A Look into 30 Years of Malware Development from a Software Metrics Perspective

Alejandro Calleja[1], Juan Tapiador[1], and Juan Caballero[2]

[1] Department of Computer Science, Universidad Carlos III de Madrid, Spain
{accortin,jestevez}@inf.uc3m.es
[2] IMDEA Software Institute, Madrid, Spain
juan.caballero@imdea.org

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 since the early 1980s to date from a software engineering perspective. We analyze the source code of 151 malware samples 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 regular software. Overall, this supports otherwise confirmed claims about the increasing complexity of malware and its production progressively becoming an industry.

Keywords: malware; source code analysis; software metrics

1 Introduction

The malware industry seems to be in better shape than ever. In their 2015 Internet Security Threat Report [5], 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 [32] 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 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* [13,20] in the flourishing underground economy of cyber crime [35,37]. 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 [36].

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 [11] and the work of Kotov and Massacci on 30 exploit kits [26].) One consequence of the lack of wide availability of malware source code is a poor understanding of the malware development process, its properties when looked at as a software artifact, and how these properties have changed in the last decades.

In this paper, we present a study of the evolution of malware from a software engineering perspective. Our analysis is based on a dataset composed of the source code of 151 malware samples ranging from 1975 to 2015, including early viruses, worms, trojans, botnets, and remote access trojans (RATs). We make use of several metrics used in software engineering to quantify different aspects of the source code of malware understood as a software artifact. 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 cost of developing the sample: 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 also use these metrics to compare malware source code to a selection of benign programs. To the best of our knowledge, our work is the first to explore malware evolution from this perspective. We also believe that our dataset of malware source code is the largest analyzed in the literature. 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.

The remaining of this paper is organized as follows. Section 2 provides an overview of the software metrics used in this work. In Section 3 we describe our dataset of malware source code. Section 4 contains the core results of this work and Section 5 discusses the suitability of our approach, its limitations, and additional conclusions. Finally, Section 7 concludes the paper.

2 Software Metrics

This section provides an overview of the software metrics concepts used in this work to quantify various aspects of malware source code. We first introduce the two most widely used measures of software size: lines of source code (SLOC) and function points (FP). We then introduce effort estimation metrics, specifically the Constructive Cost Model (COCOMO), and also measures of source code complexity and maintainability.

2.1 Measuring Software Size

The number of lines in the source code of a program (SLOC) constitutes the most commonly used measure of its size. The number of physical SLOC refers to a count of the number of lines in the source code of a program, excluding comment and blank lines. Contrarily, logical SLOC counts take into account language-specific aspects, such as terminating symbols and style or formatting conventions, to deliver an estimate of the number of executable statements. Both IEEE [23] and the Software Engineering Institute (SEI) [31] had provided definitions and counting guidelines to obtain SLOC measures.

SLOC counts have a number of shortcomings [29] and can be easily misused. Despite this, it has a long-standing tradition as the most popular sizing metric. Furthermore, SLOC is an essential input for many estimation models that aim at predicting the effort required to develop a system, its maintainability, the expected number of bugs/defects, or the productivity of programmers.

Comparing size across different programming languages can give misleading impressions of the actual programming effort: the more expressive the programming language, the lower the size. An alternative metric to using SLOCs as the estimated software size is to use a measure of its functionality. The best known of such measures is the *function-point count*, initially proposed by Albrecht [7] and later refined by Albrecht and Gaffney [8]. The function-point count refers to the overall functionality of the software and is measured by estimating four program features: external inputs and outputs, user interactions, external interfaces, and files used. The overall count also involves various weights that account for the possibly different complexity of each of the above elements. Thus, the so-called unadjusted function-point count (UFC) is computed by simply multiplying each count by the appropriate weight and summing up all values. The UFC can be subsequently adjusted through various factors that are related to the complexity of the whole system.

Programming language	SLOC/FP	Programming language	SLOC/FP
ASP / ASP.Net	69	Java	53
Assembly	119	Javascript	47
Shell / DOS Batch	128	PHP	67
C	97	Pascal	90
C#	54	Python	24
C++	50	SQL / make	21
HTML / CSS / XML / XSLT	34	Visual Basic	42

Table 1: SLOC to function-point ratios for various programming languages.

The expected size in SLOC of a software project can be estimated from function-point counts through a process known as *backfiring* [25]. This consists in the use of existing empirical tables that provide the average number of SLOC per function point in different programming languages. Software Productivity Research (SPR) [24] annually publishes such conversion ratios for the most common programming languages in what is known as Programming Languages Tables (PLT), which are empirically obtained by analyzing thousands of software projects. Table 1 shows the SLOC-to-function-point ratios provided by PLT v8.2 for the languages most commonly observed in malware. Overall, backfiring is useful as SLOC counts are not available early enough in the requirements phase for estimating purposes. Also, the resulting UFC measure is a more normalized measure of the source code size.

2.2 Effort Estimation: The Constructive Cost Model (COCOMO)

One of the core problems in software engineering is to make an accurate estimate of the effort required to develop a software system. This is a complex issue that has attracted much attention since the early 1970s, resulting in various techniques that approach the problem from different perspectives [34]. A prominent class of such techniques are the so-called algorithmic cost modeling methods, which are based on mathematical formulae that provide cost figures using as input various measures of the program's size, organizational practices, and so on.

One of the best known algorithmic software cost estimation methods is the Constructive Cost Model (COCOMO) [12]. COCOMO is an empirical model derived from analyzing data collected from a large number of software projects. These data were used to find, through basic regression, formulae linking the size of the system, and project and team factors to the effort to develop it. As in most algorithmic cost models, the number of lines of source code (SLOC) in the delivered system is the basic metric used in cost estimation. Thus, the basic COCOMO equation for the effort (in man-months) required to develop a software system is

$$E = a_b(\mathrm{KLOC})^{b_b}, \tag{1}$$

where KLOC is the estimated number of SLOC expressed in thousands. The development time (in months) is obtained from the effort as

$$D = c_b E^{d_b}, \tag{2}$$

and the number of people required is just

$$P = \frac{E}{D}. \tag{3}$$

In the equations above, the coefficients a_b, b_b, c_b, and d_b are empirical estimates dependent on the type of project (see Table 2). COCOMO considers three 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).

Software Project	a_b	b_b	c_b	d_b
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

Table 2: Basic COCOMO coefficients.

The model described above is commonly known as *Basic COCOMO* and is very convenient to obtain a quick estimate of costs. A further refinement is provided by the so-called *Intermediate COCOMO*. The main difference consists in the addition of various multiplicative modifiers to the effort estimation (E) that account for attributes of both the product and the programming process such as the expected reliability, and the capability and experience of the programmers. Since these are not known for malware, we we will restrict ourselves to the Basic COCOMO model.

2.3 Source Code Complexity and Maintainability

Software complexity metrics attempt to capture properties related to the interactions between source code entities. Complexity is generally linked to maintainability, in the sense that higher levels of complexity might translate into a higher risk of introducing unintentional interactions and, therefore, software defects [27].

One of the earliest—and still most widely used—software complexity metric is McCabe's cyclomatic complexity [28], often denoted M. The cyclomatic complexity of a piece of source code is computed from its control flow graph (CFG) and measures the number of linearly independent paths within it; that is, the number of paths that do not contain other paths within themselves. Thus, a piece of code with no control flow statements has $M = 1$. A piece of code with one single-condition IF statement would have $M = 2$, since there would be two paths through the code depending on whether the IF condition evaluates to true or false. Mathematically, the cyclomatic complexity of a program is given by

$$M = E - N + 2P, \tag{4}$$

where E is the number of edges in the CFG, N the number of nodes, and P the number of connected components. The term "cyclomatic" stems from the connections between this metric and some results in graph theory and algebraic topology, particularly the so-called *cyclomatic number* of a graph, which measures the dimension of the cycle space of a graph [16].

The cyclomatic complexity has various applications in the process of developing and analyzing software products. The most direct one is to limit the complexity of the routines or modules that comprise the system. McCabe recommended that programmers should limit each module to a maximum complexity of 10, splitting it into smaller modules whenever its complexity exceeds this value. The NIST Structured Testing Methodology [38] later adopted this practice and relaxed the figure up to 15, though only occasionally and if there are well grounded reasons to do it. The cyclomatic complexity has also implications in program testing because of its connection with the number of test cases that are necessary to achieve thorough test coverage. Specifically, M is simultaneously: (i) an upper bound for the number of test cases needed to achieve a complete branch coverage (i.e., to execute all edges of the CFG); and (ii) a lower bound for the number of paths through the CFG. Thus, a piece of code with high M would have more pathways through the code and would therefore require higher testing effort.

The cyclomatic complexity is also connected to another code metric called the maintainability index (MI), introduced by Oman and Hagemeister in [30]. The MI is a value between 0 and 100 that measures how maintainable (i.e., easy to understand, support, and change) the source code is, with high values meaning better maintainability. One of the most common definitions of the MI is given by

$$MI = 100\frac{171 - 5.2\ln\left(\overline{V}\right) - 0.23\overline{M} - 16.2\ln\left(\overline{SLOC}\right)}{171}, \tag{5}$$

where $\overline{V}$ is Halsteads average volume per module (another classic complexity metric; see [21] for details), $\overline{M}$ is the average cyclomatic complexity per module, and $\overline{SLOC}$ is the average number of source code lines per module. This is, for instance, the definition used by Visual Studio, and does not take into account the comment-to-code ratio as the original one proposed in [30]. As in the case of the COCOMO estimators, Oman and Hagemeister arrived at this formula through statistical regression over a dataset consisting of a large number of software projects tagged with expert opinions. The MI has been included in Visual Studio since 2007, and in the JSComplexity and Radon metrics for Javascript and Python. Although not exempt from criticisms, its use was promoted by the Software Engineering Institute in their "C4 Software Technology Reference Guide" [33] as a potentially good predictor of maintainability. As for its interpretation, there is no agreed upon safe limits. For example, Visual Studio flags as suspicious modules with $MI < 20$.

3 Dataset

Our work is based on a dataset of malware source code collected by the authors over several months in 2015. Collecting malware source code is a challenging endeavor be-

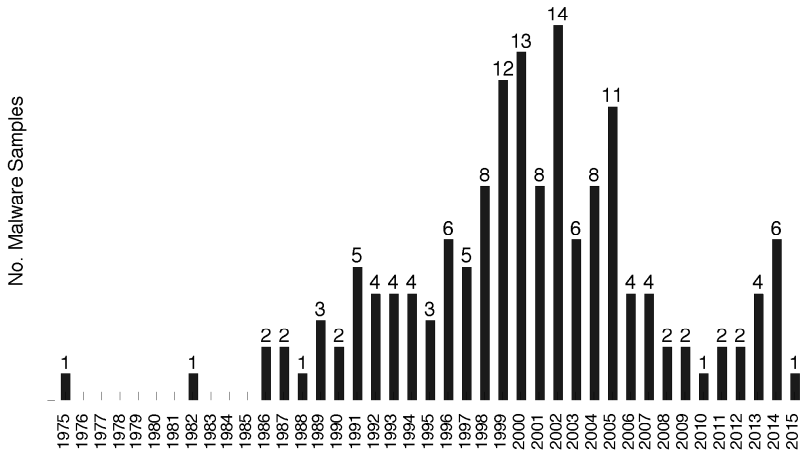


Fig. 1: Distribution of malware source code samples in the dataset.

cause 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 30+ years, constrained to the limited availability.

Samples were obtained from a variety of sources, including virus collection sites such as *VX Heaven*, code repositories such as *GitHub*, classical e-zines published by historically prominent malware writing groups such as *29A*, various malware exchange forums available in the web, 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.

Our initial collection contained 210 different samples of malware source code. Each sample was first quickly verified through manual inspection and then compiled, executed and, whenever possible, functionally tested. Approximately 30% of the obtained samples were discarded at this point, 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 151 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: Virus (V), Worm (W), MacroVirus (M), Trojan (T), Botnet (B), or RAT (R). We are aware that this classification is rather imprecise. For instance, nearly all Botnets and RATs include bots that can be easily considered as Trojans, Backdoors or Spywares and, in some cases, show Worm fea-

tures too. The same applies to some of the more modern viruses, which also exhibit Worm-like propagation strategies or behave like stand-alone Trojans. 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 151 samples. Approximately 62% of the samples (94) correspond to the period 1995-2005, with the remaining equally distributed in the 2006-2015 (27) and 1985-1994 (28) periods, plus two samples from 1975 and 1982, respectively. The largest category is Viruses (92 samples), followed by Worms (33 samples), Trojans (11 samples), RATs (9 samples), MacroViruses (3 samples), and Botnets (3 samples). A full listing of the 151 samples is provided in Table 3.

4 Analysis

This section describes our analysis over the malware source code dataset. It first details source code analytics (Section 4.1), then it estimates development cost (Section 4.2), next it discusses complexity and maintainability metrics (Section 4.3), and finally compares malware to benign code (Section 4.4).

4.1 Source Code Analytics

We next discuss various statistics obtained from the source code of the malware samples in our dataset.

Number of source code files. Figure 2a shows the distribution over time of the number of files comprising the source code of the different malware samples. 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 (LOC). 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), Rovnix (2014), and SpyNet (2014), with 267, 276, and 324 files, respectively. This increase reveals a more modular design, which also correlates with the use of higher-level programming languages discussed later.

Simple least squares linear regression over the data points shown in Figure 2a yields a regression coefficient (slope) of 1.17. (Note that the y-axis is in logarithmic scale and, therefore, such linear regression actually corresponds to an exponential fit.) This

Table 3: Malware source code samples in the dataset.

Year	Name	Type
1975	ANIMAL	T
1982	ElkCloner	V
1986	Rushrour	V
1986	V11	V
1987	Bvs	V
1987	Numberone	V
1988	MorrisWorm	W
1989	AIDS	V
1989	CIA	V
1989	Eddie	V
1990	Anthrax	V
1990	Diamond	V
1991	486	V
1991	808	V
1991	Badbrains	V
1991	Demonhyak	V
1991	Tormentor	V
1992	ACME	V
1992	Proto-t	V
1992	Rat	V
1992	Thunderbyte	V
1993	Asexual	V
1993	Beavis	V
1993	DarkApocalypse	V
1993	Nakedtruth	V
1994	Batvir	V
1994	Bluenine	V
1994	Dichotomy	V
1994	Digitisedparasite	V
1995	242	V
1995	Bluelightening	V
1995	RCE285	V
1996	Apocalyptic	V
1996	Combat	V
1996	Galicia	V
1996	Jupiter	V
1996	Mars	V
1996	Staog	V
1997	CSV	V
1997	Cabanas	V
1997	Harrier	V
1997	RedTeam	V
1997	V6000	V
1998	Anaphylaxis	W
1998	Ch0lera	W
1998	Gift	W
1998	Marburg	V
1998	PGPMorf2	V
1998	Plague2000	W
1998	Shiver	M
1998	Teocatl	M
1999	Babylon	V
1999	BeGemot	V
1999	Demiurg	V
1999	Fabi2	V
1999	IISW	W
1999	Melissa	M
1999	Nemesi	V
1999	Prizzy	V
1999	Rinim	V
1999	RousSarcoma	V
1999	YLANG	V
1999	Yobe	V
2000	Chainsaw	W
2000	Dream	V
2000	Energy	W
2000	Examplo	V
2000	ILOVEYOU	W
2000	Icecubes	W
2000	Milennium	V
2000	Rammstein	V
2000	Troodon	W
2000	Tuareg	V
2000	W2KInstaller	V
2000	XTC	W
2000	Zelda	W
2001	Anarxy	W
2001	Ketamine	V
2001	MW	W
2001	Nicole	V
2001	OU812	V
2001	Plexar	V
2001	Rudra	V
2001	Tapakan	V
2002	DW	V
2002	Efishnc	V
2002	Gemini	V
2002	Grifin	W
2002	Junkmail	V
2002	Lexotan	V
2002	Omoikane	V
2002	PieceByPiece	W
2002	Ramlide	V
2002	Simile	V
2002	Solaris	V
2002	Taichi	V
2002	Vampiro	V
2002	ZMist	V
2003	Blaster	W
2003	Mimail	W
2003	Obsidian	W
2003	Rainbow	V
2003	Seraph	V
2003	Tahorg	V
2004	Beagle	B
2004	Caribe	W
2004	Jollyroger	V
2004	Mydoom	W
2004	Netsky	W
2004	Pilif	W
2004	Sasser	W
2004	Shrug	V
2005	Assiral	W
2005	Blackhand	V
2005	Egypt	V
2005	Eternity	V
2005	Friendly	V
2005	Gripb	V
2005	Hidan	V
2005	Kroshkaenot	V
2005	Nanomites	V
2005	Spieluhr	T
2005	WRhRage	W
2006	Gurdof	W
2006	Kekule	W
2006	Macbet	W
2006	Ston	W
2007	Antares	V
2007	BO2K	R
2007	GhostRAT	R
2007	Zeus	T
2008	BatzBack	W
2008	Grum	B
2009	Cairuh	W
2009	Hexbot2	T
2010	Carberp	T
2011	KINS	T
2011	PC-RAT	R
2012	AndroR	R
2012	Dexter	T
2013	Alina	T
2013	Beetle	V
2013	Pony2	T
2013	SharpBot	R
2014	Dendroid	R
2014	Gopnik	B
2014	OmegaRAT	R
2014	Rovnix	T
2014	SpyNet	R
2014	Tinba	T
2015	Pupy	R

means that the number of files has grown at an approximate yearly ratio of 17%; or, equivalently, that it has doubled every 4.5 years.

Source code size. Figure 2d shows the distribution over time of the number of physical source lines of code (SLOC) of all samples in the dataset. For this we used `cloc` [1], an open-source tool that counts blank lines, comment lines, and SLOC, and reports them broken down by programming language. The data shown in Figure 2d was obtained

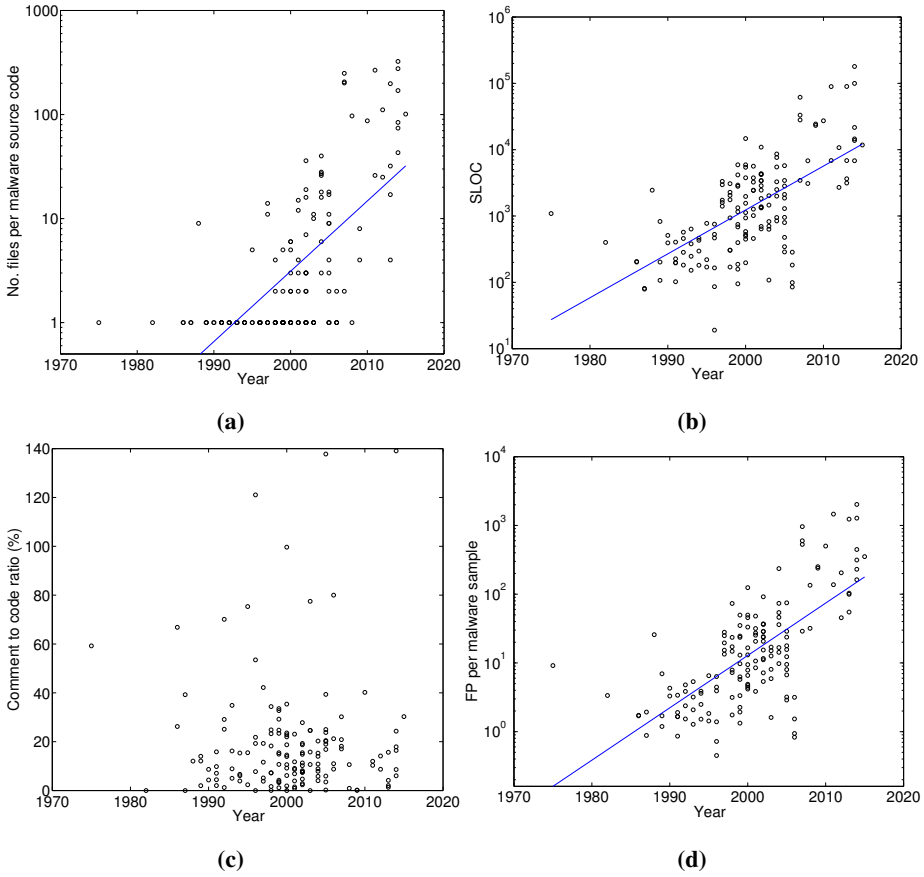


Fig. 2: Source code analytics of the malware samples in our dataset. **(a)** Number of files. **(b)** SLOC. **(c)** Comment-to-code ratios. **(d)** FP counts. Note that in **(a)**, **(b)**, and **(d)** the y-axis is shown in logarithmic scale.

by simply aggregating the SLOC counts of all source code files belonging to the same malware sample, irrespective of the programming language in which they were written.

Again, the growth over the last 30 years is clearly exponential. Thus, up to the mid 1990s viruses and early worms rarely exceeded 1,000 SLOC. Between 1997 and 2005 most samples contain several thousands SLOCs, with a few exceptions above that figure, e.g., Troodon (14,729 SLOC) or Simile (10,917 SLOC). The increase in SLOC count 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 SLOC counts 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 projects whose output is more than just one binary. 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 SLOC counts (2,701 and 3,143, respectively).

In this case the linear regression coefficient over the data points is 1.16, i.e., the number of SLOCs per malware has increased approximately 16% per year; or, equivalently, the figure doubles every 4.7 years, resulting in an increase of nearly an order of magnitude each decade.

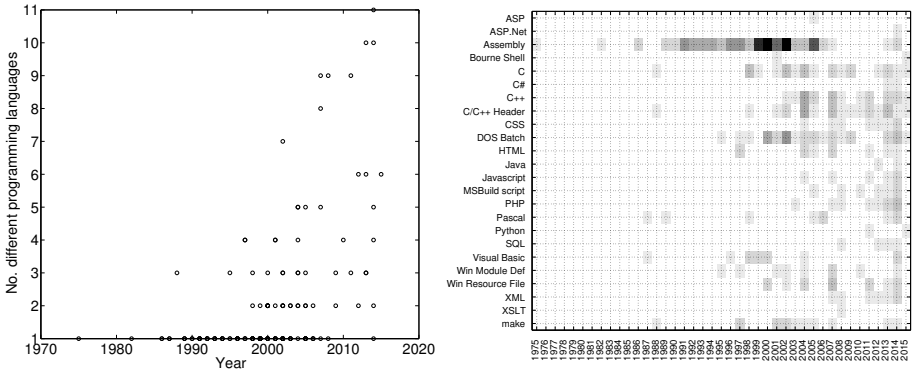
Function points estimates. We used the SLOC-to-function-point ratios provided by PLT v8.2 (see Table 1) in an attempt to use a more normalized measure of source code size for the malware samples in our dataset. To do that, we used such ratios in reverse order, i.e., to estimate function-point counts from SLOCs rather than the other way round. In doing so we pursue: (i) to better aggregate the various source code files of the same malware that are written in different languages; and (ii) to provide a fairer comparison among the sizes of the samples.

As expected, there is a clear correlation between FP and SLOC counts 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) FPs. From 2000 to 2010 the FP count concentrates between 10 and 100, with Trojans, Botnets, and RATs accounting for the higher counts. Since 2007 on, many samples exhibit FP counts of 1,000 and higher; examples include Rovnix (2014), with FP=1275.64, KINS (2011), with FP=1462.86, and SpyNet (2014), with FP=2021.79. Linear regression over the data points yields a coefficient of 1.19, i.e., FP counts per malware has suffered an approximate growth of 19% per year; or, equivalently, the figure doubles every 4 years.

Density of comments. Fig. 2c shows the comments-to-code ratios for the malware samples in the dataset. This is simply computed as the number of comment lines divided by the SLOC. There is no clear pattern in the data, which exhibit an average of 18.83%, a standard deviation of 23.44%, and a median value of 12.05%. 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 ratio of comments. We ignore if they were originally developed in this way or, perhaps, comments were cut off before leaking/releasing the code.

Programming languages. Figure 3a 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 package (e.g., HTML, XML, CSS) and scripts used to build it (e.g., BAT or make files).

Figure 3b shows the usage of different programming languages to code malware over time extracted from our dataset. The pattern reveals the prevalent use of assembly until the mid 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



(a) **(b)**

Fig. 3: (a) Number of different programming languages per malware sample in the dataset. **(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.

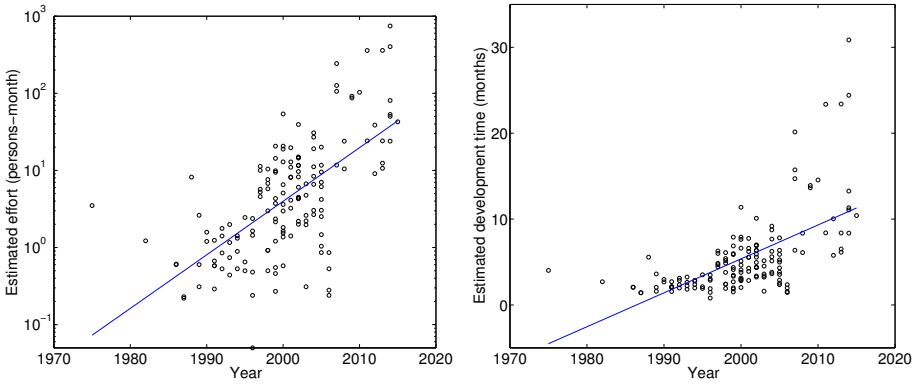
developed in PHP or Python. From 2012 to date the distribution of languages is approximately uniform, revealing the heterogeneity of technologies used to develop modern malware.

4.2 Cost Estimation

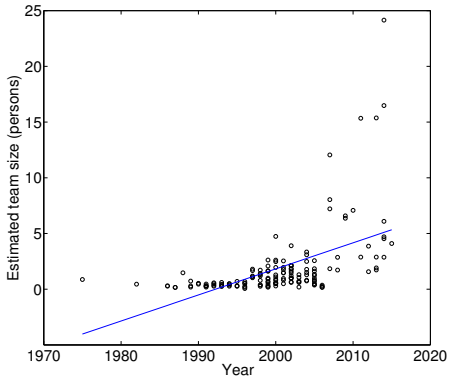
In this section we show the COCOMO estimates for the effort, time, and team size required to develop the malware samples in our dataset. One critical decision here is selecting the type of software project (organic, semi-detached, or embedded) for each sample. We decided to consider all samples as organic for two main 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 efforts which is achieved using the lowest COCOMO coefficients (i.e., those of organic projects) and can thus be seen as a (estimated) lower bound of development efforts.

Fig. 4a shows the COCOMO estimation of effort required to develop the malware samples. 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 1 man-month, this value rapidly escalates up to 10-20 in the mid 2000s, and to 100s for a few samples of the last few years. Linear regression confirms this, yielding a regression coefficient of 1.17; i.e., the effort growth ratio per year is approximately 17%; or, equivalently, it doubles every 4.5 years.

The estimated time to develop the malware samples (Fig. 4b) experiences a linear increase up to 2010, rising from 2-3 months in the 1900s to 7-10 months in the late 2000s. The linear regression coefficient in this case is 0.395, which translates into an additional month every 2.5 years. Note that a few samples from the last 10 years report



(a) **(b)**



(c)

Sample	Year	E	D	P
Anthrax	1990	1.20	2.68	0.45
Batvir	1994	0.53	1.97	0.27
AIDS	1999	0.31	1.59	0.19
IISWorm	1999	0.55	1.99	0.28
ILOVEYOU	2000	0.58	2.03	0.29
Blaster	2003	1.97	3.24	0.61
Mydoom	2004	11.13	6.25	1.78
Sasser	2004	3.03	3.81	0.80
Zeus	2007	242.85	20.15	12.05
GhostRAT	2007	126.45	15.73	8.04
Tinba	2014	53.13	11.31	4.70
Dendroid	2014	50.20	11.07	4.53

(d)

Fig. 4: 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.

a considerable higher number of months, such as Zeus (2007) or SpyNet (2014) with 20.15 and 30.86 months, respectively.

The amount of people required to develop each sample (Fig. 4c) grows similarly. Most early viruses and worms require less than 1 person (full time). From 2000 on, the figure increases to 3-4 persons for some samples. Since 2010, a few samples report person estimates substantially higher. For these data, the linear regression coefficient is 0.234, which roughly translates into an additional team member every 4 years.

Finally, the table in Fig. 4d provides some numerical examples for a selected subset of samples. For additional details, we refer the reader to the full datasets[3] with the raw data used in this paper.

[3] Available at: http://www.seg.inf.uc3m.es/~accortin/RAID_2016.html

4.3 Complexity and Maintainability

In this section we show the complexity and maintainability metrics obtained for the samples in our dataset. To compute McCabe's cyclomatic complexity, we used the Universal Code Count (`UCC`) [6], a tool that provides various software metrics from source code. While many other tools exist for measuring cyclomatic complexity (e.g., `Jhawk` [3], `Radon` [4], or the metrics plugin for Eclipse [2]), these have a strong bias towards a particular language or a small subset of them. Contrarily, `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. Despite `UCC`'s wide support for many languages, obtaining the cyclomatic measurements for each sample was not possible. As suggested by Fig. 3b, a large fraction of our samples contain a substantial amount of assembly code, which `UCC` does not support. Filtering out samples that contain at least one source file in assembly left 44 samples for analysis, i.e., approximately 33% of the dataset.

Fig. 5a shows the average cyclomatic complexity per function for each analyzed sample. Most of the samples have complexities between 2 and 5, with values higher than that being very uncommon. Only DW (2002) exhibits an average cyclomatic complexity of around 7. Two main conclusions can be drawn from Fig. 5a (note that samples are temporarily ordered). First, even if there is no clear evolution over time of the average complexity per function, there is a slight decreasing trend. This might be a consequence of a more modular design, with functions and class methods being designed with less complex control flow logic structures. Second, a closer inspection at the full output of `UCC` reveals that no function or method in the 44 samples exceeds McCabe's recommended complexity threshold of 10.

Using the metrics discussed throughout this section, we have also computed an upper bound for the maintainability index MI provided by equation (5). Note that we cannot estimate it 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 maintainability index would be lower than our estimates. Nevertheless, note that such a factor contributes the lowest to MI, so we expect our figures to provide a fair comparison among samples. Fig. 5b shows the MI values for each sample in the reduced dataset. As in the case of cyclomatic complexities, no clear trend is observed. Values are generally high, with most samples having an MI higher than 50. The most extreme values are those of Cairuh ($MI = 30.05$) and Hexbot2 ($MI = 34.99$) on one side of the spectrum, and Taichi ($MI = 78.77$), AndroRAT ($MI = 75.03$), and Dendroid (74.47) on the other. All in all, these are reasonably high values for MI.

4.4 Comparison with Regular Software

In order to better appreciate the significance of the figures presented throughout this section, we next discuss how they compare to those of a selected number of open source projects whose source code is freely available. To do this we selected 9 software packages belonging to different categories: 3 security products (the `IPTables` firewall, the

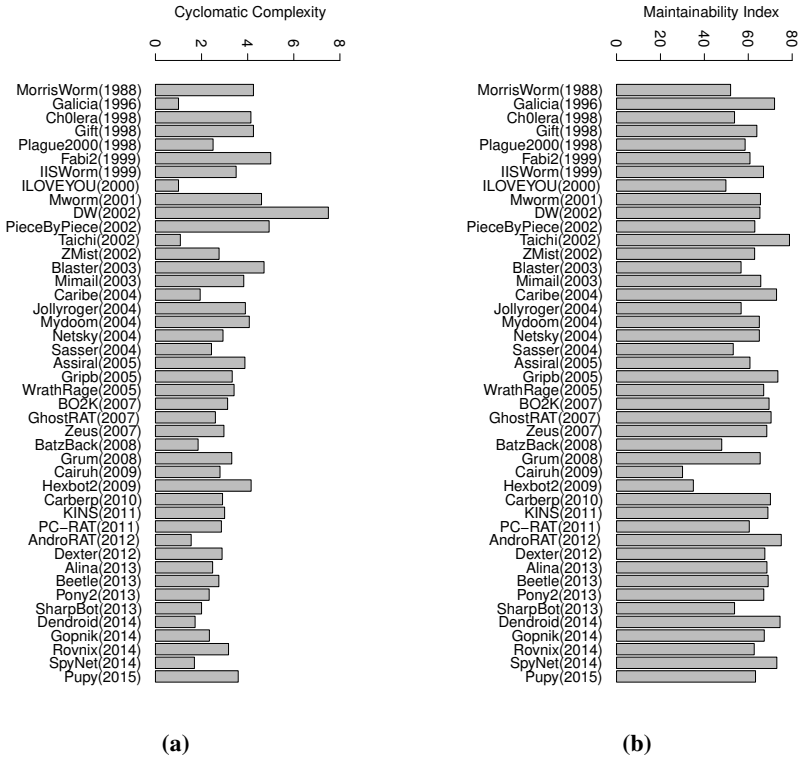


Fig. 5: (a) Average cyclomatic complexity per function and sample sorted by year. **(b)** Maintainability index per sample sorted by year.

`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 code was downloaded from the web page of each project. For each one of them we then computed the metrics discussed above for malware samples. As in the case of malware samples, we use the COCOMO coefficients for organic projects. The results are shown in Table 4 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 FP greater than 1000, or SpyNet, with an FP count comparable to that of `Bash`. In general, only complex malware projects are comparable in size and effort to these two software packages, and they are still far away from the remaining ones.

In terms of comment-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

Software	Version	Year	SLOC	E	D	P	FP	M	CR	MI
`Snort`	2.9.8.2	2016	46,526	135.30	16.14	8.38	494.24	3.31	10.32	63.27
`Bash`	4.4 rc-1	2016	160,890	497.81	26.47	18.81	2,265.35	3.40	17.08	52.42
`Apache`	2.4.19	2016	280,051	890.86	33.03	26.97	4,520.10	3.02	23.42	61.56
`IPtables`	1.6.0	2015	319,173	1,021.97	34.80	29.37	3,322.05	3.06	27.33	68.88
`Git`	2.8	2016	378,246	1,221.45	37.24	32.80	4,996.44	3.37	12.15	41.84
`Octave`	4.0.1	2016	604,398	1,998.02	44.89	44.51	11,365.09	2.52	27.69	52.42
`ClamAV`	0.99.1	2016	714,085	2,380.39	47.98	49.61	10,669.97	2.79	33.57	63.87
`Cocos2d-x`	3.10	2016	851,350	2,863.02	51.47	55.63	16,566.78	2.96	17.55	66.60
`gcc`	5.3	2015	6,378,290	2,3721.97	114.95	206.37	90,278.41	2.10	31.24	50.57

Table 4: Software metrics for various open source projects. **E**: COCOMO effort; **D**: COCOMO development time; **P**: COCOMO team size; **FP**: function points; **M**: cyclomatic complexity; **CR**: comment-to-code ratio; **MI**: maintainability index.

further investigate this point, we computed the cyclomatic complexities at the function level; i.e., for all functions of all samples in both datasets. The histograms of the obtained values is shown in Fig. 6. Both distributions are very similar, with a clear positive skewness. A Chi-squared and two-sample Kolgomorov-Smirnov tests corroborate their similarity for a significance level of $\alpha = 0.05$.

More differences appear in terms of maintainability. Up to 12 malware samples show MI values higher than the highest one for regular software—`IPtables`, with $MI = 68.88$. In general, malware samples (particularly the most recent ones) seem to have slightly higher maintainability indexes than regular software. As discussed before, two notable exception are Cairuh and Hexbot2 with surprisingly low values.

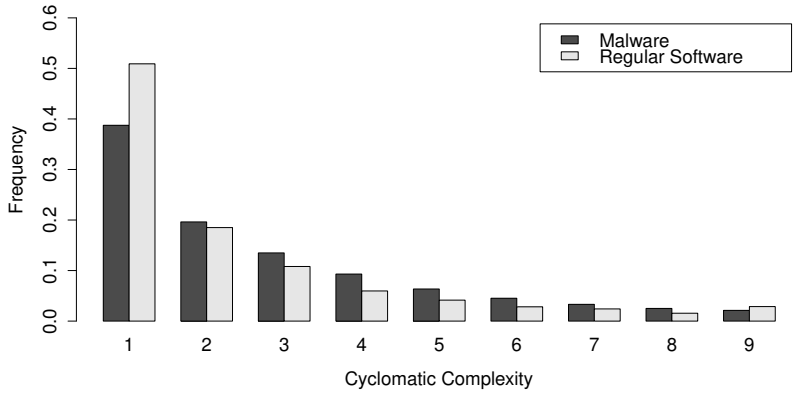


Fig. 6: Histograms of the cyclomatic complexity values computed at the function level for both malware and regular software samples.

5 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) [34]. 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.

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 3, obtaining source code of malware is hard. Still, we discuss 151 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 (one remarkable exception is ransomware, for which we were not able to find any samples). 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 and thus escape our collection. We believe those samples would emphasize the increasing complexity trends that we observe.

Main conclusions and open questions. In the last 30 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. One question is whether this trend will hold in time. If so, we could soon see malware specimens with more than 1 million SLOC. On the other hand, evolving into large pieces of software involves a higher amount of vulnerabilities and defects. This has been already observed (and exploited), e.g., in [17] and [14]. 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 [15, 19]. More generally, the results shown in this paper provide a quantified evidence of how the malware development industry has been progressively transforming into a fully fledged engineering.

6 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 [10], which upon compromising a host would upload its source code, compile it using gcc, and run the compiled executable. In 2005, Holz [22] 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 [11] 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 exploting vulnerabilities in Web browsers and their plugins. Different from client malware, exploit kits are distributed as (possibly obfuscated) source code. Kotov and Massacci [26] 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. [9] followed up on this research by building a malware lab to experiment with the exploit kits. Eshete and Venkatakrishnan describe WebWinnow [18] 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 [17] 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. [14] directly on the binary code of malware samples installed in client machines.

7 Conclusion

In this paper, we have presented a study on the evolution of malware source code over the last decades. 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

We are very grateful to the anonymous reviewers for constructive feedback and insightful suggestions. This work was supported by the MINECO grant TIN2013- 46469-R (SPINY: Security and Privacy in the Internet of You), the CAM grant S2013/ICE-3095 (CIBERDINE: Cybersecurity, Data, and Risks), the Regional Government of Madrid through the N-GREENS Software-CM project S2013/ICE-2731 and by the Spanish Government through the Dedetis Grant TIN2015-7013-R.

References

1. cloc - count lines of code. http://github.com/AlDanial/cloc, accessed: 2015-09-22
2. Eclipse metrics plugin. https://marketplace.eclipse.org/content/eclipse-metrics, accessed: 2016-04-4
3. Jhawk. http://www.virtualmachinery.com/jhawkprod.htm, accessed: 2016-04-4
4. Radon. https://pypi.python.org/pypi/radon, accessed: 2016-04-4
5. Symantec's 2015 internet security threat report. https://www.symantec.com/security_response/publications/threatreport.jsp, accessed: 2016-04-6
6. Unified code counter. http://csse.usc.edu/ucc_wp/, accessed: 2016-04-4
7. Albrecht, A.J.: Measuring Application Development Productivity. In: Press, I.B.M. (ed.) IBM Application Development Symp. pp. 83–92 (Oct 1979)
8. Albrecht, A.J., Gaffney, J.E.: Software function, source lines of code, and development effort prediction: A software science validation. IEEE Trans. Softw. Eng. 9(6), 639–648 (Nov 1983)
9. Allodi, L., Kotov, V., Massacci, F.: MalwareLab: Experimentation with Cybercrime Attack Tools. In: USENIX Workshop on Cyber Security Experimentation and Test. Washington DC (August 2013)
10. Arce, I., Levy, E.: An analysis of the slapper worm. IEEE Security & Privacy 1(1), 82–87 (2003)
11. Barford, P., Yegneswaran, V.: Malware Detection, chap. An Inside Look at Botnets, pp. 171–191. Springer (2007)
12. Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)
13. Caballero, J., Grier, C., Kreibich, C., Paxson, V.: Measuring pay-per-install: The commoditization of malware distribution. In: Proceedings of the 20th USENIX Conference on Security. pp. 13–13. SEC'11, USENIX Association, Berkeley, CA, USA (2011)
14. Caballero, J., Poosankam, P., McCamant, S., Babic, D., Song, D.: Input Generation Via Decomposition and Re-Stitching: Finding Bugs in Malware. In: ACM Conference on Computer and Communications Security. Chicago, IL (October 2010)
15. Caliskan-Islam, A., Harang, R., Liu, A., Narayanan, A., Voss, C., Yamaguchi, F., Greenstadt, R.: De-anonymizing Programmers via Code Stylometry. In: USENIX Security Symposium (2015)
16. Diestel, R.: Graph Theory, 4th Edition, Graduate texts in mathematics, vol. 173. Springer (2012)
17. Eshete, B., Alhuzali, A., , Monshizadeh, M., Porras, P., Venkatakrishnan, V., Yegneswaran, V.: EKHunter: A Counter-Offensive Toolkit for Exploit Kit Infiltration. In: Network and Distributed System Security Symposium (February 2015)
18. Eshete, B., Venkatakrishnan, V.N.: WebWinnow: Leveraging Exploit Kit Workflows to Detect Malicious Urls. In: ACM Conference on Data and Application Security and Privacy (2014)
19. Frantzeskou, G., MacDonell, S., Stamatatos, E., Gritzalis, S.: Examining the significance of high-level programming features in source code author classification. J. Syst. Softw. 81(3), 447–460 (Mar 2008), http://dx.doi.org/10.1016/j.jss.2007.03.004
20. Grier, C., Ballard, L., Caballero, J., Chachra, N., Dietrich, C.J., Levchenko, K., Mavrommatis, P., McCoy, D., Nappa, A., Pitsillidis, A., Provos, N., Rafique, M.Z., Rajab, M.A., Rossow, C., Thomas, K., Paxson, V., Savage, S., Voelker, G.M.: Manufacturing compromise: The emergence of exploit-as-a-service. In: Proceedings of the 2012 ACM Conference

on Computer and Communications Security. pp. 821–832. CCS '12, ACM, New York, NY, USA (2012)
21. Halstead, M.H.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA (1977)
22. Holz, T.: A short visit to the bot zoo. IEEE Security & Privacy 3(3), 76–79 (2005)
23. IEEE: Ieee standard for software productivity metrics (IEEE std. 1045-1992). Tech. rep. (1992)
24. Jones, C.: Programming languages table, version 8.2 (1996)
25. Jones, C.: Backfiring: Converting lines-of-code to function points. Computer 28(11), 87–88 (Nov 1995)
26. Kotov, V., Massacci, F.: Anatomy of Exploit Kits: Preliminary Analysis of Exploit Kits As Software Artefacts. In: International Conference on Engineering Secure Software and Systems (2013)
27. Lehman, M.M.: Laws of software evolution revisited. In: Proceedings of the 5th European Workshop on Software Process Technology. pp. 108–124. EWSPT '96, Springer-Verlag, London, UK, UK (1996)
28. McCabe, T.J.: A complexity measure. In: Proceedings of the 2Nd International Conference on Software Engineering. pp. 407–. ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA (1976)
29. Nguyen, V., Deeds-rubin, S., Tan, T., Boehm, B.: A sloc counting standard. In: COCOMO II Forum 2007 (2007)
30. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Proc. Conf. on Software Maintenance. pp. 337–344 (1992)
31. Park, R.E.: Software size measurement: A framework for counting source statements. Tech. Rep. CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (September 1992)
32. Security, P.: 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
33. Software Engineering Institute: C4 Software Technology Reference Guide - A Prototype. Tech. Rep. CMU/SEI-97-HB-001, 1997 (Jan 1997)
34. Sommerville, I.: Software Engineering: (Update) (8th Edition) (International Computer Science). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
35. Stringhini, G., Hohlfeld, O., Kruegel, C., Vigna, G.: 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. pp. 353–364. ASIA CCS '14, ACM, New York, NY, USA (2014)
36. Suarez-Tangil, G., Tapiador, J.E., Peris-Lopez, P., Ribagorda, A.: Evolution, detection and analysis of malware for smart devices. IEEE Communications Surveys and Tutorials 16(2), 961–987 (2014)
37. Thomas, K., Huang, D., Wang, D., Bursztein, E., Grier, C., Holt, T.J., Kruegel, C., McCoy, D., Savage, S., Vigna, G.: Framing dependencies introduced by underground commoditization. In: Workshop on the Economics of Information Security (2015)
38. Watson, A.H., Mccabe, T.J., Wallace, D.R.: Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In: U.S. Department of Commerce/National Institute of Standards and Technology (1996)