POLITECNICO DI TORINO

Collegio di Ingegneria Informatica,
del Cinema e Meccatronica
Master of Science in Computer Engineering

Master Degree Thesis

Heuristics and Evolutionary Algorithms for Android Malware Signature Optimization

Supervisors:
Prof. Giovanni Squillero
Dott. Andrea Marcelli

Candidate:
Luca Mannella
Student ID: 222325

April 2018

To those who have always believed in me.

Summary

The Android ecosystem offers an open market model, where millions of applications are downloaded by users every day. While applications from the official Google Play store undergo a review process to confirm that they comply with Google policies, other third-party markets do not. Hence, a typical pattern among malware developers is to repack popular applications from Google Play by adding malicious features and distribute them to third-party app-stores, leveraging apps popularity to accelerate malware propagation.

As a consequence antivirus software struggle to keep their signature database up-to-date, and AV scanners suffer from a considerable quantity of false negatives. Indeed creating a high quality signature able to generalize to match new malware variants, while avoiding false positives detection, is a challenging task, and requires a substantial portion of human experts time. Given the industrial interest in automatically generating new Android malware signatures, recently a new tool, named YaYaGen (Yet Another YARA Rule Generator), was proposed: it exploits a greedy-like optimization algorithm to generate YARA rules, the standard pattern matching language used to write malware signatures.

This work focuses on the research of computational intelligence techniques for the automatic malware signature generation, introducing a score system that maximize the rule efficacy, finding the best agreement between signature generality and specificity. Moreover we improve the greedy-like algorithm previously proposed, with an ensemble techniques which combines heuristics and evolutionary algorithms. In particular, we adopted the Selfish Gene algorithm, an evolutionary strategy loosely inspired by the Darwinian theory of Richard Dawkins. Experimental results show that the new version of YaYaGen gains the ability to generate more accurate rules, lowering both false positives and negatives.

Finally, the proposed approach has been tested and will be soon integrated in Koodous Brain, an artificial intelligence platform developed to assist Android malware detection in the Koodous project, an open community antivirus from Hispasec Sistemas.

Acknowledgements

Le radici stanno sepolte sotto terra e a malapena si vedono, ma senza le radici un albero non potrebbe sopravvivere. Un albero forte ha radici forti, pertanto vorrei ringraziare la mia famiglia per essermi sempre stata accanto in questi anni, per avermi supportato moralmente ed economicamente e per non avermelo mai fatto pesare.
Ringrazio mio padre per avermi insegnato a vivere come una persona onesta e per essere orgoglioso di me anche se spesso non condivide le mie scelte. Ringrazio mia madre per aver sempre assecondato la mia curiositá e avermi spinto a chiedermi di piú ogni giorno facendomi diventare l'uomo che sono oggi. Infine, ringrazio mia nonna per aver sempre creduto in me e per avermi insegnato che le vittorie nella vita vanno sudate.

Un doveroso ringraziamento va ad Andrea Marcelli e al Professor Giovanni Squillero per avermi dato l'opportunitá di lavorare con loro. In particolare ringrazio Andrea per tutti i consigli fornitimi in questi mesi e per essere sempre stato pronto a rispondere alle mie mail e ai miei messaggi, anche alle ore piú improbabili della notte.

Credo inoltre, che la forza di un uomo si misuri in base alle sfide che é in grado di affrontare, pertanto vorrei ringraziare i ragazzi di IEEE-HKN per avermi accolto — un gruppo di persone straordinarie destinate a fare grandi cose — e la famiglia che non pensavo di trovare, JEToP, la Junior Enterprise del Politecnico di Torino. Un ringraziamento particolare va al mio "successore" Lorenzo e al mio ex-responsabile, Simone Lanzafame, per avermi fatto sperimentare sulla mia pelle che nessuna sfida é troppo grande per essere affrontata. Proseguo ringraziando Nicole, per essere stata la spalla su cui potevo sempre contare durante la mia esperienza da responsabile, e tutti i miei area manager, per il lavoro impeccabile svolto durante il mio mandato.

Proseguo ringraziando le persone che hanno condiviso con me l'interminabile primo anno di Politecnico, quelli che la suddivisione per ordine alfabetico mi ha fatto incontrare e che sono passati da colleghi ad amici: Alessio, Daniele, Luca e Silvia. Ringrazio inoltre le persone che hanno condiviso con me questa laurea magistrale, un percorso affrontato sempre uniti e supportandoci l'un l'altro in cui abbiamo superato laboratori, progetti, presentazioni ed esami alternando momenti di gioia a momenti di profonda disperazione. Grazie a Flavia, Francesco, Nicoló e Stefano. Ringrazio inoltre tutti gli amici di sempre, tra cui Riccardo, per le sue innumerevoli ore di psicoterapia a titolo gratuito e per i suoi consigli mai banali, e sua sorella Veronica, per avermi spinto a tenere duro quando, al secondo anno, ho seriamente pensato di mollare tutto.

Un ringraziamento va a tutte le persone che hanno vissuto sotto il mio stesso tetto in questi anni. Tra tutti, ringrazio particolarmente quelli con cui ho cominciato questa avventura, quelli che mi hanno sopportato e supportato in tutti i modi durante il mio percorso di laurea triennale: Mattia e Stefano.

Ultima, ma non per importanza, ringrazio la mia ragazza per essermi stata di supporto in ogni modo durante questi ultimi mesi, la tua curiositá verso il mio mondo mi fa sentire meno strano.

Infine, grazie a tutte quelle persone che ho dimenticato di menzionare che ho incontrato durante il mio percorso di studi. Non sarei dove sono ora senza il contributo di ognuno di voi.

Contents

List of Figures

List of Tables

Listings

We are survival machines — robot vehicles blindly programmed to preserve the selfish molecules known as genes. This is a truth which still fills me with astonishment.
[Richard Dawkins, The Selfish Gene]

Chapter 1

Introduction

Malware analysis is like a cat-and-mouse game. As new anti-virus techniques are developed, malware authors respond with new ones to thwart analysis. Since the first days of malware, the security industry developed antivirus programs and, nowadays, having an antivirus software installed in our computers is a well-known good habit. Malicious software plays a part in most computer intrusion and security incidents. Any software that does something that causes harm to a user, computer, or network can be considered malware. Since the first known computer virus, Elk Cloner [35], malware grows in complexity and nowadays they are able to communicate with each other, exchange information about compromised systems (exchange password and user information, IP addresses to remotely execute code via a backdoor, etc), or to evolve each other by exporting and importing code modules.

Because of the huge widespread of smart devices of the last years (smartphone, tablet, IoT devices, etc.), even more malicious users started to develop Potentially Harmful Applications (PHA) and Mobile Unwanted Software (MUwS) — apps that are not strictly considered PHA, but are generally harmful to the software ecosystem — for these devices. Due to the fact that these devices have often limitations like the power consumption or a limited amount of memory and CPU, it is not so rare that there is no antivirus software installed on these devices giving to malicious users the possibility to infect them.

In the Android ecosystem, a typical pattern among malware developers is to repack popular applications from Google Play by adding malicious features and distribute them to third-party app-stores, leveraging apps popularity to accelerate malware propagation. As a consequence antivirus software struggle to keep their signature database up-to-date, and AV scanners suffer from a considerable quantity of false negatives. Indeed creating a high quality signature able to generalize enough to match new malware variants, while avoiding false positives detection, is a challenging task, and requires a substantial portion of human experts time.

Recently, to try to counter this phenomenon and to help malware analysts to create better signatures a new Python 3 tool was proposed. YaYaGen, acronym of *Yet Another YARA Rule Generator*, exploits a greedy-like optimization algorithm to generate YARA

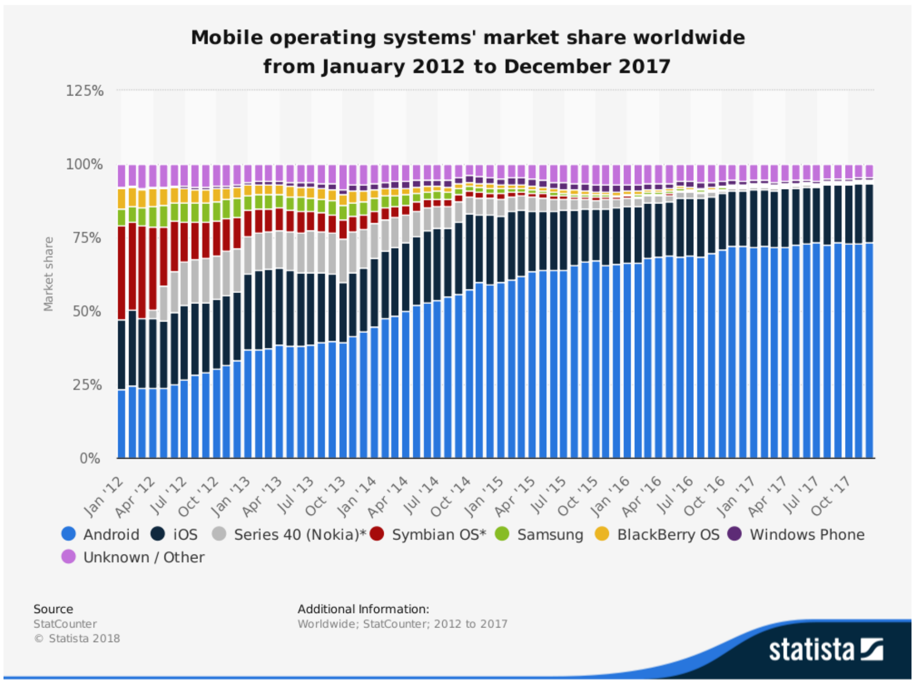


Figure 1.1: Distribution of mobile operating systems during the last years.

rules, a pattern matching language used to write malware signatures which is becoming a de facto standard. YaYaGen starts from a set of application analysis reports, possibly belonging to the same malware family, and identifies a set of clauses capable to match all the targets, then it produces a signature that can be seamlessly used by any tools that supports YARA rules.

The workflow in which YaYaGen operates is illustrated in Figure 1.2 and it is composed by the following phases:

1. *Data Analysis*: in this phase several Android application are chosen and analyzed to extract common characteristics useful for the clustering process;

2. *Clustering*: the apps are divided into malware families to facilitate the rule generation;

3. *Validation*: an analysts have to check if the malware was correctly classified. It is not yet possible to automatize this phase;

4. *YaYaGen*: the reports are converted in a working YARA rule using a greedy-like optimization algorithm. Initially this step was composed only by the rule generation phase; during the development of this thesis project was integrated also the rule optimization phase;

5. *Final Signature* memorization: the final generated signature is written on one or more files, ready to be used.

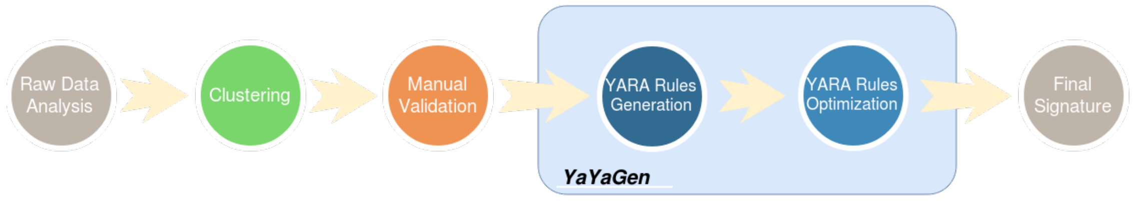


Figure 1.2: Workflow of the signature generation.

This work focuses on the research of computational intelligence techniques for the automatic malware signature generation, introducing a score system that maximize the rule efficacy, finding the best agreement between signature generality and specificity. To achieve that we improved the greedy-like algorithm previously proposed, with an ensemble techniques which combines heuristics and evolutionary algorithms. In particular, during this thesis project, we concentrated our efforts in two main research areas:

- we implemented a new signature-generation algorithm based on an Evolutionary Algorithm;
- we added a new *rule optimization* step next to the *rule generation* phase.

In particular, talking about the optimization phase, we developed two kinds of optimizer: an heuristic optimizer (called Basic Optimizer) and a second one based on an Evolutionary Algorithm developed in house (called SGX Rule Optimizer). For developing the second optimizer, we adopted the Selfish Gene Extended algorithm (SGX), an evolutionary library — exponent of the so-called Estimation of Distribution Algorithm (EDA) — loosely inspired by the neo-Darwinian theory of Richard Dawkins.

Chapter 2

Background

2.1 Android Operating System

Figure 2.1: Android Logo.

Android is a mobile operating system initially developed by Android Inc., based on a modified version of the Linux kernel and other open source software and designed initially for mobile devices such as smartphones and tablets. Later on, after the acquisition by Google in 2005, will be developed also Android Wear for wrist watches, Android TV for televisions, Android Auto for cars and Android Things for IoT devices, each of them with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics.

Android was unveiled in 2007, with the first commercial Android device launched in September 2008. The operating system has since gone through multiple major releases, with the current version being 8.1 "Oreo", released in December 2017.

Android dominates in the battle to be the top smartphone system in the world since 2013 when it was ranked as the top smartphone platform[55]. Nowadays, in the end of 2017, Android OS has over two billion monthly active users, the largest installed base of any operating system, and the Google Play store features over 3.5 million apps[44].

Figure 1.1 illustrates the distribution of the different mobile operating systems during the last years[1] meanwhile in figure 2.2 it is shown the number of Android application present on Google Play during the years[2].

[1]Real time update: `https://goo.gl/ZmH5N5`

[2]Real time update: `https://goo.gl/RxQgNv`

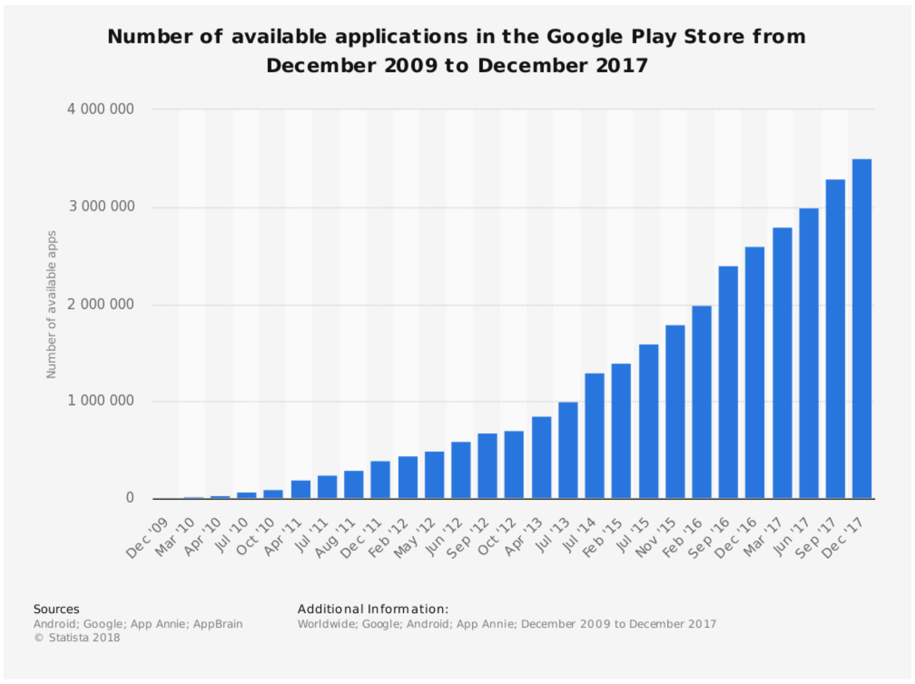


Figure 2.2: App available on Google Play during the last years.

2.1.1 Android Environment

Android is based on the Linux kernel, supplemented with middleware and libraries written in C/C++ meanwhile the official programming language of Android applications is *Java*, with the support of the Android *Software Development Kit* (SDK). Thanks to this particular architecture, Android can enjoys the full benefit of Java such as platform independence, added security and ease of app development.

On May 17, 2017, at *Google I/O*, the developer conference hold by Google Inc. every year, was officially announced that *Kotlin* will be fully supported for developing Android application[3][6].

Process Virtual Machine

Both Java and Android, to create their own *platform independence*, take advantage of a Process Virtual Machine (PVM). A PVM, sometimes also called *application virtual machine* or *Managed Runtime Environment* (MRE), runs as a normal application inside a host OS and supports a single process.

It is created when a process is started and destroyed at the end of the execution time. It is an high-level abstraction with the purpose of providing a programming environment

[3]Google I/O 2017 — Getting Started with Kotlin: https://youtu.be/czKo-jPVweg

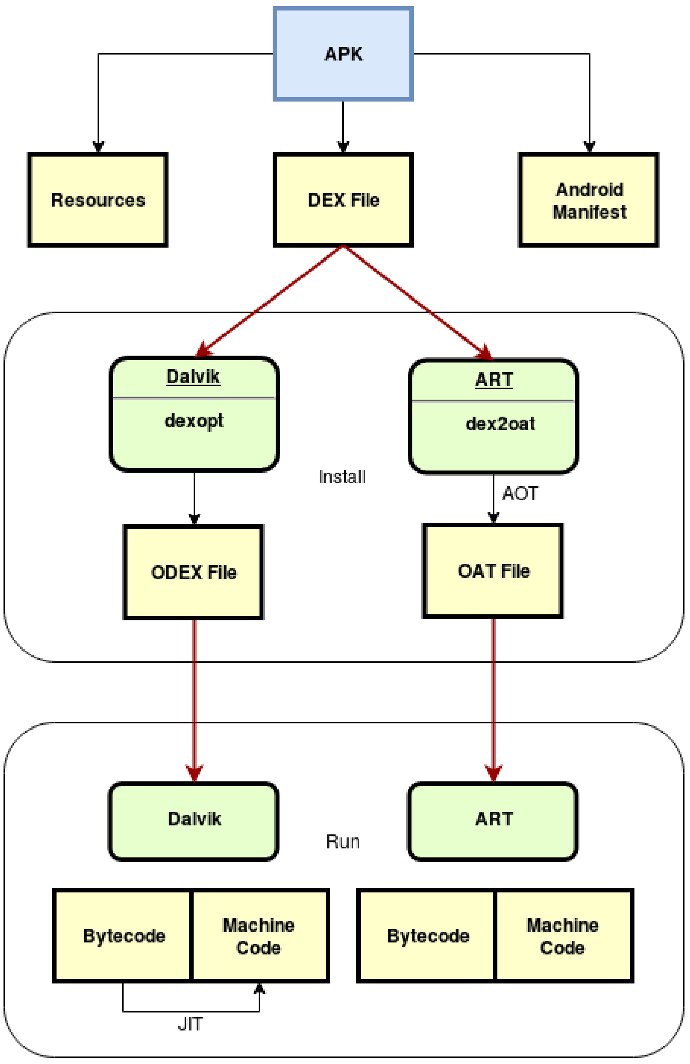


Figure 2.3: Flowchart of installation and run methods: Dalvik and ART architectures compared.

that abstracts away details of the underlying hardware or operating system, and allows a program to execute in the same way on any platform.

Dalvik Virtual Machine

Unlike a pure Java application, Android employs its own virtual machine to perform code execution called *Dalvik Virtual Machine* (DVM)[34]. Dalvik is a *Process Virtual Machine* that uses just-in-time compilation (JITC), a technique to increase execution speed of applications by compiling parts of an application to machine code at runtime. Since mobile phones did not have a lot RAM when Android was developed, Dalvik tried to make sure that the RAM management was as effective as possible.

The application source code is first compiled into Java classes and further compiled into a *Dalvik Executable* (DEX file) via dx tool. Then, the DEX program and other resource files (e.g. XML layout files, images) are assembled into the same package, called *Android application package* (APK) file.

Android RunTime

Since Android 5.0 (Lollipop), Dalvik Virtual Machine was replaced by *Android RunTime* (ART), which uses the same bytecode and .dex files, with the succession aiming at performance improvements transparent to the end users. The environment was already included as "technology preview" in Android 4.4 (KitKat), but Lollipop is the first Android version to fully support them.

While Dalvik compiles the code on the fly, ART compiles the code when the application is installed. The main reasons for this architectural change were to increase the speed of the code even more than Dalvik, improve garbage collection, better support for multi-core processors and implement 64 bit support. Unfortunately, due to the fact that applications are no more executed just-in-time, these improvements have two main disadvantages: the same application compiled for ART uses more storage memory and the installation time is slightly extended[26].

As it is possible to see in figure 2.3, the is quite similar even after the introduction of the new virtual machine[33]. The main difference is that in the case of Dalvik, the DEX file in the distributed APK was optimized using the dexopt tool and the Optimized Dalvik Executable (ODEX) file is generated, while for ART, the DEX file is compiled using the dex2oat tool and the OAT file is generated. The ODEX file is the executable file with Dalvik and, while almost identical to the original DEX file, is partially optimized so that operation code is partly changed. On the other hand, the OAT file in ART is a file in ELF format, completely different from the original DEX file, and includes the machine code generated through the compilation process.

2.1.2 Android Framework API

While an APK file is running inside the virtual machine, the Android framework code is also loaded and executed in the same domain. As a matter of fact, a DEX file acts as a plugin to the framework code and a large portion of program execution happens within the Android framework.

A DEX file interacts with the Android framework via *Application Programming Interface* (API) provided through the Android SDK. From the developers' point of view, Android API is the only channel for them to communicate with the underlying system and enable critical functionalities. Due to the nature of mobile operating system, Android offers a broad spectrum of APIs that are specific to smartphone capabilities. For instance, an Android app can programmatically send SMS messages via `sendTextMessage()` API or retrieve user's geographic location through `getLastKnownLocation()`.

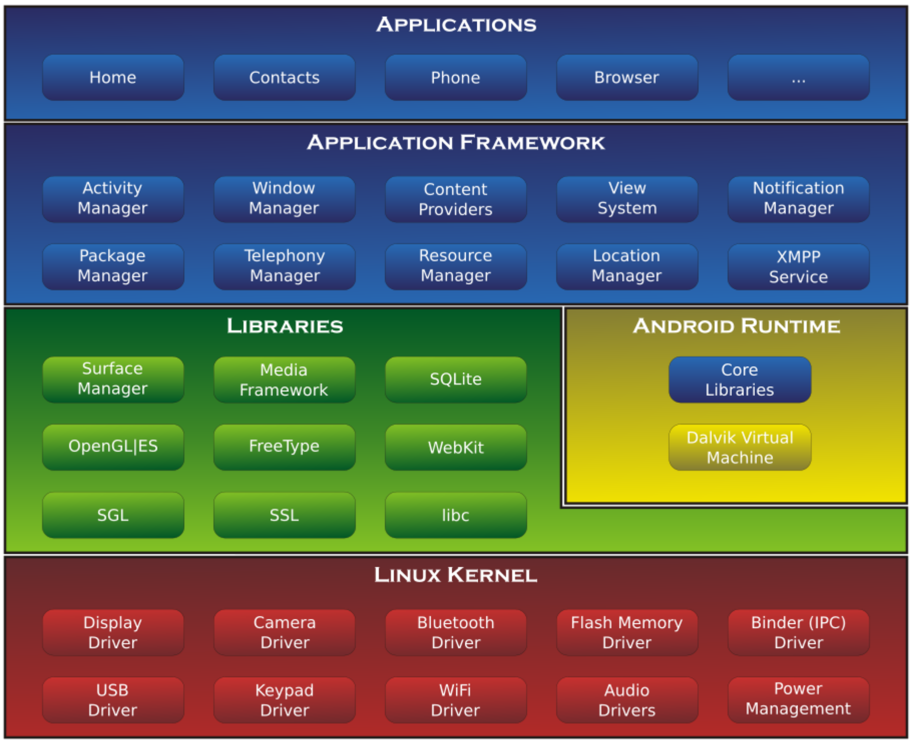


Figure 2.4: Historical Android's architecture diagram with DVM runtime environment.

2.1.3 Android Permissions

Sensitive APIs are protected by Android permissions. To enable the critical functionalities in an app or to access sensitive user data (such as contacts and SMS), a developer has to specify the needs for corresponding permissions in a manifest file `AndroidManifest.xml`. At runtime, permission checks are enforced at both framework and system levels to ensure that an app has adequate privileges to make critical API calls. Depending on how sensitive the area is, the system may grant the permission automatically, or it may ask the user to approve the request.

Right now, there are two different ways to request access to permissions:

1. *Runtime request* — Android 6.0 (API level 23) and higher
   the user will not be notified of any app permissions at install time. The app must ask the user to grant the dangerous permissions at runtime. When the permission is requested, the user will see a system dialog telling the user which permission group your app is trying to access and it has to choose if click "Deny" or "Allow". An user can also choose to select "never ask again" to grant or to revoke "definitively" a permission to the app.

2. *Install-time request* — Android 5.1.1 (API level 22) and below
   the system automatically asks the user to grant all dangerous permissions for the app

at install-time. If the user clicks Accept, all permissions the app requests are granted. If the user denies the permissions request, the system cancels the installation of the app. If an app update includes the need for additional permissions the user will be prompted to accept those new permissions before updating the app.

2.1.3.1 Classification of protection levels

Permissions are divided into several protection levels. The protection level affects whether runtime permission requests are required[22]. Since API level 27, ordered from the least to most dangerous, a permission can belongs to one of the following protection levels:

1. Normal
2. Dangerous
3. Signature
4. Signature or System

Normal permissions

Normal permissions cover areas where an application needs to access data or resources outside the app's *sandbox*, but where there's very little risk to the user's privacy or the operation of other apps; for example the permission to to connect to paired Bluetooth devices (`BLUETOOTH`) is a normal permission.

If an app declares in its manifest that it needs a normal permission, the system automatically grants the app that permission at install time. The system does not prompt the user to grant normal permissions, and users cannot revoke these permissions.

Dangerous permissions

Dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps (e.g. the ability to read the user's contacts is a dangerous permission). If an app declares that it needs a dangerous permission, the user has to explicitly grant the permission to the app when the permission request is prompted.

Signature permission

A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.

Signature or System permission

A permission that the system grants only to applications that are in a dedicated folder on the Android system image or that are signed with the same certificate as the application that declared the permission. Avoid using this option, as the signature protection level should be sufficient for most needs and works regardless of exactly where apps are installed. The `signatureOrSystem` permissions are used for certain special situations where multiple vendors have applications built into a system image and need to share specific features explicitly because they are being built together.

2.1.3.2 Other classes of permissions

Besides the classification based on the dangerous levels, it is possible to classify a permission on Android OS also in one of the other categories.

Special permissions

This class of permission is considered particularly sensitive and should not be used by most Android applications[22]. Right now, considering API level 27, belong to this class the *SYSTEM_ALERT_WINDOW* permission — that allows to an application to draw an alert window on the top of every running application — and the *WRITE_SETTINGS* permission — that allows an application to read or write the system settings. When an app needs one of these permissions, it must declare the permission in the manifest, and send an intent requesting the user's authorization. The system responds to the intent by showing a detailed management screen to the user.

Not third party app permissions

With *third party app* it is referred every other applications than the ones belongings natively to the Android Operating System or developed by a manufacturer. Usually this can be read as: everything that did not come pre-installed.

So, as it is easy to inferred by the name, this class of permissions can not be used by application developed by third party but they are reserved for Android team and manufacturers approved by Google.

Considering the classification of the protection level, these permissions could belong to any of them.

Custom Permissions

Applications can expose their functionality to other apps by defining permissions which those other apps can request. This kind of permission are the so-called *custom permission.* When a custom permission is declared the developer has also to specify the *protection level* of the declared permission.

An Android application can also define permissions which are automatically made available to any other apps which are signed with the same certificate. This is easy to check because all the APKs must be signed with a certificate whose private key is held by their developer.

2.1.4 Android Stores

An Android application, assembled in an APK package, can be shared as any other files to be installed on any Android device. One of the most common way to spread applications is to be submit them to the *Google Play store* (the official Android Market) or to one of the many Android app markets on the web. An app market serves as the hub to distribute the application products, while consumers can browse the market and purchase (or download for free) the APK files.

In recent years there has been a large increase in the number of unofficial third-party marketplaces, both in number and variety, due to the large demand placed by users. Launching an app into an unofficial store may contribute to reach a niche audience. They could be used to reach a particular user segment (e.g. games, utilities, business) or a particular nationality, for example, the Chinese market — where Google Play is not officially available — should be target through one of the Top 10 Android app stores in China.

Another reason that could drive a developer to use an unofficial store could be the revenue; alternative app stores may offer a revenue share model that is more favourable to developers than the standard 70/30 split offered by Apple and Google Play or can offer other incentives (e.g. to pay for a certain amount of downloads, to increase application's popularity, to buy "featured" placement). This ability to access additional promotional tools is one of the key advantages of using different app stores for independent developers who may find the tight editorial control of the major stores difficult to penetrate.

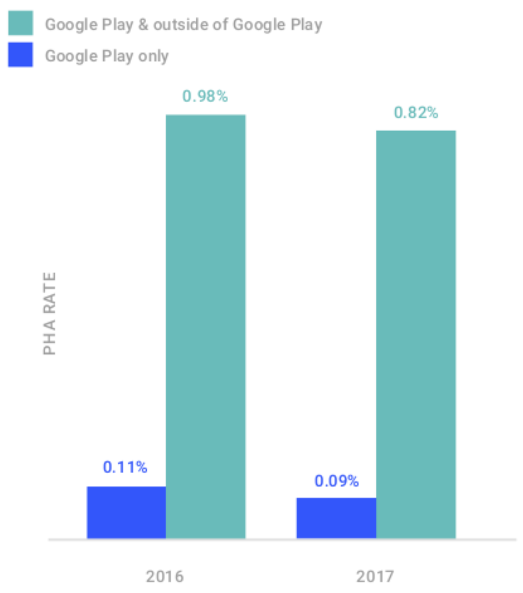


Figure 2.5: Devices with malicious app installed in 2016 and 2017.

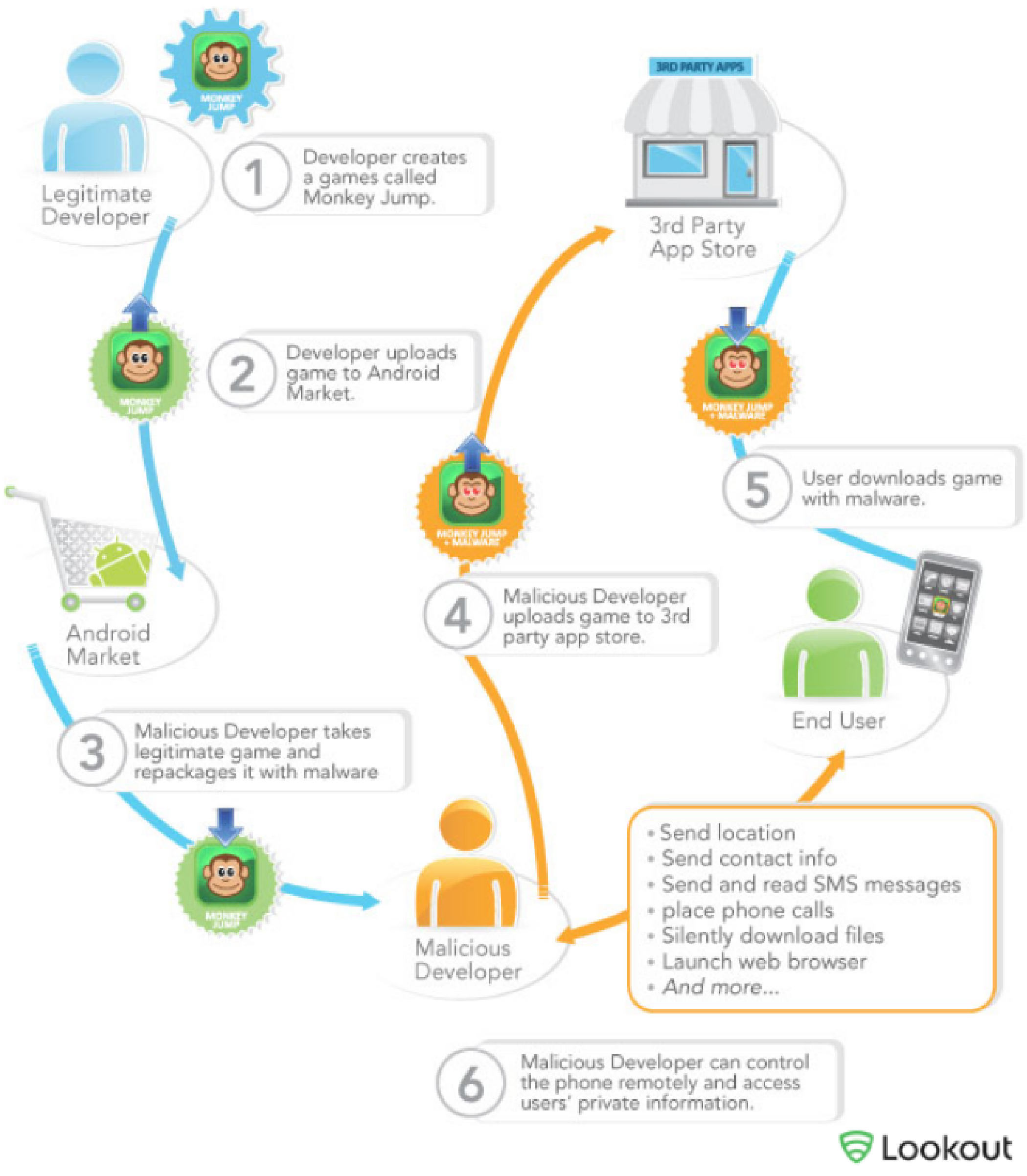
Unfortunately such degree of freedom comes with a price, along with all these stores the number of fake and malicious apps was also increased. According to the *Android Security*

report of 2017[21], when Android Security Team started to measure devices hygiene in late 2014, less than 1% of devices have malicious applications installed on average, in 2016 the number of devices affected by this kind of application was reduced to 0.77%. In 2017, 0.56% of all Android devices scanned by Google Play Protect — a security package for Android devices consisting of app scanning, browser protection, and anti-theft measures — had installed a potentially malicious application. In particular, on average, as it is shown in figure 2.5, only 0.09% of devices that exclusively used Google Play had one or more potentially harmful applications installed.
As a relevant study revealed in 2012[56], between 5% and 13% of a sample number of apps downloaded through third party marketplaces were modified from the original version available on Google Play and these percentages do not seem to decrease. For this reason scholars are continuing to work on tool able to detect this phenomenon[53].

2.1.5 App Repackaging

The phenomenon of modifying the original version of a well-known application integrating inside them malicious functionality is commonly known as *App Repackaging attack*.

This kind of attack, thanks to all the available unofficial Android marketplace, is more common on Android OS than on other operating systems. In such an attack, attackers modify a popular app downloaded from app markets, reverse engineer the application, add some malicious payloads, and then upload the modified app to app markets. Users can be easily fooled, because it is hard to notice the difference between the modified app and the original app. Once the modified apps are installed, the malicious code inside can conduct attacks (usually in the background) that can have destructive consequences for unsuspecting smart phone users, such as sending premium rate SMS, gathering personal information or even stealing money[20].


MONKEY JUMP
Legitimate Developer
1 Developer creates a games called Monkey Jump.
2 Developer uploads game to Android Market.
Android Market
3 Malicious Developer takes legitimate game and repackages it with malware
Malicious Developer
MONKEY JUMP + MALWARE
4 Malicious Developer uploads game to 3rd party app store.
3RD PARTY APPS
3rd Party App Store
5 User downloads game with malware.
End User
• Send location
• Send contact info
• Send and read SMS messages
• place phone calls
• Silently download files
• Launch web browser
• And more...
6 Malicious Developer can control the phone remotely and access users' private information.
Lookout

Taken from *Lookout Mobile Threat Report 2011*.

Figure 2.6: Conceptual workflow of a repackaging attack.

2.2 Android Malware

2.2.1 History of malicious applications

Malware stands for malicious software. Any software that does something that causes harm to a user, a computer, or a network can be considered malware[31].

Malware exists in different variants: worm, rootkit, trojan, and virus. The term *computer virus* was coined by Fred Cohen in 1983[7]. Viruses are programs able to replicate themselves and infect various system files. As many other in computer science, the idea of self-replicating software can be traced back to John von Neumann in the late 50s[50], yet the first working computer viruses are much more recent. *Creeper*, developed in 1971 by Bob Thomas, is generally accepted as the first working self-replicating computer program, but it was not designed with the intent to create damage. On the other hand, the virus *Brain*, written by two Pakistani brothers and released in January 1986, is widely considered the first real malware[4].

It is interesting to note that, in computer viruses history, the focus has gradually changed from writing for "fun" to writing for "profit" in the 2000s [4]. In the early days of malware, viruses were written by hobbyists mainly for joke or for challenge. They usually played with the user or print funny messages or graphics on the screen. Today when someone is infected by malware, does not even know to be infected. The malware runs silently in the background, without crashing the system. If it is possible, viruses are well tested and debugged in order to not slowing down the system.

During the years, malware writers developed hiding technique to escape from antivirus detection. From a simple encryption, towards the most advanced metamorphic engines, for this reason fighting against the malware is a challenging task that involve developers and researchers all around the world.

2.2.2 Potentially Harmful Applications

On Android OS, another common term used to describe malicious applications is *Potentially Harmful Applications* (PHA). Even if this term could be considered as synonym of *malware*, the *Android Security Team* prefers to talk about PHA due to the fact that the word malware lacks a well-defined and universally accepted taxonomy and it considered to much confusing[23].

According to the Android Security Report 2017 [21], a Potentially Harmful Application is an app that could put users, user data, or devices at risk. Common PHA categories include trojans, spyware, or phishing apps.

Apps that weaken or disable Android's built-in security features are potentially harmful but can also provide functionality that users find useful and desirable. To make sure that users are aware of the risks, Google Play Protect — a security package for Android devices consisting of app scanning, browser protection, and anti-theft measures — still displays a warning when they try to install these kinds of apps. For example some users choose

[4]Defcon: The History and Evolution of Malware — https://youtu.be/L81A1pNvcz4

to root their phones to access functionality that is not available in the standard Android configuration; these users are warned when they try to root the device. Power users can proceed with installation while users who were not aware of the dangers can make more informed decisions about altering their device.

2.2.2.1 PHA classification

In these years several terms were used to describe specific malware denoting their purpose, replication strategy or specific behaviors. These terms are clearly not mutually exclusive and the same 2 program may be described by several of them[31]. The PHA classifications have changed over the years along with the ecosystem, and it is expected that will continue to change and evolve. According to the *Android Security Team*[21], it is possible to categorize them in the following types:

- *Backdoor*:
  an app that allows the execution of unwanted, potentially harmful remote-controlled operations on a device that would place the app into one of the other PHA categories if executed automatically.

  In general, the backdoor is more a description of how a potentially harmful operation can happen on a device and is therefore not completely aligned with PHA categories like billing fraud or commercial spyware apps.

- *Commercial Spyware*:
  any application that transmits sensitive information off the device without user consent and does not display a persistent notification that this is happening. Commercial Spyware apps transmit data to a party other than the PHA provider. Legitimate forms of these apps can be used by parents to track their children. However, these apps can be used to track a person without their knowledge or permission if a persistent notification is not displayed while the data is being transmitted.

- *Denial of service*:
  an app that, without the knowledge of the user, executes a denial-of-service attack or is a part of a distributed denial-of-service attack against other systems and resources. This can happen by sending a high volume of HTTP requests to produce excessive load on remote servers.

- *Hostile downloader*:
  an application that is not potentially harmful by itself, but downloads other potentially harmful applications. Major browsers and file sharing apps are not considered hostile downloaders as long as they do not drive downloads without user interaction and all PHA downloads are initiated by consenting users. An app may be a hostile downloader if:

  – there is reasonable cause to assume that the app was created to spread PHAs and the app has downloaded PHAs or contains code that could download and install apps;

  - at least 5% of apps downloaded by the app are PHAs with a minimum threshold of 500 observed app downloads (25 observed PHA downloads).

- *Mobile billing fraud*:
  an app that charges the user in an intentionally misleading way. Mobile billing frauds are divided according to the type of fraud being committed into:

  - *Call fraud*:
    it charges users by making calls to premium numbers without user consent.
  - *SMS fraud*:
    an application that charges users to send premium SMS without consent, or tries to disguise its SMS activities by hiding disclosure agreements or SMS messages from the mobile operator notifying the user of charges or confirming subscription.

    SMS fraud Some apps, even though they technically disclose SMS sending behavior introduce additional tricky behavior that accommodates SMS fraud. Examples of this include hiding any parts of disclosure agreement from the user, making them unreadable, conditionally suppressing SMS messages the mobile operator sends to inform user of charges or confirm subscription.
  - *Troll fraud*:
    it tricks users to subscribe or purchase content via their mobile phone bill (e.g. Direct Carrier Billing, Wireless Access Point, or Mobile Airtime Transfer). Wireless Access Point fraud can include tricking users to click a button on a silently loaded transparent WebView. Upon performing the action, a recurring subscription is initiated, and the confirmation SMS or email is often hijacked to prevent users from noticing the financial transaction.

- *Non-Android threat*:
  an application that contains non-Android threats. These apps are unable to cause harm to the user or Android device, but contain components that are potentially harmful to other platforms.

- *Phishing*:
  an app that pretends to come from a trustworthy source, requests a user's authentication credentials and/or billing information, and sends the data to a third party. This category also applies to apps that intercept the transmission of user credentials in transit. Common targets of phishing include banking credentials, credit card numbers, or online account credentials for social networks and games.

- *Privilege Escalation*:
  an application that compromises the integrity of the system by breaking the application sandbox, or changing or disabling access to core security-related functions. Examples include:

  - an app that violates the Android permissions model, or steals credentials from other apps;

  - an app that prevents its own removal by abusing device admin APIs;
  - an app that disables Security-Enhanced Linux (SELinux) — a Linux kernel security module.

  Privilege escalation apps that root devices without user permission do not belong to this category but they are classified as rooting apps, a particular subclass of Privilege Escalation PHA.

- *Ransomware*:
  an app that takes partial or extensive control of a device or data on a device and demands payment to release control. Some ransomware apps encrypt data on the device and demand payment to decrypt data and/or leverage the device admin features so that the app can not be removed by the typical user.

- *Rooting*:
  a particular type of privilege escalation app that roots the device. There is a difference between malicious rooting apps and non-malicious rooting apps. The first ones do not inform the user that they will root the device, or they inform the user about the rooting in advance but also execute other actions that apply to other PHA categories. The second ones let the user know in advance that they are going to root the device and they do not execute other potentially harmful actions that apply to other PHA categories.

- *Spam*:
  this class of application sends unsolicited commercial messages to the user's contact list or uses the device as an email spam relay.

- *Spyware*:
  an application that transmits sensitive information off the device. Any behaviours that can be considered as spying on the user can classify an application as spyware; in particular transmission of any of the following without disclosures or in a manner that is unexpected to the user are sufficient to be considered spyware:

  - contact list;
  - photos or other files not owned by the application;
  - content from user email;
  - call log;
  - SMS log;
  - web history or browser bookmarks of the default browser;
  - information from the `/data/` directories of other apps.

- *Trojan*:
  this class of application appears to be benign and performs undesirable actions against the user. This classification is usually used in combination with other categories of harmfulness. A trojan will have an innocuous app component and a hidden

harmful component (e.g. a tic-tac-toe game that, in the background and without the knowledge of the user, sends premium SMS messages).

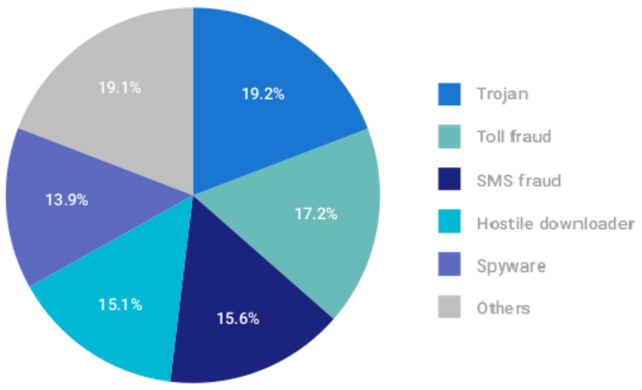


Figure 2.7: PHA categories in Google Play, 2017

Due to the fact that the Google team performs checks to avoid the presence of malicious applications on its Google Play store, this classification is also used during the evaluation process of the apps that developers submit for publication in their store.

In figure 2.7, it is possible to see the distribution of the PHA categories on Google Play in 2017.

2.2.3 Mobile Unwanted Software

Google uses the concept of "unwanted software" (UwS) as a way to deal with apps that are not strictly considered PHA, but are generally harmful to the software ecosystem. In 2016, Android took a similar approach with mobile, introducing the concept of *Mobile Unwanted Software* (MUwS). MUwS are prohibited by Google Play's policies, but even outside of Google Play they are harmful to the Android ecosystem and unwanted by most users. An example of common MUwS behavior is overly aggressive collecting of device identifiers or other metadata. Before 2016, some MUwS were categorized as PHAs, especially apps formerly described as Data Collection. Since 2016, MUwS are defined as apps that collect at least one of the following without user consent:

- device phone number;
- primary e-mail address;
- information about installed apps;
- information about third-party accounts;
- names of files on the device.

To address this problem, Google works with developers to remove this behavior from their apps or disclose their data collection practices to users.

Chapter 3

Evolutionary Algorithms

This chapter introduces the main concepts of Metaheuristic and Evolutionary computation, focusing on the evolutionary algorithms. In conclusion will be investigated the Selfish Gene algorithm, a particular exponent of the so-called Estimation of Distribution Algorithms (EDA), and its last Python implementation used inside this thesis project: the Selfish Gene Extended (SGX) library.

3.1 Introduction to Metaheuristics

Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems[27].

In particular, *Metaheuristics* are applied to "I know it when I see it" problems. They are algorithms used to find answers to problems when there is very little to help: it is not known before-hand what the optimal solution looks like, nor it is know how to go about finding it in a principled way, there is very little heuristic information to go on, and brute-force search is out of the question because the space is too large. However, if it is given a candidate solution to the problem, it is possible to test it and assess how good it is.

Hill-climbing is a simple metaheuristic algorithm; it tests new candidate solutions in the region of the current candidate and adopt the new ones if they are better. This enables to climb up the hill until a local optimum is reached.

Population-based methods keeps around a sample of candidate solutions rather than a single candidate solution. Each solution is involved in tweaking and quality assessment, but what prevents this from being just a parallel hill-climber is that candidate solutions affect how other candidates will hill-climb in the quality function. This could happen either by good solutions causing poor solutions to be rejected and new ones created, or by causing them to be Tweaked in the direction of the better solutions. It may not be surprising that most population-based methods steal concepts from biology. One particularly popular set of techniques, collectively known as Evolutionary Computation (EC), borrows liberally from population biology, genetics, and evolution. An algorithm chosen from this collection is known as an Evolutionary Algorithm (EA). Most EAs may be divided into generational algorithms, which update the entire sample once per iteration, and steady-state algorithms, which update the sample a few candidate solutions at a time. Common EAs include the Genetic Algorithm (GA) and Evolution Strategies (ES). Because they are inspired by biology, Evolutionary Computation methods tend to use (and abuse) terms from genetics and evolution[27].

3.2 History of Evolutionary Computation

Natural evolution is based on random variations: some are rejected while others preserved according to objective evaluations, and only changes that are beneficial to the individuals are likely to spread into subsequent generations. Darwin called this principle *natural selection*, a quite simple process where random variations "afford materials"[11].

From a practitioner's point of view, evolution is an optimization process that only requires to assess the effect of random changes even if the final outcome may strikingly resemble the result of an intelligent design. The field of Evolutionary Computation (EC) originates when several researchers, independently, tried to replicate evolution's dynamics to efficiently solve difficult problems. Some scholars pinpoint a paper in 1950 when Alan Turing drew attention to the similarities between learning and evolution[46] but the lack of computational power impaired their diffusion in the broader scientific community.

More commonly, the birth of EC is set in the 1960s with the appearance of three independent research lines: John Holland's *Genetic Algorithms*, Lawrence Fogel's *Evolutionary Programming*, and Ingo Rechenberg's & Hans-Paul Schwefel's *Evolution Strategies*. These three paradigms monopolized the field until the 1990s, when John Koza entered the arena with *Genetic Programming*. Moreover, *Particle Swarm Optimization* and other *swarm-based* approaches are also usually listed among EC techniques, even though they mimic the principles of social interaction rather than the struggling for survival. A comprehensive overview of the field of EC can be found in Eiben and Smith's textbook[17].

3.3 Introduction to Evolutionary Algorithm

The common idea behind all the different variants of *Evolutionary Algorithms* is the same: given a population of individuals within some environment that has limited resources, competition for those resources causes natural selection (the *survival of the fittest*). This in turn causes a rise in the *fitness* of the population.

Given a set of candidate solutions (that could be also pseudo-randomly generated) and a quality function to be maximized, it is possible to apply this abstract fitness measure to the different solutions applying the simple principle: the higher is the better. On the basis of these fitness values some of the better candidates are chosen to seed the next generation.

There are two main forces that form the basis of evolutionary systems: *variation operators* that create the necessary diversity within the population and *selection* that acts as a force increasing the mean quality of solutions in the population. The combined application of variation and selection generally leads to improving fitness values in consecutive populations; the evolution proceeds through discrete steps called generations.
To create new solutions, generally, one (or both) of these variation operators are applied: *recombination* and *mutation*.

- the first one, *recombination*, is an operator that is applied to two or more selected candidates (the so-called parents), producing one or more new candidates (the children);
- the second one, *mutation*, is applied to one candidate and results in one new candidate. It is analogous to biological mutation: it alters one or more elements inside the solution from its initial state to a new one.

Therefore executing the operations of recombination and mutation on the parents leads to the creation of a set of new candidates (the offspring); these have their fitness evaluated and then compete — based on their fitness (and possibly age) — with the old ones for a place in the next generation. This process can be iterated until a candidate with sufficient quality (a solution) is found or a previously set computational limit is reached[17].

```
BEGIN
    INITIALIZE population with random candidate solutions;
    EVALUATE each candidate;
    DO
        1. SELECT parents;
        2. RECOMBINE pairs of parents;
        3. MUTATE the resulting offspring;
        4. EVALUATE new candidates;
        5. SELECT individuals for the next generation;
    REPEAT UNTIL ( TERMINATION CONDITION is satisfied )
END
```

Listing 3.1: General scheme of an evolutionary algorithm.

It is easy to view this process as if evolution is optimizing the fitness function, by approaching the optimal values closer and closer over time. An alternative view is that evolution may be seen as a process of adaptation. From this perspective, the fitness is not seen as an objective function to be optimized, but as an expression of environmental requirements.

It should be noted that many components of such an evolutionary process are stochastic. During selection, for example, the best individuals are not chosen deterministically, and typically even the weak individuals have some chance of becoming a parent or of surviving. During the recombination process, the choice of which pieces from the parents will be recombined is made at random. Similarly for mutation, the choice of which pieces will be changed within a candidate solution, and of the new pieces to replace them, is made randomly.

The general scheme of an evolutionary algorithm is given in pseudocode in the previous listing (3.1), and it is shown as a flowchart in the following Figure (3.1).

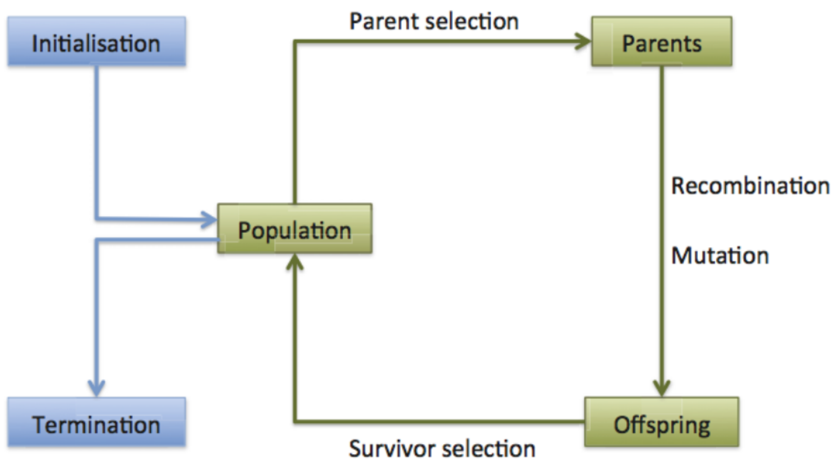


Figure 3.1: Flowchart of a general Evolutionary Algorithm

The vast majority of EC paradigms simulates a set of individuals in a population striving for survival and for reproduction. When used as optimization tools, the fitness is directly related with the ability to solve the given problem and the ultimate goal is to find an individual that maximize such a fitness value. Using a language closer to biology, each of the component previously named is called in the following way:

- A candidate solution is a *Chromosome*;
- An element of the solution is a *Gene*;
- A particular position inside a chromosome that could contain (or not) an element is a *Locus*;
- A variant form of a given Gene inside a Locus is an *Allele*.

Using this terminology it is possible to say that the fitness function evaluate the goodness of a chromosome.

To summarize it is possible to state that traditional evolutionary algorithm rely on the concept of population: a set of individuals where each of them has associated a fitness value which measures the goodness of the individual. Time is divided into discrete steps, called generations. At each generation some new individuals are generated through crossover operators and some are discarded. The choice of which individuals are used for performing reproduction usually depends on their fitness. Commonly, a mechanism called elitism is used to preserve best individuals through generations, giving them a sort of unnatural longevity, or even immortality.

As it will be explained in the following sections, almost all such elements are not present in the Selfish Gene algorithm.

3.4 Estimation of Distribution Algorithms

As it was already said in the previous section, in canonical Evolutionary Algorithms a single candidate solution is an individual and the set of all candidate solutions that exist in a particular moment represents the population. Individuals in the population strives for survival and for reproduction following Spencer's "survival of the fittest".

In each generation, the population is first expanded and then collapsed, mimicking the processes of breeding and struggling for survival. Maintaining a set of solutions, EAs are more resilient than other optimization techniques to the attraction of local optima.

The so-called *Estimation of Distribution Algorithms* (EDAs) take a quite different approach: they do not store the population as a set of distinct individual, but rather do model its relevant parameters. The main characteristics of this class of algorithms are:

- the model represents the current state of the search in terms of the relationships and distributions of the variables;
- the actual candidate solutions are *created when needed* by sampling the distribution;
- the genetic operators alter the model;

Different algorithms vary in the type of model, and in how it is tweaked but the main concepts are always there. While the *compact Genetic Algorithm* (cGA) is probably the most widely known among such population-less evolutionary algorithm[19], the idea of replacing an explicit set of individuals with a probabilistic representation of a virtual population was explored much earlier. The remarkable *Equilibrium Genetic Algorithm* (EGA) dates back to 1994[25], followed one year later by the *Population Based Incremental Learning*[2] (PBIL). Such approaches use a univariate model and assume that all variables behave independently. They represent the population storing the marginal probabilities for a gene to appear in a locus, a given location in the chromosome.

3.5 The Selfish Gene

3.5.1 The theory of Richard Dawkins

In 1976, the English biologist Richard Dawkins published: *The Selfish Gene*[13], a neo-Darwinian theory suggesting that genes strive for immortality, while individuals and species are ephemeral vehicles in that quest; a gene-centric view able to explain the behavior of social insects as well as other oddities of life on Earth. While peculiarly appealing to the general public, the work was also praised by scholars, and after more than forty years is still playing a central role in evolutionary biology and population genetics[1].

The theory was further elaborated six years later in *The Extended Phenotype*[12]. Dawkins, who is sometimes accused of ultra-Darwinism but defines himself as a mere neo-Darwinist, claims that his theory «is Darwin's theory, but expressed in a way that Darwin did not choose»[2].

Darwin's theory is based on the concept of the survival of the fittest, although Darwin himself never used this specific wording, and later scholars quarreled about the exact meaning of "fitness". The usual point of view has always been to consider the individual as the entity that can be more or less fit to survive. Indeed, individuals do not actually survive at all, but their genome does. Individuals are mortal, and their good qualities are lost with their death. On the other hand, genes are immortal: fragments of chromosome can be replicated in the offspring, and therefore survive the individual's death. All this considered, the selection mechanism can be better modeled considering "the survival of the fittest" a battle fought by genes — only genes can be more or less suited to survive, because only genes can survive.

In Dawkins' view, individuals are simply vehicles made up from the cooperation of different genes, and this cooperation is blind. A gene is not conscious, nor it has any idea about the genome it is part of and it can be a good or a bad one, depending on other genes of the individual. For instance, the gene causing the longer neck can be useful for a giraffe and potentially deadly for other animals, but it does not care about the animal it is in. The role of natural selection is to combine genes into genomes, without the need of any global information about individuals.

3.5.2 History of the Algorithm

Dawkins theory has been reflected from biology to computer science. The underlying idea that individuals could be neglected since the mechanism of selection inherently takes care of relationship and linkages between genes is counter intuitive, but it has been experimentally demonstrated. The resulting algorithms can be classified as Estimation of Distribution Algorithms (EDAs); they are univariate algorithms as PBIL[2], but, rather than assuming that variables behaved independently, they rely on the selection mechanism to compensate the lack of information.

The first Selfish Gene algorithm (SG), published in 1998[8], is a straightforward implementation of a thought experiments proposed by Dawkins in his book. Such an EDA, similarly to EGA, PBIL, and cGA, only records the first-order probabilities for genes to appear in the chromosome at specific locations. According to the gene-centric interpretation, evolution is view as a set of battles fought by alleles, that is, genes competing to occupy the very same locus. As the SG was very simple to implement, quite efficient yet more robust than pure hill climbing, it was swiftly exploited by practitioners in some

[1]It is possible to search on *Google Scholar* for having access to an updated set of articles and books.

[2]Preface of The Selfish Gene, 2nd edition, 1989

real-world applications, such as Computer-Aided Design (CAD) problems, by scholars for various test benches, and few new approaches derived from it [54, 48, 52, 51]. The SG was eventually enhanced in 1999 to better tackle deceptive functions[9].

3.5.3 The Algorithm

3.5.3.1 Virtual Population

The SG does not consider any explicit population. Instead, its Virtual Population (VP) models the *gene pool* concept defined by Dawkins. Since individuals are not explicitly listed, but implicitly represented in a pool of genes, the SG models reproduction through variations of the statistical parameters of the VP. Individuals are represented by their *genome*, a sequence of genes. The *locus* is defined as the location of the gene in the genome, while the *allele* is the binary value of the locus. Let g the number of genes in the genome: each Locus $L_i (i = 1, \dots, g)$ is occupied by an allele $\mathrm{a_i}$ which has value *0* or *1*.

In the VP, due to the number of possible combinations, genomes tend to be unique, but some alleles can be more frequent than others, hence the success of an allele is measured by the frequency with which it appears in the VP. Let p_i the marginal probability for a_i, which conceptually represents the statistical frequency of the allele a_i in locus L_i within the whole VP, regardless of the alleles found in other loci. The VP can therefore be statistically characterized by marginal probabilities of alleles a_i in the vector $p = (p_1, p_2, \dots, p_g)$.

3.5.3.2 The Generator process

The generator process simply generates individual according to VP statistics. Individual generation is further detailed inside listing 3.2, where an individual is built by choosing which allele a_i to put in each locus L, using the probability reported in p. In order to introduce further variability, a mutation can occur with the *mutation rate* probability: in that case the mutated allele is chosen in a completely random way. This function was also addressed as the process of extraction of a new individual.

```
def extract_individual():
    individual = empty genome
    for locus in individual:
        if random.uniform(0,1) < mutation_rate:
            individual[locus] = random.integer(0,1)
        else:
            if random.uniform(0,1) < p[locus]:
                individual[locus] = 0
        else:
            individual[locus] = 1
    return individual
```

Listing 3.2: New individual generation process.

3.5.3.3 The Updater process

The task of the updater is to choose what from the two individual is the best one and to update the probability distribution according to that comparison. Note that it is not necessary to have an absolute ordering between all the possible values of the candidate solutions, it is enough to being able to distinguish which individual is better between two of them. In other words, it is not necessary that he *transitive property* is valid to successfully perform the evaluation.

The Updater process modifies the VP according to the SG algorithm: the two individuals are compared, then a small constant θ is added in each locus L_i to the marginal probabilities of the allele of the winner, and removed from the marginal probabilities of the allele of the loser. The net effect is to make the extraction of individuals similar to the winner slightly more probable. The function that updates the population is shown in listing 3.3 without showing mathematical details.

```
def Updater():
    i1 = population.extract_individual()
    i2 = population.extract_individual()

    if i1.fitness == i2.fitness:
        # a tie, proceed to next comparison
        return

    if fitness(i1) > fitness (i2):
        winner = i1
        loser = i2
    else:
        winner = i2
        loser = i1

    allele_distribution.update(winner, loser)
    ...

    return
```

Listing 3.3: The population updater process.

3.5.3.4 The Evolution Mechanism

At the beginning of the evolution process, all alleles are equally probable, then the SG makes the VP randomly drift, until an allele slightly increases its marginal probability. When the given allele increases its frequency, suddenly some other allele becomes more or less desired and the positive feedback becomes effective. Ideally, the SG during the random drift selects a local optimum, and the Virtual Population quickly evolves toward such a target.

Positive feedback would quickly drive the VP towards a local optimum. The convergence speed can be tuned using the initial probability p_i: high value will make the VP moving quickly towards the first founded good solution, while a very small value will make the VP to float for a longer time before choosing which local optimum to select as a target. The algorithm is iterated until one stopping condition is reached, that is:

- an individual with maximum fitness has been evaluated;
- a maximum number of individuals have been evaluated;
- the time available has expired;
- the population cannot evolve any more because all individuals are too similar, that is, in each locus, the probability of one allele is almost 1;

Note that even if these are the default stopping conditions it is still possible to modify and expand the library according to the needs of the researcher.

3.5.3.5 Polarization

The term polarization is used to indicate the genetic variability inside a population. A population where in each locus L_i the frequency of a given gene is $p_i = 1$ while all its alleles have a frequency of zero is said to be *completely polarized.* On the contrary, when all frequencies are set to $p_i = 1/n$, the population is completely *non-polarized.* It should be noted that a completely polarized population is not composed of identical individuals, because the SG mechanism of mutation always provides a certain amount of variability. Different loci may exhibit different levels of polarization. At the end of the evolution process, the population would certainly be highly polarized. The final frequency of a gene measures, in some sense, its success in its battle for occupying the locus. However, at the beginning of the evolution process, the frequency assumes a different meaning.

A completely non-polarized initial population lets the SG start from an unspecified state. On the other hand, starting from a completely polarized population makes the SG initially behave like a Random Mutation Hill Climber (RMHC). However, after the first generations, the population will likely to become less polarized, and the SG will gradually differentiate from a pure RMHC. At the end of the evolution process, the population will be probably polarized around a different, and hopefully better, point in the solution space and therefore the SG will progressively became again more and more RMHC-fashioned.

3.5.4 The Selfish Gene Extended library

To celebrate the 20th anniversary of the original idea, the SG algorithm was rewritten from scratch in Python 3, with improved usability and several new features. The Selfish Gene Extended (SGX) library is the version of the SG algorithm that was used inside this thesis project.

In the following sections will be explained some characteristics of the library that have been used inside the development of this reasearch activity.

3.5.4.1 The fitness class

One of the most important classes of SGX library that will be extensively cited in the proposed approach is the *fitness* class. This class extends the class `tuple` and it was thought as a base class for handling and comparing different kinds of fitness values. Its constructor method (`__new__`) was developed in a very generic way giving to this class the capability of handling smoothly lists, tuples, and single numbers.

To the developer is required only to implement two methods: `__eq__` and `__lt__`, meanwhile the other methods are inferred by the library itself. Even if, for the `__eq__` method it is already provided a standard implementation, it could be useful for the developer to create its own comparison mechanism. On the other hand, if the `__lt__` method is not implemented by the developer, a `NotImplementedError` exception is raised by the library. For that reason, this class should not be used or modified directly but it is suggested to extended them according to developer's necessity.

To help the developers, the library already provides some implementation of the fitness class, in most updated version, was already developed: the `FitnessChromatic`, the `FitnessLexicase` and the `FitnessLexicographic`. In particular, the last one was the one that was used in the proposed approach, for that reason it will be discussed in the following section.

```
class Fitness(tuple):
    def __new__(cls, fit):
        if hasattr(fit, '__iter__'):
            return tuple.__new__(cls, fit)
        return tuple.__new__(cls, (fit,))

    # Standard methods
    def __bool__(self):
        """True iff all objectives are not None"""
        return all(v is not None for v in self)

    def __str__(self):
        return str(tuple(self))

    def dominate(self, other):
        """True iff self > other"""
        if not self:
            return False
        return self > other

    # Required
    def __lt__(self, other):
        """Override to implement different selection schemes"""
        raise NotImplementedError

    # Default
    def __eq__(self, other):
        return tuple(self) == tuple(other)
                or (not self and not other)

    # Methods that can be inferred
    def __ne__(self, other):
        return not self == other

    def __gt__(self, other):
        return (not self < other) and (self != other)

    def __le__(self, other):
        return self < other or self == other

    def __ge__(self, other):
        return self > other or self == other
```

Listing 3.4: The fitness class of SGX.

The lexicographic fitness

This class is a subclass of the `fitness` class and, as it was stated in the previous section, it implements the `__lt__` method. When it is used in comparison operations (i.e. >, <, ==, !=), this class performs simply a comparison in a lexicographic way, starting from the first elements of the two tuples to the last ones. To work properly the two `FitnessLexicographic` that have to be compared, are "upcasted" to `tuple` objects.

```
class FitnessLexicographic(Fitness):
    def __lt__(self, other):
        if not other:
            return False
        elif not self:
            return True
        return tuple(self) < tuple(other)
```

Listing 3.5: The lexicographic fitness class of SGX.

Chapter 4

Automatic YARA Rule Generation

This chapter analyzes well known approaches of Automatic signature generation with a particular emphasis on the YARA rule generation techniques. It introduces also the Koodous platform, YaYaGen — the tool that was improved for this thesis project — and other rule generation tools.

4.1 YARA Rules

Figure 4.1: YARA Logo

YARA is multi-platform tool primarily used in malware research and detection. It works on Windows, Linux and Mac OS X and it provides a rule-based approach to create descriptions of malware families based on textual or binary patterns.[45]. YARA rules are easy to write and understand, they work in a similar way to *regular expression* and they have a syntax that resembles the C language.

Each rule in YARA starts with the keyword *rule* followed by a rule identifier, commonly called just *rule name*. Identifiers must follow the same lexical conventions of the C programming language, they can contain any alphanumeric character and the underscore character, but the first character can not be a digit. Rule identifiers are case sensitive and cannot exceed 128 characters. Obviously YARA keywords can not be used as identifier[1].

4.1.1 YARA syntax

All the content of this section refers to YARA Rule version 3.3.0 that is currently the version that Koodous documentation suggests to consult [43]. As it is possible to see in listing 4.1, a YARA rule can be composed by three main parts: `meta`, `strings` and `condition`. Only the last section (conditions) is always required, meta and strings sections can also be left empty or not be written at all.

A YARA rule can also contain C-style comments in both forms: single-line (//) and multi-line (/* ··· */).

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

Listing 4.1: A simple example of a YARA rule.
This rule is telling that any file containing one of the three strings must be reported as silent_banker.

4.1.1.1 Metadata

The metadata section is defined with the keyword `meta` and contains identifier/value pairs. As can be seen in in listing 4.1, metadata identifiers are always followed by an equal sign and the value assigned to them. The assigned values can be strings, integers or a Boolean values in textual form (true or false). Note that identifier/value pairs defined in the metadata section can not be used in the condition section, their only purpose is to store additional information about the rule.

4.1.1.2 Strings

The `strings` definition section is where the strings that will be part of the rule are defined. Each string has an identifier consisting in a $ character followed by a sequence of alphanumeric characters and underscores in C-like style, these identifiers can be used in the condition section to refer to the corresponding string.

Strings can be defined in text or hexadecimal form, if they are defined in text form they must be enclosed on double quotes, if they are defined in hexadecimal form they can appear contiguously or separated by spaced and must be enclosed in curly brackets. A text string is case-sensitive, to work with a case-insensitive string is necessary to apply the modifier `nocase`.

4.1.1.3 Special Variables

There are some special variables, automatically defined by the language, that could be used inside the condition section without being explicitly defined. One of the special variables that we initially used inside our rule generation is the variable `filesize`. As the name suggests, the filesize variable at execution time will contain the size of the evaluated file, expressed in bytes. Obviously the use of filesize only makes sense when the rule is applied to a file, if the rule is applied to a running process, it can never match because filesize does not make sense in this context.

Note: it is also possible to use *KB* and *MB* postfixes to create Boolean expressions with the filesize operator (e.g. filesize > 200KB); in this way the associated number will be multiplied respectively by 2^{10} or by 2^{20}.

Regular Expressions

A string variable can also contain a regular expression in a Perl compatible form. They are defined in the same way as text strings, but enclosed in backslashes instead of double-quotes. Regular expressions are one of the most powerful features of YARA and they can be followed by all the modifiers available for the strings.

In previous versions of YARA externals libraries like PCRE and RE2 were used to perform regular expression matching, but starting with version 2.0 YARA uses its own regular expression engine. This new engine implements most features found in PCRE, except a few of them like capture groups, POSIX character classes and backreferences.

(Approfondimento su reg-exp?)

4.1.1.4 Condition

The `condition` section is where the logic of the rule resides. This section must contain one Boolean expression telling under which circumstances a file or process satisfies the rule or not. The condition can contain the typical Boolean operators (*AND*, *OR* and *NOT*), the relational operators (>=, <=, <, >, == and !=) and, in numberical expressions, it is also possible to use arithmetic operators (+, -, *, /, %) and bitwise operators (&, |, «, », ;̃ ;̂).

Very often (but not in the case of this thesis project), the condition will refer to previously defined strings by using their identifiers; in this context the string identifier acts as a Boolean variable which evaluate to true of the string was found in the file or process memory, or false if otherwise.

The condition part allows also to use several extremely powerful constructs that will be not shown in depth in this thesis because currently they are not exploited by YaYaGen. More details can be found in YARA documentation: http://yara.readthedocs.io/en/v3.3.0/writingrules.html.

Undefined variable logic

Note that YARA has a simple logic to manage undefined variable behaves: any arithmetic, comparison, or Boolean operation will result in an undefined value if one of its operands is undefined, this will bring the rule to fail and to does not match the given file. Except for OR operations, where an undefined operand is interpreted as `False` giving to the computation the opportunity to continue and even to match the given file.

Referencing other rules

When writing the condition for a rule you can also make reference to a previously defined rule in a manner that resembles a function invocation of traditional programming languages. In this way you can create rules that depends on others. Note that is strictly necessary to define the rule being invoked before the one that will make the invocation.

Usually, when it is desired to reference other rule, it is suggested to reference a *private rule*. A private rule is a very simple concept, it is just a rule that is not reported by YARA when it match on a given file. Private rules can serve as building blocks for other rules, and at the same time prevent cluttering YARA's output with irrelevant information. For declaring a rule as private it is necessary to add `private` qualifier before the rule declaration.
In listing 4.2 it is possible to see a YARA rule that references a private rule.

```
private rule Rule1
{
    strings:
        $a = "dummy1"
    condition:
        $a
}

rule Rule2
{
    strings:
        $a = "dummy2"
    condition:
        $a and Rule1
}
```

Listing 4.2: Example of YARA rule reference.

4.1.2 Modules

Modules are extensions to YARA's core functionality. Some modules are officially distributed with YARA and some of them can be created by third-parties. Everybody can create its own module, it is not necessary to understand how YARA works in details, it is enough to be familiar with the C programming language — YARA exposes a simple API for modules written in C — and how to configure and build YARA from source code.

The first step to use a module is importing it with the `import` statement. These statements must be placed outside any rule definition and followed by the module name enclosed in double-quotes. After importing the module, it is possible to use its features, always using `<module name>` as a prefix to any variable, or function exported by the module. An example of a YARA rule that uses a module is available in the chapter related to the proposed approach, in figure 5.1.

Koodous platform take advantage mainly of three YARA modules: Androguard, Cuckoo and Droidbox. In particular, the Androguard YARA module was developed by the the Koodous team itself and it can be used directly writing a rule on the website without any installation.

4.1.3 Running YARA

YARA can be used in C or Python programs or it can be run directly on the command line. In order to invoke YARA on the command line are required just two things: a file with the rules to use and the target to be scanned. The target can be a file, a folder, or a process.

Rule files can be passed directly in source code form, or can be previously compiled with the *yarac tool*. Rules in compiled form could be useful to save time if YARA is invoked multiple times with the same rules. Obviously, for YARA is faster to load compiled rules than compiling the same rules over and over again.

4.2 The Koodous Platform

Koodous is a *collaborative platform* for Android malware research that combines the power of online analysis tools with social interactions between the analysts over a vast APK repository[42].

KOODOUS

Figure 4.2: Koodous Logo

Koodous gives to the analyst the possibility to use a powerful YARA rule engine, and in particular it gives to the scholar two main opportunities:

- the opportunity to test if an APK is matched by one or more public YARA rules written on the platform;

- the opportunity to test if a YARA rule written by the developer is able to match one or more APK uploaded on the platform.

The APK upload operations have no restriction meanwhile to write YARA rules on Koodous is necessary to create an account. Right now, having an account on Koodous is free and it is allowed to everyone. After the creation of the account each developer can write its own rulesets[1] and he can use them in two different ways:

- searching for a particular APK and requesting the analysis of that APK:
  if the developed ruleset is able to match it, the developer will know it;
- waiting for future APK detection:
  each APK uploaded on Koodous is tested against all the active rules on the system, if an APK will be matched by the developed ruleset, the analyst will receive a notification.

Thanks to the integration of social functionalities, Koodous offers also the possibility to follow an analyst or a ruleset. If a user follows another one, he has the opportunity to be warned when an APK gives a positive match due to his rulesets.

Koodous provides millions of APKs to download and analyze. The repository is constantly being updated with new packages from several sources, both official and unofficial. Furthermore, Koodous adds the possibility to rate and comment each available APK on its platform; so, every analyst can up-score or down-score an application if he considers it relatively a goodware or a malware.
An APK is considered malware when one of these two conditions are true:

- the APK is matched by at least one *social rule*
- the APK has a score lower or equal to *-2*

4.2.1 Write a YARA ruleset on Koodous

YARA is directly integrated inside Koodous platform and does not require any installation. When an analyst starts to create a ruleset on Koodous will find on the right part of the browser screen, a menu like the one shown in figure 4.3.

- If a ruleset is set to *private*, it is visible only to its creator otherwise, if it is set to *public*, it is visible to everyone;
- if a ruleset is *disabled*, it will be ignored by Koodous platform for its analysis otherwise, if it is set to *active*, the ruleset will be considered during APK analysis;
- if *notify me* is enabled, the analyst will receive a notification when his ruleset will match new APK otherwise, if the setting is configured to *do not notify*, he will not receive any notice;

[1] A ruleset is simply a set of YARA rules stored together

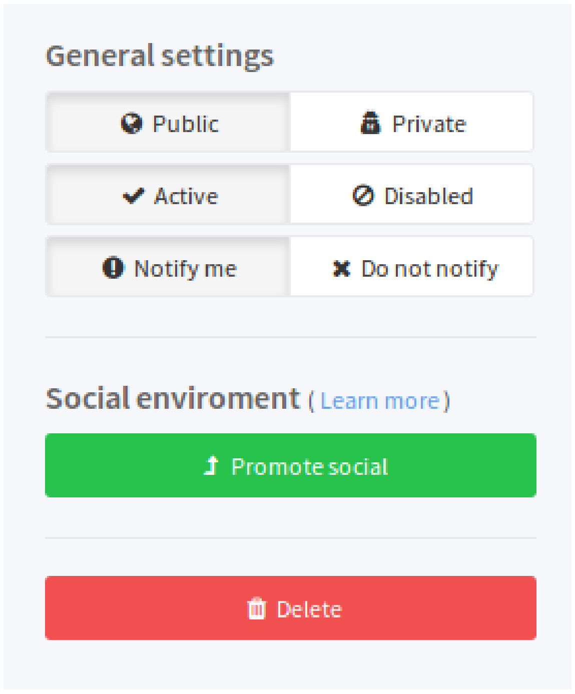


Figure 4.3: The settings shown when a ruleset is going to be created

- if the button *delete* is pressed, the ruleset will be removed by the website.

A button with a very important role in Koodous ecosystem is the *promote social* button. When this button is pressed the ruleset will be analyzed by someone of Koodous staff and, if it will be considered acceptable, the ruleset become a *Social Ruleset*. When a promote social request is delivered, the ruleset is no more modifiable until it was accepted or discarded by the team of Koodous.

A social ruleset participates in the system, warning all the analyst followers of all the positives of the rules included. If the analyst is among the most valued members of the koodous community, his ruleset will be used also to warn of the positives to all of them.

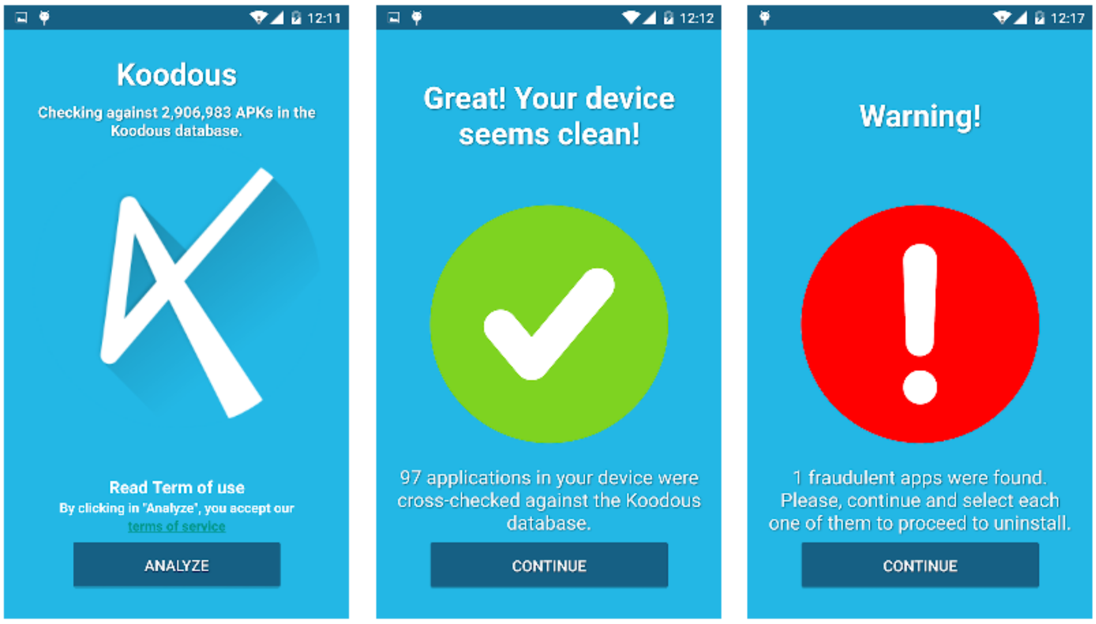
4.2.2 Koodous as anti-virus

Koodous has its own Android application to provide anti-virus functionalities to Android users[32].

Once installed the application will check if one of the installed APK is present inside Koodous database; if the application is not present, it will be automatically uploaded inside Koodous database. Otherwise if the application is present, Koodous app checks if this application has already a detection, if so the user receive the suggestion to uninstall the application.

Koodous application also provide at almost-real-time protection to the Android device on which is installed, the same check on Koodous database is also performed when a user try to install a new application on its device. If the APK is detected on Koodous backend as malware, the user will receive a notification and he can decide if he wants to complete the installation process or not.

Note, in case the user is also an analyst and he prefers to be not affected by social rulesets, he can link its own device with its account and disabling the social rulesets. Since that moment the APKs in that device will only be analysed using the rulesets developed by the analyst. In figure 4.4, it is possible to some windows displayed by the Koodous application when it is run.



Checking against 2,906,983 APKs in the Koodous database.

By clicking in "Analyze", you accept our terms of service

97 applications in your device were cross-checked against the Koodous database.

1 fraudulent apps were found. Please, continue and select each one of them to proceed to uninstall.

Figure 4.4: Screenshots taken from Android application of Koodous.

4.2.3 Clustering Android APKs

Thanks to the possibility introduced by the Koodous platform, we decided to take advantage of the already detected applications to find other APKs belonging to the same malware family. To do that we divided the applications in seven different categories that are shown in figure 4.5. These categories are:

- *Type 1*: all the APKs belonging to a family that is already correctly detected by at least a YARA rule;
- *Type 2*: all the applications belonging to a family that is already correctly detected by *at least* a YARA rule and with a score lower or equal to *-2*;
- *Type 3*: all the APKs belonging to a family that has already detected through a score lower or equal to *-2*;

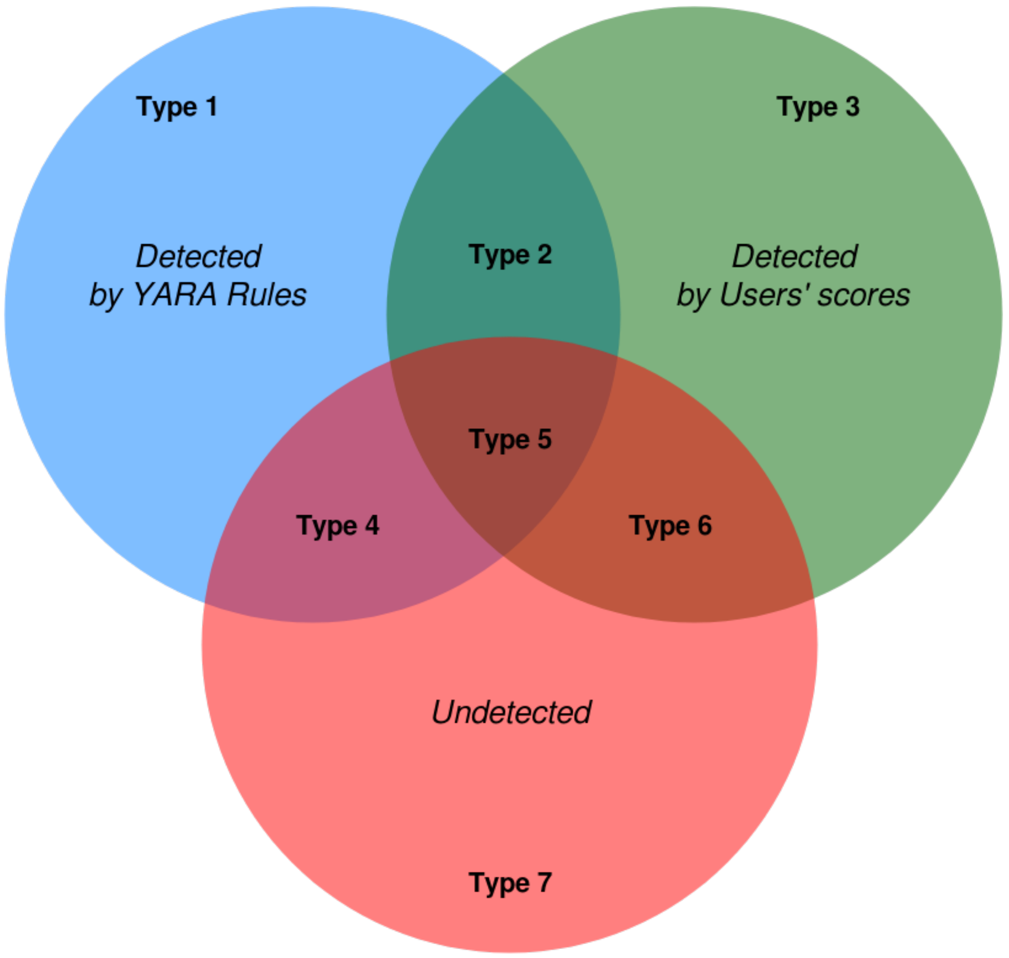


Figure 4.5: Eulero-Venn diagram of the possible clusters.

- *Type 4*: the applications contained inside this cluster belong to a family that is partially detected by *at least* one YARA rule, and partially to be still detected.

- *Type 5*: the APKs contained inside this cluster belong to a family that is already partially detected by *at least* one YARA rule and with a score lower or equal to *-2*, but this family has still malware that have to be detected;

- *Type 6*: the applications contained inside this cluster have a score lower or equal to *-2*, but the analysts did not detect yet all the possible sample of this family;

- *Type 7*: the APKs that are not yet detected, they could be malware or not.

This division could be useful to perform clustering operation and to have a preliminary idea of what kind of Koodous report an analyst is going to face off.

4.3 Signature generation

4.3.1 Introduction to Static and Dynamic analyses

Static and dynamic analyses are two of the most popular types of code security tests[28]. Before implementing one of these approach is necessary to examine precisely how both types of test can help to secure the software development life cycle. In few words they can be described in this way:

- Static analysis is performed in a non-runtime environment and involves no dynamic execution of the software under testing. Typically a static analysis tool will inspect program code for all possible run-time behaviors and seek out coding flaws, back doors, and potentially malicious code.
- Dynamic analysis adopts the opposite approach and is executed while a program is in operation. A dynamic test will monitor system memory, functional behavior, response time, and overall performance of the system. This approach is not wholly dissimilar to the manner in which a malicious third party may interact with an application.

Having originated and evolved separately, static and dynamic analysis have been mistakenly viewed in opposition. It is necessary to consider that there area number of strengths and weaknesses associated with both approaches to consider.

Static Application Security Testing (SAST) can be thought of as testing the application from the inside out — by examining its source code, byte code or application binaries for conditions indicative of a security vulnerability — meanwhile Dynamic Application Security Testing (DAST) can be thought of as testing the application from the outside in — by examining the application in its running state and trying to poke it and prod it in unexpected ways in order to discover security vulnerabilities.

Nowadays, testing an application both statically and dynamically will become increasingly important. This because some vulnerabilities can be found only with SAST testing, others with DAST; so testing in both ways yields the most comprehensive testing. Furthermore, for example, thinking about web applications development, there are many applications that would be traditionally scanned with DAST tools also use a significant amount of client-side code in the form of Javascript, Flash, Flex and Silverlight.

4.3.1.1 Strengths and Weaknesses

Having in mind the development phase, static analysis is certainly the more thorough approach and may also prove more cost-efficient with the ability to detect bugs at an early phase of the software development life cycle. For example, if an error is spotted at a review meeting or a desk-check — that are both types of static analysis — it can be relatively cheap to remedy. Had the error become lodged in the system, costs would multiply. Static analysis can also unearth future errors that would not emerge in a dynamic test.

On the other hand, dynamic analysis is capable of exposing a subtle flaw or vulnerability too complicated for static analysis alone to reveal and can also be the more expedient method of testing. However, a dynamic test will only find defects in the part of the code that is actually executed. Application type, time, and tester resources are some of the primary concerns.

While both static and dynamic tests have their shortcomings, static analysis is often considered a superior method of testing by enterprises[16]. Of course that does not necessary means that it should automatically be chosen over dynamic analysis in every situation where the choice emerges.

4.3.2 Malware detection methods

Some malware are very easy to detect and remove through antivirus software. These antivirus software maintains a repository of virus signatures (i.e. binary pattern characteristic of malicious code). Files suspected to be infected are checked for presence of any virus signatures.

This method of detection worked well until the malware writer started writing polymorphic and metamorphic malware. These variant of malware avoid detection through use of encryption techniques to thwart signature based detection.

Security products such as virus scanners look for characteristics byte sequence (signature) to identify malicious code. The quality of the detector is determined by the techniques employed for detection. A good malware detection technique must be able to identify malicious code that is hidden or embedded in the original program and should have some capability for detection of yet unknown malware. Commercial virus scanners have very low resilience to new attacks because malware writers continuously make use of new obfuscation methods so that the malware could evade detections.

Techniques used for malware detection can be broadly classified into two categories: *anomaly-based detection* and *signature-based detection*. An anomaly based detection techniques uses the knowledge of what is considered as normal to find out what actually is malicious. A special type of anomaly based detection is *specification-based detection*. Specification based detection makes use of certain rule set of what is considered as normal in order to decide the maliciousness of the program violating the predefined rule set. Thus programs violating the rule set are considered as malicious program[49].

Signature based detection uses the knowledge of what is considered as malicious to fins out the maliciousness of the program under inspection; in particular is to this class that YARA rules belong.

4.3.2.1 Signature-Based Malware detection techniques

Commercial antivirus scanners look for signatures which are typically a sequence of bytes within the malware code to declare that the program scanned is malicious in nature. Basically there are three kinds of malwares: *basic*, *polymorphic* and *metamorphic*[49]:

- *Basic malwares*: in the first category the program entry point is changed such that the control is transferred to malicious payload. Detection is relatively easy if the signature can be found for the viral code.

- *Polymorphic malwares*: this class is able to mutate while keeping the original code intact. A polymorphic malware consists of encrypted malicious code along with the decryption module. To enable the polymorphic virus, the application has got a polymorphic engine somewhere in the virus body. The polymorphic engine generates new mutants each time it is executed. Signature based detection for such a virus is difficult because each variant new signature is generated which makes signatures based detection difficult. Strong static analysis based on API sequencing is used for polymorphic virus detection.

- *Metamorphic malwares*: this third category can reprogram itself using certain obfuscation techniques so that the children never look like the parent. Such malwares evade the detections from the malware detector since each new variant generated will have different signature, hence it is impossible to store the signatures of multiple variants of same malware sample. In order to thwart detection a metamorphic engine has to be implemented with some sort of disassembler in order to parse the input code. After disassembly, the engine will transform the program code and will produce new code that will retain its functionality and would still look different from the original code.

After having specified the different kinds of malware, it is quite easy to infer that the main problems with the signature based detection method are the following:

- signature extraction and distribution is a complex task;
- the signature generation involves often manual intervention and requires strict code analysis;
- the signature can be easily bypassed as and when new signatures are created;
- the size of signature repository keeps on growing at an alarming rate.

4.3.3 YARA rules-based signature generation

According to Florian Roth — the developer of *yarGen* (4.5.1) and the creator of THOR scanner[2], a tool able to evaluate the full extend of security incidents within a computer network — a way to write «simple but sound YARA rules»[38] is to use strings quite uncommon that are not also present inside goodware applications. He suggests also to prefer UNICODE strings instead of ASCII strings because are less checked by security scanners and, for this reason, less checked by malware developers too.

In these last years, a lot of effort was put by several researcher to build YARA rules base on strings and opcode matching having in mind malwares for traditional computer and operating systems.

On the web there are also platform like the already-cited Koodous (4.2), thanks to this platform even if it is true that a YARA rule is a static analysis tool, it is possible of taking

[2]More details about THOR available at https://www.nextron-systems.com/thor/

advantage of some tools belonging to both static and dynamic analysis like Androguard, Cuckoo and Droidbox.

Thanks to these functionalities, it is possible to write YARA rules able to check also constraints that are usually analyzable only through a dynamic analysis tool. It is right having this idea in mind that YaYaGen (Yet another YARA rule Generator) was developed: creating a tool able to build powerful YARA rule more resilient to the malware obfuscation techniques.

4.4 YaYaGen

Figure 4.6: YaYaGen Logo

4.4.1 Introduction

YaYaGen is an automatic procedure, that starts from a cluster of *Koodous reports*, either identified as a malware family, or by any other mean, and eventually produces a signature in the form of a *YARA rule* that can be seamlessly used in Koodous. YaYaGen analyzes the reports of the target applications and identifies a set of clauses that are able to match all the targets[29].

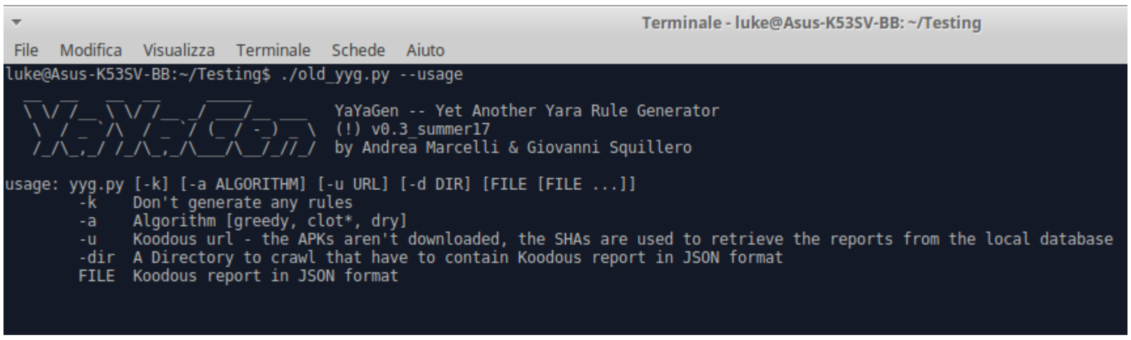
```
BEGIN
    PARSING command line parameters;
    PARSING reports from files or from local db;
    GENERATE YARA rules intersecting parsed reports;
    WRITE resulting rules on files;
END
```

Listing 4.3: General scheme of YaYaGen execution.

4.4.2 How to use it

YaYaGen accepts Koodous *JSON reports* both as positional arguments or by specifying a directory through the *-dir* option. Alternatively, it is possible to directly download them using the Koodous apk search URL (e.g., *-u* `https://koodous.com/apks?search="SHA1_A"%20OR%20"SHA1_B"%20OR%20"SHA1_C"`). Internally reports are stored in an intermediate representation and cached in a SQLite database created locally.

In figure 4.7, it is possible to see the original usage of the Python script. Even if they are not yet visible in this *usage*, it is also possible to specify to YaYaGen to avoid the database creation using option *-ndb*, and the possibility to store the generated rules on different files using option -o.


Terminale - luke@Asus-K53SV-BB: ~/Testing
File Modifica Visualizza Terminale Schede Aiuto

```
luke@Asus-K53SV-BB:~/Testing$ ./old_yyg.py --usage

 __ __ __ __      _____
 \ \/ /__ \ \/ /__ _/ ___/__ ___     YaYaGen -- Yet Another Yara Rule Generator
  \  / _ `/\  / _ `/ (_ / -_) _ \   (!) v0.3_summer17
  /_/\_,_/ /_/\_,_/\___/\__/_//_/   by Andrea Marcelli & Giovanni Squillero

usage: yyg.py [-k] [-a ALGORITHM] [-u URL] [-d DIR] [FILE [FILE ...]]
        -k    Don't generate any rules
        -a    Algorithm [greedy, clot*, dry]
        -u    Koodous url - the APKs aren't downloaded, the SHAs are used to retrieve the reports from the local database
        -dir  A Directory to crawl that have to contain Koodous report in JSON format
        FILE  Koodous report in JSON format
```

Figure 4.7: YaYaGen Usage

4.4.3 Evaluate method

A key functionality that will be exploited in the proposed approach is the `evaluate()` method. This method, contained inside class `YaraRule`, is able to give an heuristic measure of the goodness of a YARA rule.

Initially, this method returns a score that was based on the number of the attributes contained inside a rule. A rule with more attributes was considered more specific than a rule with few of them.

To summarize, it is possible to represent the score of a YARA rule in this way:

$$YR.score = \sum_{i=0}^{n} 1$$

where:

$$\begin{aligned} YR.score &= \text{the score of a YARA rule} \\ n &= \text{number of attributes inside } YR \end{aligned}$$

This approach, extremely simple, does not consider that a rule with a lot of attributes could become too much specific incurring in the phenomenon of overfitting. In the *proposed approach* (5), we will discuss how we decided to improve this method with the introduction of some heuristic strategies.

4.4.4 Rule Generation Algorithms

YaYaGen builds each YARA rule by selecting a suitable set of clauses, then picks a subset of them of variable size to build an optimal family signature. The current implementation provides two possible algorithms:

- *greedy*: select the clauses using a greedy algorithm;
- *clot*: select the clauses using a heuristic algorithm [*best choice*];

Both algorithms have the same concept implemented in two different ways: starting from a set of reports belonging to the same family, the main idea is to create a YARA rule from the intersection of all those reports. The algorithms start taking two reports and creating an intersection rule, then the created rule was intersected with another report creating a new rule; this process is iterated until all reports were used.

In figure 4.8 it is possible to see the conceptual schema of the rule generation.

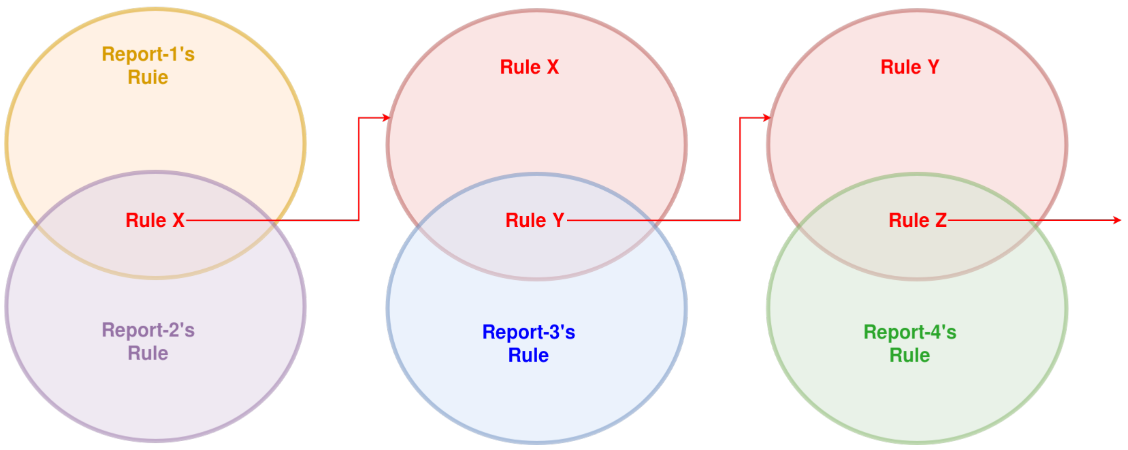


Figure 4.8: Conceptual schema of the rule generation algorithms

4.4.5 Data representation

In this section will be shown the most important data structure represented inside YaYa-Gen.

4.4.5.1 `Report` class

The class `Report` represent a Koodous report, a JSON file extracted by `koodous.com` containing all the relevant information about the malware analysis conducted on a particular APK. When it is created, a `Report` object is created with a corresponding YARA rule: that rule is represented by the class `YaraRule`. The so-created YARA rule will contains all the attributes that belongs to the Koodous report. Between the methods of the `Report` class, the most significant are:

- `_parse_jreport()`, to parse a Koodous report and convert it inside a `Report` object;
- `__init__(jreport, filename, sha256)`, that works with three different parameters:
  - if `jreport` is set, the report is creating parsing a JSON object (and eventually stored in a local database);
  - if `filename` is set, the method checks if the report was already store in the local database, otherwise it parses the file;
  - if `sha256` is set, the method checks if it is stored inside the local database a report with that SHA256.
- some matching methods useful to perform checks on a YARA rule or no a set of YARA rules given as parameters, in particular:
  - `match()` checks if a YARA rule, given as parameter, is able to match the current report (i.e. the YARA rule must has all the attributes that has also the YARA rule created from the current report);
  - `match_any()` checks if one of YARA rules contained inside the set given as command line parameter is able to match the current report (i.e. one of the rule given as parameter must has all the attributes that has also the YARA rule created from the current report);
  - `match_all()` checks if all the YARA rules given as command line parameter are able to match the current report (i.e. all the rules given as parameter must have all the attributes that has also the YARA rule created from the current report).
- the methods necessary to manipulate the local database:
  - `_db()`, for retrieving the database (the first time will be also created);
  - `_store()`, for storing reports;
  - `_fetch()`, to retrive reports previously store.

4.4.5.2 `YaraRule` class

The class `YaraRule` is an object that represent a possible YARA rule. The class `YaraRule` extends the class `set`, in this way this object is able to manage directly several type of object and it has access to all the potentiality of its superclass.

All the operation performed to generate the final signature — the final output of YaYaGen — are usually executed directly on `YaraRule` objects. Between the methods of the `Report` class, the most significant are:

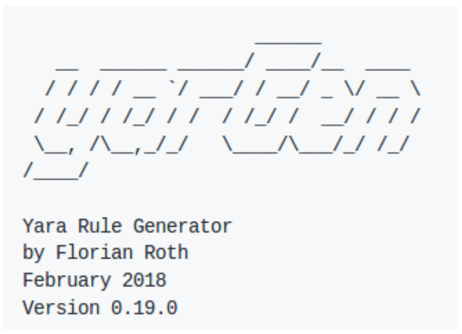
- `coverage()`: this method, given a set of reports, is able to knowing how many reports are matched by the current YARA rule;

- `evaluate()`: this method performs a rule evaluation based on the number of attributes contained inside the rule; it will be highly modified in the proposed approach;
- filter(): this method is able to reduce the dimension of the current rule according to the name of several activity;
- to_yar_format() converts the object in a textual form to be stored inside a file or inside the local database.

4.5 Other tools

YaYaGen takes advantage of several information about the applications used to generate the final YARA rules. Usually, other tools for generating YARA rules follows two ways: they are based on semantic meaningful string analysis or they are looking for particular binary values representing the signature of some well-know malicious code[39]. For this reason, YaYaGen is more robust and able to create rules able to better generalize the malware family.

4.5.1 YarGen



```
                   ______
   __  ______ ______/ ____/__  ____
  / / / / __ `/ ___/ / __/ _ \/ __ \
 / /_/ / /_/ / /  / /_/ /  __/ / / /
 \__, /\__,_/_/   \____/\___/_/ /_/
/____/

Yara Rule Generator
by Florian Roth
February 2018
Version 0.19.0
```

Figure 4.9: yarGen disclaimer, v.0.19.0

The *Yara Rule Generator* (yarGen) developed by Florian Roth [40], is a Python 3 based tool that generates YARA rules keeping a subset of strings found in malware that do not appear in goodware files. Therefore yarGen includes a big goodware strings and opcode database as ZIP archives that have to be extracted before the first use. The program uses the top 20 strings based on their score to generate the YARA rule; to help analysts it is also possible to display how a certain string is scored inside the rule or to use only strings with a certain minimum score.

The rule generation process also tries to identify similarities between the files that get analyzed and then combines the strings to so called super rules. The super rule generation does not remove the simple rule for the files that have been combined in a single super rule. This means that there is some redundancy when super rules are created. it is also possible to suppress a simple rule for a file that was already covered by super rule.

Even if it is not explicitly stated, this software was developed having in mind malwares for desktop computer (indeed it support natively the Portable Executable (PE) file format[3]).

This software is currently under development and during these years it received several updates and new functionality, in particular it can:

- partially include the goodware strings from the analysis process but they will be included with a very low score depending on the number of occurrences in goodware samples. The rules so-obtained will be included if no better strings can be found and marked with a comment: *Goodware rule*. It is possible to force yarGen to completely remove all goodware strings with a particular command (`-excludegood`);
- take advantage of the XML files generated by *PEstudio* — a software used for preliminary malware analysis;
- use a naive-bayes-classifier in order to classify the string and detect and distinguish useful words instead of meaningless strings;
- supports opcode elements extracted from the `.text` sections of PE files. During database creation it splits the `.text` sections with the regex [\x00]3, and takes the first 16 bytes of each part to build an opcode database from goodware PE files. During rule creation on sample files it compares the goodware opcodes with the opcodes extracted from the malware samples and removes all opcodes that also appear in the goodware database;
- allow the creation of multiple databases for opcodes and strings;
- support extra conditions that make use of the YARA `pe` module;
- support a "dropzone" mode in which it initializes all `strings/opcodes/imphashes/exports` only once and queries a given folder for new samples. If it finds new samples dropped to the folder, it creates rules for these samples, writes the YARA rules to the defined output file and removes the dropped samples. For example drop two files named 'identifier.txt' and 'reference.txt' together with the samples to the folder and use the parameters -b ./dropzone/identifier.txt and -r ./dropzone/reference.txt to read the respective strings from the files each time an analysis starts.

Additional tools

Since version 0.18.0, inside yarGen it was also included a tool name `db-lookup.py` that allows to query the local databases in a simple command line interface. The interface takes an input value, which can be `string`, `export` or `imphash` value, detects the query

[3]A file format for executables, object code, DLLs, FON Font files and others used in Windows.

type and then performs a lookup in the loaded databases. This allows to query the yarGen databases with string, export and imphash values in order to check if this value appears in goodware that has been processed to generate the databases. In the current version, this tool does not support the opcode lookup.

This tool could be useful for knowing if one of the previous feature appears in goodware samples or inside user database.

Installation requirements

In order to be used yarGen requires at least 4GB of RAM (8GB if opcodes are included in rule generation). According to his declaration[40], the developer tried to migrate the database to an SQLite one but the numerous string comparisons and lookups made the analysis really slow.

After the installation of the last version is required to install all dependencies from pip and then to tun the command `python yarGen.py -update` to automatically download the built-in databases (Download: 913 MB).

4.5.2 Yara Generator

Yara Generator (YG) is a work-in-progress project to build a tool for quick, simple, and effective YARA rule creation to isolate malware families and other malicious objects of interest[5]. Even if the project is defined as "work-in-progress" and it has nowadays more than 40 forks, it could be considered as a project put on stand-by: the last commit on the original repository goes back to August 29, 2013, and it has a pending pull request since February 2, 2016. Furthermore, its official website — `http://yaragenerator.com` — is currently unreachable[4].

Also in this case, even if it is not explicitly stated, this software was developed having in mind malwares for desktop computer (indeed it was integrated the support for Portable Executable (PE) files).

YaraGenerator starts from a few files from a malware family, or if it is desired to profile not executables files, it is necessary that the files contain the attribute of interest. Experiments prove that three or four samples seems to be effective for malware, however to isolate exploits in carrier documents often it is required to use many more files.

Among its latest updates, YG has included:

- support for YARA rule Tags and Unicode Wide Strings
- a Python module to work directly with PE file format;
- the possibility of selecting a particular file format to improve the generated signatures;
- the direct integration of regular expressions;

[4]Website accessed on March 2018

- a quite big database of blacklisted strings (almost 30.000 strings) divided by file format;

4.5.3 Yabin

Yabin creates YARA signatures from executable code within malware. Given one sample of malware, you can then find other samples that share code, it is considered by its developer, Chris Doman, as a prototype for testing out an approach - rather than a tool intended for production use.

It does this by looking for rare functions in a given malware sample. It identifies functions by looking for common function "prologs" which define the start of functions (e.g. 55 8B EC will often indicate the start of a function in software compiled by Microsoft Visual Studio). A whitelist taken from 100 Gb of non-malicious software is used to ignore common library functions.

Yabin could be used to help malware analysts to find malware samples that share piece of code. This could be useful mainly in two situations:

- when the scholar that is working on a malware family wants to find more sample of the same family;
- when the scholar finds a suspicious binaries and want to look if that file shares code with a malware family.

Accordin to its developer, Yabin could also be a good tool for clustering malware samples according to the re-use of their code.

Limitations

Even if Yabin is a good prototype tools, it is relatively young and it still have several limitations, for example:

- is designed to work on unpacked executables; if it is run against packed samples, it is not able to signature the sample, but it may signature the packer.
- the function prologs built in (stored in regex.txt) are designed to cover VC++, Borland and MingW compilers;
- it is not yet designed to work on .NET executables, Java software, Word documents etc.

Chapter 5

Proposed Approach

When we started working on this project, YaYaGen was still a concept and there were several challenges to face to bring it in a full prototype phase. In our research activity we worked on several fronts, in particular:

- we introduced an heuristic measure of the generated rules;
- we introduced some heuristics strategies to improve the generated rules;
- we tried to integrate the Selfish Gene Extend library (SGX) in the rule generation process;
- we take advantage of SGX library to develop an optimizer able to improve the previously generated rules: the SGX Rule Optimizer (SGX-RO);
- we introduced other heuristic strategies to improve the result of the SGX-RO.

This chapter specifies how we chose to represent the final signature and analyzes all of the previous-cited parts in detail.

5.1 Disjunctive Normal Form representation

We decided to represent the final generated signature in a way very similar to the *Disjunctive Normal Form* (DNF).

5.1.1 YARA Rule representation

5.1.1.1 Disjunctive Normal Form

The DNF is a standardization of a logical formula which is a disjunction of conjunctive clauses. Each clause is composed by one or more literals; each literal is commonly a Boolean variable. The only propositional operators in DNF are *AND*, *OR*, and *NOT*; the NOT operator can only be used as part of a literal.

The DNF can also be described as an "OR of ANDs", a *sum of products*, or (in philosophical logic) a *cluster concept*.

The DNF can be represented by the following equation:

$$DNF = \bigvee_{i=1}^{n} (\bigwedge_{k=1}^{m(i)} L_{i,k})$$

where:

$$\begin{aligned} n &= \text{number of clauses} \\ m(i) &= \text{number of literals in clause } i \\ L_{i,k} &= \text{literal } k \text{ of clause } i \end{aligned}$$

A DNF formula is in *full disjunctive normal form* if each of its variables appears exactly once in every conjunction.

A DNF important variation in the study of computational complexity is the *k-DNF*. A formula is in k-DNF if it is in DNF and each clause contains *at most k literals*.

5.1.1.2 YARA Rules DNF representation

In our case a literal is not a Boolean expression but it is one of the attributes extracted from the Koodous report (e.g., the permission of access to a resource, a target SDK, the name of an activity), for this reason, from this moment "literal" an "attribute" will be used as synonyms. Since a clause is a conjunction of literals we can assume that a clause it could be already a working YARA rule. The disjunction of several clauses help us to improve the discriminating power of the generated signatures.

To summarize, to better adapt the DNF formula to our purpose, we can rewrite them in this way:

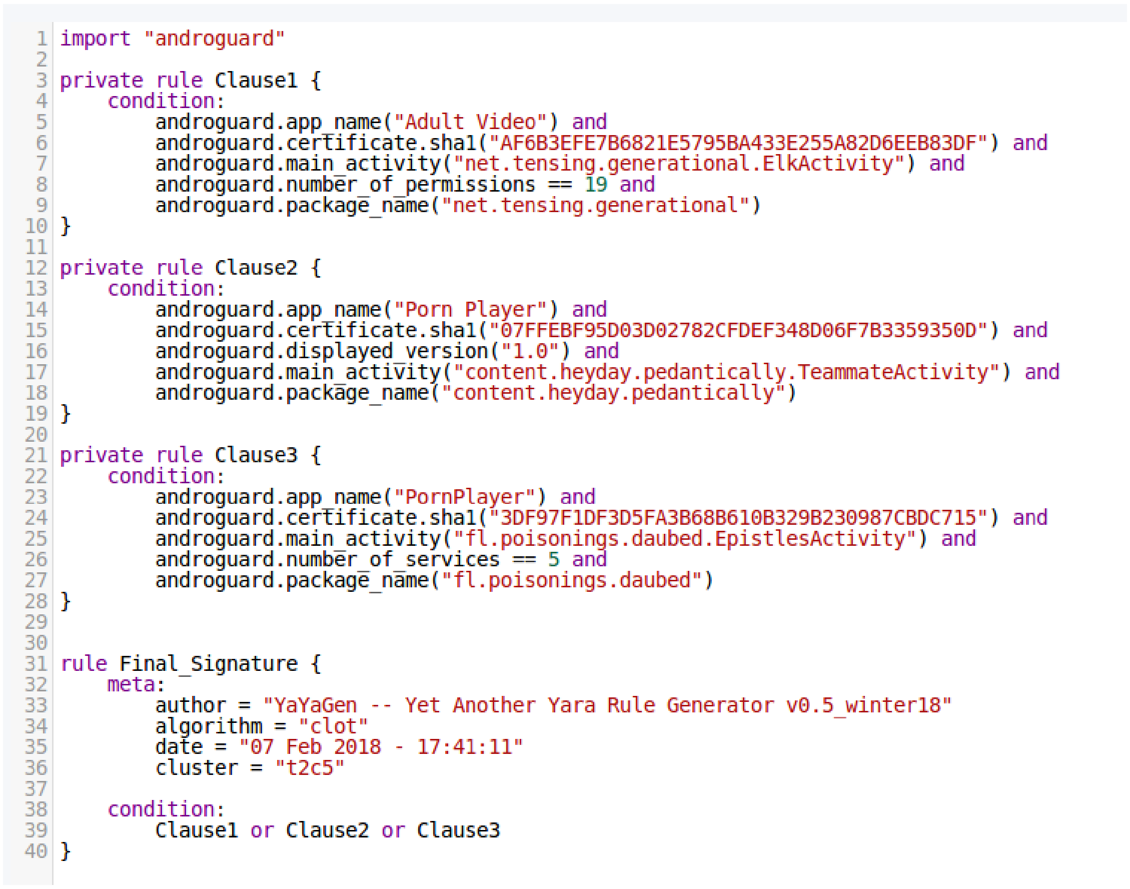
$$S = \bigvee_{i=1}^{n} YR_i$$

$$YR_i = \bigwedge_{k=1}^{m(i)} A_{i,k}$$

where:

$$\begin{aligned} S &= \text{final signature generated, a disjunction of several YARA rules} \\ YR_i &= \text{clause } i\text{, a potential working YARA rule} \\ A_{i,k} &= \text{attribute } k \text{ extracted from the Koodous report } i\text{, belonging to YARA rule } i \\ n &= \text{number of generated YARA rules} \\ m(i) &= \text{number of attributes inside YARA rule } i \end{aligned}$$

Figure 5.1 illustrates the final DNF signature generated: in the beginning, the three clauses that compose the final signature (YR_i) are declared in YARA rule form; in the end, it is possible to see a YARA rule called *Final_Signature* that invokes all the previous defined clauses.



```
import "androguard"

private rule Clause1 {
    condition:
        androguard.app_name("Adult Video") and
        androguard.certificate.sha1("AF6B3EFE7B6821E5795BA433E255A82D6EEB83DF") and
        androguard.main_activity("net.tensing.generational.ElkActivity") and
        androguard.number_of_permissions == 19 and
        androguard.package_name("net.tensing.generational")
}

private rule Clause2 {
    condition:
        androguard.app_name("Porn Player") and
        androguard.certificate.sha1("07FFEBF95D03D02782CFDEF348D06F7B3359350D") and
        androguard.displayed_version("1.0") and
        androguard.main_activity("content.heyday.pedantically.TeammateActivity") and
        androguard.package_name("content.heyday.pedantically")
}

private rule Clause3 {
    condition:
        androguard.app_name("PornPlayer") and
        androguard.certificate.sha1("3DF97F1DF3D5FA3B68B610B329B230987CBDC715") and
        androguard.main_activity("fl.poisonings.daubed.EpistlesActivity") and
        androguard.number_of_services == 5 and
        androguard.package_name("fl.poisonings.daubed")
}


rule Final_Signature {
    meta:
        author = "YaYaGen -- Yet Another Yara Rule Generator v0.5_winter18"
        algorithm = "clot"
        date = "07 Feb 2018 - 17:41:11"
        cluster = "t2c5"

    condition:
        Clause1 or Clause2 or Clause3
}
```

Figure 5.1: Signature generated in DNF form

5.2 Heuristics Strategies

According to the representation in DNF form, we assign a score to each attribute of the report, the greater is the discriminating power of the attribute, the greater was the score assigned to him. Secondly we establish that rules should have a score contained between two threshold, the lowest one (T_{min}) to avoid the creation of rules too much susceptible to match fake positives, the highest one (T_{max}) to avoid the phenomenon of overfitting. Thanks to this score, the generated signatures have a limited risk of detecting false positive in the future, yet it is general enough to catch future threats.

We established some other heuristic criteria to improve the effectiveness of the generated rules and in the end we developed a *Basic Optimizer* to reduce the score of the rules that could be considered to much specific.

5.2.1 Weighting the literals

It is clear that different literals should have been weighted in different ways. One of the firsts challenge of this project was to weight properly the different attributes that could be contained inside a rule.

To do that, we proceed using an empirical approach: we retrieve all the YARA rules present inside the *YARA Rules Project* GitHub repository[1] and we established that these rules should reach an *hypothetical score* of 100.

There are attributes that can potentially discriminate more than other, for that reason to each literal will be assigned a score according to its estimated discriminating power. For example, if an Android application asks to use a normal permission like having access to the internet, it is not very helpful to understand if this application is malware or not, this is due to the fact that accessing the web is a common behaviour for a smartphone application. On the other hand, the certificate of the developer, a particular activity name or performing requests to a particular URL are very discriminating attributes; if someone develops a malware it is highly probable that he will do it again; if inside a malware there is a particular activity, or it accesses a particular URL, it is highly probable that other malware belonging to the same family will include them again.

For these reasons, attributes judged extremely discriminating as URL, activity and service will be graded with a value able to reach the discriminating score without been aggregate with other attributes, meanwhile attributes considered not so dangerous like the normal permissions or the size of the APK will received a low score.

Table 5.1[2] shows the first scores assigned after this phase. All these scores were integrated inside the `evaluate()` method, already cited in section 4.4.3.

Thanks to this introduction, it is possible to represent the `evaluate()` method with the following formula:

$$YR.score = \sum_{i=0}^{n} score(A_i)$$

where:

$$\begin{aligned} YR.score &= \text{the score of a YARA rule} \\ n &= \text{number of attributes inside } YR \\ A_i &= \text{the attribute } i\text{, belonging to the YARA rule} \\ score(A_i) &= \text{the score associated to } A_i \end{aligned}$$

[1] Repository accessed on July 2017 — https://goo.gl/3kvx43

[2] In all the tables of this chapter, n stands for the number of the related attributes.

Literal	Assigned Score
Activity Name	100
Certificate (DN, Issuer, SHA1)	100
Number of Activities	$\lceil n/2 \rceil$
Number of Filters	$\lceil n/2 \rceil$
Number of Receivers	$\lceil n/2 \rceil$
Number of Services	$\lceil n/2 \rceil$
Filesize	10
Functionality	35
Package Name	100
Permission — Custom	20
Permission — Dangerous	11
Permission — Normal	7
Permission — Not Third Party	13
Permission — System	13
Receiver	35
Service	100
URL	100

Table 5.1: First assigned scores

After having established the scores of the attributes using this empirical approach, we decided to recalculate these score using the *Simplex Algorithm*; the results of this algorithm was later slightly modified to better fit the experimental results. The final scores can be seen in table 5.2; in the end these scores are used inside the `evaluate()` method of the class `YaraRule`.

5.2.2 Double Thresholds & Signature Optimization

The score of a rule is inversely related to its generality and it is defined as sum of the weight of its literals. The higher the score, the more a rule will be specific and less susceptible to generate false positives. The lower the score, the more a rule will be able to generalize, while more prone to unwanted detections.

In order to build effective rules and to find a balance between these two cases we introduce two thresholds: T_{min} and T_{max}; where the lowest threshold is the minimum score that a rule needs to have to be valid, meanwhile the highest threshold is used in the optimization process to avoid producing overly-specific rules.

After a conversation with the Koodous team, we decide to upscale T_{min} from 100 to 400 and to set $T_{max} = 650$.

Literal	Assigned Score
Activity Name	150
API Key	50
Certificate (SHA1)	150
Certificate (DN or Issuer)	100
Number of Activities	$\lceil n/2 \rceil$
Number of Filters	$\lceil n/2 \rceil$
Number of Receivers	$\lceil n/2 \rceil$
Number of Services	$\lceil n/2 \rceil$
Filter	150
Filter — Standard	10
Functionality	15
Main Activity Name	50
Package Name	100
Permission — Dangerous	80
Permission — Normal	7
Permission — Not Standard Value	50
Permission — Not Third Party	50
Permission — System	80
Permission — Wrong Value	150
Provider	80
Receiver	100
SDK Version (Min, Max, Target)	10
Service	150
URL	100

Table 5.2: Final assigned scores

We also have made some changes to our heuristic the more relevant changes are:

- it is not possible to have a rule with only "number of" conditions;
- it is not possible to create a rule with just one attribute even if the attribute is very discriminating as, for example, a URL;
- we introduced new parameters initially not considered like the API key;
- we remove attributes not so much relevant like the Filesize;

- we decide to distinguish more sub-attributes of attributes already considered assigning to them different scores.

The thresholds and these constraint are all used inside the `evaluate()` method of the class `YaraRule`.

5.2.3 Basic Optimizer

The goal of the Basic Optimizer is to keep the score of the rules *below* the upper threshold (T_{max}) still remaining over the lower threshold (T_{min}). To execute this task, it simply removes pseudo-randomly an attribute from the rule to optimize until the score is lower than T_{max}. Due to the fact that there are no attributes grater than the difference between the threshold ($T_{max} - T_{min} = 250$) we can safely remove an attribute without checking if we are still beyond the lower threshold.

```
def optimize(self)
    new_rule = YaraRule(self)
    while new_rule.evaluate() > YaraRule.UPPER_THRESHOLD:
        candidate = random.sample(new_rule, 1)[0]
        new_rule.remove(candidate)
    return new_rule
```

Listing 5.1: The Basic Optimizer contained inside class `YaraRule`.

The optimizing phase started after the execution of the rule generation algorithm. In that moment it was checked if each of the generated rules are under T_{max} or not; if the rule is beyond this upper threshold it will be optimized. The code of the Basic Optimizer can be seen in listing 5.1.

5.3 YaYaGen-SGX

The so-called *YaYaGen-SGX* consists in the integration of the new *Selfish Gene Extended* library inside *YaYaGen*. The main idea behind this implementation was to develop two different functionalities:

1. a new signature-generation approach alongside the already available algorithms (i.e. *Greedy* and *Clot*);

2. a new *signature-optimizer* based on the results of the already available signature-generation algorithms. This new optimizer could be used instead of the *Basic Optimizer* to have a smarter optimization.

Figure 5.2: YaYaGen-SGX

5.3.1 SGX Rule Generator

In this chapter will be explained in details the SGX Rule Generator (*SGX-RG*), the new developed signature-generation approach that take advantage of the Selfish Gene Extended library (SGX).

It will be explained how to represent a signature in a way that is easy to process for the evolutionary algorithm and it will be introduced a parallelism between the candidate solution (the individual) and the final signature. Furthermore, it will be explained how we decided to generate a set of rules to be associated with the genome and how the algorithm computes the fitness function to compare the candidate solutions.

5.3.1.1 Representing a YARA rule in SGX-compatible mode

To represent the data to be optimized by the genetic algorithm we establish to keep the classic genome representation in which each locus can contain an allele with a binary value (0/1). To the genome will be associated a set of YARA rules according to the previously defined DNF (5.1): the whole genome represents the signature, the so-called *OR of ANDs*, meanwhile each bit of the genome represent a working YARA rule, the so-called *clause*. If a clause will be present in the final signature the value of its allele will be 1, otherwise, if it will be ignored, its value will be 0. Each clause, as we already said when we spoke in detail about DNF (section 5.1), is composed by a set of literal, each literal is simply an attribute of the YARA rule.

So, even if SGX algorithm is an *Estimation of Distribution Algorithm* (EDA) and does not require to have already generated a population of individuals before starting its executions — as we stated in section 3.5.2 — it was still necessary to generate a set of rules to be associated to the genome. This is useful to have the possibility to calculate the score of each allele through the already cited `evaluate()` method (4.4.3); in this way, each bit of the genome will have a weighted value.

To create a starting set of YARA rule we developed several rule generation approaches but, before entering into the details of rule generation, we will discuss the different way in which we choose to evaluate the genome through a *fitness function*.

5.3.1.2 Fitness Function

One of the most important part necessary to make the evolutionary algorithm works was the fitness function. The fitness function is the way in which is assigned an heuristic measure to the goodness of a possible solution (an individual) [chapter 3]. Even if, it is quite easy for an analyst to know if a rule is better than another one, it is not so easy to represent this information in a way easy to process for a machine.

Simple Fitness

In our first developed fitness function, due to the fact that the final signature was represented in DNF, we established that the overall score of the generated signature was the highest score between all the contained clauses.

$$S.score = \max_{i=1}^{n}(YR_i.evaluate())$$

where:

$S.score$	= the score of the final signature generated
n	= length of the genome, number of generated YARA rules
$YR_i.evaluate()$	= the score of the clause i

Initially we decided to keep the problem simple so, in the first implementation, the best individual is the one that contains the lowest possible clause. To work properly, the fitness function requires that the clauses are already beyond the minimum threshold T_{min}. With this function is considered as dominant the individual with the lowest score. So, to summarize, the main goal of the evolutionary algorithm was to minimize the score of the generated signatures.

```
def fitness(genome, yara_rule_list):
    fitness_value = sys.maxsize
    for i in range(len(yara_rule_list)):
        if genome[i] != 0:
            if value < fitness_value:
                fitness_value = value
    return fitness_value

def Updater():
    i1 = population.extract_individual()
    i2 = population.extract_individual()
    ...

    if fitness(i1) < fitness (i2):
        winner = i1
        loser = i2
    else:
        winner = i2
        loser = i1
    ...
```

Listing 5.2: Simple fitness function

Multi-objective Fitness

To have a more robust fitness function, we decided later on to consider how many reports the clause is able to correctly detect. So, we changed the score of the final signature according to the following formula:

$$S.score = \max_{i=1}^{n}(\#detected_i * \frac{1}{YR_i.evaluate()})$$

where:

$S.score$	= the score of the final signature generated
n	= length of the genome, number of generated YARA rules
$\#detected_i$	= reports detected by the clause i
$YR_i.evaluate()$	= the score of the clause i

To do that the fitness function needs to have also the number of detection for each clause; this parameter is provided through a list associated to the first one. In this fitness we go back to consider as *dominating* the individual with the *highest score* value, we want to maximize the detections and to minimize the score of the rule. So, to summarize, the main goal of the evolutionary algorithm was to maximize the score of the generated signatures.

```
def fitness(genome, yara_rule_list, yara_rule_detection):
    fitness_value = 0

    for i in range(len(yara_rule_list)):
        if genome[i] != 0:
            rule_value = yara_rule_list[i].evaluate()
            value = yara_rule_detection[i] / rule_value

            if value > fitness_value:
                fitness_value = value

    return fitness_value
```

Listing 5.3: A multi-objective fitness function

Lexicographic Fitness

Ater the introduction of the multi-objective function we considered that having a multi-objective function could be not only more complicated, but also not very useful for our application. We decided to take advantage of one of the functionalities of the SGX library, the `FitnessLexicographic`. As it was already said in section 3.5.4.1, this class is a subclass of the class `tuple` and it was specifically designed to be managed by the SGX library. When it is used in comparison operation (i.e. >, <, ==, !=), it performs a comparison in a lexicographic way, starting from the first elements of the tuples to the last ones. The introduction of this object does not only simplify our problem but it contributes also to keep the solution more deterministic.

With the introduction of this kind of fitness we decided also what could be the most important parameter for our problem, if we want to maximize or minimize them and in which order they should be memorized. In the end, from the most to the least important, we decide to have:

1. the number of matches (to be maximized);
2. the number of clauses (to be minimized);
3. the lowest score of rules contained inside the whole signature (to be minimized — but still greater than T_{min}).

To represent the elements that have to be minimized, we decided simply to store the reverse value of these parameters inside the `FitnessLexicographic` (i.e. number of clauses and score of the genome). Furthermore, to reduce the execution time we decide to pre-calculate all the score of the genome and to introduce a third list as parameter.

To prevent the situation in which the genome will be generated with all the bits set to 0, we introduced an additional check on this value. If a genome with this characteristic will be generated, this configuration will be penalized configuring the first 2 fitness parameters with the worst possible values.

```
def fitness_function(genome, yara_rule_list, yara_rule_detection,
                     yara_rule_scores):

    best_score = sys.maxsize
    matched = set()

    for i in range(len(yara_rule_list)):
        if genome[i] != 0:
            matched.update(yara_rule_detection[i])

            if yara_rule_scores[i] < best_score:
                best_score = yara_rule_scores[i]

    m = len(matched)
    n = sum(genome)
    if n is 0:
        n = len(genome)
        m = 0

    return sgx.FitnessLexicographic((m, 1/n, 1/best_score))
```

Listing 5.4: Introduction of fitness lexicographic

So, mathematically we can represent that the score of the final signature in this way:

$$S.score = (\#detected; \frac{1}{\#allele(1)}; \max_{i=1}^{n} \frac{1}{YR_i.evaluate()})$$

where:

$S.score$	= the score of the final signature generated
$\#detected$	= reports detected by the whole signature
$\#allele(1)$	= number of allele with value set to 1
n	= length of the genome, number of generated YARA rules
$YR_i.evaluate()$	= the score of the clause i

After the introduction of this fitness, we noticed that sometimes the rule-generation phase was able to create clauses very close to maximize our fitness function. Due to the fact that maximizing the number of matching was the most important argument of our fitness, we decided to call all the clauses able to match the whole cluster from which they were generated *perfect rules*. If at least one perfect rule is present after the rule-generation phase, instead of starting the evolutionary algorithm — that usually is quite time consuming and it does not improve particularly the final signature in this case — YaYaGen simply returns as solution the perfect rule with the *lowest score* available.

Note that the clauses given to this fitness function will always must have a score greater than T_{min} to make work the algorithm properly.

5.3.1.3 YARA rule generation

In the following paragraphs will be explained the most significant steps in our rule generation approach. We started from a random generation, we introduced the concepts of *intersection rule* and power set and we concluded with some heuristic improvements of these approaches.

Random generation

Following the same approach used in the firsts heuristics strategies, we decided to start with a random approach. The first algorithm developed to generate rules to be associated to the genome of the final signature was quite-completely random.

The function shown in listing 5.5 receives as parameters the reports of the APKs given as input parameter and the desired `genome_length` and it returns as output a list of YARA rules with length equal to `genome_length`. All generated rules must have a score equal to or greater than T_{min} — that is written inside constant `YaraRule.THRESHOLD` — and they can not contain duplicate attributes; the absence of duplicate attributes is coded inside function `_get_random_attribute()`, this is the reason why `new_rule` is given as parameter to that function.

As it was already said when it was discussed the double-threshold approach (5.2.2), it is not necessary to check if the score of a rule go behiond T_{max} because an attribute with a score equal or greater than the difference of the two thresholds does not exists ($attribute\_score < T_{max} - T_{min}$).

```
def random_generation(reports, genome_length):
    new_rule_list = list()

    for i in range(genome_length):
        new_rule = YaraRule()
        while new_rule.evaluate(i) < YaraRule.THRESHOLD:
            # extracted from reports not present in new_rule
            attribute = _get_random_attribute(reports, new_rule)
            new_rule.add(attribute)

        new_rule_list.append(new_rule)

    return new_rule_list
```

Listing 5.5: Rules generated with a random approach

Random generation from a common part

The main problem of the first random approach was that very often the generated rules are unable to match even a report. To try to improve the generated rules keeping the generation process quite random we decided to generate the rules starting from a *common part*; this common part was represented by the intersection of all the reports given as input parameter. After the generation of the so-called `base_rule`, for each rule that will be associated with the final genome, from the `base_rule` will be pseudo-random extracted attributes until T_{min} is reached, or until all the common attributes are extracted. After that pseudo-random attributes are added to the new `YaraRule` until the threshold score is reached.

Even if the constraint of the absence of duplicate attributes remains valid, in respect to the previous rule generation, in this case it could happen that the found intersection between the reports is null and the resulting `base_rule` is empty. However this is not a real practical problem, indeed if there is no intersection the rules are simply randomly generated as in the previous case.

With this implementation the generated rules are able to match at least one report more often.

```python
def generate_from_common_part(reports, genome_length):
    # creating the intersection-rule from the given reports
    base_rule = get_rule_with_all_common_attributes(reports)
    new_rule_list = list()

    for i in range(genome_length-1):
        base_rule_len = len(base_rule)

        while new_rule.evaluate() < YaraRule.THRESHOLD
                and base_rule_len > 0:
            # extracted from base_rule not present in new_rule
            attribute = _get_random_attribute(base_rule, new_rule)
            base_rule_len -= 1
            new_rule.add(attribute)

        while new_rule.evaluate() < YaraRule.THRESHOLD:
            # extracted from reports not present in new_rule
            attribute = _get_random_attribute(reports, new_rule)
            new_rule.add(attribute)
            is_base_rule = False

        new_rule_list.append(new_rule)

    return new_rule_list
```

Listing 5.6: Rules generated with a random approach starting from their intersection

Pseudo-greedy generation

Even if the rules are generated from a common part the random-added attributes bring the rule to become often unable to match a single report.

To try to fix this problem we decide to check if the random generated rules are able to match at least something. Furthermore we decided to does not consider anymore the scores of the generated rules, we decided only to generate some small rules able to match something; the best combination of this small rules will be selected by the evolutionary algorithm. We decided also to explicitly check if a generated rule is equal to a previous-generated one, if so, the generated rule is discarded and it does not be added to the set of rules used by the genome.

```python
def generate_pseudo_greedy(reports, genome_length):
    new_rule_list = list()
    new_rule = YaraRule()
    i = 0
    while i < genome_length:
        attribute = _get_random_attribute(reports, new_rule)
        new_rule.add(attribute)
        # how many reports are detected by the new rule?
        detected = {c for c in reports if c.match(new_rule)}
        if len(detected) == 0:  # try again
            new_rule.remove(attribute)
            continue

        append = True
        for r in new_rule_list:
            if new_rule == r:
                append = False
                break
        # new_rule added only if not a duplicate
        if append:
            new_rule_list.append(new_rule)
            new_rule = YaraRule()
            i += 1

    return new_rule_list
```

Listing 5.7: Pseudo-greedy rule generation approach

Power set rule generation

To increase even more the space of the possible solutions we decided to do not use heuristics strategies in the rule generations but to leave all the rule-optimization to the SGX library.

To have a rule set the most generic as possible we decide to generate a *Power Set* from each Koodous report given as input parameter and merge together all the obtained clauses. On the other hand, to do not have too many clauses to be managed by the evolutionary algorithm, we decided to keep only the clauses with a score contained between the two thresholds: $T_{min} < score < T_{max}$

We defined the power set of the clause C as the set of all subsets of C, excluding the empty set and including C itself. The main difference between the classical mathematical power set operation was *the absence of the empty set.*
So, for example if C contains three attributes (x, y, z), its power set will be:

$$\mathcal{P}(C) = \{\{x\}, \{y\}, \{z\}, \{x, y\}, \{x, z\}, \{y, z\}, \{x, y, z\}\}$$

In listing 5.8, it is possible to see the method developed for the generation of the power set and the function that uses that method to generate the list of YARA rules. To introduce more flexibility to the rule generation phase, the get_powerset() method has also two parameters that could be used to specify the minimum and maximum cardinality of the generated power set. For example, if C contains four attributes (x, y, z, w) and we call the method `rule.get_powerset(2,3)` the result will be:

$$\mathcal{P}(C_{2,3}) = \{\{x, y\}, \{x, z\}, \{x, w\}, \{y, z\}, \{y, w\}, \{z, w\}, \{x, y, z\}, \{x, y, w\}, \{x, z, w\}, \{y, z, w\}\}$$

```
def get_powerset(YaraRule, min_size=1, max_size=None):
    xs = list(YaraRule)
    if not max_size max_size > len(xs):
        max_size = len(xs)
    pow_set = chain.from_iterable(combinations(xs, n)
                        for n in range(min_size, max_size+1))
    return pow_set


def powerset_rule_generation(reports):
    clause_set = set()
    i = 0
    for report in reports:
        i += 1
        powerset = report.yara_rule.get_powerset()
        for clause in powerset:
            # converting the tuple "clause" in a YaraRule object
            new_rule = YaraRule(clause)
            score = new_rule.evaluate()
            # only "valid" scores
            if score >= YaraRule.THRESHOLD
                    and score <= YaraRule.UPPER_THRESHOLD:
                # normal set can't be added to another set
                immutable_rule = frozenset(clause)
                # duplicates automatically removed
                clause_set.add(immutable_rule)

    # converting the set into a list
    return list(clause_set)
```

Listing 5.8: Power set rule generation

Unfortunately, very often the size of the power set became quite big freezing the execution. To solve this problem we changed the `get_powerset()` method to another one that uses Python *Generator functions*. A generator it is simply a function that build a Python *Iterator object* automatically. Thanks to the iterator, it is not necessary to generate the whole power set in one shot, but it is possible to retrieve from the object one element at time without saturating the memory.

After few attempts, it became clear that even with the possibility to reduce the cardinality of the power set, too many elements will be generated; due to this fact, we decide also the introduce the possibility of reducing the interval of score considered acceptable with the idea of taking a sub-part of the previous considered interval $[T_{min}, T_{max}]$.

As it is possible to see in listing 5.9, this change requires also to slightly change the logic of the calling function.

```
def powerset_generator(self, min_size=1, max_size=None):
    n = 0
    xs = list(self)
    if not max_size max_size > len(xs):
        max_size = len(xs)

    for subset in chain.from_iterable(combinations(xs, n)
                        for n in range(min_size, max_size+1)):
        yield YaraRule(subset)



def powerset_rule_generation(reports, min_len, max_len,
        min_score=YaraRule.THRESHOLD,
        max_score=YaraRule.UPPER_THRESHOLD):
    clause_set = set()
    i = 0
    for report in reports:
        i += 1
        r = report.yara_rule
        for clause in r.powerset_generator(min_len, max_len):
            score = clause.evaluate()
            if min_score <= score <= max_score:
                immutable_rule = frozenset(clause)
                clause_set.add(immutable_rule)

    return list(clause_set)
```

Listing 5.9: Power set rule generation through generator function.

First Best Matching generation

Even if we want to have a quite big initial set of YARA rules, it was quite impossible to find good values to assign to the generator function to achieve a good trade-off between results and computational time. To try to solve this problem we decided to create the desired initial pool selecting the clauses with the highest matching rate as possible.

```
def new_generation_function(reports):
    clause_set = set()
    want_to_match_number = len(reports)

    while want_to_match_number > 0:
        for report in reports:
            r = report.yara_rule
            for clause in r.powerset_generator(min_size=3):

                detected = {c for c in reports if c.match(clause)}
                # keep only best literals
                if len(detected) == want_to_match_number:
                    eval = clause.evaluate()
                    if YaraRule.THRESHOLD <= eval
                          ane eval <= YaraRule.UPPER_THRESHOLD:
                        immutable_rule = frozenset(clause)
                        clause_set.add(immutable_rule)

        if len(clause_set) == 0:
            want_to_match_number -= 1

    return list(clause_set)
```

Listing 5.10: First Best Matching generation

Note that the method `powerset_generator()` has `min_size = 3` because, with the current available literals score, the maximum possible value for a literal is 150. Considering that the value of T_{min} is fixed on 400, we can assume that is impossible to generate a clause with less than 3 literals. We introduce this small characteristic to reduce a little bit the possible generated clauses.

Best Matching from intersections of the rules

The main problem of the previous approach was that a lot of rules are generated and discarded because they do not have the desired amount of matching (i.e. the `want_to_match_number` variable shown in listing 5.10). So, to generate less rules and to improve the execution speed of this part of the algorithm, we decided to slightly change the generation logic.

As it is possible to see in the following listing (5.11), we reintroduced the concept of *intersection rule*; the function starts intersecting all the report together, then a power set was expanded from the so-generated `common_rule`. This process is reiterated for all the possible combination of reports removing initially just one report and later on two of them, three of them, and so on until all the reports minus one are removed or until the counter `i` reach the specified parameter (`to_remove`).

Generating the intersection rule from all the reports we will have the warranty that the generated rule will match all the reports, using n-1 reports will know that the rule will match all the reports minus one, and so on. In this way we are able to describe the same approach represented in the previous section improving considerably the rule-generation performance.

```python
def iterative_intersection_generation(reports, to_remove=None):
    clause_set = set()
    i = 0
    if to_remove is None:
        to_remove = len(reports)

    while i <= to_remove:
        for report_list in
                itertools.combinations(reports, len(reports)-i)
            common_rule = get_intersection_rule(report_list)
            for c in common_rule.powerset_generator(min_size=3):
                score = c.evaluate()
                if YaraRule.THRESHOLD <= score
                        and score <= YaraRule.UPPER_THRESHOLD:
                    immutable_rule = frozenset(c)
                    clause_set.add(immutable_rule)
        i += 1

    return list(clause_set)
```

Listing 5.11: Best matching using several intersection rule.

To improve performance, the previous function was later on improved with few updates:

- in the beginning of the function was added an additional check: if at least a *perfect* *rule* was found, the `while` loop is not executed and the perfect rules are immediatly returned;

- if an *intersection rule* is bigger than a specified value its power set is not generated;
- it was introduced a number of required clauses that, if reached, break the loop;
- it was introduced a sorting algorithm to return only the best clauses if they are too many;
- several reports were intersected before generating the power set; the intersection was break when the `common_rule` reach length specified as parameter.

5.3.1.4 Other implementation details

In this section will be introduced some concept worthy of note even if they are not the main concept of the SGX rule generator.

Reducing resulting ruleset

In some experiments we noticed that it is possible to reduce heuristically the number of generated rules. For example, if it is generated a ruleset starting from a cluster of five application, it is expected that, in the worst case, five rules are generated: each one of these five rules able to match one of the five reports. Unfortunately sometimes, using the SGX-RG algorithm, more than five rules are generated. To solve this problem it was developed a mechanism able to reduce the number of generated rules without reducing the effectiveness of the final signature. This mechanism should be placed between rule generation phase and rule optimization phase.

```python
def reduce_ruleset(reports, rules):
    if len(rules) > len(reports):
        global_detected = set()
        for rule in rules:
            local_detected = {c for c in reports if c.match(rule)}
            global_detected.update(local_detected)
        whole_rules_matches = len(global_detected)

        global_detected = set()
        tmp_global_detected = set()
        reduced_rule_list = list()
        for rule in rules:
            tmp_global_detected.update(
                        {c for c in reports if c.match(rule)})
            if len(tmp_global_detected) > len(global_detected):
                global_detected.update(tmp_global_detected)
                reduced_rule_list.append(rule)
                if len(global_detected) == whole_rules_matches:
                    rules = reduced_rule_list
                    break
    return rules
```

Listing 5.12: This function reduce the size of the final signature generated.

Evolutionary main function

The main steps necessary to generate a signature through the SGX library are the following:

1. creation of a pool of YARA rules according to one of the methodologies explained in 5.3.1.3;
2. the array of loci is instantiated;
3. for each locus, the score and the number of detections of its related rule was precomputed;
4. creation of the list of *perfect rules*;
5. if a perfect rule was found, the evolutionary algorithm is not triggered;
6. the genome object is generated through the class `AlleleDistribution`;
7. the optimizer was instantiated with all the necessary arguments;
8. the evolutionary process is started;
9. the resulting genome is converted in a list of YARA rules that is returned to the caller.

In the following listing (5.13) it possible to see the simplified code of the main function that generates the rules and evolves them using the SGX library.

```
def yyg_sgx(reports, args):
    global YARA_RULE_LIST
    global YARA_RULE_SCORES
    global YARA_RULE_DETECTED

    loci = list()
    perfect_rule_list = list()
    YARA_RULE_LIST = rule_generation(reports)

    for rule in YARA_RULE_LIST:
        ...
        loci.append(sgx.LocusEnum([0,1])

        # Scores pre-computation
        score = rule.evaluate()
        YARA_RULE_SCORES.append(score)

        # Detection rates pre-computation
        detected = {c for c in reports if c.match(rule)}
        YARA_RULE_DETECTED.append(detected)

        if len(detected) == len(reports):
            perfect_rule_list.append(rule)

    if len(perfect_rule_list) > 0:
        return get_best_rule(perfect_rule_list)

    # initialize and start SGX
    genome = sgx.AlleleDistribution(loci)

    opt = sgx.VanillaOptimizer(genome, fitness_function,
                               max_generations=args.max_gen,
                               mutation_probability=args.mut_prob)
    opt.evolve()

    final_genome = opt.archive.pop().genome
    to_return_list = list()
    for i in range(len(YARA_RULE_LIST)):
        if final_genome[i]:
            to_return_list.append(YARA_RULE_LIST[i])

    return to_return_list
```

Listing 5.13: The main function of the evolutionary algorithm.

5.3.2 SGX Rule Optimizer

Given the promising results of the *Basic Optimizer* and due to the high computational overhead to generate a signature from scratch, we decide to take advantage of the SGX library to introduce a new class of rule optimizer: the SGX Rule Optimizer (SGX-RO).

In this section will be introduced how we decided to represent the final product of the evolutionary process and how to compute the new fitness function. In particular, compared to the SGX-RG, the main differences in using the library are:

- it was not necessary to generate a set of rules to associate to the genome. The SGX algorithm has already a starting point to evolve;
- in this case the final product of the evolutionary computation will be only the optimized rule, not the whole signature.

5.3.2.1 Representing a YARA rule in SGX-compatible mode

Each locus contains an allele with a binary value (0/1). As we already said in section 5.3.1.1, even if the Selfish Gene is an *EDA*, in our case it was necessary to associate a pre-computed data structure to the genome to perform an appropriate evaluation. Due to the fact that this time we want to optimize a YARA rule, it was quite natural to associate to the genome the whole YARA rule. This means that each locus of the genome will contain an attribute of the rule itself; if the value of the allele will be 1, the attribute will be used in the optimized rule, if it will be 0 the value will be ignored.

Unlike the previous case, we do not need a new mechanism to evaluate the single attribute or the whole genome because it was already defined a specific value for each possible attribute and furthermore, we had already defined the `evaluate()` method of the class `YaraRule` to perform an evaluation of the single individual.

5.3.2.2 Fitness function

Lexicographic Fitness

The first developed fitness for the SGX-RO was a lexicographic fitness. As we already did for the lexicographic fitness of the SGX-RG algorithm, it was necessary to establish what are the most important parameter of our problem and if we want to maximize or minimize them. This time, from the most to the last important, these parameters are:

1. the number of reports matched by the optimized YARA rule (to be maximized);
2. the score of the optimized YARA rule (to be minimized — but still greater than T_{min});
3. the number of attributes contained inside the optimized YARA rule (to be minimized).

This kind of fitness, as we already said in section 5.3.1.2, was implemented using an extension of the class `tuple` offered by the SGX library: the class `FitnessLexicographic`.

In this case, we did not have a generation phase, so it is not possible to assume that, during the computation of the fitness value, the evaluated individual was properly configured. For this reason, if compared with the lexicographic fitness presented inside section 5.3.1.2, we had to add a penalty if the individual had a score not contained between the two thresholds (T_{min} and T_{max}). We decided to introduce a huge penalty if the rule becomes to much general going under T_{min} (the algorithm assigns the worst values to the first two attributes of the fitness), and a slightly less huge penalty if the rule could became to much specific going beyond T_{max} (worst value to the first attribute, half worst value to the second one).

Furthermore, due to the fact that we wanted to minimize the number of attributes but we did not want to fall in wrong configurations, if the genome contains only zeros the parameter `attribute` is set to the worst possible value: the length of the whole genome.

```
def optimizer_fitness_lex(genome, rule, reports):
    # counting literals
    attributes = sum(genome)
    if attributes == 0:
        attributes = len(genome)

    # rule creation from the individual
    rule = YaraRule()
    for i in range(len(genome)):
        if genome[i] != 0:
            rule.add(rule[i])

    # count matches
    matched = {c for c in reports if c.match(rule)}
    m = len(matched)

    # score evaluation
    score = rule.evaluate()
    if score < YaraRule.THRESHOLD:
        score = sys.maxsize
        m = 0
    elif score > YaraRule.UPPER_THRESHOLD:
        score = sys.maxsize / 2
        m = 0

    return sgx.FitnessLexicographic((m, 1./score, 1/attributes))
```

Listing 5.14: Fitness function of SGX Rule Optimizer that uses a `FitnessLexicographic`.

Heuristic Fitness Function

The rules generated by the previous approach, even if their score was always in the prefixed interval, were often considered unacceptable for a human expert. In particular, after several tests we observed that very often the rules generated by this optimizer contains few literals with a high score; for example, when it is possible, SGX-RO gives as result rules with three attributes of score 150 (like the name of a *service*) or with four attribute of score 100 (like a *URL*). Generally, human experts prefers having rules that includes a widespread range of different attribute to reduce the probability that their rules match fake positive samples.

For this reason, we decided to introduce some heuristic strategies in the selection process of our optimizer. To implement this approach it was not possible to use anymore the `FitnessLexicographic`, so we developed a new fitness class called `FitnessYaYaGen` and we introduced inside its comparison methods (`__eq__` and `__lt__`) some heuristic rules. So, when two YARA rules are compared:

- if one of them contains only URL is worst than the other one;
- if one of them has the functionality *SSL* is worst than the other;
- if one of them has less categories than the other is worst;
- if one of them has less *functionality* than the other is worst.

These measure are established in an empirical way, what we wanted to achieve was a system able to simulate the decision that will be taken by a YARA rule expert. For this reason, as is shown in listing 5.15, when it is possible the comparison are not performed in an hard way but they state that a rule is better than another one only if the difference between that particular attribute is beyond a certain threshold. So, for example, it is necessary that one rule has 10% more attributes than the other one to be considered better and, furthermore, it is necessary that a rule has at least 3 categories more than the second one to be considered better.

Note that this approach does not want to create a new absolute lexicographic order, it should be able just to establish which rule is better between two different rules, nothing more. This is possible because SGX library does not need to have an absolute ordering to perform successfully, it has to know just what is the better individual when two candidate solutions are compared. This new comparison mechanism makes lose the transitive property to the rule evaluated through this new fitness function.

Before using these heuristic rules, the algorithm will check if the number of matches are equal or not; this check is executed for two difference reasons:

- the first one is an implementation trick: considering how it is instantiated the `FitnessYaYaGen` object, if a rule has an invalid score, it receive a particular score — `sys.maxsize` if the score is under T_{min}; `sys.maxsize/2` if the score is beyond T_{max}. Thanks to this value it is very easy to establish which rule is better if one of them is not inside the range of valid scores;

- the second one is the possibility to create a more powerful rule; if a rule matches more reports than the other one it should be considered as more powerful rule.

```
class FitnessYaYaGen(Fitness):
    _TOLERANCE = 10  # percentage

    def __eq__(self, other):
        ...

    def __lt__(self, other):
        # 0 -> report matching
        if self[0] != other[0]:
            return self[0] < other[0]

        differences = list()
        # 1 -> URL
        if self[3].only_url() and not other[3].only_url():
            # self YaraRule is worst
            differences.append(1)
        elif not self[3].only_url() and other[3].only_url():
            # other YaraRule is worst
            differences.append(-1)

        # 2 -> categories
        cat_diff = other[3].categories() - self[3].categories()
        diff = cat_diff * 100 / YaraRule.ATTRIBUTES
        if abs(diff) > FitnessYaYaGen._TOLERANCE:
            if diff > 0:
                differences.append(1)
            else:
                differences.append(-1)

        # 3 -> functionalities
        func1 = self[3].functionalities()
        func2 = other[3].functionalities()
        diff = func2 - func1
        if diff > 3:
            differences.append(1)
        elif diff < -3:
            differences.append(-1)

        # 4 -> funcionality = SSL
        to_append = 0
        for att, val in self[3]:
            if att.startswith('androguard.functionality.ssl'):
                to_append += 1
                break
        for att, val in other[3]:
            if att.startswith('androguard.functionality.ssl'):
                to_append += -1
                break
        differences.append(to_append)

        # result
        if sum(differences) > 0:
            return True    # self rule is worst than other rule
        elif sum(differences) < 0:
            return False   # self rule is better than other rule

        # impossible use heuristics
        if self[1] == other[1]:
            if self[2] == other[2]:
                return False   # The rules are equally good
            else:
                return self[2] > other[2]   # to minimize
        else:
            return self[1] > other[1]       # to minimize
```

Listing 5.15: Fitness class developed specifically for the SGX-RO.

Note: in the previous listing the sum of the values contained inside the list `differences` is to be considered as a "negative score": when a rule has a characteristic that is worst than the characteristic of the other one a *+1* is assigned. In the end if the sum is a positive value, it means that the `self` rule is worst than the other one, on the other hand, if the sum is a negative value, the `self` rule is worst than the other one. If the sum is equal to zero it is not possible to determinate which rule is better according to these heuristic strategies, so the algorithm uses firstly the *scores* of the candidate YARA rules and then the *number of literals* to discriminate which rule is better. Due to the fact that we want to minimize the score of the best solution, it is possible to state that a rule is lower than another if its score is greater than the score of the other rule. The same concept is applied also for the number of literals.

The fitness function itself is quite identical to the one shown in the listing 5.14, the only difference is in the last row that was changed in this way `return FitnessYaYaGen((m, score, attributes, rule))`.

As we already said, the new tuple `FitnessYaYaGen` introduces also new comparison methods, for this reason it is not necessary anymore to use the reciprocal of the score and the reciprocal of the number of attributes that are now simply inserted inside the constructor of the object `FitnessYaYaGen`. Furthermore, the new comparisons requires to have access to more information about the representation of the YARA rule, for this reason the related object `YaraRule` was a required parameter for the constructor of this new object.

Additions to `YaraRule` class

To develop this new class, was also necessary to add new functionalities to the class YaraRule, in particular in the previous listing are shown the methods:

- `only_url()`, it returns a Boolean value to state if the YARA rule contains only URL or not;
- `categories()` returns the number of different categories of attributes contained inside the YARA rule (e.g. activity_name, presence of a certificate, requesting a permission, using a functionality, and so on);
- `functionalities()` returns the number of functionalities invoked inside the APK from which is extracted the YARA rule.

5.3.2.3 Biscardi Optimizer

Inside the `VanillaOptimizer` — the default optimizer provided by the SGX library — the candidate solution are evaluated using a *Pareto front*. When two candidate solutions are generated and compared, these new solutions are put in competition with a set of solution that are considered equally good. This pool of individuals is called `archive` an it contains all the individuals that are candidate to be returned as final solution. To become part of the archive, the fitness value of a candidate solution have to be greater or equal to all the other solutions contained in the archive in the moment of its generation.

```python
def pareto(individuals):
    pareto_set = set()
    for i in set(individuals):
        if all([not j.fitness.dominate(i.fitness)
                    for j in individuals]):
            pareto_set.add(i)

    return pareto_set
```

Listing 5.16: Computation of the pareto front.

To make work this mechanism, it is necessary to have an absolute ordering between all the solutions; this concept is no more feasible after the introduction of the heuristic fitness class `FitnessYaYaGen`. For this reason, we decided to implement a new optimizer that chose what individuals kept in the archive using a tournament-based approach.

After the comparison of the two candidate solutions, they will be still inserted inside the archive but the comparison is no more strictly based only on the fitness function. Each individual inside the pool is compared with all the other ones according to comparison methods of the class `FitnessYaYaGen` and according to the result of the comparison it receives:

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

This mechanism is executed in a round-trip way, so all the couple of individuals are compared two times. In the end, the individual that is kept inside the archive is the one that has the highest score; if there are more than one individual with the same score, they are all kept inside the archive [3].

In the following listing (5.17), it is possible to see the function that computes the archive and the pseudo-code of the new optimizer. The code of the optimizer is almost the same of the already provided `VanillaOptimizer`, the only significant change is the `biscardi` function, the function used to compute the archive.

```
WIN_SCORE = 3
TIE_SCORE = 1

def biscardi(individuals):
    scores = dict()
    for i in individuals:
        scores[i] = 0
    for i1 in set(individuals):
        for i2 in set(individuals):
            if i1 != i2:
                if i1.fitness > i2.fitness:
                    scores[i1] = scores[i1] + WIN_SCORE
                elif i1.fitness < i2.fitness:
                    scores[i2] = scores[i2] + WIN_SCORE
                else:
                    scores[i1] = scores[i1] + TIE_SCORE
                    scores[i2] = scores[i2] + TIE_SCORE

    max_value = max(scores.values())
```

[3] We decided to named this optimizer *Biscardi Optimizer* due to the fact that this mechanism is very similar to the group stage of the soccer tournaments and to pay tribute to a very famous Italian soccer journalist: Aldo Biscardi.

```
    biscardi_set = set()
    for i in individuals:
        score = scores.get(i)
        if score == max_value:
            biscardi_set.add(i)

    if len(biscardi_set) is 0:
        logging.error("The␣biscardi␣set␣is␣empty!")

    return biscardi_set


class BiscardiOptimizer(VanillaOptimizer):
    def evolve(self):
        ...

        new_archive = biscardi(self._archive | {i1, i2})
        ...
```

Listing 5.17: Biscardi Optimizer and computation of the archive

5.3.2.4 Implementation details

This section illustrates some interesting implementation details related to the SGX-RO.

Steady State mechanism

The steady state mechanism is useful whenever the SGX algorithm is able to reach a good solution before the fixed maximum number of generations (`max_generations`). If after a certain number of generation, arbitrarily configurable by the user, the algorithm is not able to find a better solution than the ones that are already present inside the archive the evolutionary process was interrupted and the best individual found until that moment is returned as solution.

```
def _steady_state_function(optimizer, gen_without_improve):
    to_stop = False
    for i in optimizer.archive:
        if optimizer.generation - i.birth > gen_without_improve:
            to_stop = True
            logger.warning("Steady␣state␣reached!")
            break

    return to_stop
```

Listing 5.18: Code to implement a steady-state mechanism.

The function shown in the previous listing (5.18) is evaluated after every generation during the execution of the `evolve()` method, contained inside the optimizer object.

After several experiments we observed that this mechanism is able to improve substantially the execution speed unless the initial configuration was to much distant from an optimal solution.

Automatic Initial Probability Computation

During the testing phase, we observed that the length of the genome influences the goodness of the solution and the execution time of the evolutionary computation. After some tests we noticed that this execution time could be influenced by the initial probability of having bits set to 0. In particular we saw that:

- if a genome is quite short, if the probability of having alleles set to 0 was lower, the execution time will be reduced;
- if a genome is quite long, if the probability of having alleles set to 0 was higher, the execution time will be reduced.

Furthermore, we noticed that the execution time will be particularly reduced if this trick was used together with the implementation of the *Steady State*.

For this reason, even if we decided to leave to the user the possibility to configure his own *initial probability* of having bits of the genome set to 0, we implemented a simple mechanism to automatic compute this value. Due to the fact that under no circumstances we want to have this probability configured to the absolute denial (0%) or to the absolute certainty (100%), we set also two limits values: the minimum possible probability of zero is 0.03, meanwhile the maximum possible probability of zero is 0.97. To summarize it is possible to say that:

- if the length of the genome is lower or equal to 10 alleles, $IP(0) = 0.03$
- if the length of the genome is greater or equal to 160 alleles, $IP(0) = 0.97$
- for all the other length of the genome, the initial probability of 0 is set according to the following formula:

$$IP(0) = (length(genome) - 10)\frac{0.97 - 0.03}{160 - 10} + 0.03$$

Thanks to this simple mechanism, especially if used in combination with the already cited *Steady State* mechanism, the algorithm is able to converge to a better solution in a lower amount of time. In the following listing is it possible to see how it was coded this mechanism.

```
MIN_PROB = 0.03
MAX_PROB = 0.97
MIN_LEN = 10
```

```
MAX_LEN = 160

def calculate_init_prob(length)
    if length <= MIN_LEN:
        return MIN_PROB
    elif length >= MAX_LEN:
        return MAX_PROB
    else:
        diff_prob = MAX_PROB - MIN_PROB
        diff_len = MAX_LEN - MIN_LEN
        return (length - 10) * diff_prob / diff_len + MIN_PROB
```

Listing 5.19: Automatic Initial Probability Computation.

SGX-RO main function

The main steps necessary to optimize a YARA rule are the following:

1. if not specified by the user, the initial probability of having bit set to 0 is automatically computed as it was explained in section 5.3.2.4;
2. the array of loci is instantiated;
3. the genome object is generated through the class `AlleleDistribution`;
4. the optimizer was instantiated with all the necessary arguments;
5. the *stay state* stopping condition is added to the optimizer;
6. the evolutionary process is started;
7. the resulting genome is converted in a `YaraRule` object that is returned to the caller.

In the following listing (5.20) it possible to see the simplified code of the main function of the SGX Rule Optimizer.

```
def sgx_rule_optimizer(rule, reports, args):
    if args.init_prob is None:
        args.init_prob = calculate_init_prob(len(rule))

    loci = list()
    listed_literals = list()
    for literal in rule:
        loci.append(sgx.LocusEnum([0, 1]), args.init_prob)
        listed_literals.append(literal)
```

```
genome = sgx.AlleleDistribution(loci)
# VanillaOptimizer or BiscardiOptimizer
opt = sgx.Optimizer(genome, optimizer_fitness,
                    max_generations=args.max_gen,
                    mutation_probability=args.mut_prob)
opt._stopping_conditions.append(lambda:
        _steady_state_function(opt, args.steady_state))

opt.evolve()

new_rule = YaraRule()
final_genome = opt.archive.pop().genome
for i in range(len(listed_literals)):
    if final_genome[i]:
        new_rule.add(listed_literals[i])

return new_rule
```

Listing 5.20: Pseudo Code of SGX Rule Optimizer.

5.4 Further improvements

YaYaGen has generated a great interest within the malware research team of Koodous, for that reason it is a project in under continuous development inside our research team. There are several proposals of improvements, mostly dictated by the need of increasing the practical usability of the tool. Some of them includes:

- to change the way in which the tool is configured with the introduction of several configuration files;
- introduce a tool to check maliciousness of the URLs and of the IPs before introduce them in rule generation.

5.4.1 Configuration files

The goal of the configuration file is to help Koodous system administrator to manipulate YaYaGen after the introduction in *Koodous Brain* without knowing how the code works in details. This will introduce the possibility to customize YaYaGen according to future requirements without the risk of introducing new bugs.
The configuration files that will be introduced are:

- a file named *configuration.json* that will allows to enable Cuckoo support, specify Permission and Intent filters list, keywords and values files;
- a file named *keywords.json* that will be used at the pre-processing of the Koodous JSON reports to select which literal consider during the rule generation process;
- a file named *values.json* is used to specify the weight of each literal.

5.4.2 IP and URL filtering

The second improvement is due to the fact that URLs and IP addresses are very effective in detecting malicious samples, so it was established to augment their score from 100 to 200 adding a mechanism that will filter and check for maliciousness before including them in the set of literals used for the rule generation. The module incharged of pre-processing them will be *url_checker.py*, which firstly will filter common URLs using the *Alexa Top 1 million list* (alexa), and then will use the API of *Virus Total* — a free virus, malware and URL online scanning service — to check the domain for malicious traffic. In order to increase the query performances, results can be cached in a local database.

Chapter 6

Experimental Results

Given the research nature of the thesis, experiments focused on showing the feasibility of the approach, rather than implementing a production ready tool, for this reason the results of our experiments refers only the data-set of applications that we had to our disposal during the development phase. However we successfully performed experiments on a very large data-set of 1.3 million of Android applications collected during the period between November 2016 and November 2017.

Note, all the tests performed during this research activity were executed on a server equipped with a 4-core Intel i5 processor (i5-2500 CPU @3.30 GHz), 8GB of RAM, and running Ubuntu 16.04.3 LTS.

6.1 Basic Optimizer

The Basic Optimizer produces always good results converting each YARA rule in a more specific one, without affecting the capability of each clause to correctly detect malware.

An example of the generated rules on few testing clusters is shown in table 6.1. Note, the proposed results of the Basic Optimizer are an average of 10 tests executed on the same "non-optimized" YARA rule. This is due to the fact that this optimizer does not follow a deterministic process and the obtained result is different every time.

In the Table 6.1 are shown the results of the optimizer process starting from both the already existent rule generation algorithms (the greedy and the clot). The clusters belong to the different types already explained in section 4.2.3 and they are labelled according to that convention. So for example, the cluster labeled with the name *t2c1*, is the first analyzed cluster (c1) of type 2 (t2). The clusters chosen for this table belongs to all the most significant categories — we did not work with clusters of type 1 because they are already covered by other YARA rules and we did not work with clusters of type 7 because we do not have too much information about them — and they have very different scores. They were chosen to show how the Basic Optimizer is able to optimize both clauses very close to the threshold and clauses with a huge score.

If the resulting signature contains more than one clause, they are reported with a progressive number (clause 1, clause 2, ..., clause n). Next to the clause label, it is also

Cluster	Algorithm	Clause	Match	Literals	Score	Opt. Literals	Opt. Score
t2c1	Greedy	Clause 1	2/3	64	4263	10.30	608.70
		Clause 2	1/3	165	7526	13.80	592.60
	Clot	Clause 1	2/3	64	4263	9.40	603.10
		Clause 2	2/3	18	440	-	-
t2c2	Greedy		3/3	15	1145	8.00	651.50
	Clot		3/3	15	1145	8.70	630.00
t3c1	Greedy		4/4	260	17614	9.00	625.00
	Clot		4/4	260	17614	9.20	597.00
t3c2	Greedy		3/3	163	12015	8.80	597.30
	Clot		3/3	163	12015	9.10	616.10
t4c1	Greedy	Clause 1	3/4	64	2073	20.20	612.30
		Clause 2	1/4	59	1907	19.80	620.40
	Clot	Clause 1	1/4	59	1907	20.80	615.40
		Clause 2	3/4	64	2073	18.20	620.50
t4c2	Greedy		4/4	30	904	21.50	610.10
	Clot		4/4	30	904	20.90	611.20
t5c1	Greedy		4/4	157	12215	8.70	615.00
	Clot		4/4	157	12215	7.90	620.00
t5c2	Greedy	Clause 1	14/15	13	620	-	-
		Clause 2	1/15	91	2512	19.80	603.90
	Clot	Clause 1	3/15	21	400	-	-
		Clause 2	14/15	12	435	-	-
t6c2	Greedy		8/8	128	10725	7.70	608.50
	Clot		8/8	128	10725	7.20	619.50
t6c3	Greedy	Clause 1	7/20	20	421	-	-
		Clause 2	16/20	24	438	-	-
		Clause 3	1/20	37	750	30.50	617.50
	Clot	Clause 1	1/20	37	750	30.00	596.50
		Clause 2	18/20	22	408	-	-
		Clause 3	17/20	19	406	-	-

Table 6.1: Result obtained through Basic Optimizer

reported the capability of the clause to match the reports from which it was generated. As it is possible to see, the sets of clauses are always able to match all the reports from which they were generated.

Close to these values are also reported the number of literals and the score of each clause, together with the average number of literals and scores of the related optimized clauses.

6.2 SGX Rule Generator

Unfortunately, we do not have accurate results to shown for this rule generation approach. Due to the huge memory required to generate and manage the starting pool of YARA rules and due to a very long execution time that was considered not appropriate for an almost-real-time application, no one of the rule generation attempts explained in section 5.3.1.3 was able to reach a satisfactory result.

For this reason its developing process has been put on stand-by and we decided to concentrate our research activity on the improvement of an already generated rule through the optimizing phase that was giving good results. The knowledge acquired during the development of this rule generation algorithm was re-used for developing the SGX Rule Optimizer.

6.3 SGX Rule Optimizer

Until its firsts implementations, this optimization strategy was able to achieves good results and it was able to perform better than the Basic Optimizer. The YARA rules obtained through the SGX-RO have always a lower score than the ones optimized with the Basic Optimizer and, furthermore, the rule obtained with this evolutionary optimizer are usually more appreciated by human experts.

In the next section will be discussed the results of the last version of the SGX Rule Optimizer, the so-called *Biscardi Optimizer*.

6.3.1 Biscardi Optimizer

In Table 6.2 are shown the results provided by *Biscardi Optimizer*. Like in the Basic Optimizer case, the results shown in the following table are an average of 10 executions on the same "non-optimized" YARA rule. This is due to the fact that also the SGX Rule Optimizer does not provide deterministic results.

The table is composed by the same clusters proposed in the previous table (6.1), the only difference is that the columns *Optimized Literals* and *Optimized Score* show the results obtained through this new optimizer.

Cluster	Algorithm	Clause	Match	Literals	Score	Opt. Literals	Opt. Score
t2c1	Greedy	Clause 1	2/3	64	4263	21.00	400.00
		Clause 2	1/3	165	7526	17.40	440.00
	Clot	Clause 1	2/3	64	4263	21.70	409.30
		Clause 2	2/3	18	440	-	-
t2c2	Greedy		3/3	15	1145	5.00	405.00
	Clot		3/3	15	1145	5.00	406.00
t3c1	Greedy		4/4	260	17614	11.50	401.70
	Clot		4/4	260	17614	12.60	401.80
t3c2	Greedy		3/3	163	12015	10.30	408.30
	Clot		3/3	163	12015	10.10	406.50
t4c1	Greedy	Clause 1	3/4	64	2073	39.20	555.00
		Clause 2	1/4	59	1907	39.30	550.50
	Clot	Clause 1	1/4	59	1907	38.30	536.20
		Clause 2	3/4	64	2073	40.90	583.90
t4c2	Greedy		4/4	30	904	20.30	411.70
	Clot		4/4	30	904	19.60	400.60
t5c1	Greedy		4/4	157	12215	10.00	405.00
	Clot		4/4	157	12215	10.00	405.00
t5c2	Greedy	Clause 1	14/15	13	620	-	-
		Clause 2	1/15	91	2512	41.40	611.30
	Clot	Clause 1	3/15	21	400	-	-
		Clause 2	14/15	12	435	-	-
t6c2	Greedy		8/8	128	10725	21.80	412.00
	Clot		8/8	128	10725	22.70	425.50
t6c3	Greedy	Clause 1	7/20	20	421	-	-
		Clause 2	16/20	24	438	-	-
		Clause 3	1/20	37	750	29.20	405.20
	Clot	Clause 1	1/20	37	750	29.10	405.10
		Clause 2	18/20	22	408	-	-
		Clause 3	17/20	19	406	-	-

Table 6.2: Result obtained through SGX-RO (Biscardi)

6.4 Optimizers comparison

In the last table (6.3) the results of the two developed optimizers are compared. As it is possible to see, both the optimizers are able to generate a YARA rule that has less literals and a lower score than the relative non-optimized rule.

In particular, it is possible to observe that the rules generated by the Basic Optimizers have almost always a score greater than the rules obtained through the SGX Rule Optimizer. This is due to the fact that the first optimizer removes attributes until the score of the YARA rule is lower of the upper threshold. Considering that, in this implementation, the value of T_{max} is 650, and the attributes with the higher weight have a score equal to 150, it is clear that this optimizer can not produce a YARA rule with a score lower than 500. On the other hands, due to the fact that the evolutionary optimizer takes advantage of some heuristic constraints and, when it is not possible to use these constraints, try to minimize the score of the YARA rule more than it can remaining beyond the lower threshold, it is quite evident that will be produced some rules with a score closer to the configured value of T_{min} (which is 400 in this implementation).

Another noteworthy fact is that the rules optimized with the Basic Optimizer usually present a lower number of literals than the ones optimized using the SGX-RO. This, at a first glance, could be appear counter-intuitive, however this is not so strange because the heuristic constraints imposed by the SGX Rule Optimizer tend, on one hand, to penalize the rules composed by only URL — that are one of the most heavy attributes, with a score of 100 — and, on the other hand, to reward the rules with a wide range of different categories and with several *functionalities* — that in the current implementation have a score of 15. So, in the end, considering the behaviour of the evolutionary optimizer, it is not so strange that, on average, the rules generated through the SGX-RO have more literals than the ones produced by the Basic Optimizer.

Cluster	Alg.	Clause	Match	Not Opt.		Basic Optimizer		SGX Optimizer	
				Lit.	Score	Lit.	Score	Lit.	Score
t2c1	Greedy	Clause 1	2/3	64	4263	10.30	608.70	21.00	400.00
		Clause 2	1/3	165	7526	13.80	592.60	17.40	440.00
	Clot	Clause 1	2/3	64	4263	9.40	603.10	21.70	409.30
		Clause 2	2/3	18	440	-	-	-	-
t2c2	Greedy		3/3	15	1145	8.00	651.50	5.00	405.00
	Clot		3/3	15	1145	8.70	630.00	5.00	406.00
t3c1	Greedy		4/4	260	17614	9.00	625.00	11.50	401.70
	Clot		4/4	260	17614	9.20	597.00	12.60	401.80
t3c2	Greedy		3/3	163	12015	8.80	597.30	10.30	408.30
	Clot		3/3	163	12015	9.10	616.10	10.10	406.50
t4c1	Greedy	Clause 1	3/4	64	2073	20.20	612.30	39.20	555.00
		Clause 2	1/4	59	1907	19.80	620.40	39.30	550.50
	Clot	Clause 1	1/4	59	1907	20.80	615.40	38.30	536.20
		Clause 2	3/4	64	2073	18.20	620.50	40.90	583.90
t4c2	Greedy		4/4	30	904	21.50	610.10	20.30	411.70
	Clot		4/4	30	904	20.90	611.20	19.60	400.60
t5c1	Greedy		4/4	157	12215	8.70	615.00	10.00	405.00
	Clot		4/4	157	12215	7.90	620.00	10.00	405.00
t5c2	Greedy	Clause 1	14/15	13	620	-	-	-	-
		Clause 2	1/15	91	2512	19.80	603.90	41.40	611.30
	Clot	Clause 1	3/15	21	400	-	-	-	-
		Clause 2	14/15	12	435	-	-	-	-
t6c2	Greedy		8/8	128	10725	7.70	608.50	21.80	412.00
	Clot		8/8	128	10725	7.20	619.50	22.70	425.50
t6c3	Greedy	Clause 1	7/20	20	421	-	-	-	-
		Clause 2	16/20	24	438	-	-	-	-
		Clause 3	1/20	37	750	30.50	617.50	29.20	405.20
	Clot	Clause 1	1/20	37	750	30.00	596.50	29.10	405.10
		Clause 2	18/20	22	408	-	-	-	-
		Clause 3	17/20	19	406	-	-	-	-

Table 6.3: Results samples taken from both optimizers execution.

Chapter 7

Conclusions

The main idea behind the development of this thesis project was to improve the capabilities of YaYaGen using heuristic and evolutionary techniques.

We introduced some heuristic mechanisms to establish the goodness of a generated YARA rule: we assigned to each possible literal a value, and thanks to them we are able to estimate the goodness of each generated rule. We introduced also two thresholds (T_{min} and T_{max}) to establish respectively if a rule could be considered to much generic or to much specific.

To improve the rules already generated by YaYaGen we introduced a new phase, the *optimization phase*, in which the rules generated by the original version of YaYaGen are improved using both heuristic and evolutionary strategies.

Experimental results show that the rule optimization phase gives to YaYaGen the ability to generate more accurate rules, lowering both false positives and negatives.

Our proposed approach has generated a great interest within the malware research team of Koodous, for this reason, after a deeper testing phase, it will be soon integrated inside *Koodous Brain*, an artificial intelligence platform developed to assist Android malware detection directly in the Koodous project.

7.1 Future Developments

Even if this research activity has reached several satisfactory results, it is far from being considered concluded. YaYaGen has already gained new instruments and potentialities but they have to be expanded and empowered in the following months to give to the malware analysts a powerful tool to assist their research activities. Possible improvements on this research include:

- *Execute a deeper test on a large and updated data-set and try to improve the false positive/negative rates of the clustering process.*

  As YaYaGen may be powerful, each classification technique has always a rate of false positive/negative. After a deep test it will be necessary to understand and model how will be possible to reduce these rates.

- *An IP address is more discriminating than a URL.*

  During our tests we noticed that, sometimes, several different URLs are matched on the same IP address. Implementing a mechanism to take advantage of this particularity could be helpful and very useful.

- *Develope an heuristic optimizer.*

  Heuristic strategies are the ones that allows to create rules more similar to the ones that a human expert will write. Developing a third optimizer, entirely based on heuristic strategies, could gives to YaYaGen a new powerful tool to optimize its signatures.

- *Improving the SGX Rule Generation algorithm with heuristic strategies.*

  Instead of exploring a huge set of possibilities during the creation of the initial pool of rules used by the SGX algorithm for its optimization process, it could be a good idea generate a set of rule according to some heuristic strategies. In this way, time and resources used by the algorithm could be reduced and becoming acceptable.

- *Improve the function to establish the initial probability of 0 in the SGX-RO.*

  We noticed that the initial probability of 0 was a key element for the execution time and for the quality of the experimental results. Finding a precise mathematical function able to take advantage of this characteristic could improve even more computational time and results achieved.

Bibliography

[1] Victor M. Alvarez. *YARA Documentation.* 2014. URL: http://yara.readthedocs.io/en/v3.3.0/ (visited on 03/08/2018).

[2] Shumeet Baluja. *Population-based incremental learning. a method for integrating genetic search based function optimization and competitive learning.* Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Dept Of Computer Science, 1994.

[3] William J. Buchanan, Simone Chiale, and Richard Macfarlane. "A methodology for the security evaluation within third-party Android Marketplaces". In: *Digital Investigation* 23 (2017), pp. 88 –98. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2017.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1742287617300245.

[4] Thomas M Chen and Jean-Marc Robert. "The evolution of viruses and worms". In: *Statistical methods in computer security* 1 (2004).

[5] Chris Clark. *Automatic Yara Rule Generation on GitHub - Version 0.6.1.* Aug 2013. URL: https://github.com/Xen0ph0n/YaraGenerator (visited on 03/07/2018).

[6] Mike Cleron. *Android Announces Support for Kotlin.* 2017. URL: https://android-developers.googleblog.com/2017/05/android-announces-support-for-kotlin.html (visited on 02/27/2018).

[7] Fred Cohen. "Computer viruses: theory and experiments". In: *Computers & security* 6.1 (1987), pp. 22–35.

[8] F. Corno, M. Sonza Reorda, and G. Squillero. "The Selfish Gene Algorithm: a New Evolutionary Optimization Strategy". In: *SAC: ACM Symposium on Applied Computing.* 1998, pp. 349–355. URL: http://www.cad.polito.it/pap/db/sac98.pdf.

[9] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. "An improved cellular automata-based BIST architecture for sequential circuits". In: vol. 1. IEEE, 2000, pp. 76–79.

[10] Microsoft Corporation. *Market Share and Usage Analysis of File Sharing and Antivirus Report: June 2015.* December 2017. URL: https://support.microsoft.com/en-us/help/4013263/windows-10-protect-my-device-with-windows-defender-antivirus (visited on 04/01/2018).

[11] C. Darwin. *The Origin of Species.* P. F. Collier & Son, 1909. URL: https://books.google.it/books?id=YY4EAAAAYAAJ.

[12] R. Dawkins. *The Extended Phenotype: The Gene as the Unit of Selection*. Freeman, 1982. ISBN: 9780192860880. URL: https://books.google.it/books?id=uJCUAQAACAAJ.

[13] R. Dawkins. *The Selfish Gene*. Oxford paperbacks. Oxford University Press, 1989. ISBN: 9780192860927. URL: https://books.google.it/books?id=WkHO9HI7koEC.

[14] Artyom Dogtiev. *Mobile App Stores Guide 2017*. 2018. URL: http://www.mobyaffiliates.com/guides/mobile-app-stores-list/ (visited on 03/05/2018).

[15] Chris Doman. *Yabin repository on GitHub*. Jan 28, 2018. URL: https://github.com/AlienVault-OTX/yabin (visited on 03/31/2018).

[16] Neil DuPaul. *Static Testing vs. Dynamic Testing*. URL: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing (visited on 04/01/2018).

[17] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*. Vol. 53. Springer, 2003. ISBN: 978-3-662-05094-1.

[18] Paul Gerrard. *Lean Python: Learn Just Enough Python to Build Useful Tools*. Apress, 2016, p. xvi.

[19] Georges R Harik, Fernando G Lobo, and David E Goldberg. “The compact genetic algorithm”. In: *IEEE transactions on evolutionary computation* 3.4 (1999), pp. 287–297.

[20] Rowena Harrison and Keith Mayes. “Combating Android app repackaging attacks”. In: *Royal Holloway Information Security Thesis Series* (2015). URL: http://www.computerweekly.com/ehandbook/Combating-Android-app-repackaging-attacks.

[21] Google Inc. *Android Security 2017 Year In Review*. March 2018. URL: https://source.android.com/security/reports/Google_Android_Security_2017_Report_Final.pdf (visited on 03/30/2018).

[22] Google Inc. *Permissions Overview*. 2017. URL: https://developer.android.com/guide/topics/permissions/overview.html (visited on 02/28/2018).

[23] Google Inc. *The Google Android Security Team’s Classifications for Potentially Harmful Applications*. February 2017. URL: https://source.android.com/security/reports/Google_Android_Security_PHA_classifications.pdf (visited on 03/30/2018).

[24] OPSWAT Inc. *Protect my device with Windows Defender Security Center*. June 2015. URL: https://www.opswat.com/resources/reports/market-share-usage-analysis-file-sharing-antivirus-june-2015 (visited on 04/01/2018).

[25] Ari Juels, Shumeet Baluja, and Alistair Sinclair. “The equilibrium genetic algorithm and the role of crossover”. In: *Unpublished manuscript* (1993).

[26] Tobias Konradsson. *ART and Dalvik performance compared*. 2015.

[27] Michael Lones. “Sean Luke: essentials of metaheuristics”. In: *Genetic Programming and Evolvable Machines* 12.3 (2011), pp. 333–334. URL: http://www.springer.com/10710.

[28] Neil MacDonald. *Static or Dynamic Application Security Testing? Both!* URL: https://blogs.gartner.com/neil_macdonald/2011/01/19/static-or-dynamic-application-security-testing-both/ (visited on 04/01/2018).

[29] Andrea Marcelli. *YaYaGen Documentation.* 2018. URL: https://github.com/jimmy-sonny/YaYaGen (visited on 02/28/2018).

[30] Steve McConnell. *Code complete.* Pearson Education, 2004, p. 100.

[31] Sikorski Michael and Honig Andrew. *Practical Malware Analysis - The HandsOn Guide to Dissecting Malicious Software.* No Starch Press, 2012.

[32] Koodous Mobile. *Koodous Android application on Play Store.* URL: https://play.google.com/store/apps/details?id=com.koodous.android (visited on 03/09/2018).

[33] Geonbae Na et al. "Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART." In: *J. Internet Serv. Inf. Secur.* 6.3 (2016), pp. 27–39.

[34] Hyeong-Seok Oh et al. "Evaluation of Android Dalvik virtual machine". In: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems.* ACM. 2012, pp. 115–124.

[35] Jeremy Paquette. *A History of Viruses.* July 2000. URL: https://www.symantec.com/connect/articles/history-viruses (visited on 03/31/2018).

[36] Tim Peters. "PEP 20 - The Zen of Python". In: Aug. 2004. URL: https://www.python.org/dev/peps/pep-0020/.

[37] Guido van Rossum. "Origin of BDFL". In: *All Things Pythonic Weblog* (2008). URL: http://www.artima.com/weblogs/viewpost.jsp.

[38] Florian Roth. *How to Write Simple but Sound Yara Rules.* URL: https://www.bsk-consulting.de/2015/02/16/write-simple-sound-yara-rules/ (visited on 03/15/2018).

[39] Florian Roth. *How to Write Simple but Sound Yara Rules.* Feb 16, 2015. URL: https://www.bsk-consulting.de/2015/02/16/write-simple-sound-yara-rules/ (visited on 03/08/2018).

[40] Florian Roth. *yarGen (Yara Rule Generator) on GitHub - Version 0.19.0.* Feb 2018. URL: https://github.com/Neo23x0/yarGen (visited on 03/07/2018).

[41] A. Russo and A. Sabelfeld. "Dynamic vs. Static Flow-Sensitive Security Analysis". In: *2010 23rd IEEE Computer Security Foundations Symposium.* July 2010, pp. 186–199. DOI: 10.1109/CSF.2010.20.

[42] Hispasec Sistemas. *Koodous Documentation.* URL: https://docs.koodous.com/ (visited on 03/08/2018).

[43] Hispasec Sistemas. *Koodous Documentation - Getting Started - Basic steps.* URL: https://docs.koodous.com/yara/getting-started/ (visited on 03/08/2018).

[44] statista.com. "Android - Statistics & Facts". In: 2017. URL: https://www.statista.com/topics/876/android/.

[45] Virus Total. *YARA Documentation.* 2018. URL: https://virustotal.github.io/yara/ (visited on 02/28/2018).

[46] Alan M Turing. “Computing machinery and intelligence”. In: *Mind* (1950), pp. 433–460.

[47] Bill Venners. “The making of Python”. In: *Artima.com Interviews* (Jan. 2003). URL: http://www.artima.com/intv/pythonP.html.

[48] Andrea Villagra et al. “Multirecombined evolutionary algorithm inspired in the selfish gene theory to face the weighted tardiness scheduling problem”. In: *Advances in Artificial Intelligence–IBERAMIA 2004.* Springer, 2004, pp. 809–819.

[49] P Vinod et al. “Survey on malware detection methods”. In: *Proceedings of the 3rd HackersâĂŹ Workshop on computer and internet security (IITKHACKâĂŹ09).* 2009, pp. 74–79.

[50] John Von Neumann, Arthur W Burks, et al. “Theory of self-reproducing automata”. In: *IEEE Transactions on Neural Networks* 5.1 (1966), pp. 3–14.

[51] Feng Wang et al. “SGMIT: using selfish gene theory to construct mutualinformation trees for optimization”. In: *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation.* ACM. 2009, pp. 521–528.

[52] Feng Wang et al. “Using selfish gene theory to construct mutual information and entropy based clusters for bivariate optimizations”. In: *Soft Computing* 15.5 (2011), pp. 907–915.

[53] Haoyu Wang et al. “Wukong: A scalable and accurate two-phase approach to android app clone detection”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis.* ACM. 2015, pp. 71–82.

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

[55] Mu Zhang and Heng Yin. *Android Application Security: A Semantics and Context-Aware Approach.* Springer, 2016.

[56] Wu Zhou et al. “Detecting repackaged smartphone applications in third-party android marketplaces”. In: *Proceedings of the second ACM conference on Data and Application Security and Privacy.* ACM. 2012, pp. 317–326.