

The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development

Alejandro Calleja, Juan Tapiador, and Juan Caballero

Abstract—During the last decades, the problem of malicious and unwanted software (malware) has surged in numbers and sophistication. Malware plays a key role in most of today’s cyber attacks and has consolidated as a commodity in the underground economy. In this work, we analyze the evolution of malware from 1975 to date from a software engineering perspective. We analyze the source code of 456 samples from 428 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 an exponential increment of nearly one order of magnitude per decade in aspects such as size and estimated effort, with code quality metrics similar to those of benign software. We also study the extent to which code reuse is present in our dataset. We detect a significant number of code clones across malware families and report which features and functionalities are more commonly shared. Overall, our results support claims about the increasing complexity of malware and its production progressively becoming an industry.

I. INTRODUCTION

The malware industry seems to be in better shape than ever. In their 2015 Internet Security Threat Report [1], Symantec reports that the total number of known malware in 2014 amounted to 1.7 billion, with 317 million (26%) new samples discovered just in the preceding year. This translates into nearly 1 million new samples created every day. A recent statement by Panda Security [2] provides a proportionally similar aggregate: out of the 304 million malware samples detected by their engines throughout 2015, 84 million (27%) were new. These impressive figures can be partially explained by the adoption of reuse-oriented development methodologies that make it exceedingly easy for malware writers to produce new samples, and also by the increasing use of packers with polymorphic capabilities. Another key reason is the fact that over the last decade malware has become a profitable industry, thereby acquiring the status of a *commodity* [3], [4] in the flourishing underground economy of cyber crime [5], [6]. From a purely technical point of view, malware has experienced a remarkable evolutionary process since the 1980s, moving from simple file-infection viruses to stand-alone programs with network propagation capabilities, support for distributed architectures based on rich command and control protocols, and a variety of modules to execute malicious actions in the victim. Malware writers have also rapidly adapted to new

platforms as soon as these acquired a substantial user base, such as the recent case of smartphones [7].

The surge in number, sophistication, and repercussion of malware attacks has gone hand in hand with much research, both industrial and academic, on defense and analysis techniques. The majority of such investigations have focused on binary analysis, since most malware samples distribute in this form. Only very rarely researchers have access to the source code and can report insights gained from its inspection. (Notable exceptions include the analysis of the source code of 4 IRC bots by Barford and Yegneswaran [8] and the work of Kotov and Massacci on 30 exploit kits [9]). One consequence of the lack of wide availability of malware source code is a poor understanding of the malware development process, its properties as a software artifact, and how these properties have changed in the last decades.

In this paper, we present a study of malware evolution from a software engineering perspective. Our analysis is based on a dataset collected by the authors over two years and composed of the source code of 456 malware samples ranging from 1975 to 2016. Our dataset includes, among others, early viruses, worms, botnets, exploit kits, and remote access trojans (RATs). This is the largest dataset of malware source code presented in the literature. We perform two separate analysis on this dataset. First, we provide quantitative measurements on the evolution of malware over the last four decades. Second, we study the prevalence of source code reuse among these malware samples.

To measure the evolution of malware complexity over time we use several metrics proposed in the software engineering community. Such metrics are grouped into three main categories: (i) measures of size: number of source lines of code (SLOC), number of source files, number of different programming languages used, and number of function points (FP); (ii) estimates of the development cost: effort (man-months), required time, and number of programmers; and (iii) measures of code quality: comment-to-code ratio, complexity of the control flow logic, and maintainability of the code. We use these metrics to compare malware source code to a selection of benign programs.

We also study the prevalence of source code reuse in our dataset. Code reuse—or code clone—detection is an important problem to detect plagiarism, copyright violations, and to preserve the cleanness and simplicity of big software projects [10]. Several authors have suggested that code cloning is a fairly common practice in large code bases, even if it also

A. Calleja and J. Tapiador are with the Department of Computer Science, Universidad Carlos III de Madrid, 28911 Leganes, Madrid, Spain. E-mail: {accortin, jestevez}@inf.uc3m.es.

J. Caballero is with the IMDEA Software Institute, Madrid, Spain. E-mail: juan.caballero@imdea.org.

leads to bug propagation and poorly maintainable code [11]. Given the high amount of malware discovered on a daily basis, it is a common belief that most malware is not developed from scratch, but using previously written code that is slightly modified according to the attacker’s needs [12]. Detecting clones in malware source code enables a better understanding of the mechanisms used by malware, their evolution over time, and may reveal relations among malware families.

This paper builds on our previous work that studied malware evolution using software metrics on a dataset of 151 malware samples covering 30 years [13]. In this work, we present our updated dataset, which triples the number of original samples and extends the covered time frame to four decades. We redo the analysis on malware evolution to cover the new samples, and also provide a new analysis on malware source code reuse.

The main findings of our work include:

- 1) We observe an exponential increase of roughly one order of magnitude per decade in the number of source code files and SLOC and FP counts per sample. Malware samples from the 1980s and 1990s contain just one or a few source code files, are generally programmed in one language and have SLOC counts of a few thousands at most. Contrarily, samples from the late 2000s and later often contain hundreds of source code files spanning various languages, with an overall SLOC count of tens, and even hundreds of thousands.
- 2) In terms of development costs, our estimates evidence that malware writing has evolved from small projects of just one developer working no more than 1-2 months full time, to larger programming teams investing up to 6-8 months and, in some cases, possibly more.
- 3) A comparison with selected benign software projects reveals that the largest malware samples in our dataset present software metrics akin to those of products such as `Snort` or `Bash`, but are still quite far from larger software solutions.
- 4) The code quality metrics analyzed do not suggest significant differences between malware and benign software. Malware has slightly higher values of code complexity and also better maintainability, though the differences are not remarkable.
- 5) We find quite a large number of code reuse instances in our dataset, specifically in C/C++ and Assembly code, that range from a few lines to several thousands lines of code in length. An analysis of such clones reveals that commonly shared functionality belongs to one of four groups:
 - a) Anti-analysis capabilities such as unpacking routines, polymorphic engines, and code to kill antivirus (AV) processes.
 - b) Core malware artifacts, including shellcodes for initial infection, spreading routines, and code for various actions on the victim.
 - c) Data clones such as arrays of passwords, process names, and IP addresses.
 - d) Data structures and associated functions, such as those

needed to interact with PE or ELF files, popular communication protocols, or the operating system kernel through documented and undocumented APIs.

The remaining of this paper is organized as follows. In Section II we describe our dataset of malware source code. Section III presents our quantitative measurements on the evolution of malware development. In Section IV we detail our code clone detection approach and results. Section V discusses the suitability of our approach, its limitations, and additional conclusions. Finally, Section VII concludes the paper.

II. DATASET

Our work is based on a dataset of malware source code collected by the authors over two years (2015–2016). Collecting malware source code is a challenging endeavor because malware is typically released in binary form. Only occasionally its source code is released or leaked, with its availability being strongly biased towards classical viruses and early specimens. When leaked, the source code may be difficult to access in underground forums. These challenges make it impossible to try to be complete. While we try to collect as many samples as possible, the goal is to acquire representative examples of the malware ecosystem during the last 40+ years, constrained to the limited availability.

Samples were obtained from a variety of sources, including virus collection sites such as *VX Heaven* [14], code repositories such as *GitHub*, classical e-zines published by historically prominent malware writing groups such as *29A*, malware exchange forums, and through various P2P networks. We expanded our list of sources by using a snowballing methodology, exploring previously unknown sources that were referenced in sites under examination.

A sample in our dataset corresponds to a specific version of a malware project, where a malware project is most often referred to as a *malware family*. A sample may comprise of one or multiple source code files typically bundled as an archive (e.g., a ZIP file). Those files may be set in an arbitrarily complex directory structure and may be written in one or multiple programming languages (see Section III). Most often only one version of a family has been leaked, but occasionally we collect multiple, e.g., `Cairuh.A` and `Cairuh.B`. For the vast majority of samples we do not know the author(s).

Our initial collection contained 516 samples. Each sample was first quickly verified through manual inspection and then compiled, executed and, whenever possible, functionally tested. At this point, 11.6% of the obtained samples were discarded, either because testing them was unfeasible (e.g., due to nontrivial compilation errors or unavailability of a proper testing environment), or simply because they turned out to be fake.

The 456 successfully tested samples that comprise our final dataset have been tagged with a year and a loose category. The year corresponds to their development when stated by the source, otherwise with the year they were first spotted in the wild. They are also tagged with a coarse-grained malware type:

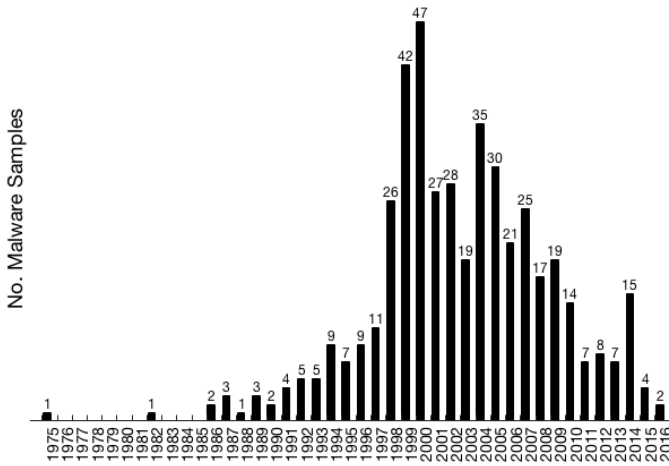


Fig. 1: Distribution of malware source code samples in the dataset.

Virus (V), Worm (W), Macro virus (M), Trojan (T), Botnet (B), RAT (R), Exploit kit (E), or Rootkit (K). We are aware that this classification is rather imprecise. For instance, nearly all Botnets and RATs can be easily considered as Trojans, and, in some cases, show Worm features too. We chose not to use a more fine-grained malware type because it is not essential to our study and, furthermore, such classifications are problematic for many modern malware examples that feature multiple capabilities.

Figure 1 shows the distribution by year of the final dataset of 456 samples. Approximately 61% of the samples (281) correspond to the period 1995-2005. The second biggest set of samples (139) correspond to the period 2006-2016. Finally, the rest of samples (36) corresponds to the period ranging from 1975 to 1994.

The largest category is Virus (318 samples), followed by Worm (58), Botnet (26), Trojan (26), RAT (12), Exploit kit (11), Macro virus (4), and Rootkit (1).

III. MALWARE EVOLUTION ANALYSIS

This section describes our analysis of the evolution of malware source code using software metrics. It first quantifies the evolution in code size (Section III-A), then it estimates development cost (Section III-B), next it measures code quality (Section III-C), and finally compares malware to benign code (Section III-D). In each section, we briefly introduce the software metrics used, and refer the reader to our original paper for more details [13].

A. Code Size

We use 3 different metrics to measure code size: number of files, number of source code lines, and function point estimates. We also measure the use of different programming languages in malware development.

Number of files. Figure 2a shows the distribution over time of the number of files comprising the source code of each sample in the dataset. Except for a few exceptions, until the

mid 1990s there is a prevalence of malicious code consisting of just one file. Nearly all such samples are viruses written in assembly that, as discussed later, rarely span more than 1,000 lines of code. This follows a relatively common practice of the 1980s and 1990s when writing short assembly programs.

From the late 1990s to date there is an exponential growth in the number of files per malware sample. The code of viruses and worms developed in the early 2000s is generally distributed across a reduced (<10) number of files, while some Botnets and RATs from 2005 on comprise substantially more. For instance, Back Orifice 2000, GhostRAT, and Zeus, all from 2007, contain 206, 201, and 249 source code files, respectively. After 2010, no sample comprises a single file. Examples of this time period include KINS (2011), SpyNet (2014), and the RIG exploit kit (2014) with 267, 324, and 838 files, respectively. This increase reveals a more modular design, which also correlates with the use of higher-level programming languages discussed later, and the inclusion of more complex malicious functionalities (e.g., network communications and support for small databases).

Simple least squares linear regression over the data points shown in Figure 2a yields a regression coefficient (slope) of 1.14. (Note that the y-axis is in logarithmic scale and, therefore, such linear regression actually corresponds to an exponential fit.) This means that the number of files has grown at an approximate yearly ratio of 14%, i.e., it has doubled every 5 years.

Number of lines. Traditionally, the number of lines in the source code of a program, excluding comment and blank lines (SLOCs), has been employed as the most common metric for measuring its size. In our analysis we use `clloc` [15], an open-source tool that counts SLOCs, blank lines, and comment lines, and reports them broken down by programming language. Figure 2b shows the SLOCs for each sample, obtained by simply aggregating the SLOCs of all source code files of the sample, irrespective of the programming language in which they were written.

Again, the growth over the last 40 years is clearly exponential. Up to the mid 1990s viruses and early worms rarely exceeded 1,000 SLOCs. Between 1997 and 2005 most samples contain several thousands SLOCs, with a few exceptions above that figure, e.g., Simile (10,917 SLOCs) or Troodon (14,729 SLOCs). The increase in SLOCs during this period correlates positively with the number of source code files and the number of different programming languages used. Finally, a significant number of samples from 2007 on exhibit SLOCs in the range of tens of thousands. For instance, GhostRAT (33,170), Zeus (61,752), KINS (89,460), Pony2 (89,758), or SpyNet (179,682). Most of such samples correspond to moderately complex malware comprising of more than just one executable. Typical examples include Botnets or RATs featuring a web-based C&C server, support libraries, and various types of bots/trojans. There are exceptions, too. For instance, Point-of-Sale (POS) trojans such as Dexter (2012) and Alina (2013) show relatively low SLOCs (2,701 and 3,143, respectively).

In this case the linear regression coefficient over the data

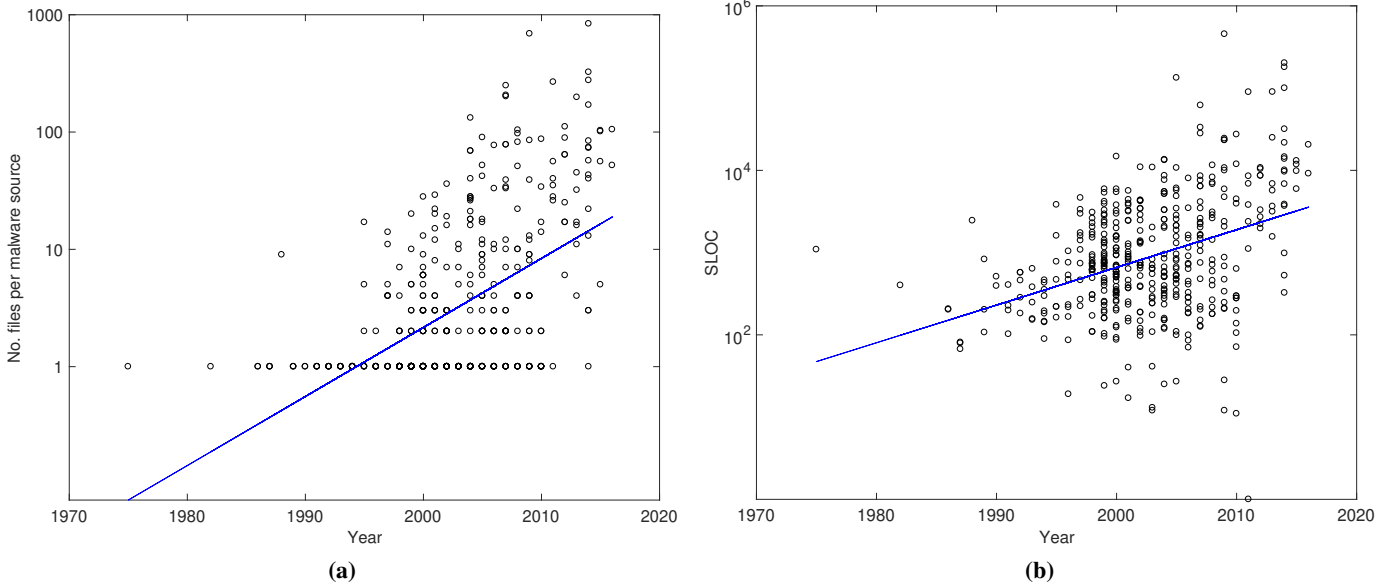


Fig. 2: Number of files (a) and SLOC (b) for each sample in our dataset. Note that the y-axis is in logarithmic scale.

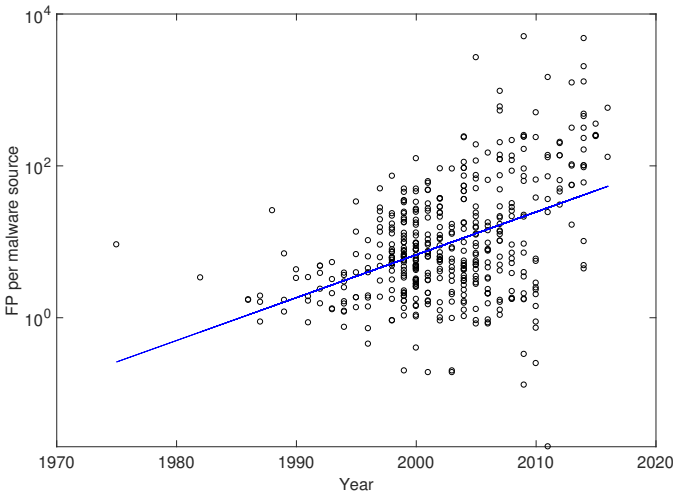


Fig. 3: Function point counts for each sample in the dataset. Note that the y-axis is in logarithmic scale.

points is 1.11, i.e., the SLOCs per malware have increased approximately 11% per year; or, equivalently, the SLOC count doubles every 6.5 years, resulting in an increase of nearly an order of magnitude each decade.

Function points estimates. Although SLOCs is the most popular metric for measuring project size, it has a number of shortcomings [16]. Most notably, when comparing the sizes of projects developed using different programming languages, SLOCs may lead to misleading conclusions since this metric does not take into account the programming language expressiveness. To address this issue, we leverage the *function-point count* [17], [18] (FPC) metric, which aims to capture the overall functionality of the software. The function-point count is measured using four program features: external inputs and outputs, user interactions, external interfaces, and files used. The expected size in SLOCs of a software project can

be estimated (before it is coded) from function-point counts through a process known as *backfiring* [19]. This process uses programming languages empirical tables (PLTs) that provide the average SLOCs per function point for different programming languages. In our analysis, we use a reversed backfiring process that uses PLT v8.2 [20] to obtain function-point counts from SLOCs. We use those function-point counts as a normalized size for malware written in different languages.

Figure 3 shows, as expected, a clear correlation between FPC and SLOCs and the conclusions in terms of sustained growth are similar. Starting in 1990, there is roughly an increase of one order of magnitude per decade. Thus, in the 1990s most early viruses and worms contain just a few (< 10) function points. From 2000 to 2010 the FPC concentrate between 10 and 100, with Trojans, Botnets, and RATs accounting for the higher counts. Since 2007, many samples exhibit FPC of 1,000 and higher; examples include Pony2 (2013), with 1,240, SpyNet (2014), with 2,028, and the RIG exploit kit (2014), with 4,762. Linear regression over the data points yields a coefficient of 1.13, i.e., FPCs per malware have suffered an approximate growth of 13% per year; or, equivalently, FPCs double every 5.5 years.

Programming languages. Figure 4a shows the distribution over time of the number of different languages used to develop each malware sample. This includes not only compiled and interpreted languages such as assembly, C/C++, Java, Pascal, PHP, Python, or Javascript, but also others used to construct resources that are part of the final software project (e.g., HTML, XML, CSS) and scripts used to build it (e.g., BAT or Make files).

Figure 4b shows the usage of different programming languages to code malware over time in our dataset. The pattern reveals the prevalent use of assembly until the late 2000s. From 2000 on, C/C++ become increasingly popular, as well as other “visual” development environments such as Visual Basic

and Delphi (Pascal). Botnets and RATs from 2005 on also make extensive use of web interfaces and include numerous HTML/CSS elements, pieces of Javascript, and also server-side functionality developed in PHP or Python. Since 2012 the distribution of languages is approximately uniform, revealing the heterogeneity of technologies used to develop modern malware.

B. Development Cost

An important problem in software engineering is to make an accurate estimate of the cost required to develop a software system [21]. A prominent approach to this problem are algorithmic cost modeling methods, which provide cost figures using as input various project properties such as code size and organizational practices. Probably the best known of these methods is the Constructive Cost Model (COCOMO) [22]. COCOMO is an empirical model derived from analyzing data collected from a large number of software projects. COCOMO provides the following equations for estimating three metrics related to the cost of a project: effort (in man-months), development time (in months), and number of people required.

$$E = a_b(\text{KLOC})^{b_b} \quad (1)$$

$$D = c_b E^{d_b} \quad (2)$$

$$P = \frac{E}{D} \quad (3)$$

In the equations above, KLOC represent the estimated SLOCs in thousands and a_b, b_b, c_b, d_b are empirically obtained regression coefficients provided by the model. The value of these coefficients depends on the nature of the project. COCOMO considers three different types of projects: (i) *Organic* projects (small programming team, good experience, and flexible software requirements); *Semi-detached* projects (medium-sized teams, mixed experience, and a combination of rigid and flexible requirements); and (iii) *Embedded* projects (organic or semi-detached projects developed with tight constraints). For our analysis, we decided to consider all samples as organic for two reasons. First, it is reasonable to assume that, with the exception of a few cases, malware development has been led so far by small teams of experienced programmers. Additionally, we favor a conservative estimate of development cost, which is achieved using the lowest COCOMO coefficients (i.e., those of organic projects). Thus, our estimates can be seen as a (estimated) lower bound of development cost.

Figure 5 shows the COCOMO estimates for the effort, time, and team size required to develop the malware samples in our dataset. Figure 5a shows the COCOMO estimation of effort in man-months. The evolution over time is clearly exponential, with values roughly growing one order of magnitude each decade. While in the 1990s most samples required approximately one man-month, this value rapidly escalates up to 10–20 man-months in the mid 2000s, and to 100s for

a few samples in the last years. Linear regression confirms this, yielding a regression coefficient of 1.11; i.e., the effort growth ratio per year is approximately 11%; or, equivalently, it doubles every 6.5 years.

The estimated time to develop the malware samples (Figure 5b) experiences a linear increase up to 2010, rising from 2–3 months in the 1990s to 7–10 months in the late 2000s. The linear regression coefficient in this case is 0.255, which translates into an additional month every 4 years. Note that a few samples from the last 10 years report a considerable higher number of months, such as Zeus (2007) or SpyNet (2014) with 18.1 and 27.7 months, respectively.

The amount of people required to develop each sample (Figure 5c) grows similarly. Most early viruses and worms require less than one person (full time). From 2000 on, the figure increases to 3–4 persons for some samples. Since 2010, a few samples report substantially higher estimates. For these data, the linear regression coefficient is 0.143, which roughly translates into an additional team member every 7 years.

Finally, the table in Figure 5d provides some numerical examples for a selected subset of samples.

C. Code Quality

We measure 2 aspects of code quality: source code complexity and software maintainability.

Complexity. To measure software complexity we use McCabe’s cyclomatic complexity [23], one of the earliest—and still most widely used—software complexity metrics.

Despite having been introduced more than 40 years ago, it is still regarded as a useful metric to predict how defect-prone a software piece is [24], hence its use in many software measurement studies. For instance Warren et al. [25] characterized the evolution of modern websites by measuring different parameters, including the complexity of their source code. More recently, Hecht et al. included McCabe’s complexity in their analysis of the complexity evolution of Android applications [26].

The cyclomatic complexity (CC) of a piece of source code is computed from its control flow graph (CFG) and measures the number of linearly independent paths within the CFG. Mathematically, the cyclomatic complexity is given by:

$$CC = E - N + 2P \quad (4)$$

where E is the number of edges in the CFG, N the number of nodes, and P the number of connected components. There are many available tools for measuring this metric [27]–[29], but most of them only support a small subset of programming languages. To compute the cyclomatic complexity we use the Universal Code Count (UCC) [30]. UCC works over C/C++, C#, Java, SQL, Ada, Perl, ASP.NET, JSP, CSS, HTML, XML, JavaScript, VB, PHP, VBScript, Bash, C Shell Script, Fortran, Pascal, Ruby, and Python. Since our dataset contains source code written in different languages, UCC best suits our analysis. Still, it limited our experiments since it is not compatible with assembly source code, which appears in many projects in

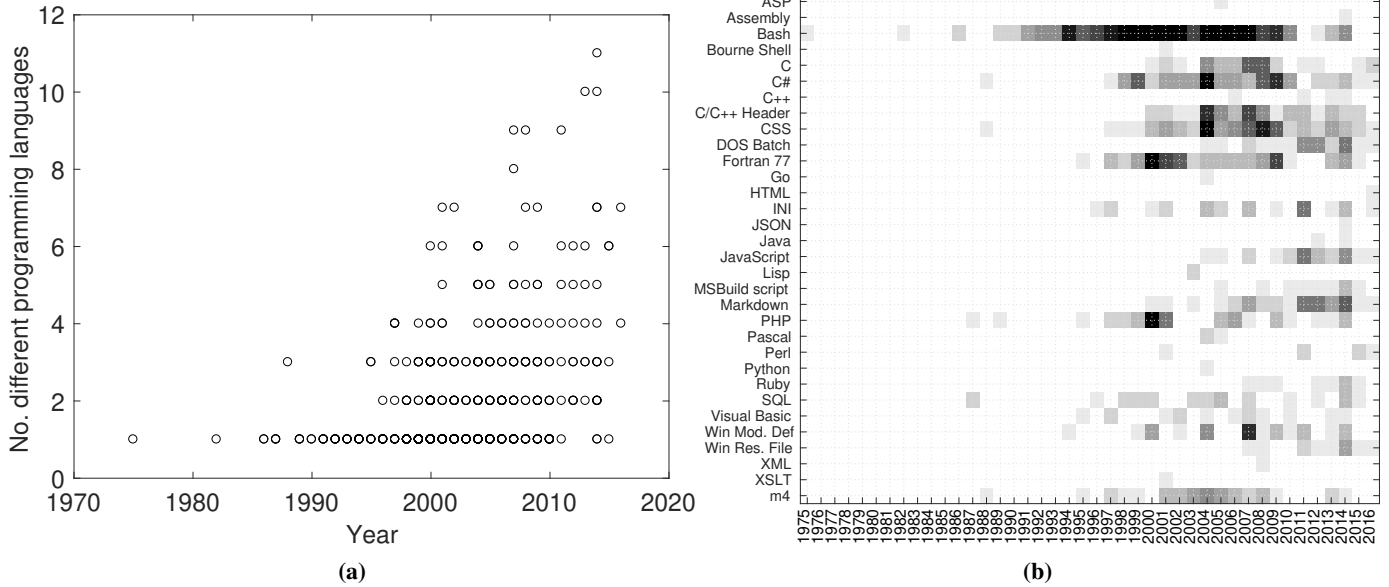


Fig. 4: (a) Number of programming languages per malware sample in the dataset. Darker circles represent overlapping data points. (b) Use of programming languages in malware samples. The chart shows the number of samples using a particular language each year, with darker colors representing higher number of samples.

our dataset (see Figure 4b). Filtering out samples that contain at least one source file in assembly left 144 samples for this analysis, i.e., 32% of the dataset.

Figure 7a shows the distribution of the average cyclomatic complexity per function for each analyzed sample. Most of the samples have functions with complexities between 3 and 8, with values in the interval [5, 6] being the most common. Overall, this can be seen as a supporting evidence of a generally modular design with a good break down into fairly simple functions and class methods. There are, however, some counterexamples. We observed a number of functions with complexity higher than 10, which exceeds McCabe’s recommended complexity threshold.

Maintainability. A concept often linked to software quality is source code maintainability. Maintainability is connected to complexity, since high complexity translates into poor maintainability [31]. Maintaining a software product generally involves fixing bugs and adding new features. The documentation found in the source code as code comments can have a great impact in facilitating this process. Thus, the comments-to-code ratio (or simply “comments ratio”) has traditionally been the metric used to measure documentation quality [32], [33].

Figure 6 shows the comments-to-code ratio for each sample, computed as the number of comment lines divided by the SLOCs. There is no clear pattern in the data, which exhibits an average of 17.2%, a standard deviation of 21.5%, and a median value of 10.7%. There are a few notable outliers, though. For example, W2KInstaller (2000) and OmegaRAT (2014) show ratios of 99.6% and 139.1%, respectively. Conversely, some samples have an unusually low comments ratio. We ignore if they were originally developed in this way or, perhaps, comments were cut off before leaking/releasing the code.

A more direct metric for measuring the maintainability of a software project is the maintainability index (MI) [32]. This metric is a quantitative estimator of how easy is to understand, support, and modify the source code of a project. A popular definition of MI is:

$$MI = 100 \frac{171 - 5.2 \ln(\bar{V}) - 0.23\bar{M} - 16.2 \ln(\overline{SLOC})}{171} \quad (5)$$

where \bar{V} is Halsteads average volume per module (another classic complexity metric; see [34] for details), \bar{M} is the average cyclomatic complexity per module, and \overline{SLOC} is the average number of source code lines per module. MI has been included in Visual Studio since 2007 [35]. Visual Studio flags modules with $MI < 20$ as difficult to maintain.

We use Equation (5) for computing an MI upper bound for each sample in our dataset. Note that we cannot estimate MI exactly since we do not have the average Halstead’s volume for each sample. Since this is a negative factor in Equation (5), the actual MI value would be lower than our estimate. Nevertheless, note that such factor contributes the lowest, so we expect our estimate to provide a fair comparison among samples.

Figure 7b shows the distribution of MI values grouped in quartiles. Most samples have an MI in the third quartile, and only 15 samples fall short of the recommended threshold of $MI < 20$, mainly because of a higher-than-expected cyclomatic complexity.

D. Comparison with Regular Software

In order to better appreciate the significance of the figures presented so far, we next discuss how they compare to those of a selected number of open source projects. We selected 9

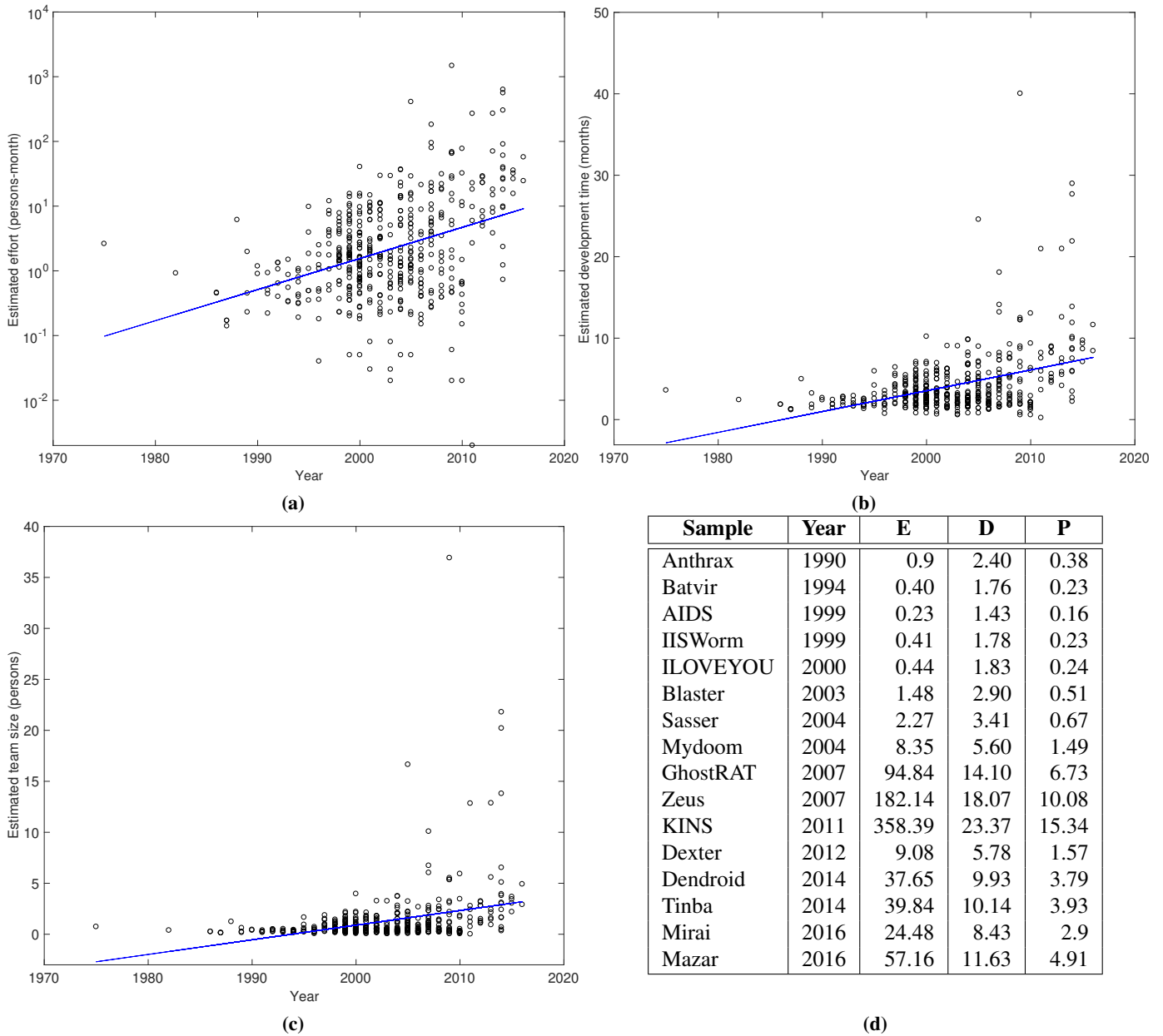


Fig. 5: COCOMO cost estimators for the malware samples in the dataset. (a) Effort (man-months). (b) Development time (months). (c) Team size (number of people). (d) Selected examples with effort (E), development time (D), and number of people (P). Note that in (a) and (b) the y-axis is shown in logarithmic scale.

software projects: 3 security products (the IPTables firewall, the Snort IDS, and the ClamAV antivirus); a compiler (gcc); a web server (Apache); a version control tool (Git); a numerical computation suite (Octave); a graphic engine (Cocos2d-x); and a Unix shell (Bash). The source code was downloaded from the web page of each project. For each project we then computed the metrics discussed above for malware. As in the case of malware, we use the COCOMO coefficients for organic projects. The results are shown in Table I in increasing order of SLOC count.

The first natural comparison refers to the size of the source code. Various malware samples from 2007 on (e.g. Zeus, KINS, Pony2, or SpyNet) have SLOC counts larger than those

of Snort and Bash. This automatically translates, according to the COCOMO model, into similar or greater development costs. The comparison of function point counts is alike, with cases such as Rovnix and KINS having an FPC greater than 1000, or SpyNet, with an FPC comparable to that of Bash. In general, only complex malware projects are comparable in size and effort to these two software projects, and they are still far away from the remaining ones.

In terms of comments-to-code ratio, the figures are very similar and there is no noticeable difference. This seems to be the case for the cyclomatic complexity, too. To further investigate this point, we computed the cyclomatic complexities at the function level; i.e., for all functions of all samples

Software	Version	Year	SLOC	E	D	P	FP	CC	CR	MI
Snort	2.9.8.2	2016	46,526	135.30	16.14	8.38	494.24	5.99	10.32	81.26
Bash	4.4 rc-1	2016	160,890	497.81	26.47	18.81	2,265.35	12.6	17.08	35.61
Apache	2.4.19	2016	280,051	890.86	33.03	26.97	4,520.10	5.45	23.42	62.58
IPtables	1.6.0	2015	319,173	1,021.97	34.80	29.37	3,322.05	4.35	27.33	49.98
Git	2.8	2016	378,246	1,221.45	37.24	32.80	4,996.44	5.21	12.15	60.78
Octave	4.0.1	2016	604,398	1,998.02	44.89	44.51	11,365.09	5.73	27.69	41.60
ClamAV	0.99.1	2016	714,085	2,380.39	47.98	49.61	10,669.97	6.36	33.57	63.01
Cocos2d-x	3.10	2016	851,350	2,863.02	51.47	55.63	16,566.78	3.47	17.55	68.18
gcc	5.3	2015	6,378,290	2,3721.97	114.95	206.37	90,278.41	3.56	31.24	64.08

TABLE I: Software metrics for various open source projects. **E:** COCOMO effort; **D:** COCOMO development time; **P:** COCOMO team size; **FP:** function points; **CC:** cyclomatic complexity; **CR:** comment-to-code ratio; **MI:** maintainability index.

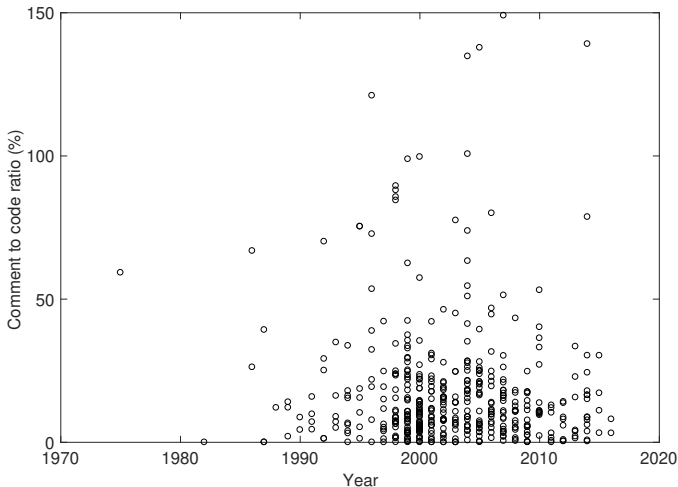


Fig. 6: Comments-to-code ratio for each sample in the dataset.

in both datasets. The histograms of the obtained values are shown in Figure 8. Both distributions are very similar, with a clear positive skewness. A Chi-squared and two-sample Kolmogorov-Smirnov tests corroborate their similarity for a significance level of $\alpha = 0.05$.

Regarding the maintainability index, no malware sample in our dataset shows an *MI* value higher than the highest for regular software—Snort, with $MI = 81.26$. However, Figure 7b shows that most *MI* values for malware source code fall within the second and third quartiles, which also holds for traditional software. Two notable exceptions are Cairuh and the Fragus exploit kit, which exhibit surprisingly low values (29.99 and 14.1, respectively).

IV. SOURCE CODE REUSE

This section presents the analysis of malware source code reuse in our dataset. Section IV-A first introduces the two techniques we use for clone detection. We present the clone detection results in Section IV-B. Finally, Section IV-C analyzes some of the clones found.

A. Detecting Code Clones

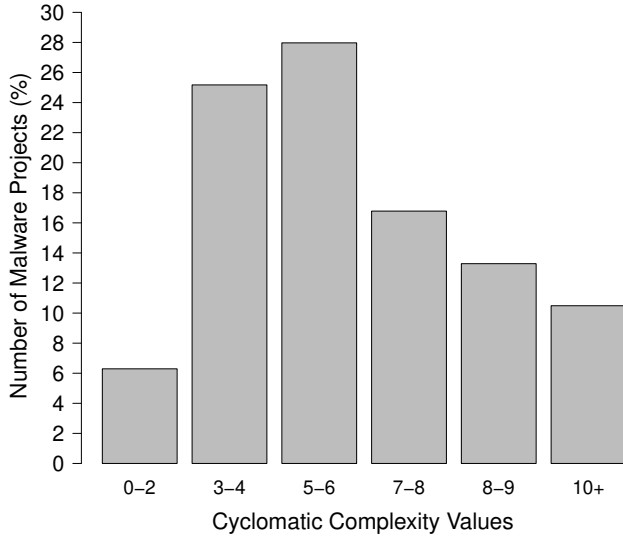
One challenge to detect code clones in our dataset is the diversity of programming languages used by the samples (Figure 4). Since samples written in C/C++ and Assembly

constitute 92% of our dataset (115 projects contain at least one file fully written in C/C++ and 304 projects contain at least one file fully written in Assembly), we need clone detection techniques that can at least cover these two languages. To achieve this goal, we use two code detection techniques detailed next.

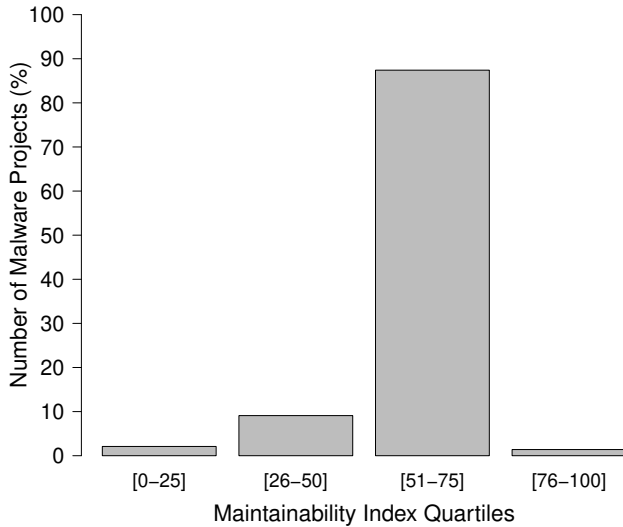
Deckard. Our first clone detection technique uses Deckard [36], a tool for detecting source code clones that was specifically designed to scale to large code bases such as the Java JDK and the Linux kernel, which comprise thousands of files. Deckard computes an Abstract Syntax Tree (AST) for each input source file. For each AST tree it produces a set of vectors of fixed length, each representing the structure of the AST subtree rooted at a specific node. These vectors are then clustered and each output cluster comprises of a set of similar ASTs, each a clone of the others. One advantage of AST-based clone detection techniques is that they can detect code clones with the same structure, despite some code changes such as variable renaming or different values assigned to a variable. Figure 9 shows an example of a code clone detected by Deckard despite changes in the names of the function, function parameters, and function variables. On the other hand, they can have high false positives since code snippets with the same AST structure may not necessarily be clones.

To limit the false positives, Deckard allows to specify the *minimum clone size* (as a number of AST tokens). This enables to remove short code sequences that appear in multiple samples simply because they are commonly used, but may not imply that the code was copied from one project into another. For example, sequences of C/C++ `#include` directives and loop statements, e.g., `for (i=0; i<n; i++)`, are not real clones. Deckard allows the user to set two additional parameters, the *stride* that controls the size of the sliding windows used during the serialization of ASTs, and the *clone similarity* used to determine if two code fragments are clones. In our experiments we tried different settings for these parameters. We obtained best results using minimum clone size of 100, stride of 2, and 1.0 similarity.

By default, Deckard offers support for the following languages: C/C++, Java, PHP, and dot. It can also support other languages if a grammar is available. Since our dataset contains a diversity of Assembly instruction sets (PPC, x86) and syntax specifications (Intel, AT&T), we would need to generate a grammar for each instruction set and syntax. That would



(a)



(b)

Fig. 7: Distributions of cyclomatic complexity (a) and maintainability index (b) for malware samples in the dataset.

require a significant effort to support Assembly samples (in some cases with a low return given the small number of samples for some combinations). Thus, we only use Deckard for finding clones among samples developed using C/C++. We apply Deckard on all projects of the same language (C/C++ or Assembly) simultaneously, which leverages Deckard’s design for efficiency.

Pairwise comparison. Our second clone detection technique compares two source code files using the Ratcliff-Obershelp algorithm for string similarity [37]. This technique measures how similar two sequences of characters are by computing the ratio between the matching characters and the total number of characters in the two sequences. The algorithm outputs

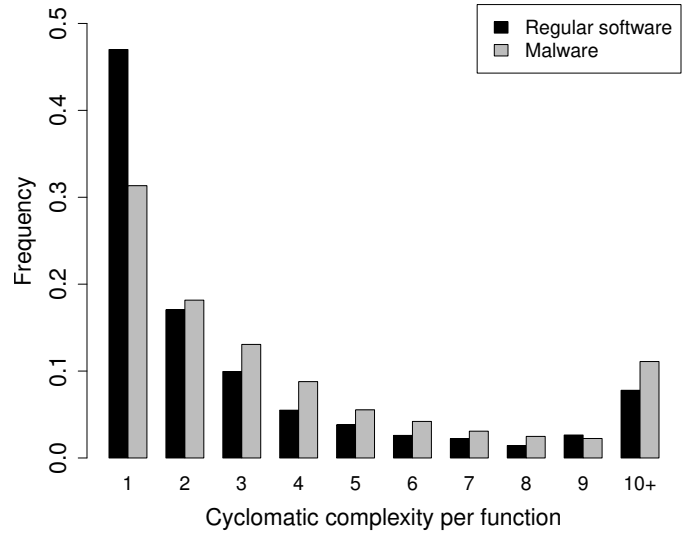


Fig. 8: Histograms of the cyclomatic complexity values computed at the function level for both malware and regular software samples.

```
int InfectExes(void){
WIN32_FIND_DATA d32;
char MyFile[256];
GetFileName(MyFile, sizeof(MyFile));
```

```
int InfectFiles(void){
WIN32_FIND_DATA w32;
char FileName[256];
GetFileName(FileName, sizeof(FileName));
```

Fig. 9: Two cloned code fragments sharing identical syntactic structure but with different names for method and variables.

matching blocks of characters containing the longest common subsequence (LCS) and characters neighboring the LCS that are similar in both strings. We consider a code clone every matching block satisfying a minimum length. We experimented with different minimum length values, achieving best results with a minimum length of 10 SLOC for Assembly and 5 SLOC for C/C++. The higher threshold for Assembly is due to its lower abstraction compared to C/C++.

Since this technique operates on two input files, for each pair of samples we compare every pair of source code files, one file from each sample. To avoid missing clones because of simple changes to the copied code we preprocess the files using these rules: remove blank spaces, tabs, and newline characters; convert to lower case; and translate the character set to UTF-8.

The main advantages of this technique are its simplicity, that it can handle any text-based language, and very low false positives. The main disadvantages are potentially high false negatives and low efficiency. False negatives can happen because this technique only detects reuse of identical code

fragments; it will not detect a clone if the code is modified (e.g., variable renaming). Low efficiency is due to the quadratic number of comparisons needed.

B. Clone Detection Results

This section presents the clone detection results using Deckard and the pairwise comparison technique. We manually examine the clones detected by both techniques to determine whether they are true positives or false positives. During our initial manual analysis we found that a large number of clones were different instances of the same cases. To speed up the manual analysis of the more than 10K detected clones, we use a clustering approach based on regular expressions and fuzzy hashing [38] to automatically group nearly identical clones. The analyst then labels each cluster, which considerably speeds up the manual analysis since the number of clusters is nearly two orders of magnitude smaller than the number of clones.

Table II summarizes the code clone detection results. For each language and detection technique, it shows the number of detected clones, the split of those clones into true and false positives, the total and per-pair runtime, and statistics on the SLOC size of the detected clones.

The C/C++ results show that Deckard detects 7,655 clones compared to 959–1,040 using the pairwise comparison technique. However, of the 7,655 Deckard clones, 87% are false positives, compared to 6.4% (raw) and 5.7% (normalized) using pairwise comparison. The very high false positive rate of Deckard is due to its AST representation, which ignores type information and constant values. For example, an array definition like `static unsigned long SP1[64] = { 0x01010400L, ... }` is considered by Deckard a clone of `static unsigned char PADDING[64] = {0x80, ... }`, even if both arrays are of different type and are initialized with different values. As another example, the function invocation `CopyFile(wormpath, "gratis.mp3.exe", 0)` is considered a clone of `add_entry(TABLE_KILLER_FD, "\x0D\x44\x46\x22", 4)`.

Clone lengths. The average clone size using Deckard is 112.3 SLOC, 52.7 using normalized pairwise comparison, and 25.9 using raw pairwise comparison. Thus, while the number of TPs is similar using Deckard and raw pairwise comparison, Deckard is able to find longer (i.e., higher quality) clones. Surprisingly, the number of TPs found by the raw pairwise comparison is higher than those found by the normalized pairwise comparison. This happens because the raw pairwise comparison breaks longer clones into multiple smaller clones, which increases the number of detected clones, but produces shorter (i.e., lower quality) clones. Thus, normalization helps to find larger clones. For example, in the C/C++ code, normalization allowed us to discover a large true clone consisting of 22K SLOC (detailed in Section IV-C).

Figure 10 shows the distributions of length values for raw and normalized clones in both languages. In both distributions the number of clones becomes smaller as the size grows, which

translates into positively skewed distributions. Noteworthy exceptions of this trend are clones in the range of 50-99 and 100-199 lines in Assembly code, and also clones larger than 100 lines in C/C++. These peaks are related to the nature of the detected clones, discussed in Section IV-C.

Small clones (i.e., shorter than 20 lines) are fairly common in both C/C++ and Assembly. Manual inspection revealed different explanations for each language. In the case of Assembly, such small cloned fragments are usually related to control flow structures such as loops, which employ the same sequence of instructions regardless of the actual data values. In addition, we also observed reuse of the names used to label Assembly code segments. In the case of C/C++ projects, we found that small clones are generally associated with preprocessor directives such as `#typedef`, `#define`, and `#include`. These short clones also include sequences of instructions to initialize data structures (e.g., arrays), and generic sequences often found at the end of functions that release allocated variables before returning a value. These clones are often exact copies of each other and are common in malware samples from the same family.

On the other hand, clones larger than 20 lines represent less than 50% of the detected clones in both languages. In particular, 350 assembly clones are larger than 20 lines. The number of Assembly clones is also notably smaller than the total number of Assembly source code files in our dataset, which is close to 800. Comparing the average lengths of Assembly source code files and large clones provides similar results. Large Assembly clones are 102.87 SLOC on average, while Assembly source files in the dataset are 498.18 SLOC. In the case of C/C++ clones, the figures are comparable. We identified 450 C/C++ clones out of 984 that are larger than 20 lines. The total number of C/C++ files in the dataset is 2,981, which almost doubles the total number of clones found. The average length of large C/C++ clones depends greatly on the normalization process: it is just 102.01 SLOC without normalization and 231.28 SLOC after normalization.

We analyzed the size of clones found in samples belonging to the same family or developed by the same author. To do so, we selected 4 pairs of projects developed in C/C++ and Assembly for which we had ground truth, i.e., they are known to share authorship or are variants of the same family. Specifically, we selected two banking trojans (Zeus & Kins) and two worms (Cairuh.A & Hunatch.a) for the C/C++ case; and two pairs of viruses written in Assembly: (methaphor.1.d & Simile) and (EfishNC & JunkMail). The average clone sizes for the C/C++ couples are 57.40 lines (37 clones) for the banking trojans and 13.86 lines (60 clones) for the worms. In the case of the Assembly samples, the average clone lengths are 179.72 lines for methaphor.1.d & Simile (54 clones) and 47.06 lines for EfishNC & JunkMail (30 clones).

Clone file distribution. Figure 11 shows the distribution of the number of files in which a clone appears. As it can be seen, in approximately 80% of the cases clones are found in just two files. The number of clones appearing in 3 or more files decreases considerably for both languages. In the case

Language	Detection technique	Clones	TPs	FPs	Runtime		Clone size (SLOC)				
					All (h)	Pair (s)	Average	Std. Dev.	Median	Min.	Max.
C/C++	Deckard	7,655	984	6,671	1.4	0.8	112.3	1,441.2	17	7	22,658
C/C++	Pairwise (raw)	1,040	973	67	197.7	107.5	25.9	69.4	7	5	1,157
C/C++	Pairwise (with normalization)	959	904	55	209.3	115.0	52.7	762.9	7	5	22,709
Assembly	Pairwise (raw)	974	972	2	97.8	7.6	50.1	100.1	19	10	1,084
Assembly	Pairwise (with normalization)	704	703	1	101.0	7.9	58.6	102.8	21	10	1,084

TABLE II: Clone detection results for C/C++ and Assembly projects using the two detection techniques. It first shows the number of code clones detected and their split into true positives and false positives. Next, it shows the total runtime in hours and the average runtime for each pair of projects in seconds. Finally, it shows clone size statistics.

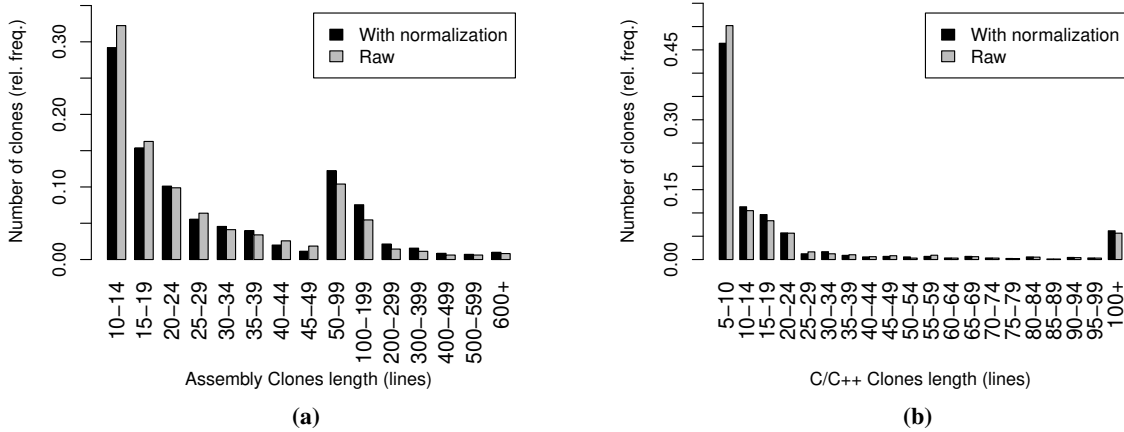


Fig. 10: Distribution of code clone sizes for Assembly and C/C++ languages

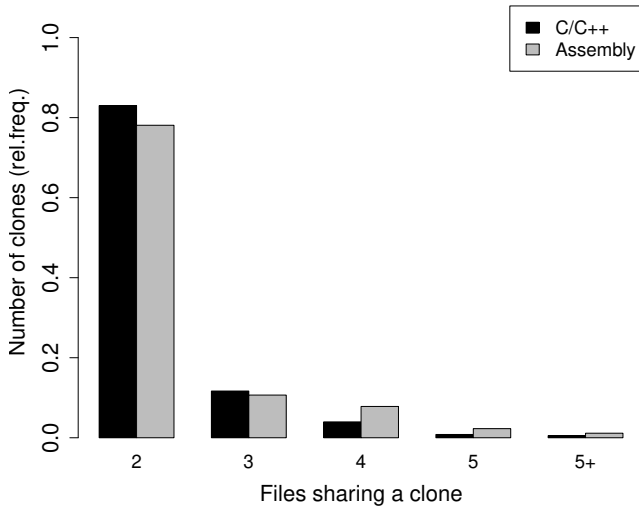


Fig. 11: Distribution of clones in 2 or more files.

of C/C++, the fraction of clones appearing in 3, 4, 5, and more than 5 files is 0.11, 0.04, 0.008, and 0.005 respectively. The pattern for Assembly clones is similar, though clones are slightly more widespread as they appear in more than 3 files more often than C/C++ clones.

Runtime. Deckard is almost two orders of magnitude faster than the pairwise comparison technique, finding clones across

all 115 C/C++ samples in 1.4 hours, compared to 8 days for the pairwise comparison. Such efficiency difference is due to Deckard parsing each file only once and to its clustering. We observe that the pairwise comparison on Assembly run much faster than on C/C++. This is due to the C/C++ projects containing more, and longer, files.

To conclude, our clone detection results show a significant amount of code reuse despite the relatively small number of samples, e.g., 984 clones of 112 SLOC on average across 115 C/C++ projects. We detail the type of clones found in Section IV-C. This constitutes an evidence that malware authors copy functionalities useful to their goals that may appear in leaked malware source code (or malware they have purchased). Of the two techniques evaluated, Deckard runs very fast and finds more and longer clones, but produces very high false positives. On the other hand, the pairwise comparison technique produces much lower (but a non-negligible 6%) false positives, but runs two orders of magnitude slower.

C. Clone Analysis

We next discuss the purpose of the reused code fragments we found. Through manual analysis, we classified clones into four main groups according to the functionality they provide. For additional details, Tables IV and V summarize the main features of a selection of representative clones of these categories we found for both C/C++ and Assembly.

Operational data structures and functions. One large group of code clones consists of libraries of data structures and

	Assembly	C/C++	Avg. length (SLOC)	
			Assembly	C/C++
Type A	22	42	364.86	193.83
Type B	55	41	238.94	151.70
Type C	15	8	101.6	121.62
Type D	8	9	613.62	2587.33

TABLE III: Clone types frequencies for Assembly and C/C++ by types. **Type A:** Operational data structures and functions, **Type B:** Core malware artifacts, **Type C:** Data clones, **Type D:** Anti-analysis capabilities.

the associated functions to manipulate system and networking artifacts, such as executable file formats (PE and ELF) and communication protocols (TCP, HTTP) and services (SMTP, DNS). We also observe a number of clones consisting of headers for several API functions needed to interact with the Windows kernel, such as the 3,054 lines long clone shared by W32.Remhead and W32.Rovnix.

Core malware artifacts. The second category of clones consists of code that implements properly malicious capabilities, such as infection, spreading, or actions on the victim. For instance, the W32.Dopebot botnet contains shellcode to exploit the CVE-2003-0533 vulnerability, and the same shellcode is found in the W32.Sasser worm. Another good example of this practice is the network sniffer shared by W32.NullBot and W32.LoexBot.

Data clones. Some of the clones are not code, but rather data structures that appear in multiple samples. An example is the array of frequent passwords present in both W32.Rbot and W32.LoexBot. Another example is the list of strings found in W32.Hunatchab and W32.Branko, containing the process names associated to different commercial AV (antivirus) software, which both bots try to disable. Furthermore, some samples also share strings containing IP addresses, for example the Sasser worm and the Dopebot botnet.

Anti-analysis capabilities. One of the most noticeable examples of this is the packer included in the W32.Cairuh worm, which is shared by the W32.Hexbot botnet. Its size is 22,709 lines and it is the biggest clone we found in our dataset. Another remarkable example is the metamorphic engine shared by the Simile and Metaphor.1d viruses, consisting of more than 10,900 lines of assembly code. Other examples of reused anti-analysis modules can be found in W32.Antares and W32.Vampiro, which share the same polymorphic engine, and also in W95.Babyloni, and W32.Ramlide, which share the same packing engine. Finally, we also found a number of reused instances of code to kill running AV processes, such as the clone found in Hunatchab.c and Branko.c

In order to estimate the number of clones for each category, we randomly sampled the set of found clones and selected 100 for each language. The 200 clones were then manually labeled according to the four semantic categories described above. Table III shows the distribution of clones together with their average length. As it can be seen, most of the cases belong to types A (operational data structures and functions) and B (core malware artifacts). In the case of Assembly, both categories amount for 84% of all clones, while in the case

of C/C++ core malware artefacts alone constitute 55% of the clones. In both cases, data clones and anti-analysis capabilities are considerably less frequent.

With respect to their lengths, Type D assembly clones are noticeably larger than clones in other categories. This is due to the presence of polymorphic and packing engines in this category, which are relatively complex code samples. Contrarily, data clones (Type C) are generally shorter, which is reasonably given their nature. In general, Assembly clones are bigger than their C/C++ counterpart, which is in line with the general results described above.

The data in Table III suggests that clone size highly depends on the nature of shared features. This is especially evident for those clones labeled as type C. In addition, the results reveal that the inclusion of evasion and anti-analysis capabilities has a noticeable impact in the size of malicious codebases.

Last but not least, we observed that in most cases code reuse usually takes place in short time spans, i.e., the samples sharing a particular fragment of code have been developed within 1-4 years of each other. This could evidence that the same author has participated in the development of such samples, or else that collaborating groups share previously developed artifacts that can be easily reused.

D. Code Sharing with Benign Source Code

We also explored if code cloning between malicious and benign source happens to the same extent as it does among malware samples. For this purpose, we analyzed the set of major open source projects used in Section III-D and extended this set adding the Bitcoin cryptocurrency and Linux kernel source code master branches.

We followed a similar approach as for the main cloning experiment. However we decided to lean exclusively on Deckard since it is faster, especially when dealing with large codebases. We ran Deckard ten times, one time per project, combining the open source project with the whole malicious source code dataset each time. Then, we processed the output as outlined in Section IV-A. Despite the high FP ratios obtained, in the experiment we found code cloning cases in 4 out of 10 source code projects. Snort, Iptables, Bash, Apache, Cocos2d and the Bitcoin projects do not share any source code snippet with any of the samples included in our dataset. We did find up to 210 relevant code clones (larger than 5 lines) in gcc, the Linux kernel, Git, and clamAV. Surprisingly, all the cloned source clones found in gcc, clamAV, and the Linux kernel are part of the Zlib compression library. In particular, the cloned fragments appear in a C header (`defutil.h`) and 3 C source files (`infbak.h`, `inflate64.c`, and `inflate.c`) in the open source project. In the malicious source code dataset, the same fragments are contained in the files `deflate.h`, `infbak.c`, `inflate.c`, and `infrees.c` included in the server source code of the XtremeRAT botnet. Git shares a set of data structures with the samples w32.Rovnix and w32.Carberp. The content of these data structures is used as padding in the implementation of the SHA1 and MD5 hashing algorithms. The shared code is located in the files

Length	Samples	Description	Category
22709	W32.Cairuh.A (Worm, 2009) W32.Simile (Worm, 2009) W32.HexBot2 (Bot, 2009)	Array containing a raw dump of an executable packing tool used after the compilation of the main binary	Anti-analysis capabilities
3054	W32.Remhead (Trojan,2004) W32.Rovnix (Virus,2014) W32.Carberp (Virus,2013)	Define several data structures used for interacting with the NT Kernel through its Native API.	Operational data structures and functions
2546	W32.MyDoom (Worm, 2004) W32.HellBot (Bot, 2005)	Share the code for deploying a SMTP relay and a fake DNS server.	Core malware artifacts
1323	W32.NullBot (Bot, 2006) W32.LoexBot (Bot, 2008)	Includes hardcoded IRC commands used to send instruction to infected clients	Data clones
328	W32.Dopebot.A (Bot, 2004) W32.Dopebot.B (Bot, 2004) W32.Sasser (Worm, 2004)	Shellcode employed for exploiting, the CVE-2003-0533 vulnerability [39]	Core malware artifacts

TABLE IV: Examples of code clones found in C/C++.

Length	Samples	Description	Category
10917	W32.Metaph0r.1d (Virus, 2002) W32.Simile (Virus, 2002)	These samples contain a complete metamorphic engine coded in pure x86 assembly.	Anti-analysis capabilities
1283	W32.EfishNC (Virus,2002) W32.Junkmail (Virus,2003)	Both declare the same structs and function headers for infection and spreading through email	Core malware artifacts
233	Lin32.Tahorg (Virus, 2003) Lin32.GripB (Virus, 2005)	Share structures and routines for reading and modifying ELF files. Includes a list of offsets for the different sections in the EFL header.	Operational data structures and functions.
1009	W32.Relock (Virus,2007) W32.Mimix (Virus,2008) W32.Impute (Virus, 2013)	These samples share an assembly implementation of Marsenne Twister PRNG. W32.Relock was the first malware piece using the <i>virtual code</i> obfuscation technique [40] which is based in memory reallocation.	Anti-analysis capabilities
100	Gemini (Virus,2003) EfisNC (Virus,2008) JunkMail (Virus,2013)	Contains offsets pointing to several functions within a MZ (DOS Executable files) manipulation library	Data clones

TABLE V: Examples of code clones found in Assembly.

sha1.c in the git source code tree and also in the files md5.c and md5.cpp included in the code of Rovnix and Carberp, respectively. The average size of the cloned fragments is 102 lines.

V. DISCUSSION

We next discuss some aspects of the suitability of our approach, the potential limitations of our results, and draw some general conclusions.

Suitability of our approach. Software metrics have a long-standing tradition in software engineering and have been an important part of the discipline since its early days. Still, they have been subject to much debate, largely because of frequent misinterpretations (e.g., as performance indicators) and misuse (e.g., to drive management) [21]. In this work, our use of certain software metrics pursues a different goal, namely to quantify how different properties of malware as a software artifact have evolved over time. Thus, our focus here is not on the accuracy of the absolute values (e.g., effort estimates given by COCOMO), but rather on the relative comparison of values between malware samples, as well as with benign programs, and the trends that the analysis suggests.

The use of comments as an efficient documentation method has been questioned by several experts. Among the stated reasons, it has been argued that often comments add redundant description of code functionality instead of clarifying design decisions and the underlying algorithmic workflow. However others authors defend that good quality comments are still valuable and necessary, specially in large collaborative projects [41]. The validity of the comments-to-code ratio nowadays

could also be criticized, given the trend to develop source code using automatically generated documentation frameworks. This trend may have reduced over time the reliability of comments-to-code ratio as a maintainability metric. Nevertheless, during our analysis we did not find any samples, using such approaches, as the only delivered documentation with the (recent) samples, are the comments written by the authors. Thus, comments seem to still play an important role in the development of malware.

As for the case of detecting code reuse, the techniques we used represent standard approaches to the problem. By using two different approaches, we obtain complementary and more robust results. For example, we can use the pairwise comparison technique to analyze assembly samples not supported by Deckard, while Deckard’s AST-based approach resists certain classes of evasion attacks, e.g. variable and function renaming, which affect the pairwise comparison technique.

Limitations. Our analysis may suffer from several limitations. Perhaps the most salient is the reduced number of samples in our dataset. However, as discussed in Section II, obtaining source code of malware is hard. Still, we analyze 456 samples, which to the best of our knowledge is the largest dataset of malware source code analyzed in the literature. While the exact coverage of our dataset cannot be known, we believe it is fairly representative in terms of different types of malware. It should also be noted that in our study, the sample concept refers to a malware family. Thus, we are not only covering 456 binary samples but a wider set of potential variants. The wide gap between the number of binary samples found in the wild and the number of malware families has been previously discussed in the community. A recent study [42] examined

23.9M samples and classified them into 17.7K families (i.e., three orders of magnitude smaller). While this phenomenon is due to different reasons, the most prominent one is the use of polymorphism and other advanced obfuscation methods employed by malware authors. We note that 428 out of 17.7K is a respectable 2.4% coverage.

In particular, we believe the coverage of our dataset is enough to quantify and analyze the trends in malware evolution (size, development cost, complexity), but we do not attempt to analyze the evolution of malware code reuse. Since we only have one (or a few) versions for each malware family and a limited number of families, our dataset may miss important instances of malware code reuse. Thus, we have focused on analyzing what type of code we observe being reused in our dataset. As we collect more samples, we should be able to obtain a more representative picture of the code sharing phenomenon in malware creation, going beyond the findings we have reported.

Another limitation is selection bias. Collection is particularly difficult for newest samples and more sophisticated samples (e.g., those used in targeted attacks) have not become publicly available. We believe those samples would emphasize the increasing complexity trends that we observe.

Finally, even if the employed pairwise comparison code clone detection technique is very simple and has poor scalability, it has performed remarkably well in terms of false positives compared with Deckard, a more sophisticated tool based on comparing syntactic structures. The large amount of false positives obtained with Deckard can be partially explained because of the way in which malware writers reuse code. As discussed in section IV, cloned code fragments are often core artifacts such as shellcodes or obfuscation engines. Given the nature of these artifacts, malware authors are forced to reuse them in a copy and paste fashion rather than rewriting some of their content. This makes very uncommon to find partial clones, consisting on slightly modified code fragments. For this reason, and despite the great variety of code-clone detection techniques available in the literature [43], [44], it is unclear whether employing more sophisticated approaches might lead to finding significantly more clones when dealing with plain malware source code.

In addition, clone detection tools based on syntactic structures depend greatly on the set of selected features. In the case of Deckard, leaving out data types and literals definitely contributes to achieving poorly accurate results, especially in our use case which differs from standard use cases for this kind of tools.

Deckard could be improved in many ways in order to obtain more precise results. Two natural ideas would be combining syntactic and semantic features, and introducing a similarity metric after the clustering step. However, in this paper we just aimed at comparing the performance of a naive approach (diff-based clone detection) against an already proposed tool, and therefore we decided to use Deckard out of the box, leaving out any improvement.

Main conclusions and open questions. In the last 40 years

the complexity of malware, considered as a software product, has increased considerably. We observe increments of nearly one order of magnitude per decade in aspects such as the number of source code files, source code lines, and function point counts. This growth in size can be attributed to various interconnected reasons. On the one hand, malicious code has progressively adapted to the increasing complexity of the victim platforms they target. Thus, as Operating Systems evolved to offer richer application programming interfaces (API), malware authors rapidly leveraged them to achieve their purposes. This translated into larger and more complex samples with a variety of computing and networking capabilities. On the other hand, malware authors have clearly benefited from newer and richer integrated development environments (IDEs), frameworks, and libraries. This explains the increasing modularity seen in the most recent samples—and, especially, the rise of complex, multi-language malware projects that would be otherwise unmanageable.

One interesting question is whether this trend will hold in time. If so, we could soon see malware specimens with more than 1 million SLOC. To translate these numbers into real-world examples, in the near future we could witness malware samples exceeding three times in size open source projects like Git or the Apache web server (see Table I). However, evolving into large pieces of software will surely involve a higher amount of vulnerabilities and defects. This has been already observed (and exploited), e.g., in [45] and [46]. In addition, such evolution requires larger efforts and thus possibly larger development teams. While we observe the trend we have not examined in detail those development teams. For this, we could apply authorship attribution techniques for source code [47], [48]. More generally, the results shown in this paper provide quantified evidence of how malware development has been progressively transforming into a fully fledged industry.

VI. RELATED WORK

While malware typically propagates as binary code, some malware families have distributed themselves as source code. Arce and Levy performed an analysis of the Slapper worm source code [49], which upon compromising a host would upload its source code, compile it using gcc, and run the compiled executable. In 2005, Holz [50] performed an analysis of the botnet landscape that describes how the source code availability of the Agobot and SDBot families lead to numerous variants of those families being created.

Barford and Yegneswaran [8] argue that we should develop a foundational understanding of the mechanisms used by malware and that this can be achieved by analyzing malware source code available on the Internet. They analyze the source code of 4 IRC botnets (Agobot, SDBot, SpyBot, and GTBot) along 7 dimensions: botnet control mechanisms, host control mechanisms, propagation, exploits, delivery mechanisms, obfuscation, and deception mechanisms.

Other works have explored the source code of exploit kits collected from underground forums and markets. Exploit kits are software packages installed on Web servers (called exploit

servers) that try to compromise their visitors by exploiting vulnerabilities in Web browsers and their plugins. Different from client malware, exploit kits are distributed as (possibly obfuscated) source code. Kotov and Massacci [9] analyzed the source code of 30 exploit kits collected from underground markets finding that they make use of a limited number of vulnerabilities. They evaluated characteristics such as evasion, traffic statistics, and exploit management. Allodi et al. [51] followed up on this research by building a malware lab to experiment with the exploit kits. Eshete and Venkatakrishnan describe WebWinnow [52] a detector for URLs hosting an exploit kit, which uses features drawn from 40 exploit kits they installed in their lab. Eshete et al. follow up this research line with EKHunter [45] a tool that given an exploit kit finds vulnerabilities it may contain, and tries to automatically synthesize exploits for them. EKHunter finds 180 in 16 exploit kits (out of 30 surveyed), and synthesizes exploits for 6 of them. Exploitation of malicious software was previously demonstrated by Caballero et al. [46] directly on the binary code of malware samples installed in client machines.

The problem of detecting duplicated or cloned code was first approached using simple text-matching solutions. The technique described in [53] consists in a pairwise comparison among source code files looking for a coincidence. While this allows to find exact copies of source code, it does not scale well and may incur in performance issues. In any case, note that text-matching approaches require a preliminary normalization step such as the one used in this work. A second group of techniques rely on data structures such as graphs or trees to represent the syntactic structure of the programs [54], [55], together with an appropriate similarity measure among them. Other works have proposed solutions based on a lexical analysis of source files. These techniques convert the source code sentences into lists of tokens, which are then compared to detect duplicated subsequences [11], [56].

In the case of code sharing in malware, most existing work has focused on binary objects [57], [58]. Even though the results reported are reasonable, one potential limitation of such works is that modern compilers can perform different optimization and cleaning tasks (e.g., loop unraveling, symbol stripping, etc.) to generate optimal binary objects in terms of size and memory consumption. This could end up altering the structure of the original code and deleting many valuable and meaningful artifacts [59]. Contrarily, working directly with the original source code gives us more precise insights on the functionality that is more frequently reused across samples.

VII. CONCLUSION

In this paper, we have presented a study on the evolution of malware source code over the last four decades, as well as a study of source code reuse among malware families. We have gathered and analyzed a dataset of 456 samples, which to our knowledge is the largest of this kind studied in the literature. Our focus on software metrics is an attempt to quantify properties both of the code itself and its development process. The results discussed throughout the paper provide a numerical

evidence of the increase in complexity suffered by malicious code in the last years and the unavoidable transformation into an engineering discipline of the malware production process.

ACKNOWLEDGMENTS

This work was supported by the Spanish Government through MINECO grants SMOG-DEV (TIN2016-79095-C2-2-R) and DEDETIS (TIN2015-7013-R), and by the Regional Government of Madrid through grants CIBERDINE (S2013/ICE-3095) and N-GREENS (S2013/ICE-2731).

REFERENCES

- [1] "Symantec's 2015 internet security threat report," <https://www.symantec.com/security-center/archived-publications>, accessed: 2017-12-6.
- [2] P. Security, "27% of all recorded malware appeared in 2015," <http://www.pandasecurity.com/mediacenter/press-releases/all-recorded-malware-appeared-in-2015>, accessed: 2016-04-6.
- [3] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. Berkeley, CA, USA: USENIX Association, 2011.
- [4] Grier et al., "Manufacturing compromise: The emergence of exploit-as-a-service," in *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2012.
- [5] G. Stringhini, O. Hohlfeld, C. Kruegel, and G. Vigna, "The harvester, the botmaster, and the spammer: On the relations between the different actors in the spam landscape," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14. New York, NY, USA: ACM, 2014.
- [6] K. Thomas, D. Huang, D. Wang, E. Bursztein, C. Grier, T. J. Holt, C. Kruegel, D. McCoy, S. Savage, and G. Vigna, "Framing dependencies introduced by underground commoditization," in *Workshop on the Economics of Information Security*, 2015.
- [7] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2, 2014.
- [8] P. Barford and V. Yegneswaran, *Malware Detection*. Springer, 2007, ch. An Inside Look at Botnets.
- [9] V. Kotov and F. Massacci, "Anatomy of Exploit Kits: Preliminary Analysis of Exploit Kits As Software Artefacts," in *International Conference on Engineering Secure Software and Systems*, 2013.
- [10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, 2009.
- [11] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002.
- [12] A. Rahimian, R. Ziarati, S. Preda, and M. Debbabi, "On the reverse engineering of the citadel botnet," in *Foundations and Practice of Security*. Springer, 2014.
- [13] 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*, Evry, France, September 2016.
- [14] "Vx heaven," <http://vxheaven.org/>, accessed: 2017-06-20.
- [15] "cloc," <http://github.com/AIDanial/cloc>, accessed: 2015-09-22.
- [16] V. Nguyen, S. Deeds-rubin, T. Tan, and B. Boehm, "A sloc counting standard," in *COCOMO II Forum 2007*, 2007.
- [17] A. J. Albrecht, "Measuring Application Development Productivity," in *IBM Application Development Symp.*, I. B. M. Press, Ed., Oct. 1979.
- [18] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Trans. Softw. Eng.*, vol. 9, no. 6, Nov. 1983.
- [19] C. Jones, "Backfiring: Converting lines-of-code to function points," *Computer*, vol. 28, no. 11, Nov. 1995.
- [20] —, "Programming languages table, version 8.2," 1996.
- [21] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [22] B. W. Boehm, *Software Engineering Economics*. Prentice-Hall, 1981.

- [23] T. J. McCabe, "A complexity measure," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976.
- [24] C. Ebert and J. Cain, "Cyclomatic complexity," *IEEE Software*, vol. 33, no. 6, pp. 27–29, 2016.
- [25] P. Warren, C. Boldyreff, and M. Munro, "The evolution of websites," in *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 1999, pp. 178–185.
- [26] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 236–247.
- [27] "Jhawk," <http://www.virtualmachinery.com/jhawkprod.htm>, accessed: 2016-04-4.
- [28] "Radon," <https://pypi.python.org/pypi/radon>, accessed: 2016-04-4.
- [29] "Eclipse metrics plugin," <https://marketplace.eclipse.org/content/eclipse-metrics>, accessed: 2016-04-4.
- [30] "UCC," http://csse.usc.edu/ucc_wpl/, accessed: 2016-04-4.
- [31] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, ser. EWSPT '96. London, UK, UK: Springer-Verlag, 1996.
- [32] P. Oman and J. Hagemester, "Metrics for assessing a software system's maintainability," in *Proc. Conf. on Software Maintenance*, 1992.
- [33] M. J. B. Garcia and J. C. G. Alvarez, "Maintainability as a key factor in maintenance productivity: a case study," in *1996 Proceedings of International Conference on Software Maintenance*, Nov 1996, pp. 87–93.
- [34] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.
- [35] "Code metrics values in microsoft's visual studio," <https://msdn.microsoft.com/en-us/library/bb385914.aspx>, accessed: 2018-03-19.
- [36] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.
- [37] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, 1988.
- [38] "ssdeep," <https://ssdeep-project.github.io/>, accessed: 2017-11-01.
- [39] "Cve-2003-0533 vulnerability," <https://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0533>, accessed: 2017-06-20.
- [40] "Virtual code obfuscation by roi g biv," <http://vxheaven.org/lib/vrg19.html>, accessed: 2017-06-20.
- [41] E. Torres, "Why code comments still matter," <https://cacm.acm.org/blogs/blog-cacm/225574-why-code-comments-still-matter/fulltext>, accessed: 2018-10-2.
- [42] C. Lever, P. Kotzias, D. Balzarotti, J. Caballero, and M. Antonakakis, "A lustrum of malware network communication: Evolution and insights," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 788–804.
- [43] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," 2007.
- [44] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *International Journal of Computer Applications*, 2016.
- [45] B. Eshete, A. Alhuzali, M. Monshizadeh, P. Porras, V. Venkatakrishnan, and V. Yegneswaran, "EKHunter: A Counter-Offensive Toolkit for Exploit Kit Infiltration," in *Network and Distributed System Security Symposium*, February 2015.
- [46] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input Generation Via Decomposition and Re-Stitching: Finding Bugs in Malware," in *ACM Conference on Computer and Communications Security*, Chicago, IL, October 2010.
- [47] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," *J. Syst. Softw.*, vol. 81, no. 3, Mar. 2008.
- [48] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing Programmers via Code Stylometry," in *USENIX Security Symposium*, 2015.
- [49] I. Arce and E. Levy, "An analysis of the slapper worm," *IEEE Security & Privacy*, vol. 1, no. 1, 2003.
- [50] T. Holz, "A short visit to the bot zoo," *IEEE Security & Privacy*, vol. 3, no. 3, 2005.
- [51] L. Allodi, V. Kotov, and F. Massacci, "MalwareLab: Experimentation with Cybercrime Attack Tools," in *USENIX Workshop on Cyber Security Experimentation and Test*, Washington DC, August 2013.
- [52] B. Eshete and V. N. Venkatakrishnan, "WebWinnow: Leveraging Exploit Kit Workflows to Detect Malicious Urls," in *ACM Conference on Data and Application Security and Privacy*, 2014.
- [53] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995.
- [54] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998.
- [55] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics." in *icsm*, vol. 96, 1996.
- [56] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, 2006.
- [57] H. Huang, A. M. Youssef, and M. Debbabi, "Binsequence: Fast, accurate and scalable binary code reuse detection," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017.
- [58] J. Jang and D. Brumley, "Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006)," *CyLab*, 2009.
- [59] N. E. Rosenblum, B. P. Miller, and X. Zhu, "Extracting compiler provenance from program binaries," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2010.



forensics and reverse engineering.

Alejandro Calleja is a Ph.D. candidate in the Computer Security Lab (COSEC) at the Department of Computer Science and Engineering of Universidad Carlos III de Madrid under the supervision of Dr. Juan Tapiador. He holds a B.Sc and a M.Sc in computer science from Universidad Carlos III de Madrid. His main research line is automatic malware source code analysis. He is also also interested in several topics related with information and systems security, such as security in smartphone and mobile devices, security in embedded devices, computer



such as ACSAC, ACNS, DIMVA, ESORICS and AsiaCCS.

Juan Tapiador Juan Tapiador is Associate Professor in the Department of Computer Science at Universidad Carlos III de Madrid, Spain, where he leads the Computer Security Lab. Prior to joining UC3M, I worked at the University of York, UK. His research focuses on various security issues of systems, software and networks, including malware analysis, attack modeling, anomaly and intrusion detection. He is Associate Editor for Computers & Security (COSE) and has served in the technical committee of several venues in computer security,



Most Influential Paper 2009-2013 award. He is an Associate Editor for ACM Transactions on Privacy and Security (TOPS). He has been program chair or co-chair for ACSAC, DIMVA, DFRWS, and ESSOS. He has been in the technical committee of the top venues in computer security including IEEE S&P, ACM CCS, USENIX Security, NDSS, WWW, RAID, AsiaCCS, and DIMVA

Juan Caballero Juan Caballero is Deputy Director and Associate Research Professor at the IMDEA Software Institute in Madrid, Spain. His research addresses security issues in systems, software, and networks. One of his focus is the analysis of malware and cyberattacks. He received his Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University, USA. His research regularly appears at top security venues and has won two best paper awards at the USENIX Security Symposium, one distinguished paper award at IMC, and the DIMVA