Callgraph properties of executables

Daniel Bilar
Wellesley College
Department of Computer Science
Wellesley, MA 02481, USA
E-mail: dbilar@wellesley.edu

This paper examines the callgraphs of 120 malicious and 280 non-malicious executables. Pareto models were fitted to in-degree, out-degree and basic block count distribution, and a statistically significant difference shown for the derived power law exponent. A two-step optimization process is hypothesized to account for structural features of the executable.

Keywords: Executables, Callgraph, HOT process, Graph-structural Fingerprint, Malware

1. Motivation

All commercial antivirus (AV) products rely on signature matching; the bulk of which constitutes strict byte sequence pattern matching. For modern, evolving *polymorphic* and *metamorphic* malware, this approach is unsatifactory. Clementi recently checked fifteen state-of-the-art, updated AV scanner against ten highly polymorphic malware samples and found false negative rates from 0-90%, with an average of 48% [5]. This development was already predicted in 2001 [29]. Polymorphic malware contain decryption routines which decrypt encrypted constant parts of the malware body. The malware can mutate its decryptors in subsequent generations, thereby complicating signature-based detection approaches. The decrypted body, however, remains constant. Metamorphic malware generally do not use encryption, but are able to mutate their body in subsequent generation using various techniques, such as junk insertion, semantic NOPs, code transposition, equivalent instruction substitution and register reassignments [4][27]. The net result of these techniques is a shrinking usable "constant base" for strict signature-based detection approaches.

Since signature-based approaches are quite fast (but show little tolerance for metamorphic and polymorphic code) and heuristics such as emulation are more resilient (but quite slow and may hinge on environmental triggers), a detection approach that combines the best of both worlds would be desirable. This is the philosophy behind a structural fingerprint. Structural fingerprints are statistical in nature, and as such are positioned as 'fuzzier' metrics between static signatures and dynamic heuristics. The structural fingerprint investigated in this paper for differentiation purposes is based on some properties of the executable's callgraph. I also propose a generative mechanism for the callgraph topology.

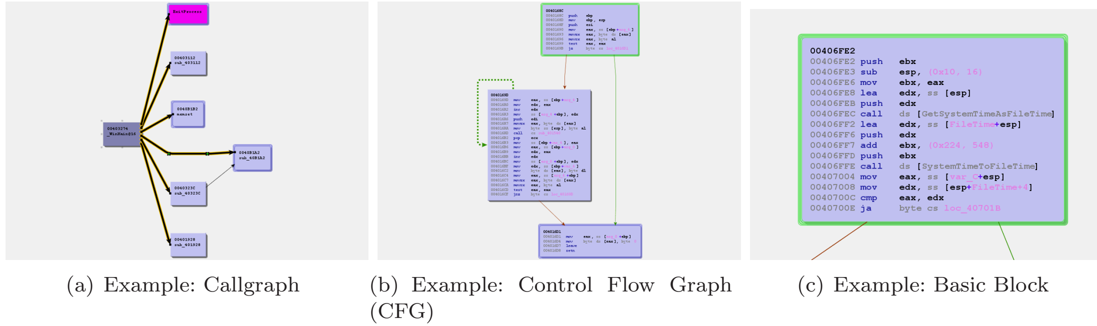
2. Generating the callgraph

Primary tools used are described in more details in the appendix.

2.1. Samples

For non-malicious software, henceforth called 'goodware', sampling followed a two-step process: I inventoried all PEs (the primary 32-bit Windows file format) on a Microsoft XP Home SP2 laptop, extracted uniform randomly 300 samples, discarded overly large and small files, yielding 280 samples. For malicious software (malware), seven classes of interest were fixed: backdoor, hacking tools, DoS, trojans, exploits, virus, and worms. The worm class was further divided into Peer-to-Peer (P2P), Internet Relay Chat/Instant Messenger (IRC/IM), Email and Network worm subclasses. For an non-specialist introduction to malicious software, see [26]; for a canonical reference, see [28]. Each class (subclass) contained at least 15 samples. Since AV vendors were hesitant for liability reasons to provide samples, I gathered them from herm1t's (underground) collection and identified compiler and (potential) packer metadata using PEiD. Practically all malware samples were identified as having been compiled by MS C++ 5.0/6.0, MS Visual Basic 5.0/6.0 or LCC,

AI Communications
ISSN 0921-7126, IOS Press. All rights reserved



(a) Example: Callgraph (b) Example: Control Flow Graph (CFG) (c) Example: Basic Block

Fig. 1. Graph structures of an executable

and about a dozen samples were packed with various versions of UPX (an executable compression program). Malware was run through best-of-breed, updated open- and closed-source AV products yielding a false negative rate of 32% (open-source) and 2% (closed-source), respectively. Overall file sizes for both mal- and goodware ranged from Θ(10kb) to Θ(1MB). A preliminary file size distribution investigation yielded a log-normal distribution; for a putative explanation of the underlying generative process, see [19] and [16]. All 400 samples were loaded into the de-facto industry standard disassembler (IDA Pro [12]), inter- and intra-procedurally parsed and augmented with symbolic meta-information gleaned programmatically from the binary via FLIRT signatures (when applicable). I exported the identified structures exported via IDAPython into a MySQL database. These structures were subsequently parsed by a disassembly visualization tool (BinNavi [6]) to generate and investigate the callgraph.

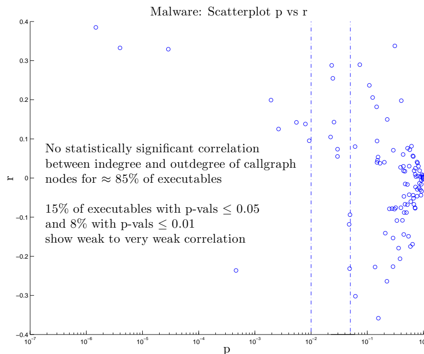
2.2. Callgraph

Following [7], we treat an executable as a *graph of graphs*. This follows the intuition that in any procedural language, the source code is structured into *functions* (which can be viewed as a flowchart, e.g. a directed graph which we call *flowgraph*). These functions call each other, thus creating a larger graph where each node is a function and the edges are calls-to relations between the functions. We call this larger graph the *callgraph*. We recover this structure by diassembling the executable into individual instructions. We distinguish between *short* and *far* branch instructions: Short branches do not save a return address while far branches do. Intuitively, short branches are normally used to pass control around within one function of the program, while far branches are used to call other functions. A sequence of instructions that is continuous (e.g. has no branches jumping into its middle and ends at a branch instruction) is called a *basic block*. We consider the graph formed by having each basic block as a node, and each short branch an edge. The connected components in this directed graph correspond to the flowgraphs of the functions in the source code. For each connected component in the previous graph, we create a node in the callgraph. For each far branch in the connected component, we add an edge to the node corresponding to the connected component this branch is targeting. Fig. 1 illustrate these concepts.

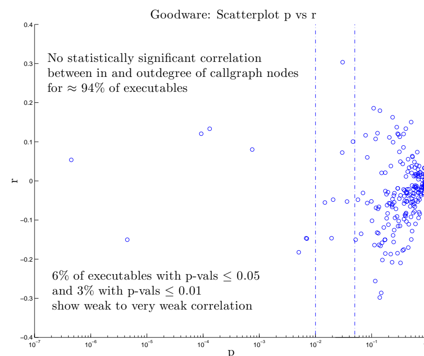
Formally, denote a callgraph CG as $CG = G(V, E)$, where $G(\cdot)$ stands for 'Graph'. Let $V = \bigcup F$, where $F \in normal, import, library, thunk$. This just says that each function in CG is either a 'library' function (from an external libraries statically linked in), an 'import' function (dynamically imported from a dynamic library), a 'thunk' function (mostly one-line wrapper functions used for calling convention or type conversion) or a 'normal' function (can be viewed as the executables own function). Following metrics were programmatically collected from CG

- $|V|$ is number of nodes in CG, i.e the function count of the callgraph
- For any $f \in V$, let $f = G(V_f, E_f)$ where $b \in V_f$ is a block of code, i.e each node in the callgraph is itself a graph, a flowgraph, and each node on the flowgraph is a basic block
- Define $IC : B \to N$ where B is defined to be set of blocks of code, and $IC(b)$ is the number of instructions in b. We denote this function shorthand as $|b|_{IC}$, the number of instructions in basic block b.

- We extend this notation $|\cdot|_{IC}$ to elements of V be defining $|f|_{IC} = \sum_{b \in V_f} |b|_{IC}$. This gives us the total number of instructions in a node of the callgraph, i.e in a function.
- Let $d_G^+(f)$, $d_G^-(f)$ and $d_G^{bb}(f)$ denote the indegree, outdegree and basic block count of a function, respectively.



(a) Malware: p vs $r_{in,out}$



(b) Goodware: p vs $r_{in,out}$

Fig. 2. Correlation Coefficient $r_{in,out}$

2.3. Correlations

I calculated the correlation between in and outdegree of functions. Prior analysis of static class collaboration networks [24][21] suggest an anti-correlation, characterizing some functions as source or sinks. I found no significant correlation between in and outdegree of functions in the disassembled executables (Fig. 2). Correlation intuitively is unlikely to occur except in the '0 outdegree' case (the BinNavi toolset does not generate the flowgraph for imported functions, i.e. an imported function automatically has outdegree 0, and but will be called from many other functions). Additional, I size-blocked both sample groups into three function count blocks, with block criteria chosen as $\Theta(10)$, $\Theta(100)$ and $\Theta(1000)$ function counts to investigate a correlation between instruction count in functions and complexity of the executable (with function count as a proxy). Again, I found no correlation at significance level $\leq$ 0.001. Coefficient values and the IQR for instruction counts (a spread measure, the difference between the 75th and the 25th percentiles of the sample) are given in Table 1. The first result corroborate previous findings; the second result at the phenomenological level agrees with the 'refactoring' model in [21], which posits that excessively long functions that tend to be decomposed into smaller functions. Remarkably, the spread is quite low, on the order of a few dozen instructions. I will discuss models more in section 4.

class	metric	$\Theta(10)$	$\Theta(100)$	$\Theta(1000)$
Goodware	r	0.05	-0.017	-0.0366
	IQR	12	44	36
Malware	r	0.08	0.0025	0.0317
	IQR	8	45	28

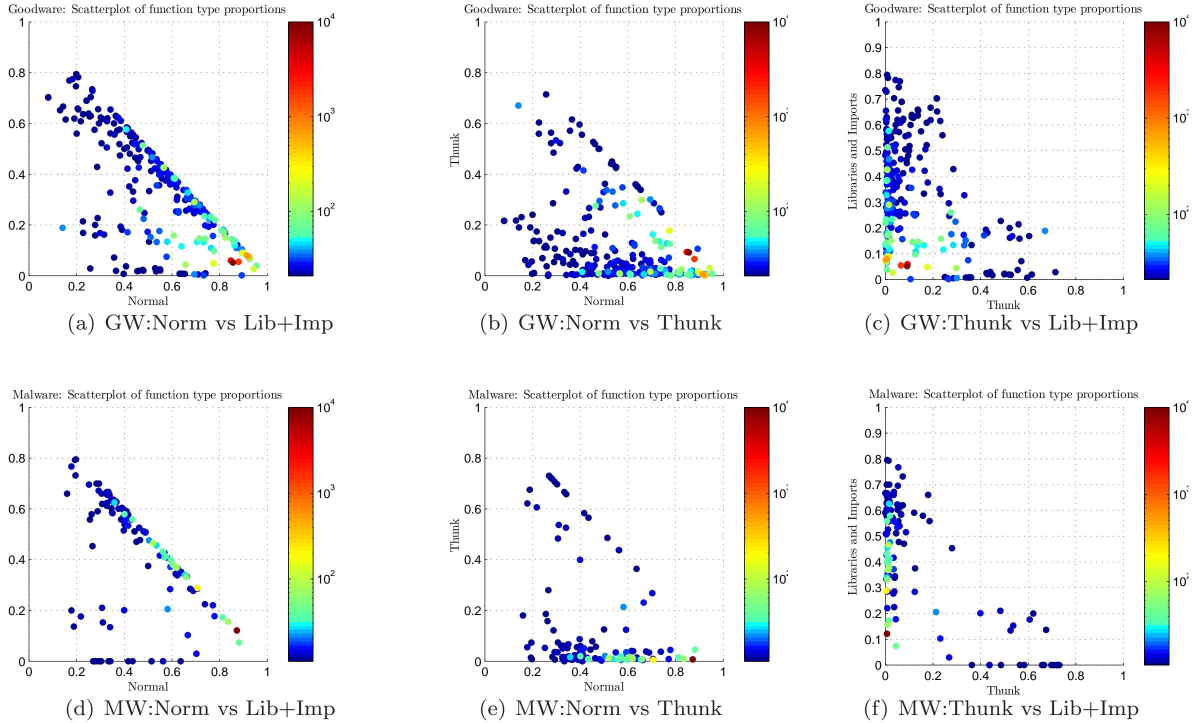
Table 1

Correlation, IQR for instruction count

2.4. Function types

Each point in the scatterplots in Fig. 3 represents three metrics for one individual executable: Function count, and the proportions of normal function, static library + dynamic import functions, and thunks. Proportions for an individual executable add up to 1. The four subgraphs are parsed thusly, using Fig. 3(b) as an example. The x-axis denotes the proportion of 'normal' function, and the y-axis the proportion of "thunk" functions in the binaries. The color of each point indicates $|V|$, which may serve as a rough proxy for the executable's size. The dark red point at (X,Y)= `(0.87, 0.007)` is `endnote.exe`, since it is the only goodware binary with functions count of $\Theta(10^4)$.

Most thunks are wrappers around imports, hence in small executables, a larger proportion of the functions will be thunks. The same holds for libraries: The larger the executable, the smaller the



(a) GW:Norm vs Lib+Imp (b) GW:Norm vs Thunk (c) GW:Thunk vs Lib+Imp

(d) MW:Norm vs Lib+Imp (e) MW:Norm vs Thunk (f) MW:Thunk vs Lib+Imp

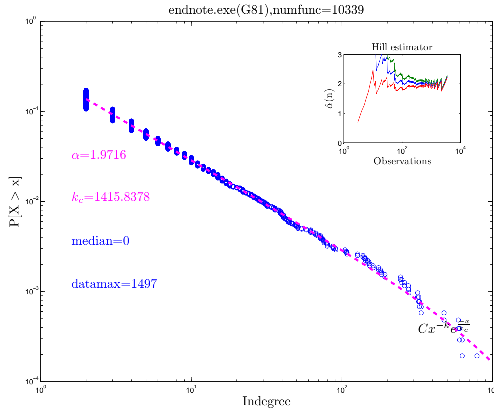
Fig. 3. Scatterplot of function type proportions

percentage of libraries. This is heavily influenced by the choice of dynamic vs. static linking. The thunk/library plot, listed for completeness reasons, does not give much information, confirming the intuition that they are independent of each other, mostly due to compiler behavior.

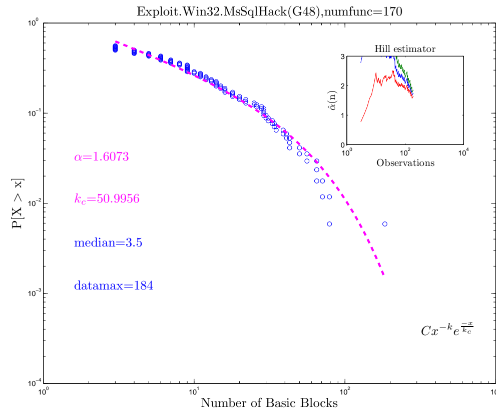
2.5. α fitting with Hill estimator

Taking my cue from [23] who surveyed empirical studies of technological, social, and biological networks, I hypothesize that the discrete distributions of $d^+(f), d^-(f)$ and $d^{bb}(f)$ follows a truncated powerlaw of the form $P_{d^\star(f)}(m) \sim m^{\alpha_{d^\star(f)}} e^{-\frac{m}{k_c}}$, where k_c indicates the end of the power law regime. Shorthand, I call $\alpha_{d^\star(f)}$ for the respective metrics α_{indeg}, α_{outdeg} and α_{bb}.

Figs. 4(a) and 4(b) show *pars pro toto* the fitting procedures for our 400 samples. The plot is an Empirical Complimentary Cumulative Density Plot (ECCDF). The x-axis show indegree, the y-axis show the CDF P[X>x] that a function in `endote.exe` has indegree of x. If P[X>x] can be shown to fit a Pareto distribution, we can extract the power law exponent for PMF $P_{d^\star(f)}(m)$ from the CDF fit (see [1] and more extensively [22] for the relationship between Pareto, power laws and Zipf distributions). Parsing Fig. 4(a)): Blue points denotes the data points (functions) and two descriptive statistics (median and the maximum value) for the indegree distribution for `endote.exe`. We see that for `endnote.exe`, 80% of functions have a indegree=1, 2% indegree >10. and roughly 1% indegree > 20. The fitted distribution is shown in magenta, together with the parameters α = `1.97` and k_c = `1415.83`. Although tempting, simply 'eyeballing' Pareto CDFs for the requisite linearity on a log-log scale [11] is not enough: Following [19] on philosophy and [25] on methodology, I calculate the Hill estimator $\hat{\alpha}$ whose asymptotical normality is then used to compute a 95% CI. This is shown in the inset and serves as a Pareto model self-consistency check that estimates the parameter α as a function of the number of observations. As the number of observations i increase, a model that is consistent along the data should show roughly $CI_i \supseteq CI_{i+1}$. For an insightful exposé and more recent procedures to estimate Pareto tails, see [33][8].



(a) GW sample: Fitting for $alpha_{indeg}$ and k_c



(b) MW sample: Fitting for $alpha_{bb}$ and k_c

Fig. 4. Pareto fitting ECCDFs, shown with Hill estimator inset

To tentatively corroborate the consistency of our posited Pareto model, 30 (goodware) and 21 (malware) indegree, outdegree and basic block ECCDF plots were uniformly sampled into three function count blocks, with block criteria chosen as $\Theta(10)$, $\Theta(100)$ and $\Theta(1000)$ function counts, yielding a sampling coverage of 10 %(goodware) and 17%(malware). Visual inspection indicates that for malware, the model seemed more consistent for outdegree than indegree at all function sizes. For basic block count, the consistency tends to be better for smaller executables. I see these tendency for goodware, as well, with the observation that outdegree was most consistent in size block $\Theta(100)$; for $\Theta(10)$ and $\Theta(1000)$. For both malware and goodware, indegree seemed the least consistent, quite a few samples did exhibit a so-called 'Hill Horror Plot' [25], where $\hat{\alpha}$s and the corresponding CIs were very jittery.

class	Basic Block	Indegree	Outdegree
GW	N(1.634,0.3)	N(2.02, 0.3)	N(1.69,0.307)
MW	N(1.7,0.3)	N(2.08,0.45)	N (1.68,0.35)
t	2.57	1.04	-0.47

Table 2

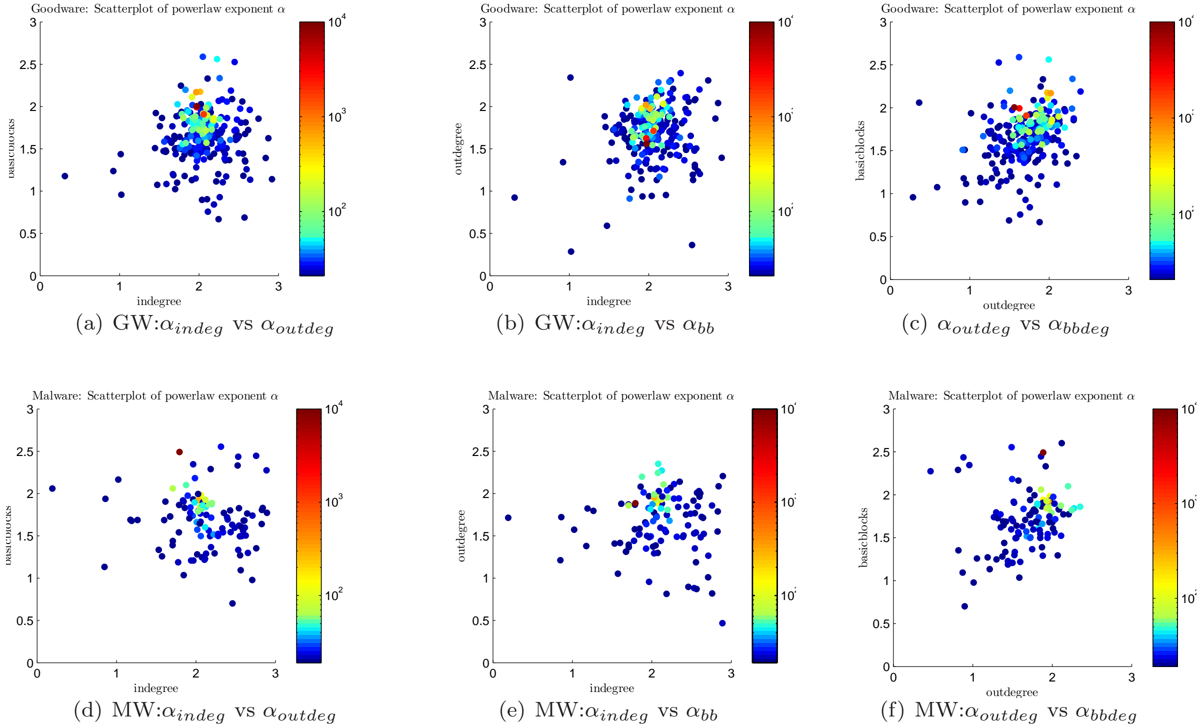
α distribution fitting and testing

The fitted power-law exponents α_{indeg}, α_{outdeg}, α_{bb}, together with individual functions' callgraph size are shown Fig. 5. For both classes, the range extends for $\alpha_{indeg} \approx$ [1.5-3], $\alpha_{outdeg} \approx$ [1.1-2.5] and $\alpha_{bb} \approx$ [1.1-2.1], with a slightly greater spread for malware.

2.6. Testing for difference

I now check whether there are any statistically significant differences between (α, k_c) fit for goodware and malware, respectively. Following procedures in [34], I find α_{indeg}, α_{outdeg} and α_{bb} distributed approximately normal. The exponential cutoff parameters k_c are lognormally distributed. Applying a standard two-tailed t-test (Table 2), I find at significance level 0.05 ($t_{critical}$=1.97) only $\mu(\alpha_{bb,malware}) \geq \mu(\alpha_{bb,goodware})$.

For the basic blocks, $k_c \approx LogN(59.1, 52)$ (goodware) and $\approx LogN(54.2, 44)$ (malware) and $\mu(k_c(bb, malware)) = \mu(k_c(bb, goodware))$ was rejected via Wilcoxon Rank Sum with $z = 13.4$. The steeper slope of malware's α_{bb} imply that functions in malware tend to have a lower basic block count. This can be accounted for by the fact that malware tends to be simpler than most applications and operates without much interaction, hence fewer branches, hence fewer basic blocks. Malware tends to have limited functionality, and operate independently of input from user and the operating environment. Also, malware is usually not compiled with aggressive compiler optimization settings. Such a regime leads to more inlining and thus increases the basic block count of the individual functions. It may be possible, too, that malware authors tend to break functions into simpler components than 'regular' programmers. The smaller cutoff point for malware seems to corroborate this, as well, in that the power law relationship holds over a shorter range. However, this explanation should be regarded as speculative pending further investigation.



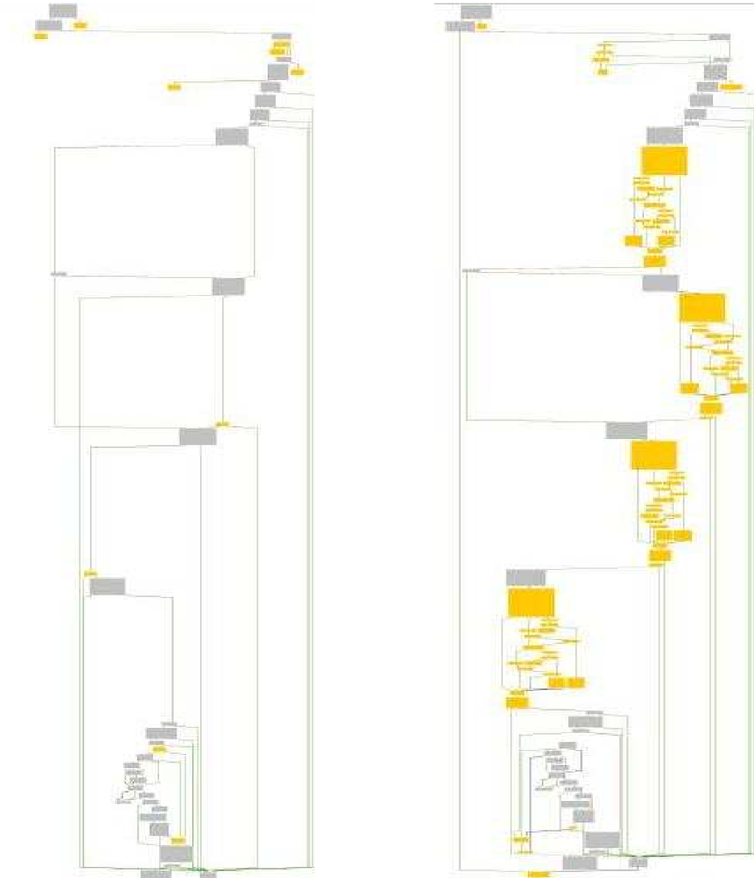
(a) GW:α_{indeg} vs α_{outdeg}
(b) GW:α_{indeg} vs α_{bb}
(c) α_{outdeg} vs α_{bbdeg}
(d) MW:α_{indeg} vs α_{outdeg}
(e) MW:α_{indeg} vs α_{bb}
(f) MW:α_{outdeg} vs α_{bbdeg}

Fig. 5. Scatterplots of α's

3. Related work

A simple but effective graph-based signature set to characterize statically disassembled binaries was proposed by Flake [9]. For the purposes of similarity analysis, he assigned to each function a 3-tuple consisting of basic blocks count, count of branches, and count of calls. These sets were used to compare malware variants and localize changes; an in-depth discussion of the involved procedures can be found in [7]. For the purposes of worm detection, Kruegel [14] extracts control flow graphs from executable code in network streams, augments them with a colouring scheme, identifies k-connected subgraphs that are subsequently used as structural fingerprints.

Power-law relationships were reported in [30] [21] [31] [3]. Valverde et al [30][31] measured undirected graph properties of static class relationships for Java Development Framework 1.2 and a racing computer game, ProRally 2002. They found the $\alpha_{JDK} \approx 2.5 - 2.65$ for the two largest (N_1=1376, N_2=1364) connected components and $\alpha_{game} \approx 2.85 \pm 1.1$ for the game (N=1989). In the context of studying time series evolution of C/C++ compile-time "#include" dependency graphs, $\alpha_{in} \approx 0.97 - 1.22$ and an exponential outdegree distribution are reported. This asymmetry is not explained. Focusing on the properties of directed graphs, Potanin et al [24] examined the binary heap during execution and took a snapshot of 60 graphs from 35 programs written in Java, Self, C++ and Lisp. They concluded that the distributions of incoming and outgoing object references followed a power law with $\alpha_{in} \approx 2.5$ and $\alpha_{out} \approx 3$. Myers [21] embarked on an extensive and careful analysis of six large collaboration networks (three C++ static class diagrams and three C callgraphs) and collected data on in/outdegree distribution, degree correlation, clustering and complexity evolution of individual classes through time. He found roughly similar results for the callgraphs, $\alpha_{in} \approx \alpha_{in} \approx 2.5$, and noted that it was more likely to find a function with many incoming links than outgoing ones. More recently, Chatzigeorgiou et al [3] applied algebraic methods to identify, among other structures, heavily loaded 'manager' classes with high in- and outdegree in three static OO class graphs.



(a) Compiler: CFG without loop unrolling (b) Compiler: CFG with loop unrolling

Fig. 6. Basic Block differences in CFG under compiler optimization regimes

4. A HOT Process

I hypothesize that the call-graph features described in the preceding sections may be the phenomenological signature of two distinct, domain-specific HOT (Highly Optimized Tolerance) optimization processes; one involving human designers and the other, code compilers. HOT mechanisms are processes that induce highly structured, complex systems (like a binary executable) through processes that seek to optimally allocate resources to limit event losses in an probabilistic environment [2].

4.1. Human design and coding as HOT mechanism

The first domain-specific mechanism that induces a cost-optimized, resource-constrained structure on the executable is the human element. Humans using various best-practice software development techniques [15][10] have to juggle at various stage of the design and coding stages: Evolvability vs specificity of the system, functionality vs code size, source readability vs development time, debugging time vs time-to-market, just to name a few conflicting objective function and resource

constraints. Humans design and implement programs against a set of constraints, taking implicitly (rarely explicitly) the probability of the event space into consideration, indirectly through the choice of programming language (typed, OO, procedural, functional etc) and directly through the design choice of data structures and control flow. Human programmers generally design for average (or even optimal) operating environments; the resulting programs deal very badly with exceptional conditions effected by random inputs [18][17] and resource scarcity [32]. These findings are consistent with an optimization-based code generation process.

4.2. Compiler as HOT mechanism

The second domain-specific mechanism that induces a cost-optimized, resource-constrained structure on the executable is the compiler. The compiler functions as a HOT process. Cost function here include memory footprint, execution cycles, and power consumption minimization, whereas the constraints typically involves register and cache line allocation, opcode sequence selection, number/stages of pipelines, ALU and FPU utilization. The interactions between at least 40+ optimization mechanisms (in itself a network graph [20, pp.326+]) are so complex that meta-optimization [13] have been developed to heuristically choose a subset from the bewildering possibilities. Although the callgraph is largely invariant under most optimization regimes, the more aggressive mechanisms can have a marked effect on callgraph structure. Fig. 6(a) shows a binary's CFG induced by the Intel C++ Compiler 9.1 under a standard optimization regime. The yellow section are loop structures. Fig. 6(b) shows the binary CFG of the same source code, but compiled under a more aggressive inlining regime. We see that the compiler unrolled the loops into an assortment of switch statements, vastly increasing the number of basic blocks, and hence changing the executable's structural features.

5. Conclusion

I started by analyzing the callgraph structure of 120 malicious and 280 non-malicious executables, extracting descriptive graph metrics to assess whether statistically relevant differences could be found. Malware tends to have a lower basic block count, implying a simpler structure (less interaction, fewer branches, limited functionality). The metrics under investigation were fitted relatively successfully to a Pareto model. The power-laws evidenced in the binary call-graph structure may be the result of optimization processes which take objective function tradeoffs and resource constraints into account. In the case of the callgraph, the primary optimizer is the human designer, although under aggressive optimization regimes, the compiler will alter the callgraph, as well.

Appendix

The goodware samples were indexed, collected and meta-data identified using *Index Your Files - Revolution! 3.1*, *Advanced Data Catalog 1.51* and *PEiD 0.94*, all freely available from `www.softpedia.com`. The executable's callgraph generation and structural identification was done with *IDA Pro 5* and a pre-release of *BinNavi 1.2*, both commercially available at `www.datarescue.com` and `www.sabre-security.com`. Programming was done with Python 2.4, freely available at `www.python.org`. Graphs generated with and some analytical tools provided by *Matlab 7.3*, commercially available at `www.matlab.com`.

I would like to thank Thomas Dullien (SABRE GmbH) without whose superb tools and contributions this paper could not have been written. He serves as the inspiration to me for this line of research. Furthermore, I would like to thank Walter Willinger (AT&T Research), Ero Carrera (SABRE), Jason Geffner (Microsoft Research), Scott Anderson, Frankly Turbak, Mark Sheldon, Randy Shull, Frédéric Moisy (Université Paris-Sud 11), Mukhtar Ullah (Universität Rostock), David Alderson (Naval Post Graduate School), as well as the anonymous reviewers for their helpful comments, suggestions, explanations, and arguments.

References

[1] L. Adamic and B. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.

[2] J. M. Carlson and J. Doyle. Highly optimized tolerance: A mechanism for power laws in designed systems. *Physical Review E*, 60(2):1412+, 1999.

[3] A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides. Application of graph theory to OO software engineering. In *WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, pages 29–36, New York, NY, USA, 2006. ACM Press.

[4] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Security '03:Proceedings of the* 12th *USENIX Security Symposium*, pages 169–186. USENIX Association, USENIX Association, August 2003.

[5] A. Clementi. Anti-virus comparative no. 11. Technical report, Kompetenzzentrum IT, Insbruck (Austria), August 2006. `http://www.av-comparatives.org/seiten/ergebnisse/report11.pdf`.

[6] T. Dullien. Binnavi v1.2. `http://www.sabre-security.com/products/binnavi.html`, 2006.

[7] T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *SSTIC '05: Symposium sur la Sécurité des Technologies de l'Information et des Communications*, Rennes, France, June 2005.

[8] Z. Fan. *Estimation problems for distributions with heavy tails*. PhD thesis, Georg-August-Universität zu Göttingen, 2001.

[9] H. Flake. Compare, Port, Navigate. Black Hat Europe 2005 Briefings and Training, March 2005.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[11] M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. *European Journal of Physics B*, 41(2):255–258, September 2004, cond-mat/0402322.

[12] I. Guilfanov. Ida pro v5.0.0.879. `http://www.datarescue.com/idabase/`, 2006.

[13] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *CF '05: Proceedings of the* 2nd *conference on Computing Frontiers*, pages 180–188, New York, NY, USA, 2005. ACM Press.

[14] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In A. Valdes and D. Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2005.

[15] J. Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[16] E. Limpert, W. A. Stahel, and M. Abbt. Log-normal distributions across the sciences: Keys and clues. *BioScience*, 51(5):341–352, May 2001.

[17] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings of the 1st International workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM Press.

[18] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communication of the ACM*, 33(12):32–44, 1990.

[19] M. Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–334, 2004.

[20] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1998.

[21] C. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 68(4):046116, 2003.

[22] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.

[23] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167, 2003.

[24] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in OO programs. *Commununication of the ACM*, 48(5):99–103, 2005.

[25] S. Resnick. Heavy tail modeling and teletraffic data. *Annals of Statistics*, 25(5):1805–1869, 1997.

[26] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River (NJ), 2003.

[27] P. Szor. *The Art of Computer Virus Research and Defense*, pages 252–293. In [28], February 2005.

[28] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Upper Saddle River (NJ), February 2005.

[29] P. Szor and P. Ferrie. Hunting for metamorphic. In *VB '01: Proceedings of the* 11th *Virus Bulletin Conference*, September 2001.

[30] S. Valverde, R. Ferrer Cancho, and R. V. Solé. Scale-free networks from optimal design. *Europhysics Letters*, 60:512–517, November 2002, cond-mat/0204344.

[31] S. Valverde and R. V. Sole. Logarithmic growth dynamics in software networks. *Europhysics Letters*, 72:5–12, November 2005, physics/0511064.

[32] J. Whittaker and H. Thompson. *How to break Software security*. Addison Wesley (Pearson Education), June 2003.

[33] W. Willinger, D. Alderson, J. C. Doyle, and L. Li. More normal than normal: scaling distributions and complex systems. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 130–141. Winter Simulation Conference, 2004.

[34] G. T. Wu, S. L. Twomey, , and R. E. Thiers. Statistical evaluation of method-comparison data. *Clinical Chemistry*, 21(3):315–320, March 1975.