Initial lexicon

To bootstrap the inference process, our algorithm requires an initial lexicon about malware labels. Generally, a small but widely accepted lexicon can be found online in specialized knowledge bases. We stress that, in EUPHONY, such a lexicon does not have to be exhaustive for our algorithm to work correctly. For example, in our experimental setting, we have leveraged a limited lexicon including only most well know types, platforms, and information enumerated by the Microsoft Malware Protection Center[2]. Table 5.5 provides statistics and examples of tokens contained in this list. In particular, we observed that important words such as "Android", "Malware" or family names are not present. We demonstrate the automated nature of the inference system by relying only on this available lexicon without any modifications of its entries.

Field	**# of Entries**	**Example**
TYPE	34	adware, backdoor, spyware, trojan, worm
PLATFORM	74	linux, androidos, iphoneos, java, win32
INFORMATION	18	dll, rootkit, plugin, pak, gen

Table 5.5.: Initial database entries of EUPHONY

[2] Malware Naming Conventions: http://bit.ly/2f3vKlu

5.3. Clustering of malware families

After parsing malware labels to identify family names given by different antivirus to each sample, EUPHONY builds a graph representing the association links between family names based on their assignment on some samples. Then, based on a threshold parameter that determines the granularity of grouping, clusters of family names are separated. Figure 5.4 provides an overview of the process. In the rest of the chapter, we will often use the terms “name” instead of the full expression “family name”.

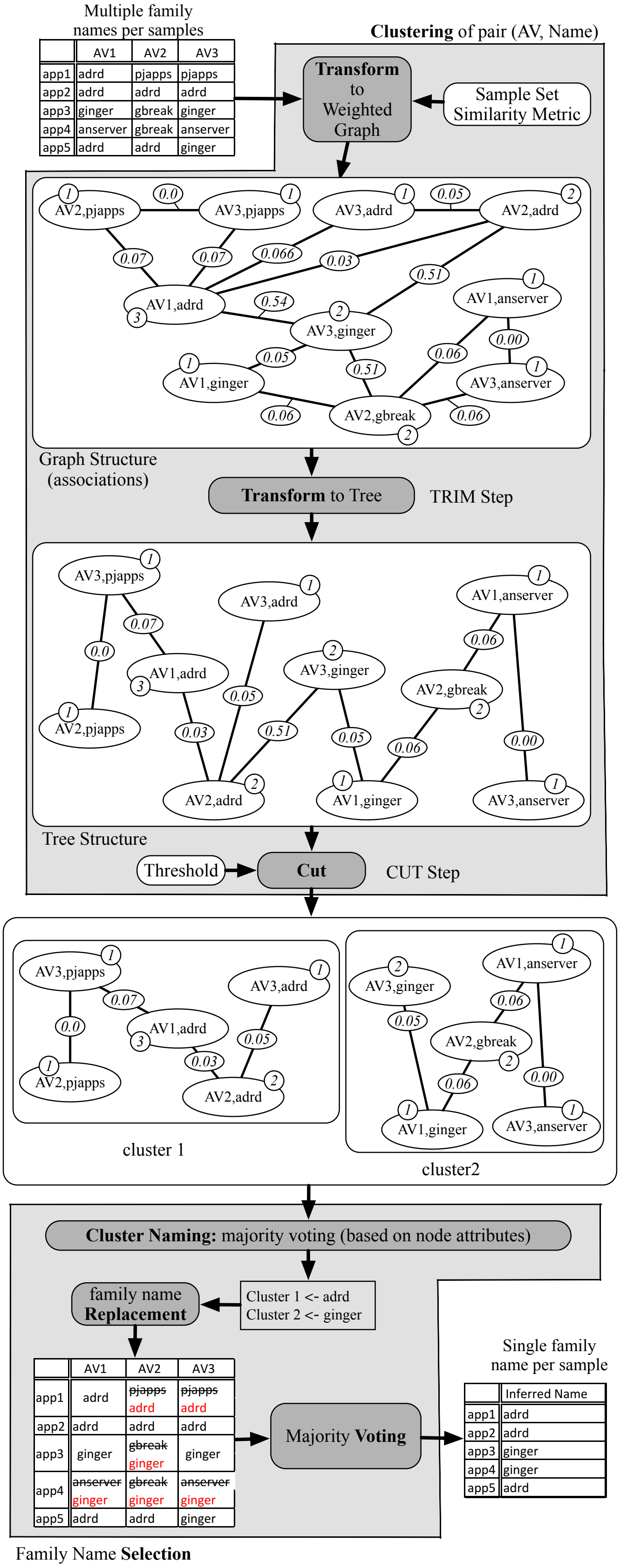
5.3.1. Associating family names

At the end of the previous stage, EUPHONY has a new dataset where each sample is associated with multiple malware family names reported by antivirus. These potentially syntactically different names may include mislabeling noises and misclassification errors, which make the process of selecting unique names more difficult.

We study the associations between antivirus family names to group together commonly related names. We found that the most natural method to analyze potential associations was to construct a weighted graph $G = (N, E)$, where a node $n \in N$ represents a name that an antivirus assigned: $n = (av_a, name)$, and an edge $e = [(av_a, name_x), (av_b, name_y)] \in E$ indicates that both antivirus av_a and av_b have labeled the same sample with $name_x$ and $name_y$ respectively. From the information attached to a node, EUPHONY can identify the corresponding Sample Set $S_{av_a,name}$ and computes, for each node, the size of its sample set (i.e., the number of times the family name is given by that antivirus in the dataset). It also computes, for each edge, the overlapping between the related sample sets (i.e., the number of times antivirus agree on the name given at that node). Both pieces of information are then used to compute the *Imbalance* and *Exclusion* metric, and eventually the *Sample Set Distance* metric, also taking into account the string distance between family names. Imbalance and Exclusion metric values are used as edge attributes in the graph, while the Sample Set Distance metric value is used as the edge weight (in Figure 5.4 only the edge weight is represented). Note that the weight of an edge represents the degree to which the sample sets associated with the connected nodes are dissimilar: the lower the weight, the more likely both sample sets belong to the same cluster.

5.3.2. Clustering family names

Given the large number of associations that we have observed among family names in our datasets, we expect the weighted graph to be highly connected and thus include very few identifiable subgraphs. For instance, a generic name can create additional edges with more specific names and thus tie together components that were otherwise weakly related. Processing mistakes during label fields extraction can also introduce fake associations among family names. It is therefore essential to remove such undesired associations from the graph and only keep



	AV1	AV2	AV3
app1	adrd	pjapps	pjapps
app2	adrd	adrd	adrd
app3	ginger	gbreak	ginger
app4	anserver	gbreak	anserver
app5	adrd	adrd	ginger

	AV1	AV2	AV3
app1	adrd	~~pjapps~~ adrd	~~pjapps~~ adrd
app2	adrd	adrd	adrd
app3	ginger	~~gbreak~~ ginger	ginger
app4	~~anserver~~ ginger	~~gbreak~~ ginger	~~anserver~~ ginger
app5	adrd	adrd	ginger

	Inferred Name
app1	adrd
app2	adrd
app3	ginger
app4	ginger
app5	adrd

Figure 5.4.: Second stage - clustering & Third stage - inference of family name

subgraphs with strongly related nodes. To that end, we build a technique that comprises two successive steps, referred to as TRIM and CUT:

- In the TRIM step, we use Prim's algorithm [Pri57] to transform the graph into a Minimum Spanning Tree (i.e., the sum of its edge weights is minimum). The goal of this operation is to reduce the complexity of the original structure and keep the most similar edges as long as they do not introduce cycles in the graph. The complexity of this algorithm is $O(|E|\log|N|)$ in our current implementation.
- The CUT step takes the tree structure and applies a filter function to remove edges whose weights exceed a given threshold value. As a result, the input tree is divided into connected components that can be interpreted as clusters of strongly related family names. The complexity of this algorithm is $O(|E|)$ in our implementation.

5.3.3. Inferring family names

In the last stage, we study the relation between the different family names to assess the prevalence of predominant naming schemes used by the different antivirus. In particular, we created a list of associations that put in relation all family names within the cluster where they are grouped. More specifically, we first associate a single name to each cluster. This name is inferred as the most frequent name present in the cluster by taking into account the attribute of each node. In the step illustrated at the bottom of Figure 5.4, cluster 1 is associated with *adrd* and cluster 2 to *ginger*. Then, each family name present in a cluster is replaced by the cluster name. For instance, in Figure 5.4, $pjapps$ given by $AV2$ is replaced by $adrd$. More generally, in this example, all occurrences of $pjapps$ are replaced by $adrd$ and all occurrences of *anserver* or *gbreak* by *ginger*.

Once the replacements are done, to infer a single family name per sample, we implement a majority voting where we compute the frequency of each family name and select the most frequent one per sample. In the case of a tie, we use the highest frequency of the names within the dataset to choose between the candidates and break the conflict.

5.4. Analysis of EUPHONY results

5.4.1. Datasets and metrics

The evaluation of EUPHONY is based on two different sets of samples: (i) *reference datasets*, and (ii) an *in the wild dataset*. We next describe the source of each of the datasets used (see Table 5.6 for a summary) and the metrics used to evaluate our approach.

Reference datasets: These datasets have been distributed by the research community together with a reference ground truth of malware families, and have been widely used in the literature recently [ASH$^+$14, DSTK$^+$16, LNP15]. For our study, we consider *MalGenome* [ZJ12], a dataset manually vetted and collected between 2008 and 2010, and which includes 1 262 samples regrouped into 44 families. Similarly to previous works [SRKC16], we update this dataset by grouping into a single family all variants of *DroidKungFu* (*DroidKungFu*1, *DroidKungFu*2, *DroidKungFusApp*, etc.). Additionally, we also consider *Drebin* [ASH$^+$14], a dataset collected between 2010 and 2012, and which includes all samples from *MalGenome* as well as an additional set of 3 998 more samples. *Drebin* includes 178 families.

In the wild dataset: We collected recent samples from *Androzoo* [ABKT16], a repository that shares samples from a variety of sources as well as their antivirus labels provided by *VirusTotal*. For our study, we leveraged the public download API and retrieved 402 600 samples created between January 2015 and August 2016[3]. We ensured that all samples were classified as malware by at least one antivirus[4].

Table 5.6.: Datasets used in EUPHONY evaluation

Reference	Wild	Samples	Families	Anti-Virus	Collection Period
MalGenome [ZJ12]	✗	1 262	44	58	08/2008 - 10-2010
Drebin [ASH$^+$14]	✗	5 260	178	57	08/2010 - 10/2012
Androzoo [ABKT16]	✓	402 600	unknown	63	01/2015 - 08/2016

Evaluation metrics: Let S be a sample dataset, $G = \{G_1, \dots, G_s\}$ be the set of s “ground truth” clusters from S, and $C = \{C_1, \dots, C_n\}$ be the set of n clusters output by a given tool over S. Similarly to previous works [SRKC16], we define the following metrics:

- **Precision:** $Prec = \frac{1}{n} \times \sum_{j=1}^{n} \max_{k=1,\dots,s}(|C_j \cap G_k|)$
- **Recall:** $Rec = \frac{1}{s} \cdot \sum_{k=1}^{s} \max_{j=1,\dots,n}(|C_j \cap G_k|)$
- **F1 score:** $F1 = 2 \times \frac{Prec \times Rec}{Prec + Rec}$

While precision measures the effectiveness of a tool to map outputted clusters into ground truth clusters, recall quantifies the effectiveness of the tool to map ground truth clusters into outputted clusters. Finally, the F Measure represents the harmonic mean between precision and recall.

In this chapter, we first investigate the precision and recall reported when clustering malware samples in the reference datasets. These metrics allow us to compare our approach with previous works quantitatively [SRKC16]. We then use the samples collected in the wild to evaluate several statistical metrics such as the number of families, the number of singletons and the most relevant labels.

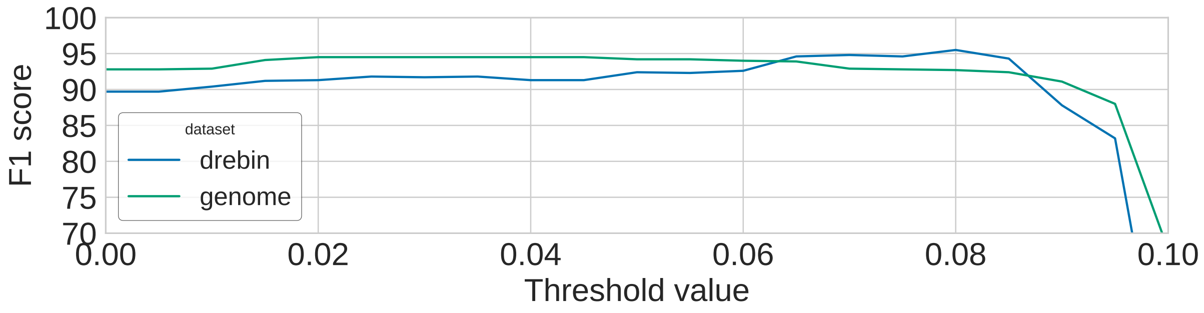


Figure 5.5.: Parameter selection of the threshold value

Table 5.7.: Performance of EUPHONY against state-of-the-art (in %)

	EUPHONY				*AVClass*											
	-				***AVClass* Config 1**			***AVClass* Config 2**			***AVClass* Config 3**			***AVClass* Config 4**		
	-				*as reported in [SRKC16]*			*with default files in Git*			*new aliases only*			*new generics & aliases*		
Dataset	Prec	Rec	F1	#	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
MalGen.	86.7	99.7	92.7	33	87.2	98.8	92.6	86.5	98.0	91.9	86.2	99.0	92.1	53.9	65.2	59.0
Drebin	95.0	96.1	95.5	142	95.2	92.5	93.9	95.4	93.0	94.2	95.6	90.6	93.0	29.6	69.8	41.6

5.4.2. Performance evaluation

EUPHONY uses a threshold to control the clustering sensitivity by breaking edges whose weight exceeds the given value. On Figure 5.5, we observed that a threshold of 0.07 represented an excellent trade-off between noise reduction and accuracy.

We now evaluate the performance reported over the reference datasets. The leftmost part of Table 5.7 shows the precision, recall and F1 measure for EUPHONY. Results show that clustering *MalGenome* is, overall, more challenging than clustering *Drebin*, with a F1 measure of 92.7% and 95.5% respectively.

These results can be explained by looking at the precision, which shows that not all predicted clusters can be mapped to their respective reference clusters. Interestingly, the score obtained for the recall indicates that almost all referenced clusters (i.e., 99.7% of the malware families) in *MalGenome* have been correctly predicted. Instead, only 96.1% of the referenced clusters in *Drebin* have been correctly predicted.

When analyzing *MalGenome* results, we observe that some antivirus prefer to combine some reference clusters to form one single super family. For instance, families *ADRD* (22 samples) and $Pjapps$ (58 samples), are perceived as one large family called $Pjapps^*$ (with 80 samples). Similarly, *BaseBridge* (122 samples) and *AnserverBot* (187 samples) are perceived as *Basebridge** (with 309 samples). This is understandable as authors in [ZJ12] believe that *AnserverBot* evolved from *BaseBridge*, inheriting common features. Other recent works have also confirmed this and pointed out that some other families are also strongly related to each other [STTPLB14]. Based on this, one can conclude that the perception of the antivirus is,

[3] *Androzoo* bases its timeline on the DEX compilation date.

[4] Overall, the samples were labeled by 63 antivirus

in some general cases, more coarse-grained. Thus, they can treat two similar clusters as one single family.

As for the results obtained with *Drebin*, we found some cases where the antivirus agree on using a more fine-grained definition of some families than the one given in the reference ground truth. For example, most of the samples from the reference family *Opfake* (952 samples) are subdivided into two sub families, i.e., *Opfake* (546 samples) and *SMSSend* (25 samples). Similarly, most of the samples in *ExploitLinuxLotoor* (70 samples) are subdivided into three subfamilies: *Lotoor** (58 samples), *GingerBreak* (9 samples) and *AsRoot* (8 samples).

Comparison against the state-of-the-art

To provide further insights on the performance achieved by EUPHONY, we compare our results with *AVClass* [SRKC16]. Since the authors have analyzed their approach on *MalGenome* and *Drebin* datasets, we use the results in their paper as our evaluation baseline (Table 5.7, *AVClass* Config 1). We also replicate their experiments by taking into account the difference between the inputs requirements of both tools. On the one hand, EUPHONY uses a small list of some well-known words (not including family names) about malware labels. On the other hand, *AVClass* requires a list of malware families to construct a set of *generics tokens* and *aliases*. These sets are built in a two-step process. First, *AVClass* uses the list of malware families to distinguish these family tokens from other so-called *generics tokens* (e.g., types, platforms, information). Second, *AVClass* strips generics tokens from malware labels to discover *aliases* among malware family names. Finally, the sets of *generics tokens* and *aliases* are leveraged to produce the final output of the tool: a single family name per malware sample using a plurality voting. Several factors may thus impact the performance of AVClass, including the exhaustiveness of the inputted list of malware families, and the error rate in the generation of the sets of generics and aliases. Consequently, we consider three different scenarios for our evaluation:

- An updated version of *AVClass*—we use the last version of the tool released on GitHub, taking into account recent code fixes, as well as updates to complete the list of generic terms and aliases[5]. The results are reported in Table 5.7, *AVClass* Config 2.
- Automatic inference of *aliases* only—we use the authors' script with the default settings to generate a list of aliases based on our two reference datasets (Table 5.7 Config 3).
- Automatic inference of both *generics* and *aliases*—we use the authors' scripts with the default settings on the union of our two reference datasets to build the knowledge necessary to *AVClass*'s functioning (Table 5.7 Config 4).

All results are reported in Table 5.7. For all four configurations, EUPHONY performs better than *AVClass* in terms of F1 score (harmonic mean between precision and recall). Nonetheless, we observed that their precision is comparable to ours in those configurations where prior

[5] AVClass repository cloned on Oct 24, 2016. Commit head: 80c14adcc29978ab813b41c73dd485072e576140

knowledge (aliases and generic terms) is provided. When we inferred both (i) aliases, and (ii) aliases and generic terms using the samples given in the reference datasets (i.e.,: *MalGenome* and *Drebin*)[6], we observed that the performance drops drastically. Typical families included in Config. 4 are: android (537 samples), trojan (377 samples) and basebridge (68 samples). These observations show that the performance of *AVClass* is driven by the input of an initial knowledge—which should be collected by the final user. In contrast, EUPHONY does not require any guiding process or pre-defined knowledge of the families. The only prior knowledge required by our framework is a basic understanding of some common types of malware (i.e., trojan, virus, etc.), execution platforms (i.e., android, linux, Win32, etc.) and information (e.g., dll, pak, gen).

5.4.3. Evaluation of samples in the wild

This section reports our experiments in the wild. We analyze the number of samples that EUPHONY can group with respect to *AVClass*. Note that in this section we report results using the same experimental setting used above. As for *AVClass*, we choose the most favorable configuration[7]. Table 5.8 summarizes the results obtained on the *Androzoo* dataset.

Table 5.8.: Results of EUPHONY for *Androzoo* (402 600 samples)

	Labeled	**Clusters**	**Singletons**	**Runtime**
EUPHONY	319 100	735	165	216s
AVClass	178 471	453	135	114s

Results show that EUPHONY managed to cluster 79% of the samples (319 100 out of 402 600). In contrast, *AVClass* clustered 44% (178 471 out of 402 600). These results mean that a practitioner using *AVClass* would not obtain labels for more than half of the recent dataset. This can be partly explained by the strategy used by *AVClass* to handle generic terms in labels, as well as aliases in family names.

On the contrary, our approach does not present distinctions between generic and specific malware families, and may thus find more associations. This can further provide a better understanding of the appropriate set of samples in every cluster. In this regard, EUPHONY has split the dataset into 735 clusters and produces 165 clusters with one single sample (namely, singletons). Instead, *AVClass* proposed 453 clusters and 135 singletons. Note that the runtime overhead of EUPHONY is negligible compared to *AVClass*, even in this setting where the creation of *generics tokens* and *aliases* for *AVClass* was skipped.

Table 5.9 shows the Top 10 clusters (in terms of size) for both EUPHONY and *AVClass*. Both approaches report clusters of the same order of magnitude and with similar family names.

[6] Note that *AVClass* published results [SRKC16] were obtained using the knowledge on aliases and generics that was built from larger datasets

[7] This is, using the default lists of aliases and generics collected by the authors. These lists may have been manually improved to guarantee the labeling system out of the lab

Table 5.9.: Top 10 clusters of EUPHONY and *AVClass*

EUPHONY		**AVClass**	
Family	**Samples**	**Family**	**Samples**
dowgin	37 739	kuguo	38 532
kuguo	25 005	dowgin	22 643
addisplay	20 862	secapk	20 492
jiagu	20 705	airpush	13 209
anydown	19 621	jiagu	8 987
secapk	18 224	smsreg	8 427
generic	17 836	feiwo	7 399
agent	17 596	revmob	6 376
inmobi	16 203	leadbolt	5 348
airpush	13 267	anydown	5 147

This table indicates that EUPHONY reaches similar conclusions than *AVClass* for the most popular families, but without prior knowledge of generic families.

We can also observe that our approach can deal with generic antivirus labels. For instance, a common field used by antivirus is "trojan.androidos.generic.a", with 94 255 occurrences (4%). We can further observe 576 261 occurrences (23%) of the string "gen" in the list of labels. This result contrast with the most occurring family, *Dowgin*, with 311 593 times (14%). Moreover, it explains the lack of coverage reported by *AVClass*. We position here that being aware of these types of clusters are important to filter out samples that might interfere with the proper classification of other clusters. In practice, adware and other type of grayware [STTPLR14] could be identified. Nevertheless, to account for corner cases where a generic term is selected as a family name, practitioners can inject into EUPHONY their knowledge on how a specific token must be associated with a label field. Since a significant portion of samples remained unlabeled by *AVClass* and EUPHONY, we only consider the subset of samples that are labeled by both tools to investigate the similarities and differences among reported clusters. We provide in Table 5.10 the statistics on the new Top 10 clusters (dropping samples that are unlabeled by either tool) and information on the extent to which they overlap. Most top families strongly overlap. For example, *Dogwin*, the most prevalent family in EUPHONY, overlaps with the also labeled *Dogwin* family in *AVClass* with a ratio of 95%. For 7 of the top 10 clusters given by EUPHONY, we find that the corresponding cluster by *AVClass* overlaps at over 85%. If we take the particular case of samples labeled as *Kuguo* by *AVClass*, EUPHONY splits them into mainly three families (*kuguo*, *addisplay*, *hiddeninstall*) with an overlap of 99%, 54%, and 93% respectively.

5.5. Support of threat intelligence services

As the lack of human experts disrupts our ability to analyze malware in the large [ICS18], threat intelligence services will become essential to enable the detection and the classification

Table 5.10.: Top 10 clusters of EUPHONY compared to *AVClass*

EUPHONY		***AVClass***		**Intersection**	
Family	**samples**	**Family**	**samples**	**samples**	**overlap (in %)**
dowgin	33 297	dowgin	22 617	21 035	93.0
kuguo	24 273	kuguo	38 532	24 072	99.2
secapk	17 889	secapk	20 492	17 825	99.6
addisplay	11 203	kuguo	38 532	6 055	54.0
airpush	10 055	airpush	13 202	10 017	99.6
jiagu	7 215	jiagu	8 987	7 211	99.9
smsreg	6 294	smsreg	8 427	5 819	92.5
agent	6 088	feiwo	7 399	1 014	16.7
revmob	6 061	revmob	6 376	6 058	99.9
generic	5 663	anydown	5 147	1 890	36.7

of Android malware further. Online services such as *VirusTotal* already contribute to this effort by providing diversified antivirus labels for malware samples. However, the results of *VirusTotal* cannot be leveraged as-if to support the creation of ground truth datasets due to the lack of naming convention between antivirus engines. With EUPHONY, we developed a state-of-the-art approach that unifies the output of antivirus systems and assigns valuable information from the knowledge embedded in antivirus labels.

EUPHONY improves over *AVClass* by overcoming its main limitations of requiring a substantial amount of initial knowledge on malware families and antivirus vendors to bootstrap the labeling process. Moreover, reference datasets need to be regularly updated without prior knowledge on generic terms that antivirus vendors use to label samples as new malware families appear. From a small and relatively stable list of common tokens, EUPHONY can (1) infer missing information on tokens in antivirus label strings using heuristics, and (2) group similar families together according to a comprehensive distance metric that takes into account typos in naming, imbalance in label assignment among antivirus sample sets, and overlapping of sets. While EUPHONY can be used off-the-shelf, without any requirement of expertise on malware labels, advanced users may also build on top of our framework and specify their heuristics, metrics or knowledge of malware labels to build their unification process.

Since the creation of better ground truth datasets is an important objective toward the development and the adoption of machine learning based systems, the security community must continue to provide a better interpretation on the relation between malware families and their structural features. Recent approaches [STDA$^+$17] have been efficient at identifying syntactic and resource-centric features that characterize Android malware. We position that adding these features to our algorithm could contribute to alleviate disagreements among antivirus vendors and support the extraction of malicious artifacts associated with specific malware families.

Chapter 6.

AP-GRAPH: dissection of malware artifacts

Android markets such as Google Play must continuously protect their users against exposure to malicious applications. In this arms race, up-to-date information is crucial to adapt security solutions to the current malware landscape. However, despite the most recent efforts of the research community, this knowledge is still severely lacking. On the one hand, our understanding of malicious applications comes from private companies who are not willing to share their information. On the other hand, recent techniques developed to characterize malicious applications are generating too much data for manual reviews.

In this chapter, we propose AP-GRAPH as a solution to address these two challenges. AP-GRAPH can analyze large sets of malicious applications at scale and find their most discriminative artifacts based on the partial knowledge provided by antivirus solutions. We evaluated our approach on 1 million Android malware and observed that AP-GRAPH identified the specific characteristics of a dozen of malware families. Moreover, AP-GRAPH is capable of reducing the number of artifacts generated by other characterization approaches to limit the noise generated by such techniques. This information can help the research community in locating the malicious parts hidden inside malware applications and recommend the inspection of suspicious artifacts to security experts.

This chapter is based on yet unpublished material.

Database: https://androzoo.uni.lu/apksearch

Analysis: https://androzoo.uni.lu/apklyze/

Table of Contents

To improve the security of Android markets, research groups around the world have developed machine learning based systems that detect malicious applications before they impact mobile users [BZNT11, WJ12, WMW$^+$12, GYAR13, ADY13, CGC13, ASH$^+$14, CWL$^+$15, AQR$^+$16, MOA$^+$17]. Even though these systems have been effective in the lab, their adoptions have been comparatively stalled in real-world use cases [ABJ$^+$16, SP10, CSD$^+$17, RDG$^+$12]. One reason to explain this lack of success is that the proposed systems must rely on an extensive set of qualified samples to learn the boundary between benign and malicious applications properly. Unfortunately, the constant evolution of malicious applications makes the validation of malware samples a bottleneck, as human resources are not sufficient to vet ground truth datasets at such scale [ICS18].

As an alternative, the security community relies on antivirus engines to classify malware samples before their use in machine learning experiments. However, this approach is not without flaws. Industrial actors do not share the details of their analysis to the public as it would threaten their business model. At best, antivirus reports provide a condensed label that mentions the type and the name of a malware. Without a transparent database of knowledge, supervised learning systems cannot be trained for explaining and locating malicious behaviors concealed inside malicious applications. Moreover, this limitation prevents our community from working with more precise malware definitions and limits our ability to fine-tune our detection systems to the most advanced threats.

In this work, we propose a data mining solution named AP-GRAPH to uncover artifacts associated with malicious behaviors of Android malware families. With only the information provided by antivirus engines, AP-GRAPH can isolate the specific components of a malware family, categorize their nature and locate their presence inside applications. The information we collect can then be used for two primary purposes. On the one hand, security analysts can adopt AP-GRAPH as a triage system to evaluate which artifacts are good candidates and find malicious behaviors in unknown malware samples. On the other hand, the output of AP-GRAPH can characterize the distinctive aspects of a malware family to complement the use of machine learning based systems and vet large malware ground truth datasets at scale.

6.1. Specification of malware artifacts

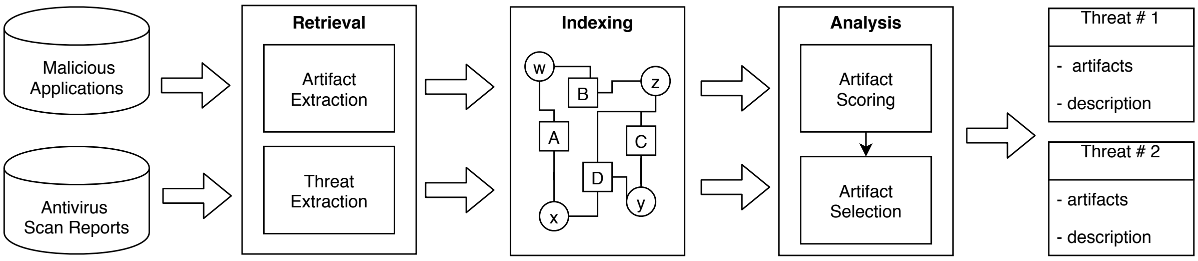


Figure 6.1.: Architecture of AP-GRAPH divided into 3 stages

The goal of our approach is to find the internal components that are specific to malware family by using the information publicly available to the security community: a broad set of malicious applications and antivirus scan reports.

The overall approach of AP-GRAPH can be summarized in Figure 6.1.

In this section, we will detail the process used to retrieve, index and analyze this information.

6.1.1. Information retrieval

6.1.1.1. Family extraction

A **family** t is a name given to a group of malware $t = \{p_1, p_2, ..., p_n\}$ that shares the same artifacts $\{a_1, a_2, ..., a_m\}$. Given a **Classifier** $\mathscr{C}$, we define the **Scanning Function** $\mathscr{S}(p, \mathscr{C}) = t$ that associates a family t to an application p. In this context, an antivirus product can be reduced to a Scanning Function $\mathscr{S}$ that returns a label. An antivirus label contains various information about the malware, such as its type, name and execution platform. In our approach, the family is only associated with the name included in the antivirus label.

Unfortunately, extracting information from antivirus labels is not a trivial process, as other researchers found that the naming convention followed by antivirus products is not standard across industrial vendors [Bon05, Har09]. Moreover, the ability of antivirus to correctly classify malware varies greatly across products as we demonstrated in Chapter 4. In order to collect this information, we rely on EUPHONY Chapter 5 to achieve state-of-the-art results in unifying the labels of multiple antivirus vendors.

We further note that the selection of families $\{t_1, t_2, ..., t_n\}$ and classifiers $\{\mathscr{C}_1, \mathscr{C}_2, ..., \mathscr{C}_n\}$ is independent of the approach presented in this chapter. Any clustering of malicious applications can be used as source material in attempting to identify the artifacts specific to a given malware family. In particular, we note that multiple taxonomies can be tested in parallel, as the extraction of artifacts is independent of the extraction of families during the later stage of the analysis. The only requirement is that the families provided to AP-GRAPH are associated with more than one malicious applications.

6.1.1.2. Artifact extraction

We define the **artifact** a of an **application** p as internal components such that $a \subset p$. Artifacts are collected through an **Extraction function** $\mathscr{E}(p) \rightarrow \{a_1, a_2, ..., a_n\}$ associated with either a static or dynamic analysis. The **Verification Function** $\mathscr{V}(a, p, \mathscr{E}) \rightarrow \{True, False\}$ can validate that an artifact a either exists or does not exist inside an application p, given an Extraction Function $\mathscr{E}$ as a parameter.

The output of the Verification Function $\mathscr{V}$ is represented by a boolean value such that

- $\mathscr{V}(a,p,\mathscr{E}) = True$ when $a \subset p$: the artifact a exists inside the application p given the Extraction Function $\mathscr{E}$.
- $\mathscr{V}(a,p,\mathscr{E}) = False$ when $a \subset p$: the artifact a does not exist inside the application p given the Extraction Function $\mathscr{E}$.

Every artifact a must be encoded by an **Indexing Function** $\mathscr{I}(a) \rightarrow i$ to guarantee that each a is associated with a unique identifier i and each a can be located unambiguously inside an application p. Thus, the Indexing Function $\mathscr{I}$ is a bijection from the set of artifacts $\{a_1, a_2, ..., a_n\}$ to the set of identifiers $\{i_1, i_2, ..., i_n\}$. The **Locating Function** $\mathscr{L}(i) \rightarrow a$ is the inverse of the Indexing Function, such that $\mathscr{L}(\mathscr{I}(a)) = a$.

The nature of an artifact a depends on the Extraction Function $\mathscr{E}$ that was chosen for the analysis. For instance, static analysis retrieves a list of symbols and constants from the source code of an application. On the other hand, dynamic analysis executes the application to gather system calls and memory footprints. These elements can constitute a set of artifacts as long as we can define an Indexing Function and a Locating Function for them.

6.1.2. Information indexing

6.1.2.1. Data models

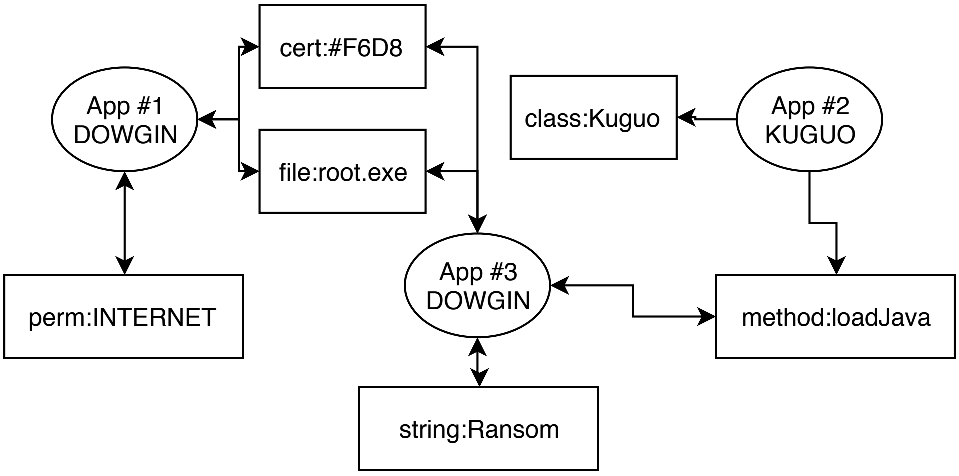


Figure 6.2.: Indexing graph produced by AP-GRAPH: applications are represented as ellipses nodes and artifacts are represented as rectangle nodes.

The **Indexing Graph** g is a data structure that stores the relations between a set of applications $\{p_1, p_2, ..., p_m\}$ and a set of artifacts $\{a_1, a_2, ..., a_n\}$. These elements are represented as nodes in the indexing graph and can contain additional attributes such as the family associated with a malicious application. More formally, the set of nodes of the Indexing Graph can be defined as $\mathscr{N} = \{p_1, p_2, ..., p_m\} \cup \{a_1, a_2, ..., a_n\}$. The edges of the graph are retrieved by the Verification Function $\mathscr{V}$ and represent the inclusion of an artifact a in an application p when

$\mathscr{V}(a,p) = True$. More formally, the set of edges of the Indexing Graph can be defined as $\mathscr{N} = \{(p_i,a_j) | \mathscr{V}(a_j,p_i) = True\}$.

Figure 6.2 shows an example of an Indexing Graph for a small set of applications and artifacts. Nodes that are applications are represented with elliptical forms and nodes that are artifacts are represented with rectangular forms. We can see that applications #1 and #3 are associated with the *dowgin* family, while application #2 is associated with the *kuguo* family. The relation between the applications and the artifacts are represented with arrows in our graph. Thus, application #1 is associated with 3 artifacts in this case: perm:internet, cert:#f6d8 and file:root.exe. It is also possible that generic artifacts are included by different families, like the artifact method:loadJava which is included by application #1 (*dowgin*) and #2 (*kuguo*).

6.1.2.2. Data queries

The main benefit of the Indexing Graph compared to other data structures is that it offers the ability to navigate efficiently between artifacts and applications. Given an application p, we can easily retrieve all the artifacts $\{a_1,a_2,...,a_n\}$ that are associated with p. Similarly, we can also retrieve all the applications $\{p_1,p_2,...,p_n\}$ that are associated with artifact a.

More formally, let us note g an Indexing Graph, p_i an application, and a_j an artifact, we define the following queries that are supported by AP-GRAPH. For each query, we also provide an example based on the graph from Figure 6.2:

1. $Q1(g,p) = \{a_1,a_2,...,a_n\} \in p$: retrieve the set of artifacts $\{a_1,a_2,...,a_n\}$ that are included in application p
   - $Q1(g,App\#2) = \{class : Kuguo, method : loadJava\}$
2. $Q2(g,a) = \{p_1,p_2,...,p_n\} \ni a$: retrieve the set of applications $\{p_1,p_2,...,p_n\}$ that include artifact a
   - $Q2(g,method : loadJava) = \{App\#2, App\#3\}$
3. $Q3(g,a) = \{t : a \in p \wedge t \ni p\}$: retrieve the set of families $\{t_1,t_2,...,t_n\}$ that are associated through the set of applications $\{p_1,p_2,...,p_n\}$ that includes artifact a
   - $Q3(g,method : loadJava) = \{dowgin, kuguo\}$
4. $Q4(g,a) = \mid Q2(g,a) \mid$: compute the total number of applications that include artifact a
   - $Q4(g,string : Ransom) = 1$
5. $Q5(g,t,a) = \mid \{p : p \ni a \wedge p \in t\} \mid$: compute the number of applications associated with family t that include artifact a
   - $Q5(g,DOWGIN,file : root.exe) = 2$

6.1.3. Information analysis

6.1.3.1. Artifact scoring

To isolate the artifacts which are specific to families, AP-GRAPH relies on a single metric $\mathscr{M}(g,a) \in [0,1]$ computed from the queries introduced in the previous section. This metric quantifies the importance that a single family has compared to the other families. Thus, if an artifact is present uniformly in multiple families, its $\mathscr{M}$ value will be close to 0. On the contrary, if the artifact is present mostly in a single family, its $\mathscr{M}$ value will be close to 1.

More formally, we define the metric $\mathscr{M}$ as follow:

$$\mathscr{M}(g,a) = \max(\{Q5(g,t,a) \ : \ t \ \in \ Q3(g,a)\})/Q4(g,a)$$

Let us consider a concrete example, where an artifact a appears in 4 families: $\{t_1,t_2,t_3,t_4\}$. The distribution of artifact a across these families is: $Q5(g,t_1,a) = 100$, $Q5(g,t_2,a) = 200$, $Q5(g,t_3,a) = 200$, $Q5(g,t_4,a) = 1000$ with $Q4(g,a) = 1500$. The value of the metric, in this case, is $\mathscr{M}(g,a) = \max(100,200,200,1000)/1500 = 2/3$. This value is associated with the specificity that the artifact a has on the family t_4.

6.1.3.2. Artifact selection

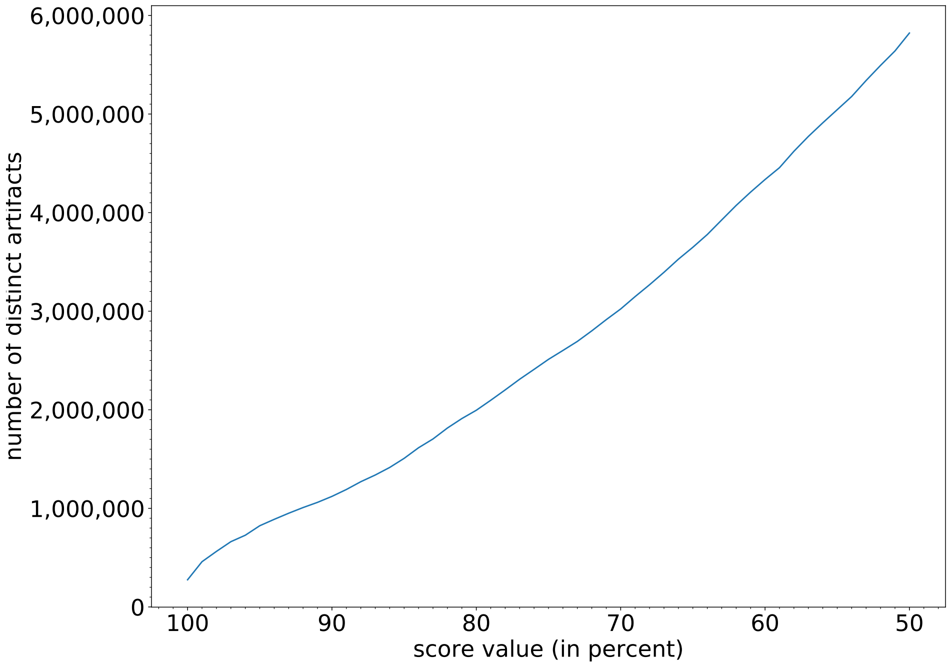


Figure 6.3.: Number of distinct artifacts related to scoring value thresholds ($\mathscr{M}$)

AP-GRAPH use the scoring value $\mathscr{M}$ as a high-pass filter[1] to discriminate the artifacts specific to a single family. As we described in the previous section, the value of $\mathscr{M}$ will be close to 1 when an artifact appears mostly in a single family. In particular, the value of $\mathscr{M}$ will be precisely 1 when the artifact is associated with a single family. While a perfect scoring value would be an excellent theoretical target, antivirus products are known to produce noisy predictions and misclassification as we observed in Chapter 4. To take into account this problem, we set a **threshold value** τ to reduce our sensitivity to misclassification by antivirus products.

In the absence of ground truth, the threshold value for our analysis is selected based on the information at our disposal. On Figure 6.3, we plot the total number of distinct artifacts retrieved by AP-GRAPH for different values of τ. We can see that the association between these two variables is linear. Thus, a smaller value of τ will uncover more specific artifacts than a higher value. On the other hand, a smaller value will also include more errors due to antivirus misclassification.

In our analysis, we selected a conservative threshold value of 0.95 to take into account a small amount of noise introduced by antivirus misclassification. Therefore, the most important family for a given artifact must account for at least 95% of the total of times an artifact is encountered in malware samples or $\mathscr{M}(g,a) > 0.95$. For instance, if an artifact appears 1000 times in total, this artifact is considered specific to a family by AP-GRAPH if an only if it is present at least 950 times in this single family.

6.2. Creation of malware knowledge base

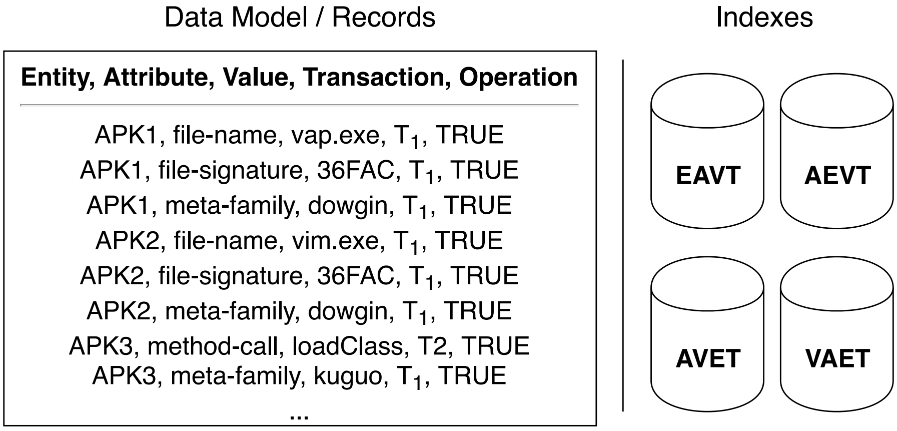
In this section, we discuss three architectures designed to support the indexing of malware artifact at scale.

Each part of this section will discuss the trade-off associated with the proposed architectures and their benefits on computing statistics from malware datasets.

6.2.1. Architecture A: Datomic

Datomic [Cog] is a commercial database released in 2012 by Cognitech. The data model of Datomic is inspired by the Resource Description Framework (RDF), as documents are serialized in a 5-tuples containing the entity, attribute, value, transaction, and operation status. Figure 6.4 shows an example of tuples associated with three Android malware. In this example, malware entities are described based on information gathered from static analysis, such as the list of file names and signatures embedded in the application or the list of methods that will be called during its execution. Datomic features a flexible querying system as records can be saved in four different indexes to support row-oriented (EAVT), column-oriented (AEVT and AVET) and graph-oriented (VAET) access. Compared to relational database systems, Datomic

[1] a filter that passes values higher than a threshold value



Entity	Attribute	Value	Transaction	Operation
APK1	file-name	vap.exe	T_1	TRUE
APK1	file-signature	36FAC	T_1	TRUE
APK1	meta-family	dowgin	T_1	TRUE
APK2	file-name	vim.exe	T_1	TRUE
APK2	file-signature	36FAC	T_1	TRUE
APK2	meta-family	dowgin	T_1	TRUE
APK3	method-call	loadClass	T2	TRUE
APK3	meta-family	kuguo	T_1	TRUE
...				

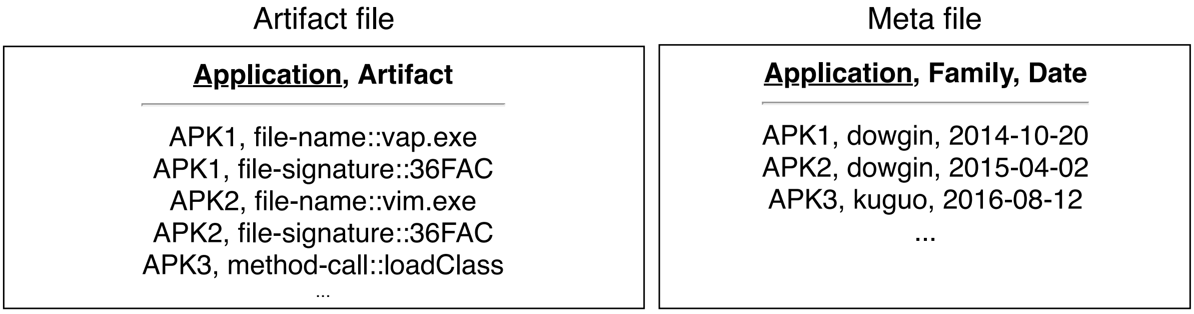
Figure 6.4.: Structure of a knowledge base powered by Datomic

information are never updated nor deleted and can only be accumulated over time. Thus, an operator can query the state of the knowledge base at a particular point in time and find the delta that was added before or after a point in time.

Datomic fits the requirements of AP-GRAPH for several reasons. On the one hand, Datomic supports a wide variety of indexing patterns to optimize the performance of read queries. The EAVT index can be leveraged to request information about a particular Android application while the AEVT and AVET indexes can be used to compute statistics on the distribution of malware artifacts. On the other hand, the data schema can be extended with new records, as the Datomic model can be adapted to represent information without impacting the existing structure of the database.

However, Datomic is not suited for use cases that require intensive write operations. Indeed, Datomic supports ACID properties by limiting the writing process to a single transactor (i.e., a single machine core across the cluster). Thus, and despite great read performance, Datomic cannot power the indexing of malware ground truth as large as Androzoo [ABKT16].

6.2.2. Architecture B: Flat file



Application	Artifact
APK1	file-name::vap.exe
APK1	file-signature::36FAC
APK2	file-name::vim.exe
APK2	file-signature::36FAC
APK3	method-call::loadClass
...	

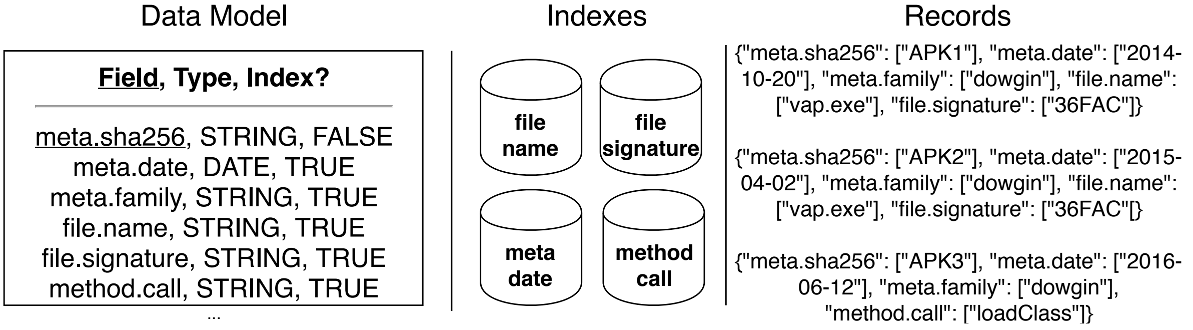
Application	Family	Date
APK1	dowgin	2014-10-20
APK2	dowgin	2015-04-02
APK3	kuguo	2016-08-12
...		

Figure 6.5.: Structure of a knowledge base powered by flat files

Flat files provide a simple format and rely on the operating system to perform read and write operations. Information can be organized into a delimited file where each line corresponds to a new record. Figure 6.5 represents two flat files that describes the relationship between applications, artifacts, family and time. On the one hand, the artifact file, once sorted, can be used to retrieve information by application or by artifact to support the query required by AP-GRAPH. On the other hand, the meta file can be joined to the artifact file to add additional information that allows the analysis of malware families over time.

However, the simplicity of flat files is both their greatest strength and their greatest weakness. Flat files do not support indexing or query capabilities found in other database systems beside sequential read and write operations. Moreover, indexing the artifact file either by application or by artifact requires twice the amount of disk space both for sorting and storing the result file. In the end, a flat file architecture is a fast prototype solution that works as long as a single machine has enough disk capacity during the sorting process.

6.2.3. Architecture C: Elastic



Field, Type, Index?

meta.sha256, STRING, FALSE
meta.date, DATE, TRUE
meta.family, STRING, TRUE
file.name, STRING, TRUE
file.signature, STRING, TRUE
method.call, STRING, TRUE
...

```
{"meta.sha256": ["APK1"], "meta.date": ["2014-10-20"], "meta.family": ["dowgin"], "file.name": ["vap.exe"], "file.signature": ["36FAC"]}

{"meta.sha256": ["APK2"], "meta.date": ["2015-04-02"], "meta.family": ["dowgin"], "file.name": ["vap.exe"], "file.signature": ["36FAC"[}

{"meta.sha256": ["APK3"], "meta.date": ["2016-06-12"], "meta.family": ["dowgin"], "method.call": ["loadClass"]}
```

Figure 6.6.: Structure of a knowledge base powered by ElasticSearch

Elastic [Ela] is an open source database built to retrieve information from structured documents such as JSON files. The database has been used by many professional actors to search a large corpus of text, as the indexing capabilities of Elastic are tailored toward this use case. Moreover, an Elastic cluster can be deployed across multiple computers to aggregate the processing and storing capacity from more than one machine.

Figure 6.6 presents a structure for an Elastic knowledge base. Artifacts are listed in a schema file that contains the name, the type and the operation index for the fields. In Elastic, fields are automatically indexed to enumerate documents that contain them efficiently. For instance, the file signature ’36FAC’ is found both in ’APK1’ and ’APK2’, which means that these two documents will be returned when a user issues a query.

The main benefit of an Elastic based architecture is to power the queries of AP-GRAPH at scale. The system can retrieve information about a single application, a single artifact or complex analytic queries. Since data are distributed across multiple machines, computer resources can be shared to speed up the analysis or support more massive ground truth datasets.

Table 6.1.: Distribution of applications and malware per market (note: an application can be distributed on multiple markets)

Markets	Malware	Total	%
play.google.com	579,840	4,315,707	13
anzhi	501,748	742,788	67
appchina	334,544	593,110	56
mi.com	71,584	113,583	63
1mobile	15,922	57,530	27
angeeks	17,528	55,794	31
slideme	8,645	52,467	16
praguard	0	10,186	0
torrents	126	5,294	2
freewarelovers	169	4,145	4
proandroid	346	3,683	9
hiapk	1,153	2,512	45
fdroid	40	2,023	1
genome	1,247	1,247	100
apk_bang	121	363	33
unknown	0	57	0

However, the solution is the most complex of the three as computer clusters are harder to maintain than a system deployed on a single computer. Furthermore, while existing documents can be updated to add more fields, this task is a costly operation as it requires to reindex the whole database file. Hence, this architecture solution should be reserved for complex use cases or when information cannot be stored on a single computer.

6.3. Characterization of malware families

In this section, we report the results of AP-GRAPH in characterizing malware families.

In the first part, we describe the datasets, artifacts, and families comprised in our analysis. Then, we evaluate the performance of AP-GRAPH in collecting specific artifacts while simultaneously dropping the ones which are not relevant in discriminating malware families. Finally, we perform a case study analysis to assess the quality of the information retrieved by AP-GRAPH.

6.3.1. Dataset

The evaluation of AP-GRAPH is based on a large set of Android malware collected by the Androzoo project [ABKT16]. This project aims to provide a representative sample of Android applications that other researchers can use in their experiments. Androzoo contains more than 6 million Android applications gathered from various Android markets. The dataset also includes nearly 1 million apps considered malicious by at least one antivirus referenced on VirusTotal [noa]. The distribution of applications and malware per market can be found in Table 6.1. In the Android ecosystem, *play.google.com* is recognized as the official market, while other market places are alternative sources of applications. From this table, we can observe that alternative markets have a higher proportion of malicious applications than the official market.

6.3.1.1. Families

The families of our evaluation are assembled from Androzoo++ [LGH$^+$17], a collection of metadata related to the Androzoo dataset. In particular, this project includes the label of every antivirus referenced on VirusTotal [noa]. As we noted in Chapter 4, antivirus labels can provide noisy output and contain various information which are not directly related to the family name. To handle this lack of consistency, we use EUPHONY to extract the name from antivirus labels and to unify them across multiple antivirus products.

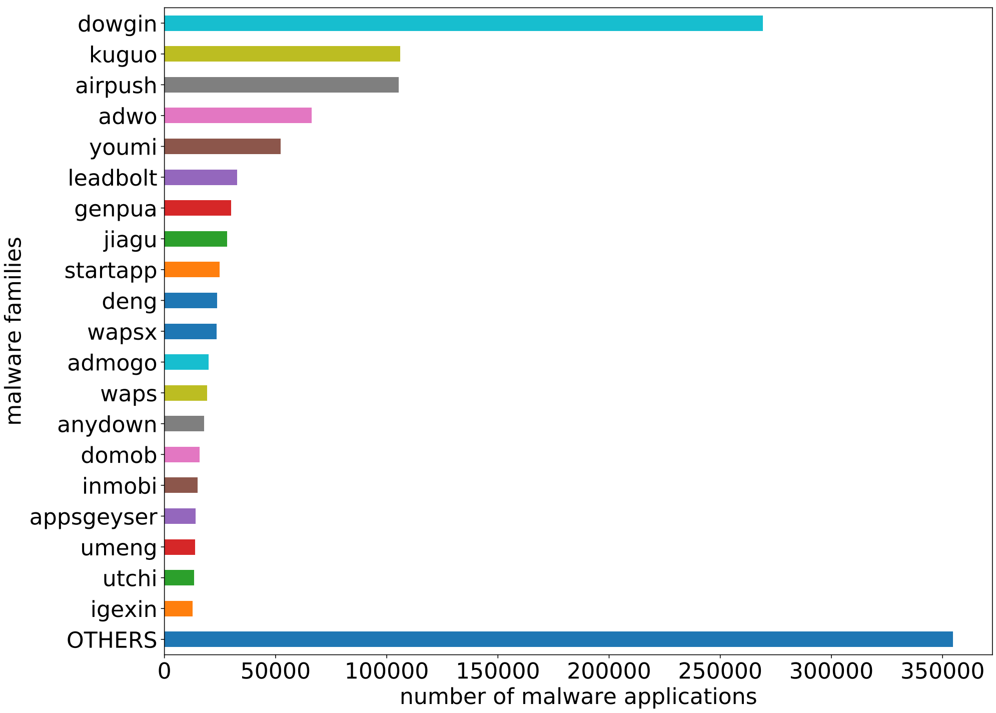


Figure 6.7.: Distribution of malware families with at least 100 samples according to Euphony

Figure 6.7 shows the distribution of malware families for the labels unified by Euphony. We can see that the distribution has a long tail of small families, as *dowgin* dominates the ranking while smaller families are grouped into the OTHERS bin. In total, Euphony proposes 3600 different malware families for the Androzoo dataset. Our study focused on the larger 20 families proposed by the following antivirus: ESET-Nod32, Sophos, GData, F-Secure (in addition to Euphony unified names). Antiviruses are selected in order to maximize the number of positive detections based on Androzoo++ data.

6.3.1.2. Artifacts

The artifacts included in our analysis are extracted with Androguard [DG], an open source tool that decompiles Android applications to retrieve their resources and bytecodes. This tool is classified as a static analysis system, as it does not execute the application on a virtual machine to collect runtime information. Instead, one of the main benefits of this approach is the global coverage that the solution provides. Every static component in the application (e.g., strings, methods, files) can be indexed as an artifact identifier by AP-GRAPH. In total, our evaluation includes 732 million distinct artifacts collected from 1 million malicious applications. This type of static analysis is also fast and scalable, as it requires only 8 seconds on average for a single machine core to decompile the applications and collect its artifacts.

The artifacts considered in our analysis are grouped into 46 categories, and displayed in Figure 6.8. Each category is associated with a specific artifact type. For instance, the strings present in dex files (a packaged version of the application source code) are identified by the entry dex::string. As another example, the source code of methods is hashed and identified by the entry dex::code. The listing below provides an overview of the artifact categories covered in our evaluation:

DEX information (starting with dex) : hash value computed from source codes (*code*), string values (*string*), method invocations (*invoke*), field names (*field*), method names (*method*), class names (*class*), package names (*package*), superclass names (*super*), dex magic number (*magic*), dex format (*format*).

File information (starting with file) : hash value computed from file content (*signature*), file names (*name*).

Manifest information (starting with manifest) : activity names (*activity*), service names (*service*), provider names (*provider*), receiver names (*receiver*), process names (*process*), intent names (*intent*), data keys (*data*), metadata keys (*meta*), application package (*package*), application version (*version*), application features (*app*), application libraries and permissions (*uses*), custom permissions (*perm*), user identifier (*sharedUserId*), protection levels (*protection*), instrumentation classes (*instru*).

Resources information (starting with resource) : string values for both keys and entries (*string*).

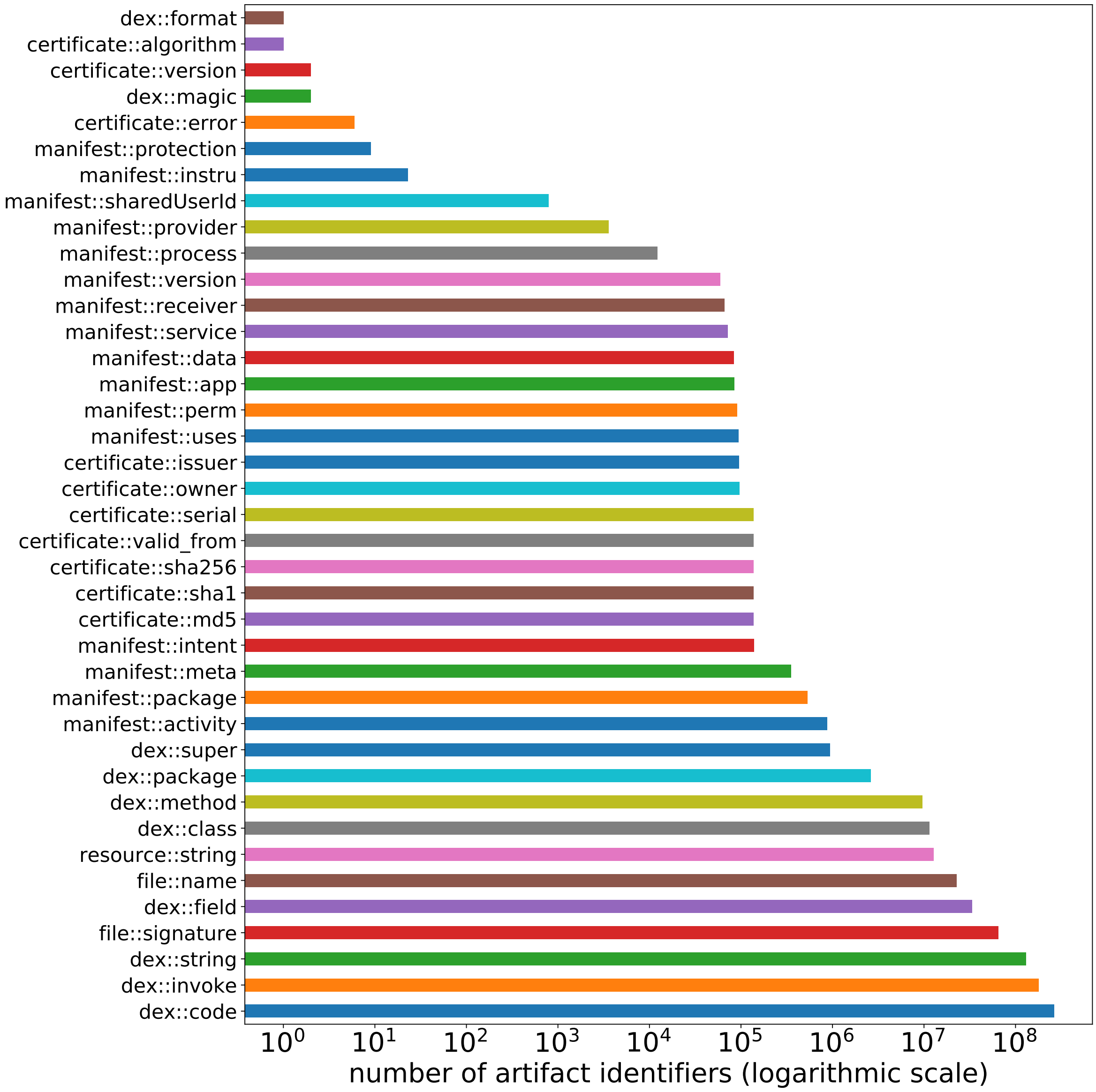


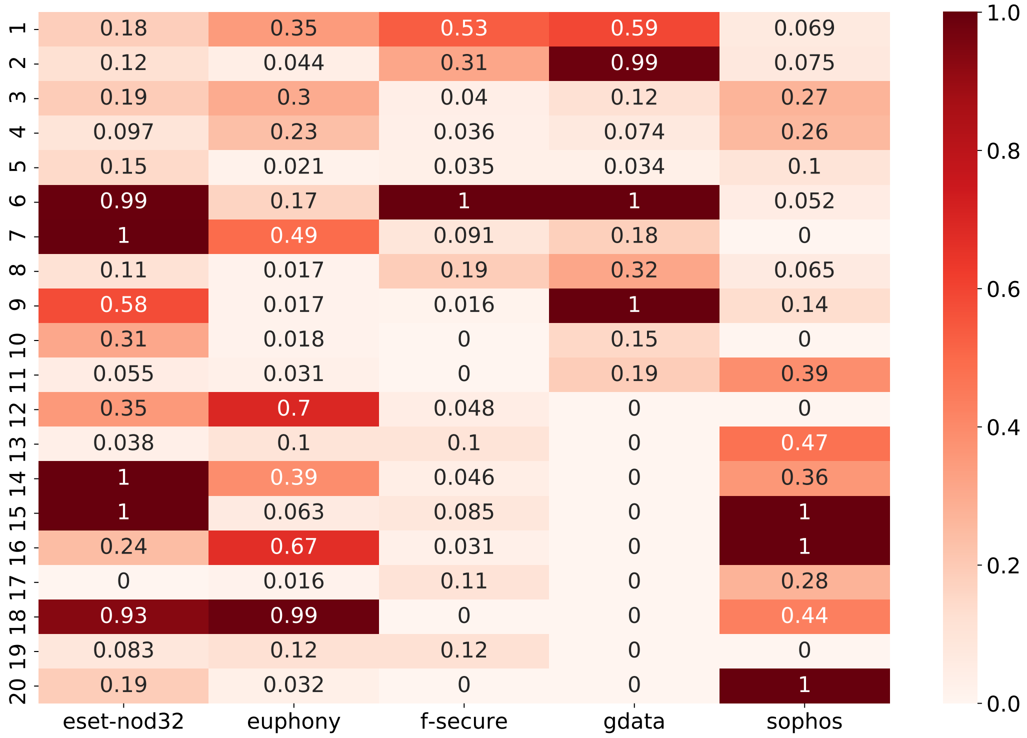
Figure 6.8.: Distribution of artifact identifiers by category

Certificate information (starting with certificate) : SHA1, MD5 and SHA256 signatures (*sha1, md5, sha256 respectively*), beginning of validity period (*valid_from*), serial number (*serial*), version number (*version*), certificate owner (*owner*), certificate issuer (*issuer*), signature algorithm name (*signature_algorithm_name*), certificate errors (*keytool_error*).

6.3.2. Performances

6.3.2.1. Characterization

To evaluate the capacity of AP-GRAPH in characterizing Android malware, we consider each antivirus and family introduced in the previous section. For each pair of antivirus and family, we report the artifact which is the most specific of this pair. For instance, if 5 artifacts a_1, a_2, a_3, a_4, a_5 are discriminative of the family t with an occurrence of 100, 200, 300, 400, 500 respectively, a_5 is the most specific artifact as it appears in more applications for this family (500 times) than the other artifacts. We then normalize the number of occurrences by dividing this value with the total number of applications associated with the pair of antivirus and family.



	eset-nod32	euphony	f-secure	gdata	sophos
1	0.18	0.35	0.53	0.59	0.069
2	0.12	0.044	0.31	0.99	0.075
3	0.19	0.3	0.04	0.12	0.27
4	0.097	0.23	0.036	0.074	0.26
5	0.15	0.021	0.035	0.034	0.1
6	0.99	0.17	1	1	0.052
7	1	0.49	0.091	0.18	0
8	0.11	0.017	0.19	0.32	0.065
9	0.58	0.017	0.016	1	0.14
10	0.31	0.018	0	0.15	0
11	0.055	0.031	0	0.19	0.39
12	0.35	0.7	0.048	0	0
13	0.038	0.1	0.1	0	0.47
14	1	0.39	0.046	0	0.36
15	1	0.063	0.085	0	1
16	0.24	0.67	0.031	0	1
17	0	0.016	0.11	0	0.28
18	0.93	0.99	0	0	0.44
19	0.083	0.12	0.12	0	0
20	0.19	0.032	0	0	1

Figure 6.9.: Maximum proportion of malware identified by AP-GRAPH per antivirus (columns) and families (rows)

We can see the results of our characterization analysis in Figure 6.9. The values range from 0 (no artifacts were found for the antivirus and family) to 1 (one artifact is present in all samples for this antivirus and family). For each antivirus (columns), the families (rows) are ordered from the larger (top) to the smaller family (bottom). As this order is different for each antivirus, we replaced the family names by an ordinal value from 1 to 20.

We observe that AP-GRAPH was able to find some key artifacts that are discriminative of whole families. This is the case for $t_6, t_7, t_{14}, t_{15}, t_{18}$ of *eset-nod32*, t_{18} of *euphony*, t_6 of *f-secure*,t_2, t_6, t_9 of *gdata*, t_{15}, t_{16}, t_{20} of *sophos*. These results show that AP-GRAPH can find

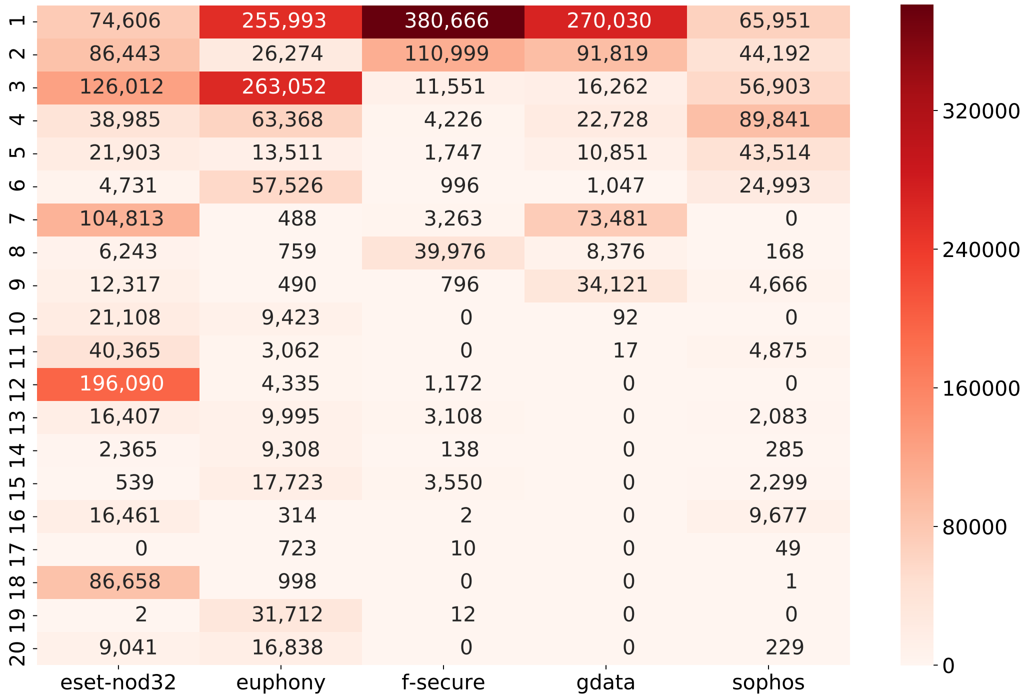


Figure 6.10.: Number of characteristic artifacts discovered by AP-GRAPH per antivirus (columns) and families (rows)

at least an artifact present in and only in every malicious application of these combinations of antivirus and family. To complement this analysis, we also display on Figure 6.10 the total number of characteristic artifacts identified by AP-GRAPH. This figure shows that out of 100 antivirus and family combinations, 74 combinations have more than 100 characteristic artifacts, 61 combinations have more than 1,000 characteristic artifacts and 39 have more than 10,000 characteristic artifacts.

We also observed from Figure 6.9 and Figure 6.10 that some antivirus and families are not fully characterized by the artifacts uncovered by AP-GRAPH. Multiple reasons could explain this result. First, antivirus labels may be associated with some hidden artifacts that are not included in this study (e.g., runtime values). Second, the antivirus labels could contain too much noise or provide information that are not granular enough to find common denominators between them. Finally, some malware families could be very generic and include a broad set of artifacts present in multiple families. We discuss the opportunity of improving our characterization scheme in the Discussion section.

6.3.2.2. Feature processing

Selecting relevant features is one of the most challenging aspects of building detection systems based on machine learning. In this regard, AP-GRAPH can assist practitioners by eliminating artifacts that do not relate to the malware families that these systems must classify. To evaluate

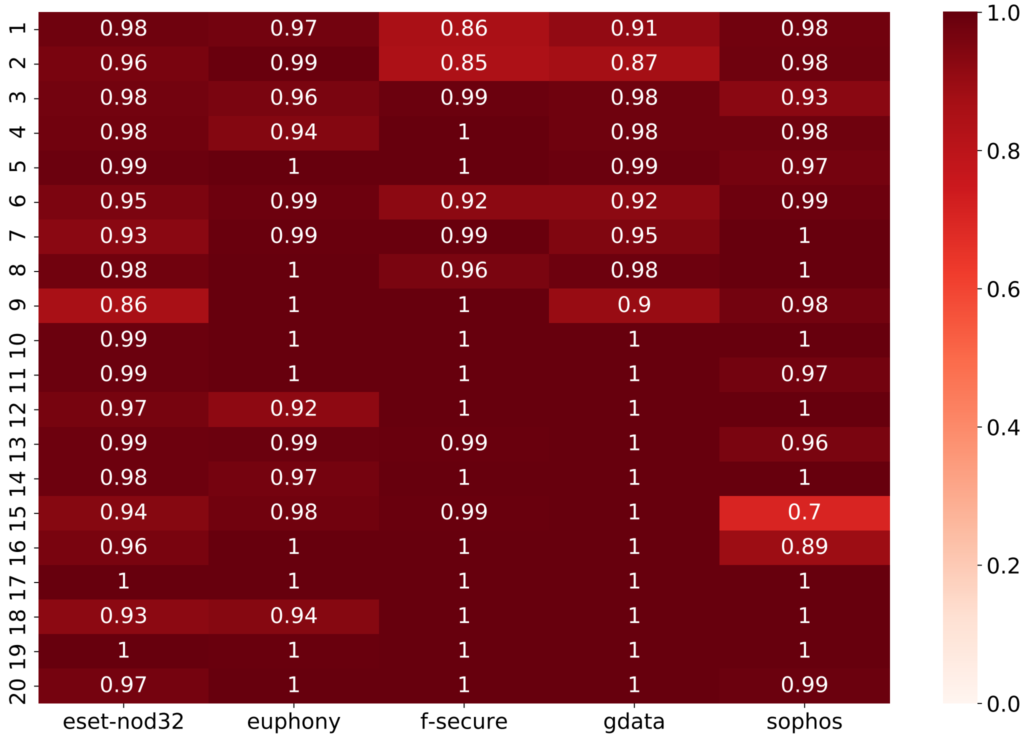


Figure 6.11.: Proportion of artifacts dropped by AP-GRAPH for each antivirus (columns) and families (rows)

the capacity of AP-GRAPH in filtering these features, we compute for each antivirus and malware families the proportion of artifacts that were dropped by our system as they were found not to be discriminative of any malware family.

We can see in Figure 6.11 the result of our feature processing. For each cell, a value close to 1.0 means that the number of dropped artifacts is closer to the total number of artifacts associated with the antivirus and family combinations. We notice that the minimum value on this figure is 0.7 for the 15th most important family of the Sophos antivirus. Thus, AP-GRAPH can remove a vast majority of artifacts that are not specific to a particular family. This ability further complements the capacity of AP-GRAPH in characterizing malware families by revealing the artifacts which are NOT associated with a particular behavior.

6.3.3. Case studies

In this section, we consider the particular case of the family *adwo* and the antivirus *gdata*. This family contains 44,828 applications and corresponds to the result of the second row and fourth column in Figure 6.9. The next parts will present the artifacts selected by AP-GRAPH for this family and the reverse analysis of a single application taken at random from Androzoo dataset.

Table 6.2.: Most specific artifacts identified by AP-GRAPH for the family *adwo* of antivirus *gdata*

Location	Type	Name	found
manifest	activity	adwoadbrowseractivity	44,162
dex	invoke	com/adwo/adsdk/i-><init>	33,889
dex	invoke	com/adwo/adsdk/h-><init>	33,078
dex	class	FSAd	32,551
dex	invoke	com/adwo/adsdk/AdwoSplashAdActivity->requestWindowFeature	32,541
dex	invoke	com/adwo/adsdk/AdwoSplashAdActivity->getWindow	32,541
dex	string	lcom/adwo/adsdk/fsad;	32,536
dex	code	#583b5	32,536
dex	invoke	com/adwo/adsdk/FSAd-><init>	32,536
dex	string	http://r2.adwo.com/adfs	32,518
dex	string	vlijlzzz	32,505
dex	code	#1b25f	32,485
dex	invoke	com/adwo/adsdk/V-><init>	32,268
dex	code	#de7d0	32,103
dex	code	#baf7b	32,103
dex	string	malformed click url.will try to follow anyway.	32,093
dex	invoke	com/adwo/adsdk/AdwoAdBrowserActivity->a	32,090
dex	string	fsad.htmlcontent	32,090
dex	code	#e5b18	32,088
dex	invoke	com/adwo/adsdk/AdwoSplashAdActivity->a	32,023

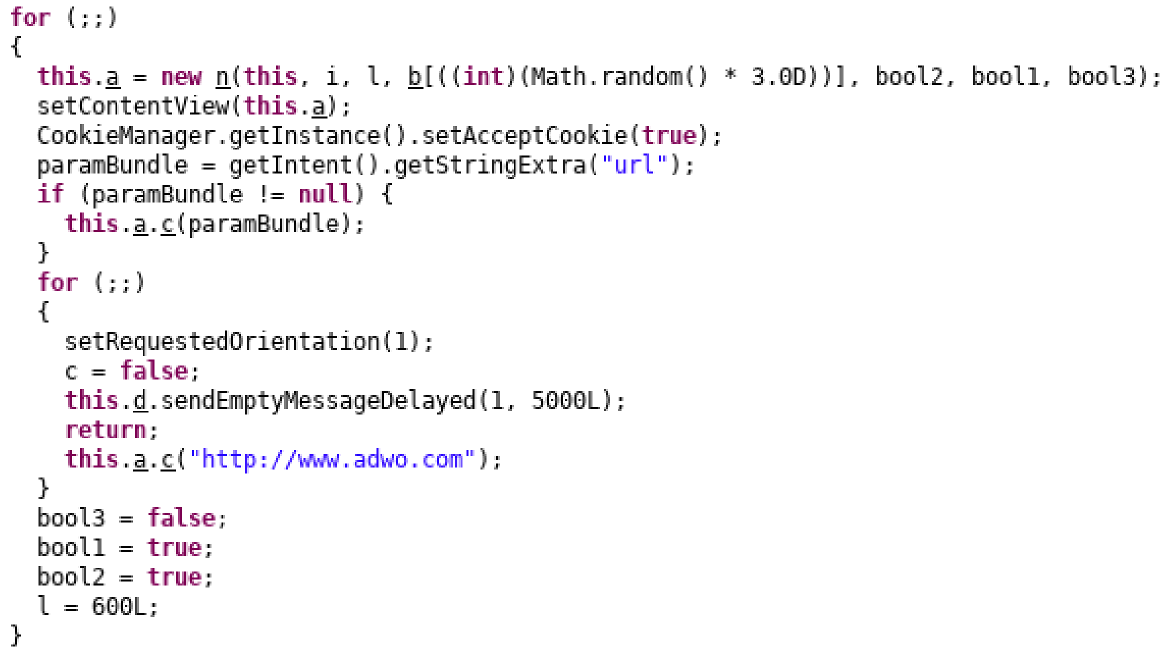
6.3.3.1. Distinctive artifacts

The artifacts identified by AP-GRAPH are listed on Table 6.2. Each row contains the location, type, name, and the number of times this artifact was found in the family. As an example, the first row shows that the activity AdwoAdBrowserActivity located in the Manifest file was found in 44,162 applications out of the 44,828 applications of the family (98%). The fourth row corresponds to the class FSAd present in the DEX file and included in 32,551 applications of the family (72%).

We see from this table that most entries are related to the family, as the artifacts contain its name. This is the case for the invoke calls that start with com/adwo and the activity AdwoAdBrowser. We can also notice a URL encoded as a string that contains the domain name r2.adwo.com. These artifacts are good indicators that AP-GRAPH can retrieve components which are related to the family under investigation. However, while some artifacts can be explained from their identifiers, others are more obscure and require some manual analysis. We explore these elements uncovered by AP-GRAPH in the next part of this section.

6.3.3.2. Reverse engineering

To demonstrate the assistance that AP-GRAPH provides in reversing malicious applications, we selected a single application at random from our set and analyzed the content of its artifacts. As a starting point, we considered every application associated with the list of artifacts showed in Table 6.2, which corresponds to the antivirus gdata and the family adwo. The SHA256 signature of the application is 000A0B1022EC485473DFAACA433F9548911BC7089B6A3 C7B47F9EC5541005CA1.



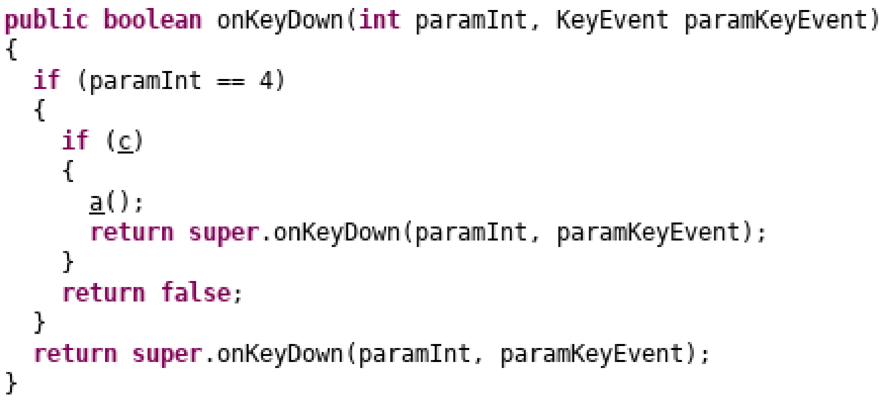
```
for (;;)
{
  this.a = new n(this, i, l, b[((int)(Math.random() * 3.0D))], bool2, bool1, bool3);
  setContentView(this.a);
  CookieManager.getInstance().setAcceptCookie(true);
  paramBundle = getIntent().getStringExtra("url");
  if (paramBundle != null) {
    this.a.c(paramBundle);
  }
  for (;;)
  {
    setRequestedOrientation(1);
    c = false;
    this.d.sendEmptyMessageDelayed(1, 5000L);
    return;
    this.a.c("http://www.adwo.com");
  }
  bool3 = false;
  bool1 = true;
  bool2 = true;
  l = 600L;
}
```

Figure 6.12.: The application contacts the primary server to download the ads

In Figure 6.12, we analyzed the content of the *onCreate* method of the AdwoAdBrowserActivity, as is it the first artifact reported by AP-GRAPH. We notice that this method constructs an object of class n and contacts the adware server at the URL: http://www.adwo.com.

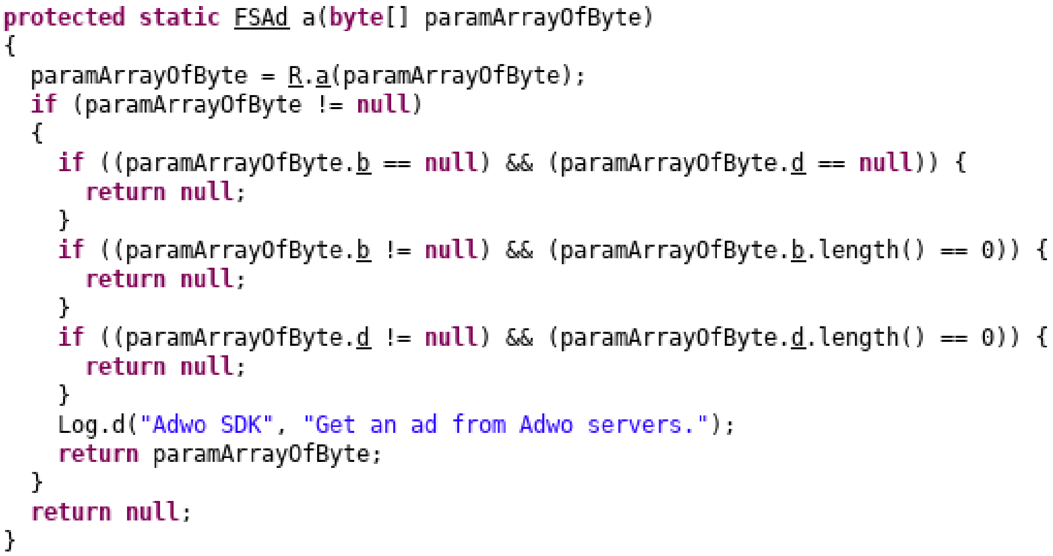
The method that triggers the ads on the screen can be found in class AdwoSplashAdActivity, another artifact identified by AP-GRAPH in Table 6.2. We show the content of this method in Figure 6.13. It corresponds to the *onKeyDown* method that was identified by AP-GRAPH under the code identifier BAF7BE622BB53166EDACE516FFA985ED2E687BC6AB295368 69D1EB709573C9A6 (or #baf7b in Table 6.2). We do not know what the variable c controls in this context, but its value is responsible for triggering the advertisement associated with this activity.

We also analyzed the class FSAd, as it is the fourth artifacts identified by AP-GRAPH in Table 6.2. The content on its constructor, displayed in Figure 6.14. It shows the construction of the advertisement from an array of bytes (we deduce this behavior from the log entry created



```
public boolean onKeyDown(int paramInt, KeyEvent paramKeyEvent)
{
  if (paramInt == 4)
  {
    if (c)
    {
      a();
      return super.onKeyDown(paramInt, paramKeyEvent);
    }
    return false;
  }
  return super.onKeyDown(paramInt, paramKeyEvent);
}
```

Figure 6.13.: The application setups the event listener to trigger the ads



```
protected static FSAd a(byte[] paramArrayOfByte)
{
  paramArrayOfByte = R.a(paramArrayOfByte);
  if (paramArrayOfByte != null)
  {
    if ((paramArrayOfByte.b == null) && (paramArrayOfByte.d == null)) {
      return null;
    }
    if ((paramArrayOfByte.b != null) && (paramArrayOfByte.b.length() == 0)) {
      return null;
    }
    if ((paramArrayOfByte.d != null) && (paramArrayOfByte.d.length() == 0)) {
      return null;
    }
    Log.d("Adwo SDK", "Get an ad from Adwo servers.");
    return paramArrayOfByte;
  }
  return null;
}
```

Figure 6.14.: The application constructs the ads from an array of bytes

inside the constructor). This method could be monitored to retrieve the content of the ads display to the user.

From our analysis of the application, we see that AP-GRAPH was able to characterize:

- the class responsible for contacting the adware server (AdwoAdBrowserActivity)
- the method responsible for triggering the ads (*onKeyDown*)
- the class responsible for handling the content of the add (FSAd)

Given these information, an analyst can quickly prioritize the artifacts that are specific to a group of applications and explore their content at a faster pace.

6.4. Evolution of malware families over time

As a first attempt to monitor the deployment of artifacts by malware authors over time, this section presents a short analysis of the evolution of three pairs of antivirus and malware family. The goal of this section is to illustrate how artifacts proposed by AP-GRAPH can be used to spot anomalies in the development of popular malware families and lead to further investigations.

Each figure contains information about 60 prominent artifacts identified by AP-GRAPH within a given malware family. We compare their distribution with the total number of Android applications classified for the same malware family in our dataset.

6.4.1. ESET NOD32 - Igexin

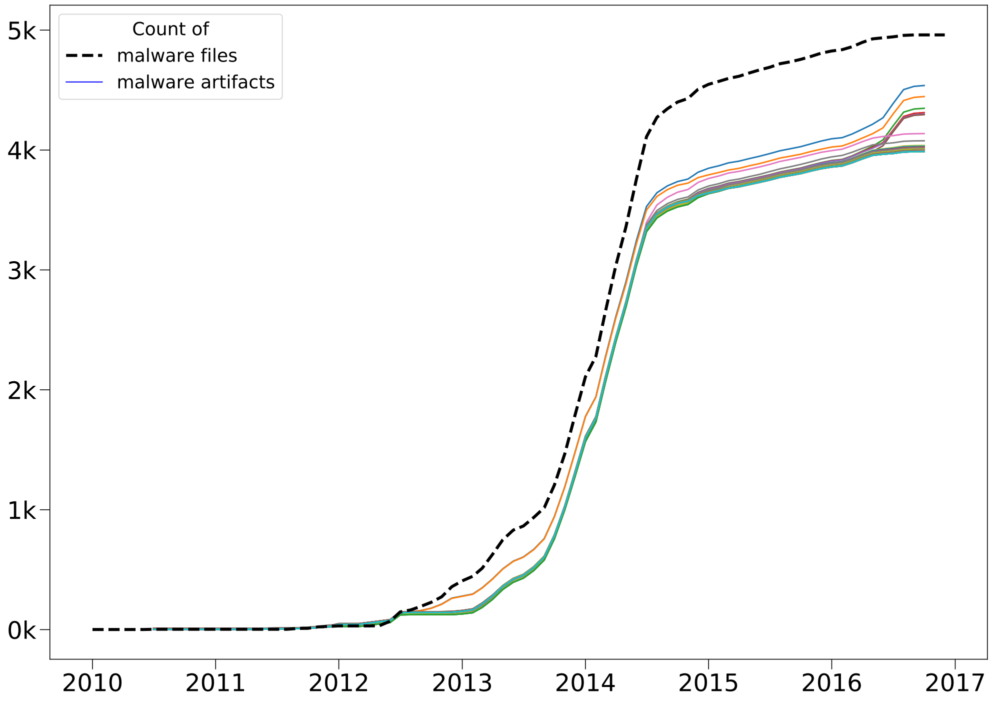


Figure 6.15.: Evolution of 60 artifacts identified by AP-GRAPH compared to the total number of malware files associated with the family ESET NOD32 - Igexin

Figure 6.15 represents an evolution of malware artifacts that coincides with the release of new malware samples associated with the family. We can see from 2010 to 2014 that the number of artifacts follows the number of malware files, even when the number of files increases considerably in 2013. However, a gap starts to appear after 2014 as the number of malware files continues to grow while the number of artifacts associated with the family stagnates.

Malicious applications found in this gap are interesting for a human analyst, as their presence is not tracked by the indicators currently revealed by AP-GRAPH. This difference could explain a drop of performance in machine learning algorithms or the presence of a new variant derived from the previous version of Igexin. We can further note that the number of artifacts starts to catch up by the end of 2017 for an unknown reason.

6.4.2. EUPHONY - AppsGeyser

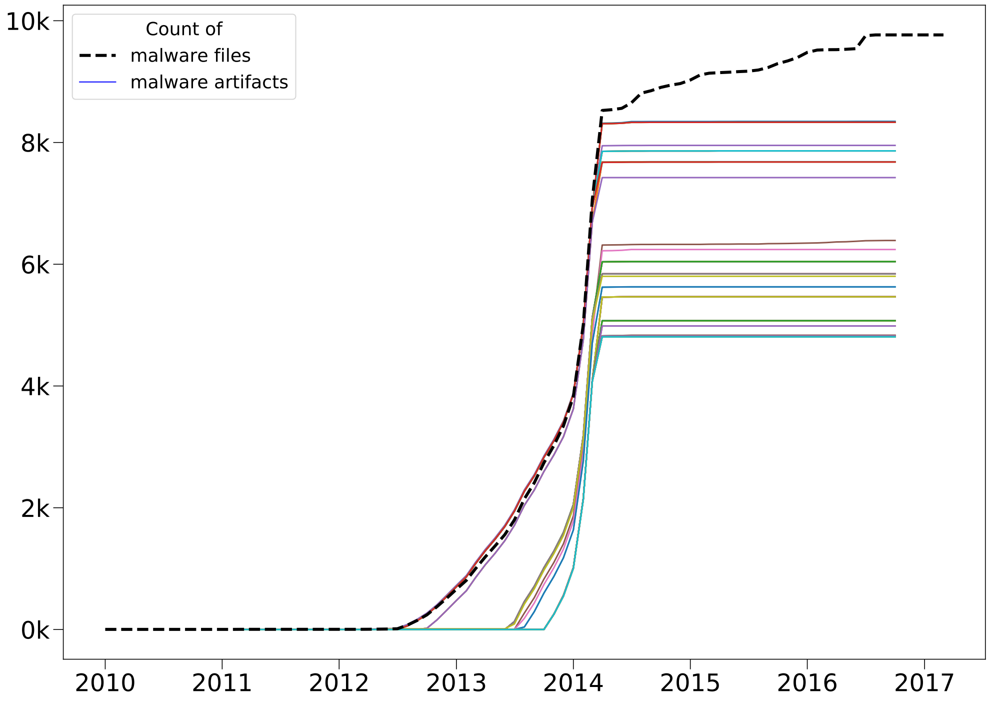


Figure 6.16.: Evolution of 60 artifacts identified by AP-GRAPH compared to the total number of malware files associated with the family EUPHONY - AppsGeyser

Figure 6.15 displays a more difficult use cases of malware evolution. On the one hand, we observe that malware artifacts associated with this family are scattered around the y-axis, indicating a difference in the composition of the malware in this set. On the other hand, we notice that artifact lines start to become flat in 2014 while the number of malware samples associated with the malware family increases.

This figure reveals either that the malware family used a completely different set of artifacts after 2014 or that artifacts were obfuscated to circumvent defense mechanisms. Such a scenario could force the investigation of new artifacts to track the malware family during its transformation period.

6.4.3. G DATA - SMSpay

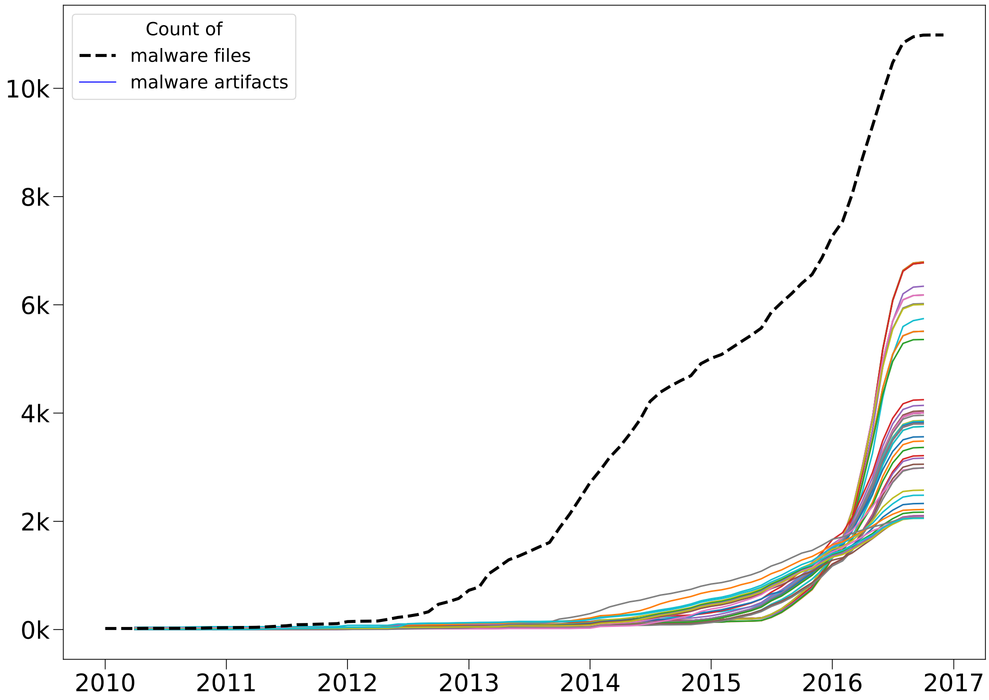


Figure 6.17.: Evolution of 60 artifacts identified by AP-GRAPH compared to the total number of malware files associated with the family G DATA - SMSpay

Figure 6.17 shows a different use case as the artifacts identified by AP-GRAPH were found after 2016, long before the rise of the malware family. Thus, we observe a period of 3 years during which no specific artifacts could be found to track the evolution of the family accurately.

The most plausible explanation is that AP-GRAPH was able to catch a new variant after 2016, but not beforehand. Otherwise, it could indicate that malware of this family were more generic before 2016, and that a recent change introduced specific artifacts that could be tracked with AP-GRAPH.

6.5. Challenges of malware classification

6.5.1. Obfuscation and variations

We noted in our study that a high proportion of malware is obfuscated by a simple scheme, including class or method renaming. Similar to other static analysis approaches, tracking the link between the original and the obfuscated content remains a challenging task. On the one hand, obfuscation techniques can make the indexing of artifacts more complicated. On the other hand, these techniques can lead to an explosion of features and introduce a wide variety of unrelated artifacts. To avoid these constraints in the short term, our study includes a diverse set of artifacts related to the same objects. For instance, a method is referenced both by its name and the hashing of its code instruction. Similarly, external files are indexed by their name and also by a sha256 signature based on their content.

Still, the mitigation techniques we adopted in our analysis cannot wholly prevent the impact of obfuscation. To handle this problem in the long term, new approaches must be designed to find artifacts whose original structure are related. One of the main benefits of AP-GRAPH in this context is its independence to a predefined type of analysis. Dynamic or other analysis can be used as an input to AP-GRAPH, as long as these techniques produce identifiers that can be indexed in our database. Thus, we think AP-GRAPH can be a viable technique to verify that newer analysis techniques can break the obfuscation scheme of malware.

6.5.2. Noisy antivirus classifications

In the evaluation section, we observed that AP-GRAPH could extract discriminative artifacts for some families, while other families do not yield artifacts specific to their whole population. Multiple reasons could explain these results:

1. the artifacts adopted by antivirus vendors are different from the one we used in our evaluation
2. obfuscation schemes are deployed at large scale by malware authors to bypass the type of analysis we applied in our evaluation
3. the malware families proposed by antivirus vendors do not reflect the malicious artifacts included in Android malware

For the first and second scenario, finding the right combination of artifacts would be sufficient to solve the detection issue. Indeed, if antivirus vendors rely mostly on dynamic analysis to detect Android malware, AP-GRAPH could consume these artifacts and verify that they correspond to a known malware family. The best approach, in this case, would be to test AP-GRAPH with other features and classifiers until a good match is found between the two sets.

The same cannot be said however for the third scenario. As there is no reference information to assess the results of our evaluation, we cannot guarantee that there is a link to be found between artifacts and the malware families reported by antivirus vendors. This lack of transparency has an impact on our research community, as we cannot verify that the input we feed to our algorithm is grounded in something concrete. If this is the case, our community must continue its work on creating better and more transparent malware ground truth.

6.5.3. Going from correlation to causation

Establishing the correlation between malware applications and their artifacts grants us the opportunity to explore what is the cause of their malicious behaviors. Future works could investigate the artifacts we uncovered to find if they are the root cause or at least trigger some malicious activity. Our approach also enables the exploration of malware over time to visualize the correlation between malicious applications and their artifacts. Searching for trends could provide a better picture of the current malware landscape and guide the deployment cycle of more accurate detection models.

Part III.

Summary and future research directions

Chapter 7.

Conclusion

This chapter proposes a conclusion on the creation of malware ground truth.

The first section outlines our contributions to create better malware ground truth.

The second section suggests research directions to further improve malware datasets.

Table of Contents

7.1. Summary

7.1.1. Definition of Android malware

As explained in Chapter 1, security practitioners must have a clear and unambiguous definition of Android malware to detect malicious applications before they impact Android users [SP10, RDG$^+$12]. While previous research groups worked with a partial [ASH$^+$14] or obsolete [ZJ12] definition of malware to support their approach, the current state of the Android security ecosystem[Goo18] imposes more transparency on the representativeness of malware samples [CSD$^+$17] and the results of machine learning experiments [ABJ$^+$16].

In Chapter 4, we proposed to evaluate the property of popular malware sets to observe their structure from a high-level perspective with STASE. Moreover, we formulated some recommendations about the desirable properties of malware ground truth to avoid the introduction of biases in experimental settings. As malware datasets are the primary source of what defines Android malware, our framework provides the railroads to prevent drifts in malware definitions across machine learning experiments.

In Chapter 5, we built a solution named EUPHONY to retrieve valuable tokens from antivirus reports and better qualify Android malware datasets in the large. Since the security industry is one of the only actors with enough human resources to systematically analyze new malware breeds, the extraction and the unification of antivirus results is an essential step towards the exploration similarities between malicious samples. EUPHONY can help practitioners in this regard by parsing the results of antivirus reports to eliminate naming confusion and propose a single definition through a majority voting scheme.

In Chapter 6, we investigated the relation between malicious artifacts and malware families with AP-GRAPH to uncover suspicious elements contained in Android applications. We based our approach on previous studies that showed [AJB$^+$14, LLB$^+$17] that the expression of malicious behaviors is supported by the presence of artifacts controlled by malware authors. AP-GRAPH can assist the description of malware families and Android malware in general by retrieving a list of artifacts related directly or indirectly to malicious behaviors found in Android applications.

Despite the solutions we proposed, our contributions do not provide a definitive definition for Android malware. On the one hand, malware families suggested by EUPHONY depend on the quality of antivirus results. Thus, the quality of the proposed names has an upper limit based on a black box labeling system. On the other hand, the artifacts retrieved by AP-GRAPH are analyzed from the output of antivirus systems and share the same limitation than EUPHONY. While our work attempted to address short term needs of the security community, we think that the quest for better malware definitions remains. This challenge might be addressed with the creation of white box systems to provide alternative results based on transparent approaches.

7.1.2. Automation of security decisions

In Chapter 1, we pointed out that both the lack of security experts [ICS18] and the use of automation techniques by malware authors [AT19] threaten the current balance between security defenders and attackers. To address these shortcomings, the security community must rely more and more on automated solutions on its own to keep pace with the proliferation of malware. With recent advancements in machine learning based systems, our community might improve our ability to prevent the surge of malicious applications that target Android markets. To support these algorithms, we reviewed several research contributions in this dissertation.

With STASE in Chapter 4, we proposed a set of metrics that can be integrated into machine learning pipelines to vet the properties of large malware datasets automatically. For instance, we saw in this chapter that different experimental settings could influence the distribution of output classes adopted to train statistical models. In place of careful manual reviews performed by security practitioners, STASE metrics can be used to track down biases and improve the confidence in machine learning based approaches.

With EUPHONY in Chapter 5, we created a fully automated solution to parse antivirus labels and suggest meaningful clusters of names based on the co-occurrence of names in malware reports. Thanks to the knowledge database integrated into our solution, EUPHONY can remember past associations and improve its suggestion over time as the research community discovers new malware names. Compared to existing solutions [SRKC16], EUPHONY can bootstrap its learning process without an extensive list of generic tokens or malware family names.

With AP-GRAPH in Chapter 6, we implemented a large scale data mining solution to analyze artifacts associated with popular malware families automatically. Our experiments showed that AP-GRAPH was able to retrieve a broad set of artifacts related to malicious behaviors found in Android malware. Moreover, the indexing and querying scheme of AP-GRAPH can be applied to provide a higher level analysis framework over common malware datasets to further automate security decisions.

While parts of our protection against Android malware can be automated, our contributions merely focused on the creation of better malware ground truth. Indeed, automated decision algorithms are always at risk at proposing an output that does not reflect the reality in the field if they are not adequately vetted or if their input is not representative of the population [SP10, RDG$^+$12, ABJ$^+$16]. We expect that quality metrics such as STASE and automatic tagging systems like EUPHONY and AP-GRAPH can assist practitioners in the construction of more sophisticated security solutions.

7.1.3. Progression of human comprehension

We recognized in Chapter 1 that Android malware fall under the curse of dimensionality [Bel13] as the sheer size of information contained in a standard application is too difficult to apprehend for human experts. Even if automated solutions can handle security decisions at large scale, we postulate that human analysts must comprehend malware more easily to craft more creative approaches against Android malware. The techniques we proposed take this point into consideration to assist the tasks of security analysts.

The metrics proposed by STASE in Chapter 4 were designed to be comprehensible by human operators and provide an overview of malware ground truths. STASE metrics proposed a human-readable interpretation of malware datasets, as the results we report are size independent and with a clear explanation for the extreme and intermediate values. Thus, the solution we propose can be embedded in a dashboard system or a monitoring solution to help experts navigate the complexity of their malware landscape.

The output produced by EUPHONY in Chapter 5 can be leveraged by a security analyst to comprehend the structure of malware labels and explore the relationship between security decisions. Since EUPHONY relies on classical search algorithms and graph data structures, the analysis steps can be decomposed to track down the advancement progress and report intermediate results. Moreover, the knowledge base instantiated by EUPHONY can be audited by external experts to ensure that the decisions we suggest are in line with the expectations of the analysts.

The artifacts proposed by AP-GRAPH in Chapter 6 can served security experts on two fronts. On the one hand, AP-GRAPH artifacts describe key features associated with malware families to get an initial idea of the behavior of a malware group. On the other hand, AP-GRAPH enumerates potential entry points for reverse engineers to start their analysis process based on malware components that were not seen in other malware families.

However, Android malware are probably one of the most complex pieces of software handled by humans analysts. Several limitations, such as our reliance on noisy labeling systems and the use of obfuscation by malware authors might impact the performance of the approaches we proposed. To overcome these shortcomings in the short term, we designed our solutions to be transparent for human operators starting from the processing of raw data to the delivery of the final results.

7.2. Future research directions

7.2.1. Malware forecast

In Chapter 6, we explored the relationship between Android artifacts and malware families with AP-GRAPH to extract the characteristics of popular malware variants. However, as malware authors continue to develop new features and introduce new malware breeds, it is crucial for our community to keep track of these changes and react efficiently according to the current malware landscape. On the one hand, a tracking solution would allow security analysts to adapt our security infrastructure based on the most recent trends in malware development.

On the other hand, the monitoring of artifacts would allow a proportionate response to malware threats, allowing the allocation of more resources when the complexity or the number of variants arise.

Our time analysis of malware artifacts is a small step toward a global indicator that could serve as a malware forecast system. Instead of focusing on the top malware variants, the artifacts uncovered by AP-GRAPH could be applied to the creation of new metrics that track changes in the composition of Android malware. For instance, the absence of a known artifact inside a malware family could indicate that the artifacts were updated or replaced by a new one. A surge of some malicious artifacts commonly associated with a malware family could also indicate either the resurgence of a known malware family or deployment of a brand new variant. In both cases, adding the dimension of time into the distribution of malicious artifacts would be a valuable asset to support better and more efficient security responses.

7.2.2. Apprenticeship learning

Through this dissertation, we took the postulate that antivirus labels are a trustworthy source of information that can be mined and analyzed to leverage the efforts of industrial security actors. However, our analysis and experience revealed that antivirus reports could return noisy results that do not fully support the confidence we expect in state of the art security solutions. While the information we gathered with EUPHONY and AP-GRAPH can be viewed as a partial reference of the decision rules built in antivirus products, our approaches do not provide a complete alternative to black box classifiers yet.

In the long term, the security community might benefit from building its own decision rules and share them publicly to create even better malware ground truths. As manual inspections require too much effort for a small community, a new type of artificial intelligence techniques must be explored to augment the performance of human analysts [VAK^{+}16]. A possible solution might be the use of apprenticeship learning [AN04], a kind of inverse reinforcement learning algorithms that derives a reward function from the decision of human experts. For instance, a security analyst could create a script that extracts suspicious artifacts within Android applications and concludes about the nature of an application based on a weighted formula. Similarly, the analyst actions could be learned and applied by an algorithm that reproduces the expert steps while a machine learning model could emulate the expert formula. Moreover, reinforcement learning algorithms would provide an efficient framework of analysis to avoid the systematic inspection of the numerous parts that composed an Android application. The main benefit of an apprenticeship learning based approach in this context would be to support the creative work of security analysts while automating their most tedious tasks as we face a shortage of security experts [ICS18].

7.2.3. Learning from machine learning

We envision that the field of artificial intelligence will remain more art than science as long as experts in the domain are not able to theorize the performance of machine learning algorithms [TKP$^+$17, Lip18]. This problem will also impact the detection of Android malware, as our community operates in an adversarial setting where a single model can be potentially exploited as a single point of failure [SP10, RDG$^+$12, PMG$^+$16]. To improve the robustness of our solutions, the security community must engage in the reproduction and the comparison of machine learning based models to understand their weaknesses but also to compensate for their inherent limitations.

A possible solution might be the creation of meta machine learning systems, able to recommend the best set of models for a given operational setting. For example, malware created more than one year ago could be better handled by an older model while new variants would require more advanced models instead. Similarly, some malware families may be better detected with statistical models created from dynamic features while other families would be handled more efficiently by a simpler model. Learning from the output of existing machine learning algorithms could improve the diffusion of research approaches outside of the lab and support the development of a new generation of hierarchical machine learning pipelines.

Bibliography

[AB18] A. Adadi and M. Berrada. Peeking Inside the Black-Box: A Survey on Explainable Artificial Intelligence (XAI). *IEEE Access*, 6:52138–52160, 2018. doi:10.1109/ACCESS.2018.2870052.

[ABJ+16] Kevin Allix, Tegawendé F. Bissyandé, Quentin Jérome, Jacques Klein, Radu State, and Yves Le Traon. Empirical assessment of machine learning-based malware detectors for Android: Measuring the gap between in-the-lab and in-the-wild validation scenarios. *Empirical Software Engineering*, 21(1):183–211, February 2016. URL: http://link.springer.com/10.1007/s10664-014-9352-6, doi:10.1007/s10664-014-9352-6.

[ABKT16] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471, May 2016. doi:10.1109/MSR.2016.056.

[ADY13] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, volume 127, pages 86–103. Springer International Publishing, Cham, 2013. URL: http://link.springer.com/10.1007/978-3-319-04283-1_6, doi:10.1007/978-3-319-04283-1_6.

[AJB+14] Kevin Allix, Quentin Jerome, Tegawende F. Bissyande, Jacques Klein, Radu State, and Yves Le Traon. A Forensic Analysis of Android Malware – How is Malware Written and How it Could Be Detected? pages 384–393. IEEE, July 2014. URL: http://ieeexplore.ieee.org/document/6899240/, doi:10.1109/COMPSAC.2014.61.

[AK14] Kevin Allix and Jacques Klein. Machine Learning-Based Malware Detection for Android Applications: History Matters! page 17, May 2014.

[AN04] Pieter Abbeel and Andrew Y. Ng. Apprenticeship learning via inverse reinforcement learning. page 1. ACM Press, 2004. URL: http://portal.acm.org/citation.cfm?doid=1015330.1015430, doi:10.1145/1015330.1015430.

[And19a] Android. Android Content License, January 2019. URL: https://source.android.com/setup/start/licenses.

[And19b] Android. Android History, January 2019. URL: https://www.android.com/history/.

[And19c] Android. Developer Policy Center, January 2019. URL: https://play.google.com/about/developer-content-policy/.

[AQR$^+$16] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. DroidNative: Semantic-Based Detection of Android Native Code Malware. page 18, February 2016.

[ASH$^+$14] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. Internet Society, 2014. URL: https://www.ndss-symposium.org/ndss2014/programme/drebin-effective-and-explainable-detection-android-malware-your-pocket/, `doi:10.14722/ndss.2014.23247`.

[AT19] AV-TEST. Malware Statistics, 2019. URL: https://www.av-test.org/en/statistics/malware/.

[BCH$^+$09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. page 18, 2009.

[Bel13] Richard Bellman. *Dynamic programming*. Courier Corporation, 2013.

[BH08] Pierre-Marc Bureau and David Harley. A dose by any other name. *Virus Bulletin Conference*, 8:224–231, 2008.

[Bha18] Anup Bhande. What is underfitting and overfitting in machine learning and how to deal with it., May 2018. URL: https://medium.com/greyatom/what-is-underfitting-and-overfitting-in-machine-learning-and-how-to-deal-with-it-6803a989c76.

[BKLTM12] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. pages 27–38. ACM Press, 2012. URL: http://dl.acm.org/citation.cfm?doid=2259051.2259056, `doi:10.1145/2259051.2259056`.

[BKvOS10] David Barrera, H. G üne ş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. page 73. ACM Press, 2010. URL: http://portal.acm.org/citation.cfm?doid=1866307.1866317, `doi:10.1145/1866307.1866317`.

[BOA$^+$07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637, pages

178–197. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. URL: http://link.springer.com/10.1007/978-3-540-74320-0_10, doi:10.1007/978-3-540-74320-0_10.

[Bon05] Dr Vesselin Bontchev. Current Status of the CARO Malware Naming Scheme. page 29, October 2005.

[BZNT11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. page 15. ACM Press, 2011. URL: http://dl.acm.org/citation.cfm?doid=2046614.2046619, doi:10.1145/2046614.2046619.

[CGC13] Jonathan Crussell, Clint Gibler, and Hao Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134, pages 182–199. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. URL: http://link.springer.com/10.1007/978-3-642-40203-6_11, doi:10.1007/978-3-642-40203-6_11.

[CIR] CIRCL. MISP - Open Source Threat Intelligence Platform & Open Standards For Threat Information Sharing. URL: https://www.misp-project.org/.

[CL14] Mihaly Csikszentmihalyi and R Larson. *Flow and the foundations of positive psychology*. Springer, 2014.

[Cog] Cognitech. Datomic Website. URL: https://www.datomic.com/.

[CRF03] William W Cohen, Pradeep Ravikumar, and Stephen E Fienberg. A Comparison of String Metrics for Matching Names and Records. page 6, 2003.

[CRTE13] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. MAST: triage for market-scale mobile malware analysis. page 13. ACM Press, 2013. URL: http://dl.acm.org/citation.cfm?doid=2462096.2462100, doi:10.1145/2462096.2462100.

[CSD$^+$17] Julio Canto, Hispasec Sistemas, Marc Dacier, Sophia Antipolis, Engin Kirda, and Corrado Leita. Large scale malware collection: lessons learned. page 6, December 2017.

[CWL$^+$15] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. page 16, 2015.

[DBH18] F. K. Došilović, M. Brčić, and N. Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 0210–0215, May 2018. `doi:10.23919/MIPRO.2018.8400040`.

[Dev08] Android Developers. Announcing the Android 1.0 SDK, release 1, September 2008. URL: https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html.

[Dev19a] Android Developers. Application Sandbox, January 2019. URL: https://source.android.com/security/app-sandbox.

[Dev19b] Android Developers. Application Security, January 2019. URL: https://source.android.com/security/overview/app-security.

[Dev19c] Android Developers. Intents and Intent Filters, January 2019. URL: https://developer.android.com/guide/components/intents-filters.

[Dev19d] Android Developers. Security Essential Checklist, January 2019. URL: https://developer.android.com/topic/security.

[DG] Anthony Desnos and Geoffroy Gueguen. Androguard. https://github.com/androguard/androguard. URL: https://github.com/androguard/androguard.

[Dic45] Lee R. Dice. Measures of the Amount of Ecologic Association Between Species. *Ecology*, 26(3):297–302, July 1945. URL: http://doi.wiley.com/10.2307/1932409, `doi:10.2307/1932409`.

[Dic19] Oxford Dictionnary. Malware, 2019. URL: https://en.oxforddictionaries.com/definition/malware.

[DSTK+16] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. DroidScribe: Classifying Android Malware Based on Runtime Behavior. pages 252–261. IEEE, May 2016. URL: http://ieeexplore.ieee.org/document/7527777/, `doi:10.1109/SPW.2016.25`.

[DVK17] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[Ela] Elastic. Elastic Website. URL: https://www.elastic.co.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. page 38, 2011.

[FADA14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. pages 576–587. ACM Press, 2014. URL: http://dl.acm.org/citation.cfm?doid=2635868.2635869, `doi:10.1145/2635868.2635869`.

[FBR+16] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Applications. pages 377–396. IEEE, May 2016. URL: http://ieeexplore.ieee.org/document/7546513/, doi:10.1109/SP.2016.30.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. page 627. ACM Press, 2011. URL: http://dl.acm.org/citation.cfm?doid=2046707.2046779, doi:10.1145/2046707.2046779.

[FLW+17] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. DAPASA: Detecting Android Piggybacked Apps Through Sensitive Subgraph Analysis. *IEEE Transactions on Information Forensics and Security*, 12(8):1772–1785, August 2017. URL: http://ieeexplore.ieee.org/document/7887707/, doi:10.1109/TIFS.2017.2687880.

[For18] Forbes. The Cybersecurity Talent Gap Is An Industry Crisis, September 2018. URL: https://www.forbes.com/sites/forbestechcouncil/2018/08/09/the-cybersecurity-talent-gap-is-an-industry-crisis/#9c53e10a6b36.

[GAF+15] André Ricardo Abed Grégio, Vitor Monte Afonso, Dario Simões Fernandes Filho, Paulo Lício de Geus, and Mario Jino. Toward a Taxonomy of Malware Behaviors. *The Computer Journal*, 58(10):2758–2777, October 2015. URL: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/bxv047, doi:10.1093/comjnl/bxv047.

[GHM18] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Transactions on Software Engineering and Methodology*, 26(3):1–29, January 2018. URL: http://dl.acm.org/citation.cfm?doid=3177743.3162625, doi:10.1145/3162625.

[Goo18] Google. Android Security 2017 Year In Review, 2018. URL: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf.

[Goo19a] Google. Google Play, January 2019. URL: https://play.google.com/store.

[Goo19b] Google. Google Play Protect, January 2019. URL: https://www.android.com/play-protect/.

[Goo19c] Google. Permissions overview, 2019. URL: https://developer.android.com/guide/topics/permissions/overview.

[GSM+13] Ilir Gashi, Bertrand Sobesto, Stephen Mason, Vladimir Stankovic, and Michel Cukier. A study of the relationship between antivirus regressions and label changes. pages 441–450. IEEE, November 2013. URL: http://ieeexplore.ieee.org/document/6698897/, doi:10.1109/ISSRE.2013.6698897.

[Gun17] David Gunning. Explainable artificial intelligence (xai). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2017.

[GYAR13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. pages 45–54. ACM Press, 2013. URL: http://dl.acm.org/citation.cfm?doid=2517312.2517315, `doi:10.1145/2517312.2517315`.

[Har09] David Harley. The Game of the Name Malware Naming, Shape Shifters and Sympathetic Magic. In *CEET 3rd Intl. Conf. on Cybercrime Forensics Education & Training, San Diego, CA*, 2009.

[Hub19] Open Hub. Android Language Breakdown, January 2019. URL: https://www.openhub.net/p/android/analyses/latest/languages_summary.

[Hur] Médéric Hurier. *Definition of Ouroboros*.

[ICS18] ICS2. Cybersecurity skills shortage soars, nearning 3 million, October 2018. URL: https://blog.isc2.org/isc2_blog/2018/10/cybersecurity-skills-shortage-soars-nearing-3-million.html.

[JBV11] Jiyong Jang, David Brumley, and Shobha Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. page 309. ACM Press, 2011. URL: http://dl.acm.org/citation.cfm?doid=2046707.2046742, `doi:10.1145/2046707.2046742`.

[JYM+16] Jae-wook Jang, Jaesung Yun, Aziz Mohaisen, Jiyoung Woo, and Huy Kang Kim. Detecting and classifying method based on similarity matching of Android malware behavior with profile. *SpringerPlus*, 5(1), December 2016. URL: http://www.springerplus.com/content/5/1/273, `doi:10.1186/s40064-016-1861-x`.

[Kel10] Tom Kelchner. The (in)consistent naming of malcode. *Computer Fraud & Security*, 2010(2):5–7, February 2010. URL: http://linkinghub.elsevier.com/retrieve/pii/S1361372310700075, `doi:10.1016/S1361-3723(10)70007-5`.

[KLLVTT16] Nicolas Kiss, Jean-François Lalande, Mourad Leslous, and Valérie Viet Triem Tong. Kharon dataset: Android malware under a microscope. In *Learning from Authoritative Security Experiment Results*, San Jose, United States, May 2016. The USENIX Association. URL: https://hal-univ-orleans.archives-ouvertes.fr/hal-01300752.

[KTA+15] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D. Joseph, and J. D. Tygar. Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels. pages 45–56. ACM Press, 2015. URL: http://dl.acm.org/citation.cfm?doid=2808769.2808780, `doi:10.1145/2808769.2808780`.

[LBB$^+$15] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. ApkCombiner: Combining Multiple Android Apps to Support Inter-App Analysis. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455, pages 513–527. Springer International Publishing, Cham, 2015. URL: http://link.springer.com/10.1007/978-3-319-18467-8_34, doi:10.1007/978-3-319-18467-8_34.

[LBP$^+$17] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 2017.

[LDFM$^+$12] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. page 349. ACM Press, 2012. URL: http://dl.acm.org/citation.cfm?doid=2420950.2421001, doi:10.1145/2420950.2421001.

[Les18] Paige Leskin. The 21 scariest data breaches of 2018, December 2018. https://www.businessinsider.fr/us/data-hacks-breaches-biggest-of-2018-2018-12. URL: https://www.businessinsider.fr/us/data-hacks-breaches-biggest-of-2018-2018-12.

[LGH$^+$17] Li Li, Jun Gao, Mederic Hurier, Pingfan Kong, Tegawende F Bissyande, Alexandre Bartel, Jacques Klein, and Yves Le Traon. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. page 21, September 2017.

[Lip18] Zachary Chase Lipton. The Mythos of Model Interpretability. *ACM Queue*, 16:30, 2018.

[LL17] Scott M Lundberg and Su-In Lee. A Unified Approach to Interpreting Model Predictions. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4765–4774. Curran Associates, Inc., 2017. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[LLB$^+$17] Li Li, Daoyuan Li, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics and Security*, 12(6):1269–1284, June 2017. URL: http://ieeexplore.ieee.org/document/7828100/, doi:10.1109/TIFS.2017.2656460.

[LLGR10] Peng Li, Limin Liu, Debin Gao, and Michael K Reiter. On Challenges in Evaluating Malware Clustering. page 18, 2010.

[LNP15] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis. pages 422–433. IEEE, July 2015. URL: http://ieeexplore.ieee.org/document/7273650/, doi:10.1109/COMPSAC.2015.103.

[LNW+14] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. ANDRUBIS – 1,000,000 Apps Later: A View on Current Android Malware Behaviors. pages 3–17. IEEE, September 2014. URL: http://ieeexplore.ieee.org/document/7446031/, doi:10.1109/BADGERS.2014.7.

[MA14] Aziz Mohaisen and Omar Alrawi. AV-Meter: An Evaluation of Antivirus Scans and Labels. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, and Sven Dietrich, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550, pages 112–131. Springer International Publishing, Cham, 2014. URL: http://link.springer.com/10.1007/978-3-319-08509-8_7, doi:10.1007/978-3-319-08509-8_7.

[MBSZ11] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding Non-trivial Malware Naming Inconsistencies. In Sushil Jajodia and Chandan Mazumdar, editors, *Information Systems Security*, volume 7093, pages 144–159. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. URL: http://link.springer.com/10.1007/978-3-642-25560-1_10, doi:10.1007/978-3-642-25560-1_10.

[MOA+17] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. Internet Society, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/mamadroid-detecting-android-malware-building-markov-chains-behavioral-models/, doi:10.14722/ndss.2017.23353.

[MSK+19] W James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. Interpretable machine learning: definitions, methods, and applications. *arXiv preprint arXiv:1901.04592*, 2019.

[noa] VirusTotal. https://www.virustotal.com/about/. URL: https://www.virustotal.com/about/.

[NZ17] Robin Nix and Jian Zhang. Classification of Android apps and malware using deep neural networks. pages 1871–1878. IEEE, May 2017. URL: http://ieeexplore.ieee.org/document/7966078/, doi:10.1109/IJCNN.2017.7966078.

[OMJ+13] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, and Eric Bodden. Effective Inter-Component Communication Mapping in Android with

Epicc: An Essential Step Towards Holistic Security Analysis. page 16, August 2013.

[OZA15] Takashi OZAKI. Decision Boundaries for Deep Learning and other Machine Learning classifiers, June 2015. URL: https://www.kdnuggets.com/2015/06/decision-boundaries-deep-learning-machine-learning-classifiers.html.

[Par] Mila Parkour. Contagio MiniDump. https://contagiominidump.blogspot.lu/. URL: https://contagiominidump.blogspot.lu/.

[Plo] Daniel Plohmann. Malpedia. https://malpedia.caad.fkie.fraunhofer.de/. URL: https://malpedia.caad.fkie.fraunhofer.de/.

[PLP09] Darius Pfitzner, Richard Leibbrandt, and David Powers. Characterization and evaluation of similarity measures for pairs of clusterings. *Knowledge and Information Systems*, 19(3):361–394, 2009. doi:10.1007/s10115-008-0150-6.

[PMG+16] Nicolas Papernot, Patrick D. McDaniel, Ian J. Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples. *CoRR*, abs/1602.02697, 2016. URL: http://arxiv.org/abs/1602.02697.

[Pri57] R. C. Prim. Shortest Connection Networks And Some Generalizations. *Bell System Technical Journal*, 36(6):1389–1401, November 1957. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6773228, doi:10.1002/j.1538-7305.1957.tb01515.x.

[PU12] Roberto Perdisci and ManChon U. VAMO: towards a fully automated malware clustering validity analysis. page 329. ACM Press, 2012. URL: http://dl.acm.org/citation.cfm?doid=2420950.2420999, doi:10.1145/2420950.2420999.

[RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. Internet Society, 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/sites/25/2017/09/harvesting-runtime-values-android-applications-feature-anti-analysis-techniques.pdf, doi:10.14722/ndss.2016.23066.

[RCE13] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: automatic security analysis of smartphone applications. page 209. ACM Press, 2013. URL: http://dl.acm.org/citation.cfm?doid=2435349.2435379, doi:10.1145/2435349.2435379.

[RDG+12] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. pages 65–79. IEEE, May 2012. URL: http://ieeexplore.ieee.org/document/6234405/, doi:10.1109/SP.2012.14.

[RLVS] Fernando Ramírez, Francisco López, Daniel Vaca, and Antonio Sánchez. Koodous. https://koodous.com/about. URL: https://koodous.com/about.

[RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144, 2016.

[SFE$^+$13] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. page 1808. ACM Press, 2013. URL: http://dl.acm.org/citation.cfm?doid=2480362.2480701, doi:10.1145/2480362.2480701.

[SGK17] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning Important Features Through Propagating Activation Differences. *CoRR*, abs/1704.02685, 2017. URL: http://arxiv.org/abs/1704.02685.

[SKGM18] Raphael Spreitzer, Felix Kirchengast, Daniel Gruss, and Stefan Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 749–763. ACM, 2018.

[Sol] Solvusoft. How to Remove Android:AccuTrack-A. URL: https://www.solvusoft.com/en/malware/potentially-unwanted-application/android-accutrack-a/.

[Sop19] Sophos. SophosLabs 2019 - Threat Report, 2019. URL: https://www.sophos.com/en-us/medialibrary/pdfs/technical-papers/sophoslabs-2019-threat-report.pdf.

[Sou] Android Source. Android Framework. URL: https://source.android.com/security/.

[SP10] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. pages 305–316. IEEE, 2010. URL: http://ieeexplore.ieee.org/document/5504793/, doi:10.1109/SP.2010.25.

[SRKC16] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AVclass: A Tool for Massive Malware Labeling. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9854, pages 230–253. Springer International Publishing, Cham, 2016. URL: http://link.springer.com/10.1007/978-3-319-45719-2_11, doi:10.1007/978-3-319-45719-2_11.

[SSB] Fridrik Skulason, Alan Solomon, and Vesselin Bontchev. CARO. http://www.caro.org/articles/naming.html. URL: http://www.caro.org/articles/naming.html.

[SSDM16] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. MADAM: Effective and Efficient Behavior-based Android Malware Detection and Prevention. page 14, 2016.

[sta19a] statcounter. Mobile Operating System Market Share Worldwide, 2019. http://gs.statcounter.com/os-market-share/mobile/worldwide/. URL: http://gs.statcounter.com/os-market-share/mobile/worldwide/.

[Sta19b] Statista. Number of available applications in the Google Play Store from December 2009 to December 2018, 2019. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/. URL: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[STDA$^+$17] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. pages 309–320. ACM Press, 2017. URL: http://dl.acm.org/citation.cfm?doid=3029806.3029825, `doi:10.1145/3029806.3029825`.

[STS18] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned. page 18, January 2018.

[STTPLB14] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4):1104–1117, March 2014. URL: http://linkinghub.elsevier.com/retrieve/pii/S0957417413006088, `doi:10.1016/j.eswa.2013.07.106`.

[STTPLR14] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, Detection and Analysis of Malware for Smart Devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014. URL: http://ieeexplore.ieee.org/document/6657497/, `doi:10.1109/SURV.2013.101613.00077`.

[Sym19] Symantec. Android.Kuguo, 2019. URL: https://www.symantec.com/security-center/writeup/2014-040315-5215-99.

[Sys19] Upstream Systems. Secure-D uncovers pre-installed malware com.tct.weather on Alcatel Android smartphones manufactured by TCL, January 2019. URL: https://www.upstreamsystems.com/secure-d-uncovers-pre-installed-malware-alcatel-android-smartphones-manufactured-tcl/.

[Tet18] Sai Tetali. Keeping 2 billion Android devices safe with machine learning, May 2018. URL: https://security.googleblog.com/2018/05/keeping-2-billion-android-devices-safe.html.

[TFA+17] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The Evolution of Android Malware and Android Analysis Techniques. *ACM Computing Surveys*, 49(4):1–41, January 2017. URL: http://dl.acm.org/citation.cfm?doid=3022634.3017427, doi:10.1145/3017427.

[TKFC15] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. Internet Society, 2015. URL: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/copperdroid-automatic-reconstruction-android-malware-behaviors/, doi:10.14722/ndss.2015.23145.

[TKP+17] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[VAK+16] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. AI2: Training a Big Data Machine to Defend. pages 49–54. IEEE, April 2016. URL: http://ieeexplore.ieee.org/document/7502263/, doi:10.1109/BigDataSecurity-HPSC-IDS.2016.79.

[Vas18] Gowthamy Vaseekaran. Machine Learning: Supervised Learning vs Unsupervised Learning, September 2018. URL: https://medium.com/@gowthamy/machine-learning-supervised-learning-vs-unsupervised-learning-f1658e12a780.

[Ver19] Verizon. 2018 Data Breach Investigations Report, 2019. https://enterprise.verizon.com/resources/reports/dbir/. URL: https://enterprise.verizon.com/resources/reports/dbir/.

[Web19] Android Website. Android Security, January 2019. URL: https://www.android.com/enterprise/security/.

[WHA+18] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Brad Reaves, Patrick Traynor, and Sascha Fahl. A Large Scale Investigation of Obfuscation Use in Google Play. page 14, January 2018.

[WJ12] Yajin Zhou Zhi Wang and Wu Zhou Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. page 13, 2012.

[WLC15] Xiaozhou Wang, Jiwei Liu, and Xueer Chen. Microsoft Malware Classification Challenge (BIG 2015): First Place Team: Say No to Overfitting. page 7, May 2015.

[WLR+17] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep Ground Truth Analysis of Current Android Malware. page 20, January 2017.

[WMGH14] Ting Wang, Shicong Meng, Wei Gao, and Xin Hu. Rebuilding the Tower of Babel: Towards Cross-System Malware Information Sharing. pages 1239–1248. ACM Press, 2014. URL: http://dl.acm.org/citation.cfm?doid=2661829.2662086, doi:10.1145/2661829.2662086.

[WMW$^+$12] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. pages 62–69. IEEE, August 2012. URL: http://ieeexplore.ieee.org/document/6298136/, doi:10.1109/AsiaJCIS.2012.18.

[YBK13] Guanhua Yan, Nathan Brown, and Deguang Kong. Exploring Discriminatory Features for Automated Malware Classification. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 7967, pages 41–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. URL: http://link.springer.com/10.1007/978-3-642-39235-1_3, doi:10.1007/978-3-642-39235-1_3.

[YLCJ10] Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. page 95. ACM Press, 2010. URL: http://dl.acm.org/citation.cfm?doid=1835804.1835820, doi:10.1145/1835804.1835820.

[YLX16] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. DroidDetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, February 2016. URL: http://ieeexplore.ieee.org/document/7399288/, doi:10.1109/TST.2016.7399288.

[YXG$^+$14] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, volume 8712, pages 163–182. Springer International Publishing, Cham, 2014. URL: http://link.springer.com/10.1007/978-3-319-11203-9_10, doi:10.1007/978-3-319-11203-9_10.

[YY12] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. page 16, 2012.

[ZH05] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the royal statistical society: series B (statistical methodology)*, 67(2):301–320, 2005.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. pages 95–109. IEEE, May 2012. URL: http://ieeexplore.ieee.org/document/6234407/, doi:10.1109/SP.2012.16.

[ZZG$^+$13] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of "Piggybacked" mobile applications. page 185. ACM Press, 2013. URL: http://dl.acm.org/citation.cfm?doid=2435349.2435377, doi:10.1145/2435349.2435377.