Android Malware Source Code Analysis

Master's Thesis in Cybersecurity, 2021/2022

Ramón Costales
Student
Department of Computer Science,
Universidad Carlos III de Madrid,
Madrid, Spain

Juan Tapiador
Advisor
Department of Computer Science,
Universidad Carlos III de Madrid,
Madrid, Spain

***Abstract*—Since the emergence of malware in the 1970s, these malicious programs have steadily increased in number and sophistication. The increasing profits generated by the use of malware have led to a growing demand, turning malware into a commodity of the underground economy. In this thesis, we analyze the evolution of Android malware from 2012 to date from a software engineering perspective. We analyze the source code of 97 samples from 83 unique families and obtain measures of their size, code quality, and estimates of the development costs (effort, time, and number of people). Our results suggest a linear increment per year in aspects such as number of malware samples and size, as well as a rapid increase in development cost. In terms of complexity and maintainability, we observe a low score compared to malware on other operating systems. Overall, our results are not conclusive enough to support claims about the increasing complexity of Android malware and its production progressively becoming an industry. This could be due to the fact that the Android malware industry is still young, as the operating system itself was launched just over ten years ago, or that there has been little change in the computer industry since the release of Android compared to the progress made in previous decades.**

***Index Terms*—Android; malware; source code analysis; software metrics**

I. Introduction

Every day, the mobile landscape grows in size. Recently, Google announced that Android had surpassed *three billion* users [1], securing the largest mobile market share for another year. Every year, the number of mobile users increases, causing malware to follow that trend. But malware is not only growing in the mobile landscape. As AVTEST's Malware Statistics clearly show [2], 2021 saw an extreme increase in new malware discovered, exceeding *150,000,000* new samples that year. Android is not far behind, as *3,000,000* new samples were discovered for this operating system.

A 2021 Report by Malware Bytes [3] confirmed that malware as a business is a growing trend, taking up more real estate in the cyber-threat landscape, making malware development more profitable. On Android, most malware developers fund their operations by generating ad revenue, while others deploy ransomware or large botnets for profit. It is also mentioned that stalkerware and spyware-type applications experienced a detection increase of *1,677%* in 2021, which is consistent with the types of malware present in our dataset. Some malware developers even leveraged the current global situation to use the COVID-19 pandemic as a cover to deploy malware and infect unsuspected victims; two samples in our dataset are ransomware disguised behind this facade.

As the number and profitability of Android malware increases, so does the sophistication and impact of attacks. In this thesis, we present a study of the evolution of Android malware from a software engineering approach. Our analysis is based on a dataset collected by the authors over several months and composed of the source code of 97 Android malware samples ranging from 2012 to 2022. Our dataset includes, among others, RATs, Trojan-Bankers, Keyloggers, Ransomware, Spyware, and Lockers. This is the largest Android malware source code dataset presented in the literature. We perform several analyses on this dataset. First, we review the most prevalent malware types and the most common permissions and capabilities used by the collected samples, as well as their antivirus detection rate. We also measure the evolution of malware development as a function of size, cost and quality.

Size dimensions are measured with several metrics, mainly the number of source files, the number of source lines of code (SLOCs), the number of functions, and the number of different programming languages used. Development cost is calculated with three estimates: effort, development time, and team size. Finally, code quality measures are computed using the cyclomatic complexity, the maintainability index, and the density of comments present in the code.

We then compare the results obtained with other similar works [4], [5] that performed the same measurements, but without specializing in a single platform, as we did with Android. This thesis is based on those works; we wanted to know if the results obtained were also applicable to Android malware.

To our knowledge, our work is the first to analyze the code evolution of Android malware from this perspective. We also believe that our dataset of Android malware source code is the largest analyzed in the literature.

The main findings of our work include:

- In the Android malware landscape there is a high tendency towards spying malware, such as Spyware, RATs, Trojan-Spy, Keyloggers, etc.

- The number of malware samples increases at a rapid rate every year.
- Antivirus detection rates are severely skewed towards Lockers, Trojan-Bankers, Ransomware, Rootkits, Keyloggers, and RATs, as the rest of the malware types hardly raised a single detection.
- There is a high annual increase in the number of source code files, SLOCs, functions and programming languages used.
- There is a big difference in the number of files, SLOCs, functions and programming languages between malware types, as Backdoors, Trojan-Bankers and RATs often outnumber the other types in some of these categories.
- Android malware samples have a high value of effort, development time, and team size, which is steadily increasing every year.
- Android malware samples have a low value of complexity, maintainability index, and comment ratio, which slowly decreases every year.

This paper is structured as follows. In Section II we describe our malware source code dataset and the capabilities of the malware contained in it. Section III presents measurements on the evolution of malware development, in terms of size, cost and quality. In Section IV we compare the results of our Android malware with those obtained from non-specific malware. In Section V we discuss the limitations of our approach, and future work that we have set aside. Section VI reviews previous similar work. Finally, Section VII concludes the paper.

II. Dataset

To perform the analysis, an Android malware dataset is required. Due to the nature of this analysis, the dataset must be composed of open source samples, so binary or decompiled sources are discarded. Gathering malware in this format can prove to be extremely challenging. Not only is it difficult to obtain malware samples as-is, but they are often shared in their binary format; in the case of Android malware, the APK is the usual means of distribution. For the malware source code to be publicly available, the sample may have been open-sourced from the beginning, voluntarily opened later in the development or leaked. In the case of the source code samples we managed to find, most of them were either used in the wild and subsequently leaked or open-sourced early on as a functional proof-of-concept (PoCs). We have spent several months searching for as much Android malware source code as we could find. For this reason, we believe that during this process we have assembled the largest collection of Android malware source code in existence.

To accomplish this task, we scoured *GitHub* [6] and public repositories for source code samples. One of the most interesting repositories is *MalwareSourceCode* [7], from *vx-underground* [8], as it contains many well-known leaked Android malware source code samples. It is likely that there are samples found in underground forums that have not been published on GitHub, but since we already had enough samples for the limited workforce we had, we did not venture into those forums. Nevertheless, it would be worthwhile to further this research with such samples to try to complete the dataset with them.

The dataset consists of malware samples. A sample is a specific variant of a malware project. There can be several variants of a specific malware project; they are said to share the same malware family. To distinguish between samples and variants, we have labeled each of them using a system similar to the CARO naming scheme [9]: **MalwareType:OperatingSystem/MalwareFamily.Variant**. As an example, one of the samples in our dataset is labeled as *Trojan-Banker:AndroidOS/Anubis.k*.

In our dataset, the samples are bundled in ZIP archives in a GitHub repository [10]. A sample can consist of one or several source code files. In the case of Android malware, that code usually corresponds to the logic and views behind the malicious application to be installed, but may also involve the C2 server to which the malicious app will connect or the type of payload to be sent in between, as well as web page code to load at runtime; it will depend on the type of malware and its capabilities. Since the code may serve different purposes and systems, the source code files may be written in multiple programming languages and have different directory structures, but they will usually all share the same structure for the code of the Android application itself.

The 97 samples that make up our final dataset have been labeled with a unique ID and tagged with a year and a generic description of their behavior. The year corresponds to the last date of their development when indicated by the source, otherwise with the year they were uploaded to the GitHub repository. They are also labeled with a coarse-grained malware type that best matches their behavior and fine-grained malware categories (called tags in this research) into which they also fit. Table I shows the distribution of malware types in our dataset, while Table II shows the frequency of different malware tags across all samples.

TABLE I: Distribution of Malware Types

Malware Type	**No. Malware Samples**
RAT	31
Spyware	13
Trojan-Spy	9
Keylogger	8
Trojan-Banker	7
Rootkit	5
Locker	4
Ransomware	4
Phishing	3
Trojan-SMS	3
Dropper	2
Trojan-Backdoor	2
Backdoor	1
Downloader	1
Password-Stealing-Ware	1
Scareware	1
Trojan	1
Trojan-Wiper	1

TABLE II: Frequency of Malware Tags

Malware Tag	No. Malware Samples
Spyware	72
Botnet	60
Backdoor	44
C2	44
Billing-Fraud	40
Trojan	35
RAT	34
Downloader	31
Elevated-Privilege-Abuse	27
Locker	19
Keylogger	17
Mailfinder	12
Wiper	12
Password-Stealing-Ware	11
Phishing	9
Encryption-Ransomware	8
Screen-Locking-Ransomware	8
Overlay	7
Clicker	6
Loader	6
Dropper	5
Rootkit	5
Rooting	3
DoS	2
Proxy	2
Spam	2
Scareware	1

To make these distinctions and correctly label each sample, we manually reviewed the source code of all samples and noted which malware tags they fit into and which individual malware type best described them, as well as the aforementioned description and other data, such as permissions and capabilities, which further aided in the classification of the samples.

As can be seen in Table I, most of the samples collected are related to spying on the victim. In fact, the five most abundant malware types in our dataset are spyware variants (RAT, Spyware, Trojan-Spy, Keylogger and Trojan-Banker). This fact should not come as a surprise, since smartphones are the most coveted targets for this type of malware, as the feedback obtained from these devices is richer than that of any desktop. After all, cell phone users always carry it everywhere, so the information obtained from the GPS location, camera or microphone will be more abundant and accurate. Another benefit of spying on smartphones is gaining access to calls and SMS (useful for 2FA, spam or billing fraud), instant messaging, contacts, and email and bank accounts, further enhancing the capabilities of mobile spyware compared to its desktop counterpart. Also noteworthy is the appearance of rootkits, lockers and ransomware (Android ransomware comes in two variants, both file-encrypting and device-locking).

The malware tags also reveal that nearly 75% have some spyware capability. It also shows that 60 of the 97 samples are used to create a botnet; 44 out of those 60 use a remote C2 server to manage it, while others use SMS, email or other means to administer their bots. It is also apparent that 40 samples can perform billing fraud by either sending SMS or

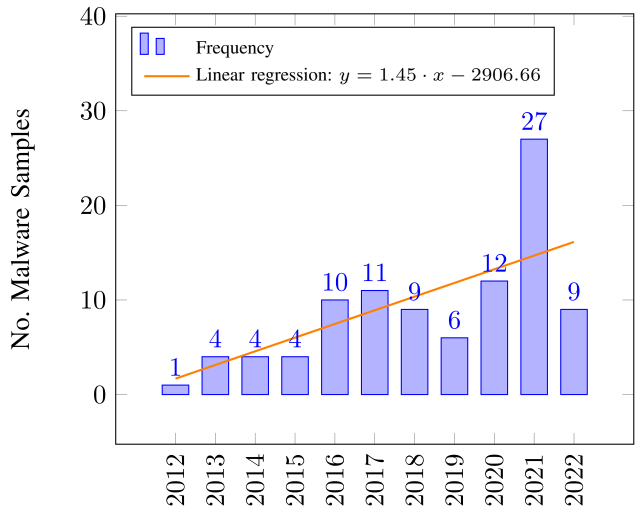

Fig. 1: Malware Distribution by Year

calls without the user's knowledge. 31 samples can download files from the Internet, some of which subsequently install those files as applications or are used to update the malware itself to gain more capabilities. It is also worth noting that 7 of the total samples perform overlay attacks, where the malicious app injects its own view on top of a legitimate one when the victim opens a specific app; this attack is mostly used by banking Trojans to overlay on top of legitimate banking apps and steal the victim's credentials.

Figure 1 shows the year distribution of the final dataset of 97 samples. Approximately 50% of the samples correspond to the last three years (2020-2022). The second largest set of samples consists of about 40% of the overall samples, corresponding to the period 2016-2019. Finally, the remaining samples, ranging form 2012 to 2015, account for less than 15% of all samples. It is interesting to note that almost 30% of all samples were developed in 2021. The linear regression clearly depicts the increase in the number of malware over the years, as it increases *1.45* samples each year. To measure how well the proportion of variation in the samples is described by the linear regression, we use the coefficient of determination, also known as R^2. This value ranges from 0 to 1; the closer it is to 1, the better the linear regression fits the sample values. In the case of the malware distribution by year, this coefficient has a value of *.47*, which means that approximately half of the variation can be explained by the regression.

Table III shows the most common permissions requested by the malicious apps that make up our dataset. There are 10 samples that had no permissions, as they are not Android applications (scripts written in Python, kernel rootkits in C, etc.), or whose *AndroidManifest.xml* file was not found. To extract the permissions automatically, a Python tool was developed [11]. It can also extract package names and actions from the manifest, and imports from the code, as they can give the researcher a general idea of the project's capabilities. It also has the ability to add information to each permission (description, group, group description, protection level and deprecation status), pulling it from a JSON file that we developed by aggregating public information and reports from *Android Developers* [12] and other sources in what is arguably the most extensive public Android permission database to date, containing both new and deprecated permissions from older Android versions. It is also capable of counting the frequency of permissions and imports across various projects and displaying a comprehensive graph with that information.

TABLE III: Frequency of Malware Permissions

Permission	No. Malware Samples
INTERNET	66
RECEIVE_BOOT_COMPLETED	55
WRITE_EXTERNAL_STORAGE	51
READ_SMS	47
ACCESS_NETWORK_STATE	45
READ_PHONE_STATE	44
READ_CONTACTS	44
SEND_SMS	39
WAKE_LOCK	35
ACCESS_FINE_LOCATION	35
RECEIVE_SMS	33
RECORD_AUDIO	31
READ_EXTERNAL_STORAGE	31
CAMERA	31
CALL_PHONE	26
READ_CALL_LOG	24
ACCESS_COARSE_LOCATION	21
SYSTEM_ALERT_WINDOW	20
BIND_ACCESSIBILITY_SERVICE	18
VIBRATE	16
BIND_DEVICE_ADMIN	16
GET_TASKS	14
ACCESS_WIFI_STATE	14
WRITE_SETTINGS	13
WRITE_CONTACTS	13
FOREGROUND_SERVICE	13
BIND_NOTIFICATION_LISTENER_SERVICE	13
WRITE_SMS	12
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	12
PROCESS_OUTGOING_CALLS	12
GET_ACCOUNTS	12
READ_HISTORY_BOOKMARKS	10
WRITE_CALL_LOG	10
BROADCAST_SMS	10

In this case, the most frequent permission in all samples is *Internet* access request. This permission is so common because, as seen before, most samples are spyware related, and therefore need to exfiltrate data remotely. The second most common permission is used to receive a notification when the boot process has finished. This is leveraged to launch the application each time the notification is received, therefore the app will always run at startup. This is done to add persistence; in the case of spyware to always remain in the background collecting information, and in the case of ransomware or lockers, to ensure that the device remains inaccessible even after reboot. Another permission is used to read the *phone state*, which is used to get cellular network information and know when phone calls are being made. Lockers usually draw their own view on top of everything else to prevent the victim from using their phone, and phishing malware also use this technique to inject their view into legitimate ones to ask for credentials in a credible manner; this can be done by requesting the *SYSTEM_ALERT_WINDOW* permission. The *GET_TASKS* permission is also commonly used with it to check which applications are in the foreground an correctly inject specific views over them. Most keyloggers use *BIND_ACCESSIBILITY_SERVICE* to capture keystrokes, and other malware also request this permission to read the text present on the screen, alert on transitions between applications, and spoof clicks and presses (clicker, clickjacking) which allow the app to grant itself permissions, click on specific ads to generate revenue for the developer, and more. To escalate privileges, some malware requests *BIND_DEVICE_ADMIN*, which provides administration features. Another interesting permission is *REQUEST_IGNORE_BATTERY_OPTIMIZATIONS*, as it prevents system optimizations from killing the app if it runs in the background during Doze mode, ensuring its persistance. There are also a large number of permissions that request access to SMS, contacts, location, camera, call log, etc. These permissions clearly depict the main interests of spyware samples.

After manual review of each sample, we collected all the capabilities that were inferred from the source code. Tables IV and V are the result of counting the frequency of occurrence of each capability and joining them in one table. In this case, the table is split in two, as it did not fit on a single page. Both tables only show the most frequent capabilities, as the less common ones did not reflect the entire dataset. The most frequent capabilities show profound similarities to the Tables II and III; after all, permissions and malware types are closely related to malware capabilities.

The *device information* capability refers to malware samples being able to extract the IMEI number, manufacturer, hardware specifications, etc., which are often employed to uniquely identify the infected device in order to properly distinguish it from other components of a botnet. Most malware also has the ability to manage internal files, such as *creating*, *modifying* and *deleting* them. A C2 server can also prompt the infected device to *list the files* present in a folder and *upload* them to the server, or *download files* from the server or the Internet. This can then be used to *install new applications*. Other malware has the ability to exfiltrate the *list of installed apps*, *screenshots*, *open specific applications* or *open URLs* in the default browser. Some have the ability to *hide the app icon* so that it is not visible from the launcher, while others have the ability to *uninstall applications*, even *themselves* so as to not leave any traces. It is also common for malware to check if *Internet connectivity* is available or if the phone is *rooted* in order to perform privilege actions, such as executing root commands on a terminal, as some malware has the ability to open a *remote shell*.

Another interesting approach to our dataset is to check whether antivirus software flags newer Android malware as malicious more often than older samples. To this end, we used the online service VirusTotal [13] to upload the ZIP files and wrote down the detection rate of each sample as a percentage of the number of antivirus solutions that flagged the sample

TABLE IV: Frequency of Malware Capabilities I

Capability	No. Malware Samples
Internet	69
Remote Data Exfiltration	61
Receive Boot Completed	58
Run at Startup	58
Run in Background	57
Bot	54
Write External Storage	53
Read SMS	50
Access Network State	49
Device Information	47
Read Contacts	46
Read Phone State	44
Send SMS	41
Access Fine Location	39
Wake Lock	38
Receive SMS	35
Upload Files	35
Camera	32
Read External Storage	32
Record Audio	31
Stealth	31
Download Files	30
Call Phone	27
Icon Hiding	27
Create Files	26
Access Coarse Location	25
Delete Files	25
Persistance	25
Read Call Log	25
Privilege Escalation	23
System Alert Window	22
List Files	20
Lock Device	20
Periodical Connection With Server	20
Activate Admin	19
Input Capture	18
Vibrate	18
Bind Accessibility Service	17
Keystrokes Monitoring	17

TABLE V: Frequency of Malware Capabilities II

Capability	No. Malware Samples
List Installed Apps	17
Make Toasts	17
Access Wifi State	16
Bind Device Admin	16
Data Encoding	15
Open URL	15
Check Screen Locked	14
Bind Notification Listener Service	13
Foreground Service	13
Get Tasks	13
Write Contacts	13
Write Settings	13
Credential Theft	12
Get Accounts	12
Read Files	12
Read History Bookmarks	12
Remote Shell	12
Root Check	12
Write SMS	12
Connectivity Check	11
Data Decoding	11
Open Apps	11
Persist on Screen	11
Process Outgoing Calls	11
Screenshot	11
Broadcast SMS	10
Change Wifi State	10
Draw Over Other Apps	10
Local Data Exfiltration	10
Uninstall Itself	10
Write Call Log	10
Write Files	10
File Decryption	9
File Encryption	9
Install Apps	9
Logging	9
Request Ignore Battery Optimizations	9
Uninstall Apps	9
Volume	9

as malware. Overall, our dataset has an average detection rate of *8.026%*. As shown in Figure 2, there is a slight increase in detections, which turns out to be almost constant over the years. In this case, the coefficient of determination has a value equal to *.0026*, which implies that the linear regression does not fit the data properly, as the values of each sample are extremely scattered.

To look at detection rates from a different perspective, instead of sorting the number of VirusTotal detections by year, we arranged them by malware type. Figure 3 shows that there is indeed a clear imbalance between the different malware types, which can be divided into four blocks. Lockers (LCK), Trojan-Bankers (TBK) and Ransomware (RNM) samples raise by far the most alerts. The second most detected block is made up of Rootkits (RTK), Keyloggers (KLG) and RATs (RAT). Next, Droppers (DRP), Spyware (SPY) and Trojan-Spy (TSY) samples are detected occasionally, but do not have a high rate. Finally, the remaining types of malware are not detected even once. One hypothesis that could explain this imbalance is that the noisier and pernicious malware types have been more vocal

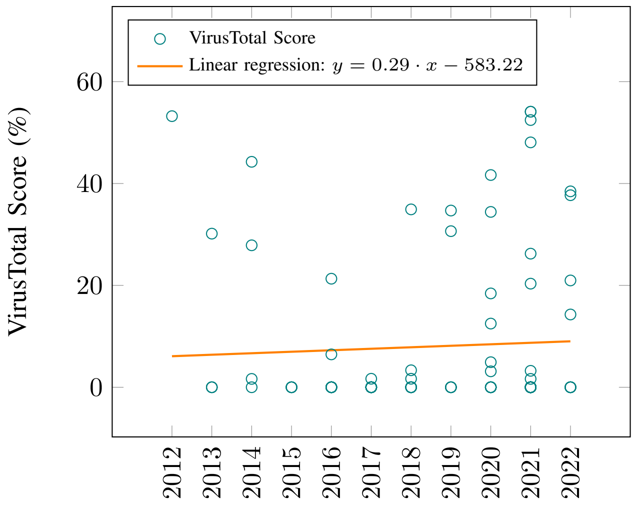

Fig. 2: VirusTotal Score by Year

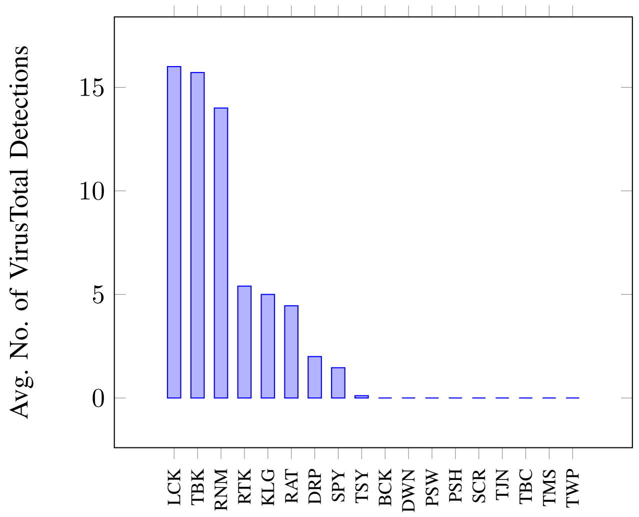

Fig. 3: Average Number of VirusTotal Detections by Malware Type

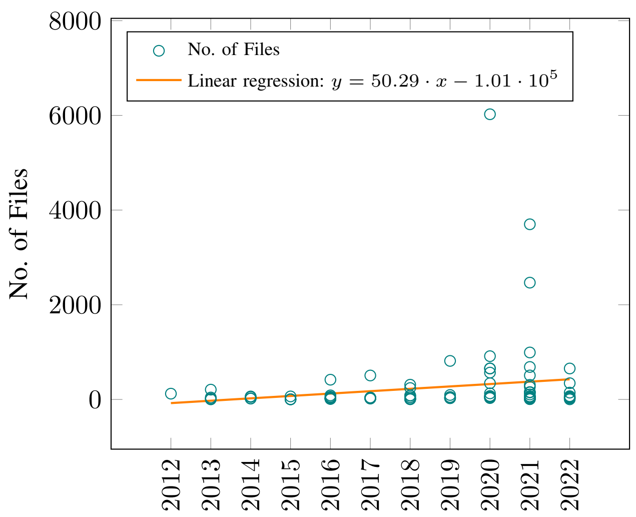

Fig. 4: Number of Files by Year

and are therefore more reported than the stealthier samples. It could also be due to the fact that their behavior is more distinct compared to that of the benevolent software. Another possibility is that most of the samples of the detected malware types posted on GitHub have been used in the wild, while the other types of malware found in public repositories tend to be PoCs and the like. Unfortunately, this thesis does not focus on obtaining an answer to this conundrum, so we will leave it with these speculations.

III. Malware Evolution Analysis

This section outlines our analysis of the evolution of Android malware source code employing software metrics. It first quantifies the evolution in Code Size, then it estimates Development Cost, and it finally measures Code Quality. In each section, we briefly introduce the software metrics and methodology used, and refer the reader to the papers on which this thesis is based for more details [4], [5].

A. Code Size

We utilize 3 different metrics to measure code size: **Number of files**, **Number of source code lines** (SLOCs), and **Number of functions**. We also measure the use of different **Programming languages** in malware development.

*1) **Number of files**:* To count the number of files, we use *Cloc* [14], an easy-to-use open source tool that has a wide range of recognized languages and simultaneously calculates the number of files present in a given project, the blank lines, comment lines, and source lines of code broken down by programming language, which were also useful for the measurements in the following sections. When counting files, it discards binary files and repeated source code files.

Collectively, our dataset has an average of *256.74* files per sample, totaling to *24904* files. Figure 4 shows the distribution over time of the number of files comprising the source code for each sample in the dataset. With 2 exceptions, no malicious code exceeds 100 files before 2016. From 2016 to 2019, there are 5 samples exceeding 100 files, one of which almost manages to approach 1000 files. From 2020 onward, several samples surpass that limit, with one of them reaching 6024 files. In this range of years, 21 samples exceed 100 files. Only in the years 2017, 2019 and 2020 no sample falls bellow 10 files. Only 1 malware consists of 1 file: a Rootkit named *Mindtrick*, written in C in 2015.

Therefore, the increase in the number of files per year is evident. The linear regression even shows that for each year, the number of files is increased by *50.3*. This implies that, every two years, the size increases by 100 files. The coefficient of determination for this estimation has a value of *.031*, which is somewhat low, as there are 3 samples that fall outside the norm when compared to the other samples: *AhMyth* (6024 files), *Bootloader-Backdoor* (3700 files) and *Arbitrium* (2467 files). Although the linear regression does not perfectly represent the real values, there is no doubt that the number of files has increased in recent years.

Regarding *Bootloader-Backdoor*, most of its files are C/C++ code, PO files, shell and Python scripts, HTML, XML, Kotlin and Java code, most of which are external tools and libraries that were imported and used by the malware. In the case of *AhMyth* and *Arbitrium*, since they are RAT samples, their files are mostly related to the server front end: JavaScript, CSS, HTML, JSON, TypeScript, Markdown and Java.

Figure 5 shows the samples organized by malware type rather than by year. Note that the y-axis is labeled on a logarithmic scale, so the difference between the maxima of the bars is larger that it may appear at first glance. The backdoor type has the most average number of files out of all malware types by a wide margin, as the only sample corresponding to this type is the aforementioned *Bootloader-Backdoor* (3700 files). Next are Trojan-Bankers, as they use overlay techniques to inject malicious views on top of legitimate ones and need code that mimics those views as well as the logic behind them.

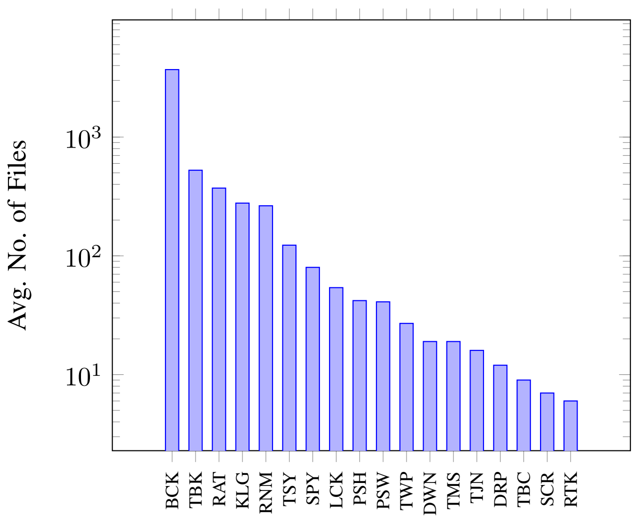

Fig. 5: Average Number of Files by Malware Type

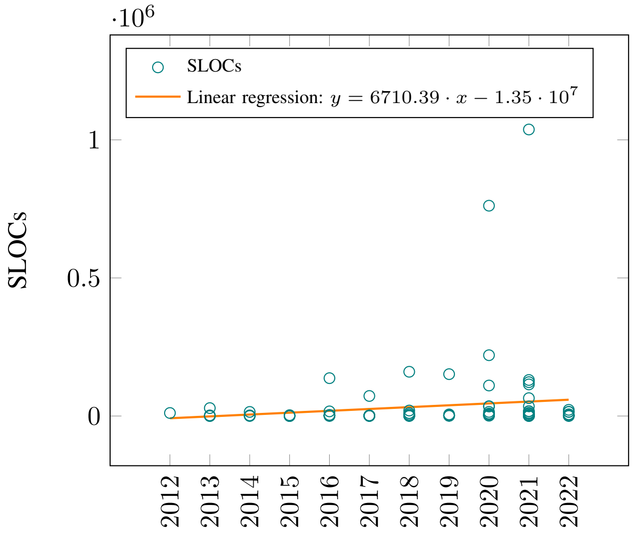

Fig. 6: Number of SLOCs by Year

They are followed by RATs, which need both client and server code to function properly. Most of the latest types of malware shown in the graph rely on a small number of files because they do not need more to perform their functions: Rootkits only need a file to hook system calls, Scareware only requires displaying intimidating messages to scare the victim, Droppers and Trojan-Backdoors are just the first step to deploy larger and more complex malware, etc.

*2) **Number of source code lines:*** In order to measure the size of software programs, the most commonly used metric is to count the number of lines present in the source code, excluding blank lines and comments, which is called SLOCs (source lines of code). To obtain this number from the samples in our dataset, we use *Cloc* [14], the same tool used in the previous section. When measuring the SLOCs of a sample, we consider all the code in it, regardless of the programming language or its functionality. The average SLOCs per sample is *36481.27*, adding up to *3538683* SLOCs. Figure 6 is the result of sorting the measured SLOCs in years.

Similar to the number of files, the growth of SLOCs is evident. Up to 2015, the *AndroidSurveillance* RAT is the biggest sample in terms of SLOCs (28688), while the smallest is once again the *Mindtrick* Rootkit (68). During this time period, only 3 samples exceed 3100 lines of code, while there are 5 samples bellow 300 SLOCs. Between 2016 and 2019, no sample falls under 300 SLOCs, while 10 samples exceed 3100 lines of code. The largest sample during this time, from the *Slempo* Trojan-Banker family, reaches one hundred thousand SLOCs (160267). From 2020 to date, several samples exceed the hundred thousand mark and one of them even reaches one million SLOCs. The three largest samples in terms of SLOCs are: *Bootloader-Backdoor* (2021) with 1037561 lines, *AhMyth* (2020) with 761,610 and *Cerberus* (2020) with 220,173.

This analysis bears a strong resemblance to the number of files. The same pattern is present in this case, and even two out of the three largest samples are the same; some of the

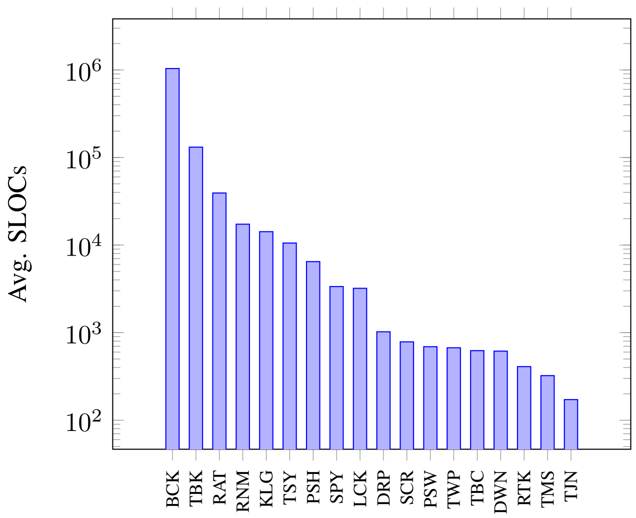

Fig. 7: Average SLOCs by Malware Type

smaller samples also remain unchanged. The linear regression shows an increase of *6710.39* SLOCs per year. Moreover, the coefficient of determination is equal to *.018*, which makes it clear that there is a very large variance in the SLOCs in our dataset.

Sorting the SLOCs by malware type, the resulting graph (Figure 7) closely resembles the one presented in the previous section (Figure 4). The 6 largest malware types remain almost unchanged, while the smallest ones experience a slight shift. As expected, there is a strong relationship between files and SLOCs, so the conclusions in terms of sustained growth are similar.

*3) **Number of functions:*** Although SLOCs is the most popular metric for measuring project size, it is not foolproof, as it does not take into account that different programming languages produce different lengths of code for the same functionality, as they are different in terms of expressiveness.

For that reason, we also decided to count the number of functions present in the code, as it not only illustrates the size of the project, but can also lead to insightful conclusions in terms of code quality, as it is generally better to split the code into smaller functions, as it improves legibility, maintainability and avoids repetition of commonly used code.

To count the number of functions per sample, we use the *Unified Code Count - Java (UCC-J)* [15]. This is a code metrics tool developed by the *Boehm Center for Systems and Software Engineering* [16] that allows to count physical and logical lines of code, obtain Cyclomatic Complexity and Maintainability Index results, count functions, and more. This specific release supports different languages: Ada, ASP/ASP.NET, Bash, C/C++, C Shell Script, COBOL, ColdFusion, ColdFusion Script, CSS, C#, DOS Batch, Fortran, Go, HTML, Java, JavaScript, JSP, Makefiles, MATLAB, NeXtMidas, Pascal, Perl, PHP, Python, R, Ruby, Scala, SQL, VB, VBScript, Verilog, VHDL, XML, and XMidas. Unfortunately, it does not support Kotlin, so any samples containing code written in Kotlin will be discarded. We took note of the number of functions detected, as well as the number of files form which those functions were extracted, as we will review the average number of functions per file later. Summing over the entire number of functions present in our dataset uncovers *20813* functions, with an average of *233.85* per sample.

Figure 8 reveals yet another correlation between the number of files, SLOCs and number of functions, as this graph continues to look similar to the previous ones. In this case, however, the values are slightly more balanced between the years. Already in 2012, the *AndroRAT* malware has 890 functions, which is almost 4 times the average. Apart from this rarity, the range of years from 2013 to 2015 is bellow average, where the lowest sample has 2 functions (*FakeFacebook*) and the largest one has 265 (*Dendroid*), which is the only one that manages to exceed the average value. From 2016 to 2019, four samples exceed the average number of functions. The lowest value in this period is 7 (*Rootkit-Android*), and the highest is 645 (*GmBot*). Finally, the period between 2020 and 2022 has the samples with the highest number of functions. These are *Lokiboard* (2020), with 1283, *Apps_Keylogger* (2021), with 1283, *Lokiboard-mod* (2020), with 1350, and *Covid-Locker* (2021), with 5846. In this case, these are three Keyloggers of the same family and a Ransomware. None of these samples stood out this much in the previous analyses; this is possibly due to the fact that this tool is limited in the number of languages it can interpret, so there are some samples with a high volume of code written in these non-interpreted languages that are not being counted.

Linear regression suggests that there is an increase of *34.24* functions each year. The coefficient of determination stands at *.02*, which, again, is a low score, indicative of the variability in the data.

Figure 9 shows the average number of functions per file. As mentioned above, in general it is better to have fewer functions per file to improve readability and maintenance of the code. As can be seen, there are a few files that deviate from the norm,

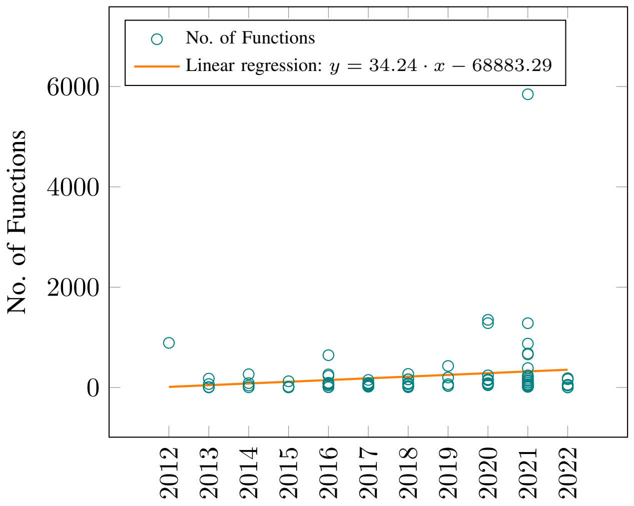

Fig. 8: Number of Functions by Year

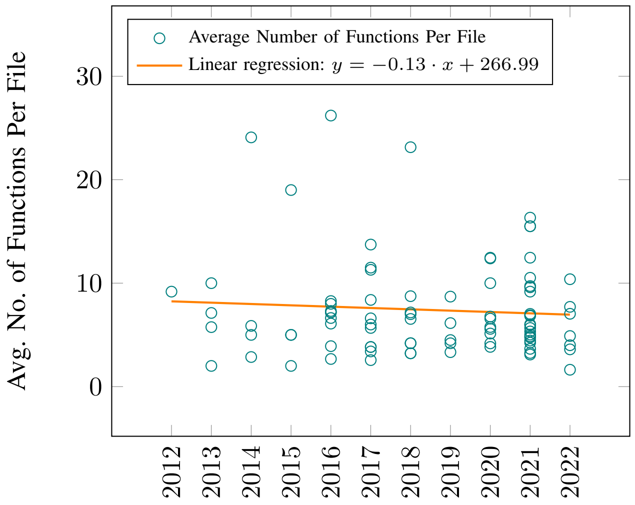

Fig. 9: Average Functions Per Files by Year

but most of them are close to the overall average, which is close to *7.33*. The most outlier samples are *BetterAndroRAT* (2016), *Dendroid* (2014), *rdroid* (2018), and *Android-Rootkit* (2015). From 2019 to 2022 most samples fall bellow average. Linear regression shows a decrease of *0.13* and a coefficient of determination equal to *.0054*, as the data are highly balanced and scattered.

*4) **Programming languages**:* Figure 10 shows the year distribution of the number of distinct languages used to develop each malware sample. This includes compiled and interpreted languages, such as C#, C/C++, Java, Kotlin, PHP, Python, or JavaScript, languages used to construct resources, which include HTML, XML, and CSS, and scripts used to build the project, that is, BAT or Make files. To obtain the languages for each project, we use *Cloc* [14], which was also used for the number of files and SLOCs.

The languages are mostly arranged around the linear regression, having only two far outlayer samples: *AhMyth* (2020) and

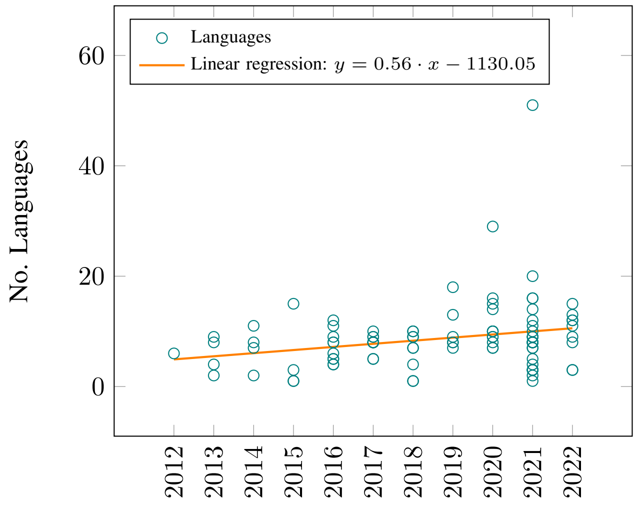

Fig. 10: Number of Different Languages used by Year

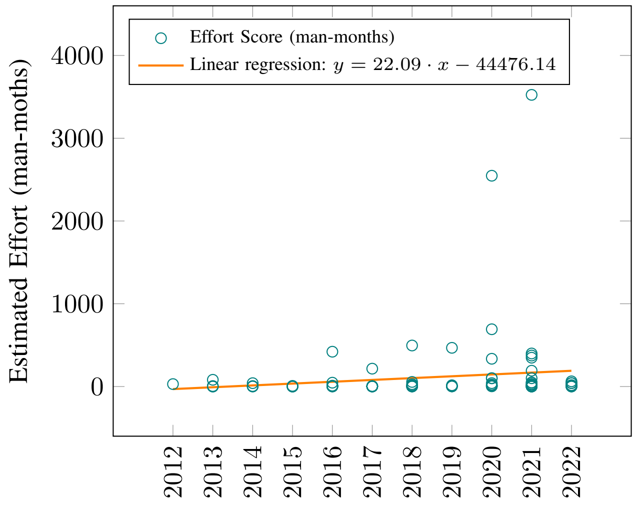

Fig. 11: Estimated Effort by Year (man-months)

Arbitrium(2021), both of which are RAT malware and were mentioned in the **Number of files** section, as these malware samples make heavy use of different server-side technologies and languages. The average number of languages per project is *8.75*. This is a large number, since Android applications are usually developed with at least 5 languages: Java/Kotlin, XML, Gradle, Text and Properties. Linear regression suggests that every 2 years, another language is added to the samples, as the slope has a value of *0.56*. Once again, the coefficient of determination has a low value: *0.58*.

B. Development Cost

The *Constructive Cost Model* (*COCOMO*) [17] is a reliable regression model based on source code lines to measure and predict software project development cost metrics, such as **Effort**, **Development time**, **Team size** and Code Quality. Effort corresponds to the amount of labor that is required to complete the project (measured in man-months). Development time is the estimated time required to complete the project (measured in months). Team size is an estimated of the people required to develop the project (measured in persons). The following equations represent the three metrics:

$$E = a(KLOC)^b$$

$$D = cE^d$$

$$P = \frac{E}{D}$$

In the preceding equations, KLOC represent the estimated SLOCs in thousands and a, b, c, and d are constant values obtained empirically and provided by the model as a function of the characteristics and nature of the project (Table VI). Three types of projects where considered for this model:

- Organic: small team, good experience, low complexity, flexible software requirements.
- Semi-detached: medium-sized team, mixed experience, medium complexity, combination of rigid and flexible requirements.
- Embedded: large team, high level of experience, high complexity, rigid software requirements.

For our dataset, we consider that all samples fall into the Organic project category, as malware development is usually led by small teams of programmers, and for this project we are inclined to conservatively estimate development costs rather than overestimate them.

TABLE VI: COCOMO constants

Software Projects	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

*1) **Effort**:* Figure 11 shows the COCOMO estimate of effort for each sample over the years. The global average effort is *115.98*, and the linear regression has a slope of *22.09*, showing that the effort increases rapidly each year. There are two samples (*Bootloader-Backdoor* and *AhMyth*) that probably do not fit into the Organic project category, as there is a large discrepancy between their values and the rest. Therefore, the coefficient of determination shows a low value, *.017*. Nevertheless, the value of the effort increases steadily each year.

*2) **Development time**:* Similar to the effort metric, the estimated time to develop samples present in our dataset (Figure 12) shows the same two samples deviating from the norm. The average development time is *8.32*, and these samples have an estimate 6 times larger than the average. Linear regression shows that, starting from 3.96 months in 2012, the value has transformed to 10.65 months in 2022, as its coefficient is *0.67* months per year. The coefficient of determination is *.038*.

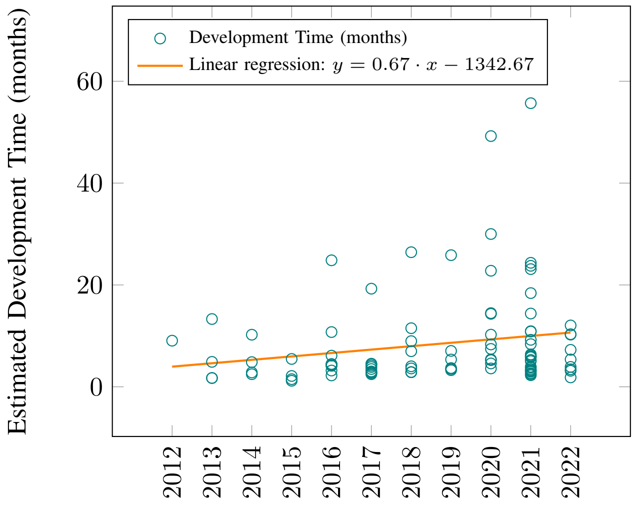

Fig. 12: Estimated Development Time by Year (months)

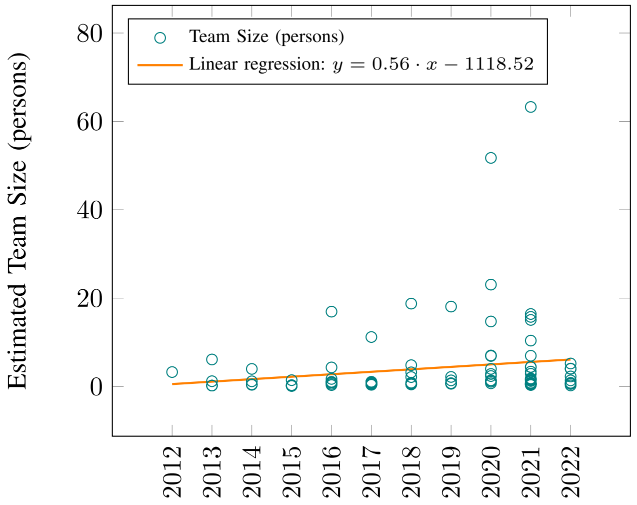

Fig. 13: Estimated Team Size by Year (persons)

*3) **Team size**:* As both previous metrics have shown, the estimated team size (Figure 13) also depicts the two outlier malware samples. In this case, the earlier malware required about 3 persons (full time). In 2019, the average (*4.31*) is exceeded, and in 2022 it is estimated that 6 persons are necessary to develop a malware sample. The increase per year is *0.67*, which can be translated into 2 persons per 3 years. The coefficient of determination is *0.26*.

C. Code Quality

In this section, we measure three aspects of code quality: **Complexity**, **Maintainability** and **Density of comments** of the samples.

*1) **Complexity**:* To measure software complexity of our samples, we use McCabe's *cyclomatic complexity* [18], one of the most commonly used software complexity metrics. The cyclomatic complexity (CC) of source code is computed from its control flow graph, and measures the number of linearly

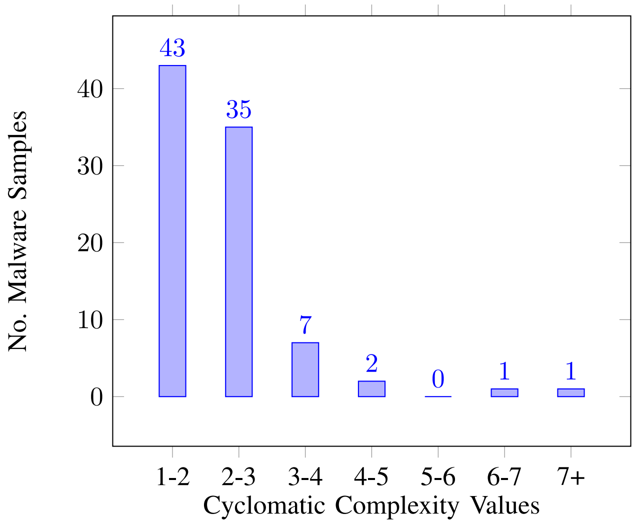

Fig. 14: Number of Samples By Cyclomatic Complexity Values

independent paths within the graph, taking into account the number graph nodes (N), edges (E), and the connected components (P):

$$CC = E - N + 2P$$

To calculate the cyclomatic complexity, we use the *Universal Code Count* (*UCC-J*) [15] (the same tool used in **Number of functions**), as it supports a wide range of languages, which is extremely useful for our analysis, since Android malware samples use many diverse languages for a single project. Still, it is not compatible with Kotlin source code, which appears in many projects in our dataset that were therefore excluded from this analysis.

Figure 14 illustrates the distribution of the average cyclomatic complexity per function for each malware sample. Most of the projects have functions with complexities ranging from 1 to 3. The most common range of values is [1,2], and the average of all samples is *2.29*. Overall, this can be seen as evidence in support of a generally monolithic design with poor break down into simple functions. However, there are some counterexamples. There is a sample with complexity equal to 6, called *SARA* (Ransomware, 2022), and one equal to 8, called *Adore* (Rootkit, 2014).

Looking at the annual distribution of the cyclomatic complexity values (Figure 15), it can be observed that the complexity decreases slightly each year, at a rate of *-0.0322*. The coefficient of determination, which is *.0078*, shows a scattered set of values.

*2) **Maintainability**:* Another useful metric related to software quality is the maintainability of source code. There are several reasons to maintain code, as it can be improved, fixed or extended. Maintainability can be measured with the maintainability index (MI), which is an estimate of how easy it is to comprehend, sustain and alter code. Both maintainability

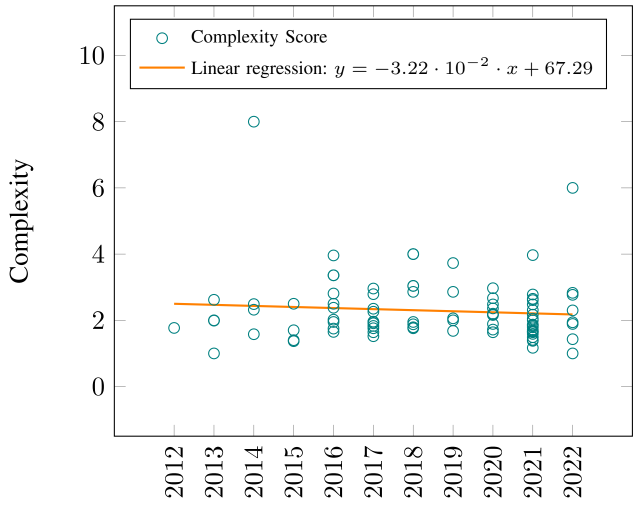

Fig. 15: Cyclomatic Complexity by Year

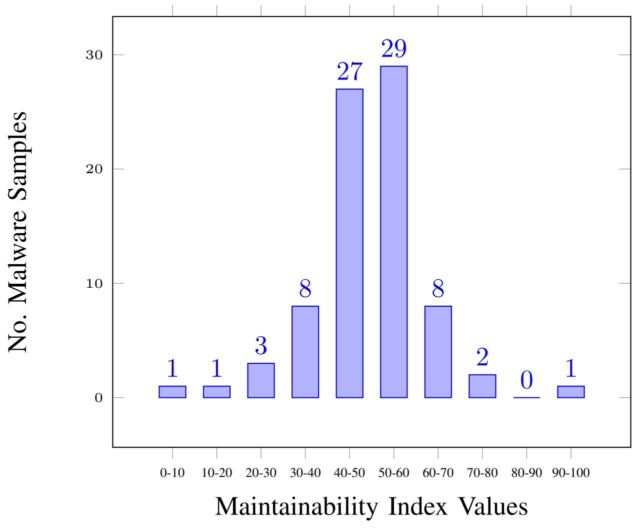

Fig. 16: Number of Samples by Maintainability Index

and complexity are often related, as more complex code translates into a low maintainability index.

To obtain the MI, we use *UCC-J* [15], as it calculates it automatically by function. In the case of the MI, this tool only calculates it for Java code, which further decreased the number of samples analyzed, as 17 of them were not written in Java, accounting for 17.53% of out dataset. We then computed the average of these indices to obtain the average MI for each malware. This number can be any number less than 171. In order to make the result easier to understand, we followed the same technique used by *Visual Studio* [19]: any negative number is treated as 0, after which we rebase the range between 0 and 171 to be from 0 to 100. The maintainability index results are considered very low if the MI is between 0-10, moderately low if it ranges from 10-20; any MI above 20 is considered good enough.

Figure 16 shows the distribution of MI values grouped into 10-point intervals. Most samples have an MI between 40 and 60, with only 2 samples falling bellow 20, which is the recommended threshold: *AndroidTrojanStarter* (Dropper, 2016) and *GetFiles* (Spyware, 2021). The average MI is *48.60*. There is one sample that falls in the last 10-point interval: *FakeFacebook* (Trojan, 2013).

The annual distribution (Figure 17) shows a decreasing trend in maintainability of *-0.2*, with a *.0018* coefficient of determination. It is worth noting that, in this case, complexity is not related to maintainability, as both decrease each year rather than being inversely proportional.

*3) **Density of comments:*** Code documentation is of great importance in determining the maintainability of code. For this reason, it is relevant to check the ratio of comments compared to SLOCs. From Figure 18 it can be seen that the tendency to include comments in the code is increasing. Specifically, the linear regression has a slope of *0.34* and a coefficient of determination of *.003*. The average comment ratio is *14.81%*.

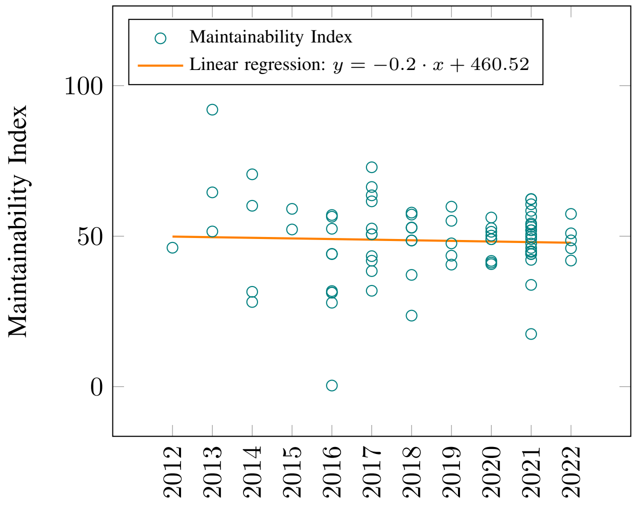

Fig. 17: Maintainability Index by Year

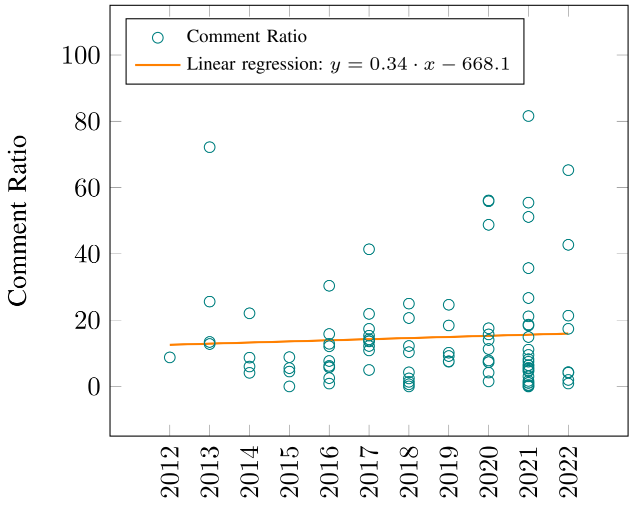

Fig. 18: Comment Ratio by Year

IV. Comparison with Other Malware

In this section, we compare our results with those obtained by Calleja, Tapiador and Caballero in their work [5]. This will provide information on the differences present between general and Android-specific malware.

First of all, there is a big difference in the size of the dataset. Theirs contains 456 samples, being almost 5 times larger than ours, and spans from 1975 to 2016, which is an interval 4 times larger than that of our dataset.

In terms of size, Android samples tend to be larger: few of their samples manage to exceed the average number of files, SLOCs and number of programming languages obtained from Android malware. This can be explained by the prevalence of small samples in their dataset that only consist of 1 file and have few lines of code, but also because Android programs usually need a large number of files, as they must contain all view, logic and build files. The same is true for the values of effort, development time and team size: Android samples present higher values compared to non-Android malware.

But this is reversed when complexity and maintainability are compared. Android malware shows extremely low values in terms of complexity compared to non-specific malware. Maintainability and feedback ratios are also poor in comparison. In essence, one could argue that Android malware has larger sizes and development costs, but is less complex and has lower quality compared to other types of malware.

V. Discussion

Bellow, we discuss some aspects of the nature of our approach, mainly the Limitations of our methodologies and results, and pose some Open questions for future work.

A. Limitations

One of the potential shortcoming of this work is the low number of samples, as it renders the analysis less reliable and representative. However, obtaining source code is an arduous task, even more so in the case of malware samples, especially Android ones. Considering this fact, we believe that our dataset is representative enough to at least ensure that the analysis can extract some genuine results.

Our dataset possibly suffers from a collection bias, as all samples have been gathered from public GitHub repositories, which do not necessarily represent malware used in the wild. Still, we have a decent number of leaked malware samples that have been engineered by malware developers and openly deployed. Nevertheless, the richness of our dataset would increase considerably if more samples were collected from specialized underground forums and other means.

We considered the number of functions as a metric of code size, but it would also have been valuable to consider *function point estimates* [20], which measure external inputs and outputs, files used, user interactions and external interfaces to capture the overall functionality of the software.

UCC-J [15] has proven to be a limited tool, especially in the maintainability index calculations, considering that Kotlin, which along with Java is the most common language for developing Android applications, is not supported.

Lastly, we consider that it is still too early to draw meaningful conclusions, as only a decade has passed since Android was released. Another decade would provide more data that would allow researches to draw more accurate results.

B. Open questions

There are some analyses that were planned, but were not carried out, as we did not have enough time to perform them. One of them is to analyze whether certain malware types have increased in popularity over the years.

Another analysis we left behind is to distribute languages by malware type, to see which programming languages are most used by each type of malware. In this paper we already mentioned that RATs and Spyware tend to have server-side languages apart from client code, but it would be interesting to make that comparison with all malware types.

It would also be insightful to contrast the metrics of malware software with those of regular software, to see if there are major differences between the costs, maintainability and complexity of regular development teams compared to malware-focused ones.

We also wanted to compile each sample to see possible errors, as well as its behavior in a sandbox environment and perform a dynamic analysis.

For each application, we also wanted to know how many components, services, etc. the sample has and what uses each of them serves.

Finally, code reuse and clone detection was one of the most important analyses we had to leave out. It would have been extremely revealing to measure plagiarism, code sharing, auto-generated code and code reuse among the malware samples.

VI. Related Work

Calleja, Tapiador and Caballero [4], [5] followed the same approach as this paper, as we have based this thesis on their works. They analyzed a large number of malware source code samples from different operating systems and from different time periods for code size, code quality, development costs and code reuse. They found that malware production is progressively becoming an industry, as code size, effort and complexity experienced an exponential increase over the years.

Mercaldo, Di Sorbo, Visaggio, Cimitile, and Martinelli [21] demonstrated that malware writers devote effort and skill to improve the quality of their code to produce high quality malware, similar to the evolution observed in goodware applications.

Yajin Zhou and Xuxian Jiang [22] collected 1200 Android malware samples covering major families and arranged them according to various aspects, after which they revealed the rapid evolution of malware to evade anti-virus software, revealing that some antivirus software are not up to the malware's efforts. Tam, Feizollah, Anuar, Salleh and Cavallaro [23] similarly analyzed detection techniques against Android malware, seeking to demonstrate the evolution of Android malware, and also discuss related malware statistics.

VII. Conclusion

In this thesis, we have conducted a study on the evolution of Android malware source code over its entire lifetime, which for now spans a period of 10 years. We have collected and analyzed 97 samples, which is the largest dataset of Android malware source code to our knowledge. We have quantified the code size and estimated its cost and quality using well-known software metrics. The results extracted from this work indicate an increase in size and cost, but a small decrease in complexity and maintainability. Therefore, we conclude that the results are not conclusive enough to support the claim of a developing malware production industry.

A CSV file with all the information we have collected and extracted from the malware samples during this research can be obtained from the following GitHub repository [24].

References

[1] A. Cranz, “There are over 3 billion active Android devices,” The Verge, 18-May-2021. [Online]. Available: https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. [Accessed: 19-Sep-2022].

[2] “Malware Statistics & Trends Report: AV-TEST,” AV. [Online]. Available: hrefhttps://www.av-test.org/en/statistics/malware/https://www.av-test.org/en/statistics/malware/. [Accessed: 19-Sep-2022].

[3] A. Kujawa, J. Segura, T. Reed, N. Collier, J. Taggart, H. Jazi, A. Brading, M. Stocklev, D. Ruiz, J. Umawing, C. Boyd, P. Arntz, “2021, State of Malware Report,” Malware Bytes. [Online]. Available: https://www.malwarebytes.com/resources/files/2021/02/mwb_stateofmalwarereport2021.pdf. [Accessed: 19-Sep-2022].

[4] A. Calleja, J. Tapiador, and J. Caballero, “A Look into 30 Years of Malware Development from a Software Metrics Perspective,” in Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses, pp. 325-345, Evry, France, September 2016, doi: 10.1007/978-3-319-45719-2_15.

[5] A. Calleja, J. Tapiador and J. Caballero, ”The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development,” in IEEE Transactions on Information Forensics and Security, vol. 14, no. 12, pp. 3175-3190, Dec. 2019, doi: 10.1109/TIFS.2018.2885512.

[6] “Where the world builds software,” GitHub. [Online]. Available: https://github.com/. [Accessed: 16-Sep-2022].

[7] Vxunderground, “MalwareSourceCode: Collection of malware source code for a variety of platforms in an array of different programming languages.,” GitHub. [Online]. Available: https://github.com/vxunderground/MalwareSourceCode. [Accessed: 16-Sep-2022].

[8] “vx-underground,” vx-underground. [Online]. Available: https://www.vx-underground.org/. [Accessed: 16-Sep-2022].

[9] “A new virus naming convention (1991),” CARO. [Online]. Available: http://www.caro.org/articles/naming.html. [Accessed: 16-Sep-2022].

[10] d-Raco, “android-malware-source-code-samples: Android malware source code dataset collected from public resources.,” GitHub. [Online]. Available: https://github.com/d-Raco/android-malware-source-code-samples. [Accessed: 16-Sep-2022].

[11] d-Raco, “android-malware-capabilities-analyzer: A tool for analyzing Android malware source code capabilities.,” GitHub. [Online]. Available: https://github.com/d-Raco/android-malware-capabilities-analyzer. [Accessed: 16-Sep-2022].

[12] “Manifest.permission: android developers,” Android Developers. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission. [Accessed: 16-Sep-2022].

[13] “VirusTotal - Home,” VirusTotal. [Online]. Available: https://www.virustotal.com/gui/home/upload. [Accessed: 17-Sep-2022].

[14] AlDanial, “Cloc: Cloc Counts Blank Lines, comment lines, and physical lines of source code in many programming languages.,” GitHub. [Online]. Available: https://github.com/AlDanial/cloc. [Accessed: 18-Sep-2022].

[15] “CSSE tools / Unified Code Count Java - UCC-J / UCC-java 2018.05,” GitLab. [Online]. Available: https://cssedr.usc.edu:4443/csse-tools/ucc-j/UCC-Java-2018.05. [Accessed: 18-Sep-2022].

[16] “Boehm Center for Systems and Software Engineering,” CSSE. [Online]. Available: https://csse.usc.edu/. [Accessed: 18-Sep-2022].

[17] B. W. Boehm, ”Software Engineering Economics,” in IEEE Transactions on Software Engineering, vol. SE-10, no. 1, pp. 4-21, Jan. 1984, doi: 10.1109/TSE.1984.5010193.

[18] T. J. McCabe, ”A Complexity Measure,” in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976, doi: 10.1109/TSE.1976.233837.

[19] “Code metrics - maintainability index range and meaning - Visual Studio (Windows),” Visual Studio (Windows) — Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022. [Accessed: 19-Sep-2022].

[20] A. J. Albrecht, “Measuring Application Development Productivity,” in IBM Application Development Symp., I. B. M. Press, Ed., Oct. 1979.

[21] F. Mercaldo, A. Di Sorbo, C. A. Visaggio, A. Cimitile, and F. Martinelli, “An exploratory study on the evolution of Android Malware Quality,” Journal of Software: Evolution and Process, vol. 30, no. 11, 2018.

[22] Y. Zhou and X. Jiang, ”Dissecting Android Malware: Characterization and Evolution,” 2012 IEEE Symposium on Security and Privacy, 2012, pp. 95-109, doi: 10.1109/SP.2012.16.

[23] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, “The evolution of Android malware and Android Analysis Techniques,” ACM Computing Surveys, vol. 49, no. 4, pp. 1–41, 2017.

[24] d-Raco, “android-malware-source-code-analysis: Analysis of Android malware families using available source code.,” GitHub. [Online]. Available: https://github.com/d-Raco/android-malware-source-code-analysis. [Accessed: 16-Sep-2022].