Malware Analysis and Attribution Using Genetic Information

Avi Pfeffer, Catherine Call, John Chamberlain, Lee Kellogg, Jacob Ouellette, Terry Patten, Greg Zacharias
Charles River Analytics

Arun Lakhotia, Suresh Golconda
University of Louisiana, Lafayette

John Bay, Robert Hall, Daniel Scofield
Assured Information Security

Abstract

As organizations become ever more dependent on networked operations, they are increasingly vulnerable to attack by a variety of attackers, including criminals, terrorists and nation states using cyber attacks. New malware attacks, including viruses, Trojans, and worms, are constantly and rapidly emerging threats. However, attackers often reuse code and techniques from previous attacks. Both by recognizing the reused elements from previous attacks and by detecting patterns in the types of modification and reuse observed, we can more rapidly develop defenses, make hypotheses about the source of the malware, and predict and prepare to defend against future attacks. We achieve these objectives in Malware Analysis and Attribution using Genetic Information (MAAGI) by adapting and extending concepts from biology and linguistics. First, analyzing the "genetics" of malware (i.e., reverse engineered representations of the original program) provides critical information about the program that is not available by looking only at the executable program. Second, the evolutionary process of malware (i.e., the transformation from one species of malware to another) can provide insights into the ancestry of malware, characteristics of the attacker, and where future attacks might come from and what they might look like. Third, functional linguistics is the study of the intent behind communicative acts; its application to malware characterization can support the study of the intent behind malware behaviors. To this point in the program, we developed a system that uses a range of reverse engineering techniques, including static, dynamic, behavioral, and functional analysis that clusters malware into families. We are also able to determine the malware lineage in some situations. Using behavioral and functional analysis, we are also able to identify a number of functions and purposes of malware.

1. Introduction

As organizations become ever more dependent on networked operations, they are increasingly vulnerable to attack by a variety of attackers, including criminals, terrorists and nation states using cyber attacks. New malware attacks, including viruses, Trojans, and worms are constantly and rapidly emerging threats. A lack of a deep, automated analysis of malware makes it hard to identify novel attacks, characterize the source of attacks (e.g., where does it come from and from what type of attacker), and forecast future attacks. Currently, manual analysis is relied upon, which is extremely time consuming and does not scale to the large amount of malware being encountered. Meanwhile, current automated systems use heuristics, such as signatures, to detect malware. These fail to detect most novel attacks because they do not provide a way to connect novel malware to known malware. They also provide no basis for characterizing malware sources or forecasting future malware developments.

Our approach to developing a deep understanding of malware relies on two key insights, each of which produces a useful analogy. First, attackers often *reuse code and techniques* from one malware product to the next. They want to avoid having to write malware from scratch each time, but they also want to avoid detection, so they try to *hide similarities*. If we can understand reuse patterns and recognize reused elements, we will be able to connect novel malware to other malware from which it originated, create a searchable database of malware with relationships based on similarity, and predict and prepare to defend against future attacks.

This insight suggests a *biological analogy*, in which a malware sample is compared to a living organism. Just like a living organism, the malware sample has a *phenotype*, consisting of its observed properties and behavior (e.g. eye color for an organism, packer type of a sample). The phenotype is not itself inherited between organisms; rather it is the expression of the

genotype, which is inherited. Likewise, it is the source code of the malware that is inherited from one sample to the next. By reverse engineering the malware to discover its intrinsic properties, we can get closer to the genetic features that are inherited.

The second insight is that the *function* of malware has a crucial role to play in understanding reuse patterns. The function of malware – what it is trying to accomplish – is harder to change than the details of how it is accomplished. Therefore, analysis of the function of malware is a central component of our program. In our analysis of function, we use a *linguistic analogy*: a malware sample is like a linguistic utterance. Just like an utterance, malware has temporal structure, function, and context in which it takes place. The field of *functional linguistics* studies utterances not only for their meaning, but also for the function they are trying to accomplish and the context in which they occur.

In this paper, we describe our Malware Analysis and Attribution Using Genetic Information (MAAGI) program. The program, which is currently at the end of its second year, has already produced promising results in determining malware similarity, clustering malware into families, determining the temporal ordering of malware, generating malware lineages, and identifying behaviors and purposes of malware. MAAGI uses an array of reverse engineering techniques, including semantic and functional analysis, coupled with a central evolutionary analysis component that fuses these analyses to determine the malware lineage. After providing some background in Section 2, we prevent an overview of the system architecture in Section 3. We describe our reverse engineering components in Section 4 and our approach to evolutionary analysis in Section 5, before concluding in Section 6.

2. Background

One of the advantages of our approach was that it largely did not compete with existing research; it borrowed significantly from it. While the genetic, evolutionary, and linguistic-based malware analysis tools are innovative, especially in their ability to reason about attackers, they are also related to a variety of existing efforts that each attempt to analyze malware for the purposes of recognizing and identifying attacks. By using a centralized arbiter, we are able to incorporate and reuse many of these existing techniques (e.g., [1-3]) and leverage their particular strengths and insights into the analysis of the phylogeny of malware and the classification of malware instances.

There has been significant work over the past two decades in detecting and classifying malware. For instance, static, binary-level analysis is still the most common type of analysis used by anti-virus software (e.g., [4;5]). This approach looks at the object code of the incoming executable to try to identify bit strings that have been seen in previous malware attacks. This approach is able to rapidly detect previously seen malware and some instances of novel malware where there is significant binary-level overlap. Unfortunately, it is easily fooled into giving false negatives by minor changes to the code at the higher-level program level (e.g., moving functions around, adding no-ops or uncalled dummy functions), changing the packing algorithm used, or changing which compiler is used (or which flags are given to a compiler). It can also give false positives because non-maliciously packed software can share a packer header that is bitwise identical.

To address these weaknesses, a range of new techniques have been developed, each of which makes an attempt to abstract away from the raw, static binary. For instance, a simple improvement is to do a bitwise comparison on the *unpacked* code (e.g., [6-9]). It is also possible to analyze the call-graph structure to detect which code is actually callable (e.g., [3]). Or, one can use dynamic analysis where the suspected malware is executed in a safe sandbox and its runtime performance is evaluated against the behavior of historical instances of malware [2;10]. All of these, however, have their own weaknesses. Static analysis, even of unpacked code, is still vulnerable to many forms of obfuscation, such as code transposition. Call-graph analyses are vulnerable to other forms of obfuscation, such as creating callable but un-called functions. And dynamic analyses are vulnerable to mimicry, attacks on the analysis system itself, and malicious behavior that only happens after some time (e.g., on a particular date).

The use of many types of analytic techniques is similar to the multiple types of analysis used by biologists who study organisms' external anatomy (e.g., looking at the presence of hair), internal anatomy (e.g., looking at organs), behavior (e.g., nocturnal animals, nesting animals), the ecological habitat (e.g., which animals live in a certain ecological niche), genetics (e.g., which animals share significant amounts of genetic material and which genes), and evolution (e.g., which species descended from which other species). Just as in biology, where each of these types of analysis is useful but insufficient to characterize the subject fully, each of the current approaches to analyzing malware has specific advantages and disadvantages.

3. System architecture

The general architecture of our approach is shown in Figure 1. Malware (or potential malware) enters the system either as historical instances or live data coming in from the network. MAAGI consists of two major subsystems. *Reverse Engineering* operates on each individual malware to obtain features of the malware that are not present at the surface. *Evolutionary Analysis* operates on the entire set of malware as a whole to cluster the malware and produce a lineage graph. The outputs of the system are *Clusters* of the malware into families, a *Lineage Graph*, and identified *Traits* and *Purposes* of the malware.

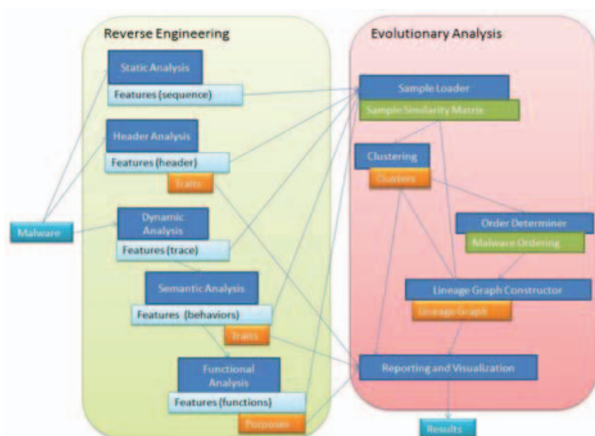

Figure 1: MAAGI architecture

4. Reverse engineering

One of the main principles behind MAAGI is that combining a variety of analyses can produce better results than each individual analysis method. Accordingly, the Reverse Engineering component consists of five different kinds of analysis, each of which produces malware *Features*:

Header Analysis is a simple component of the MAAGI system, but one that produces much useful information. It processes the malware's PE header, which is useful both for characterizing the malware and for clustering and lineage construction. The header features are passed to the evolutionary analysis.

Dynamic Analysis analyzes the behavior of the malware, which is executed in a *Secure Sandbox.* Dynamic analysis consists first of t*race analysis*, which uses AIS's Cuckoo dynamic trace environment with special-purpose hooks to produce the execution trace of the malware. The traces become features for the evolutionary analysis.

Trace analysis then feeds into *Semantic Analysis*, which analyzes the sequence of system calls made by the malware, together with a semantic understanding of the calls and their arguments, to identify behaviors of the malware. The Semantic Extractor (SemEx) merges dynamic API traces and expert programming knowledge to build an RDF graph that can be searched for patterns that identify high level behaviors. The patterns can contain many relations, including temporal orders and data flow relations. For example, the semantic extractor can determine whether the malware makes a network connection, receives data on that specific connection, writes that specific data to a file, and then executes that specific file.

According to functional linguistic theory, it is as important to understand the function and context of an utterance and its components as it is the plain meaning of the words. We apply the same principles to malware through our *Functional Analysis* component. The functional analysis takes in behaviors found by the semantic analysis and analyzes them using the framework of *probabilistic systemic functional grammars* (PSFGs). Behaviors are parsed using a PSFG to infer functional classes. We then use an explicit many-many mapping between functional classes and characterization/attribution classes. For example, given input behaviors ReadDriverSetupLog and CopySelfToUSB, we can infer that the malware is air gap capable and self-replicating, and possibly that the attacker has anti-forensics knowledge.

For an example of semantic analysis, we now describe the semantic analysis we performed on directory walks. In a trace file we observed a set of related invocations. The relationships include things like data dependencies and juxtapositions. The simple directory walk we have observed looks something like: (**FindFirstFile**, **CreateFile**, **ReadFile**, **ReadFile**, **ReadFile**, **CloseFile**, **Send**, **FindNextFile**, **CreateFile**, **ReadFile**, **ReadFile**, **ReadFile**, **CloseFile**, **Send**, **FindNextFile**, **FindClose**). In this case, the **FindFirstFile** invocation returns a value of type HANDLE which is consumed by the subsequent **FindNextFile** and **FindClose** invocations. This allows us to infer the existence of a directory walk containing all of these invocations. Note that it is important (but not required) that the application have a failed **FindNextFile** and successful **FindClose** at the end of the sequence, as these allow us to know that the walk has ended due to normal completion.

To break this down, and label individual invocations as part of higher level behaviors, we create a table of invocations labeled with semantic tags, or lower level behaviors (LLBs), where each invocation is associated with the intermediate level behaviors (ILBs) and high level behaviors (HLBs) to which it belongs. The table is then encoded in an RDF structure. The first part of the table for our example is in Table 1.

Table 1: Part of behavioral analysis table

Inv-ID	Invocation	Semantic Tags (LLB)	Intermediate Behavior		HLB
Inv-1	FindFirstFile	Success x Opens x ProgramaticFileListing	DirectoryWalk-n-Start	Directory-Walk-n-Step-1	DirectoryWalk-n (ProgramaticDirectoryWalk)
Inv-2	CreateFile	Success x Opens x File			
Inv-3	ReadFile	Success x Reads x File			
Inv-4	ReadFile	Success x Reads x File			
Inv-5	ReadFile	Failure x Reads x File			
Inv-6	CloseFile	Success x Closes x File			
Inv-7	Send	Success x Writes x Network			
Inv-8	FindNextFile	Success x ProgramaticFileListing	DirectoryWalk-n-Step-2		
Inv-9	CreateFile	Success x Opens x File			

In addition to the wider directory walk, we are able to infer other HLBs and ILBs that occur inside of the directory walk steps. A specific example of this in the example above is that we read two files until the end of the file. In our ontological system we add containment relationships to capture these additional behaviors, e.g., "DirectoryWalk-n has steps 1-2; DirectoryWalk-n-Step-1 contains behavior ReadUntilEndFile-j; DirectoryWalk-n-Step-2 contains behavior ReadUntilEndFile-k".

Our semantic and functional analyses have already made significant progress towards being able to characterize useful properties of malware. For example, we have performed an automated analysis of a Koobface sample in which we found a number of behaviors, including directory walks and file searches, file downloads and concurrent execution. Our system has the potential to assist malware analysts by automatically discovering behaviors that previously required time-consuming analysis to find. In addition, we are able to detect a number of purposes of malware in some cases, such as Remote Access Tool (RAT), virus, rootkit, dropper, reconnaissance, keylogger, and destruction.

5. Evolutionary analysis

The first step of Evolutionary Analysis is to compare the similarity between malware samples based on their features. Similarity is computed by a memory-efficient component called the *Sample Loader*, which uses a trie structure to produce a similarity matrix from the features of all the samples. In this similarity matrix, rare features are weighted more highly, because sharing a rare feature provides stronger evidence of a relationship between samples than a common feature.

Our hypothesis has been that using multiple feature types to measure malware similarity produces significantly better results than any single feature. This hypothesis is borne out in our experiments. Figure 2 shows the individual similarity matrices obtained from seven features, as well as the matrix obtained from combining all these features. The data for these experiments is a collection of 140 malware samples from eight families. Each similarity matrix has samples across the rows and columns, so that each entry indicates the similarity between the row and column sample. Brightness indicates the degree of similarity. The samples are sorted by true family, so the ideal similarity matrix would contain eight bright squares along the diagonal. Of the input matrices, the one from static analysis comes closest to this ideal, but lacks definition in many of the families. Some of the behavior and header features find greater similarity within some of the families, but also find strong similarities between samples in different families, as indicated by the bright red regions away from the diagonal. In addition, these features are completely missing in some of the samples, indicated by the solid blue. Our combined output is, qualitatively, by far the closest to the ideal similarity matrix, and it is robust to missing inputs for some feature types.

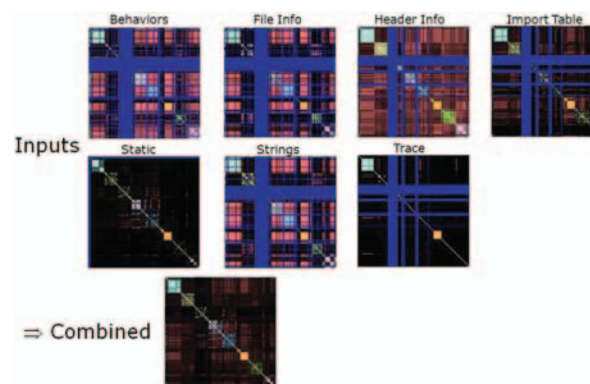

Figure 2: Individual and combined similarity matrices

After the similarity matrices are computed, the *Clustering* component clusters the malware into families. The goal is to group samples that are strongly related, maximizing intra-cluster similarity (cohesion) and minimizing inter-cluster similarity (adhesion). We

use hierarchical clustering, choosing the level of the hierarchy to maximize the cohesion/adhesion tradeoff. We use two alternative clustering algorithms: the Louvain algorithm [11], which is an agglomerative algorithm that maximizes the modularity of clusters based on in-links and out-links; and the Girvan-Newman algorithm [12], which is a partitional algorithm that iteratively removes edges that are most between two clusters. Since clustering is strongly dependent on good similarity, it naturally improves as we combine more features.

To determine the lineage of malware, it is essential to know the order in which samples were generated. Without this information, it would be hard to determine the direction of parent-child relationships. The compiler timestamp in the header is an indication of the time at which malware is generated, but it may be missing or obfuscated. However, it should not be ignored completely, as it provides an accurate signal in cases where it is not obfuscated. The challenge of determining ordering is to (a) determine which samples have obfuscated timestamps, and (b) determine the correct times of those samples. To achieve these goals, our *Order Determiner* uses two additional pieces of information. First, it uses the size of the sample, as a proxy for sample complexity, since samples within a family tend to grow in complexity over time. Second, it uses the similarity between samples as a signal, since similar samples should be close to each other in time.

We fuse all this information using a probabilistic graphical model, containing variables representing the true and observed timestamp of each sample, as well as a variable indicating whether the timestamp was obfuscated. We place soft constraints on the timestamps enforcing the size and similarity heuristics. The graphical model is implemented in our Figaro probabilistic programming language [13], which enables probabilistic models to be expressed as programs making it easy to express these complex constraints.

Because our malware corpora do not include ground truth about the generation times of malware, we use benign software to evaluate our ordering algorithm. Figure 3 shows results for one such dataset, the MCMap Github repository. In each experiment, we obfuscated the timestamps of some of the samples and attempted to recover the ordering of all samples. The horizontal axis shows the percentage of timestamps that were obfuscated, while the vertical axis shows the percentage of pairs that were correctly ordered with respect to each other. The blue line shows the baseline performance achieved by taking the observed timestamps as correct. The other lines show the performance of the graphical model using various weightings of size and similarity. The similarity heuristic alone performs extremely well with smaller amounts of obfuscation but tails off with larger amounts, while size alone performs less well with smaller amounts but holds up well even with 100% obfuscation. Combining size and similarity provides the benefit of both. In particular, Combined 1:2, which weights size twice as highly is similarity, performs almost as well as similarity alone with lower levels of obfuscation, and as well as size alone with higher levels.

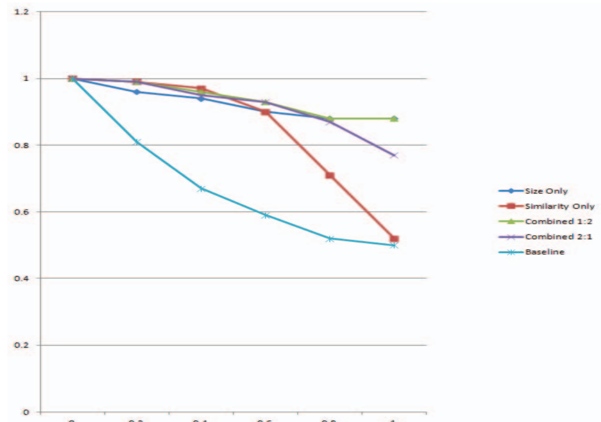

Figure 3: Ordering results for MCMap dataset

The final step in evolutionary analysis is the *Lineage Graph Constructor*. We view lineage construction as an optimization problem. We want to provide the best possible explanation of the features of each sample. In general, a feature can be explained in one of three ways: it can be inherited from a parent, a mutation of a feature of a parent, or fresh (generated de novo). We prefer the first explanation; however, we also prefer to minimize the number of parents of a sample, since most samples will be generated from a small number of parents (often one). In addition, we prefer simpler structures for the lineage as a whole.

While we have explored many algorithms to solve this problem, our final algorithm is a combination of a directed best parent algorithm, which optimizes individual parent-child relationships, and a minimum spanning tree algorithm, which optimizes the lineage as a whole. Our algorithm, called best parent with minimum spanning merging, combines the best of both approaches. First, it identifies the best parent of each sample. Then, it creates strong straight line sublineages using the best parents. Finally, it merges sublineages using the minimum spanning tree. Lineages are constructed for each cluster separately, and then merged as appropriate.

Figure 4 shows results from our evaluation of our lineage construction algorithm on benign samples from Github as well as a hand-generated dataset consisting

of fifteen malware samples with known ground truth (labeled as “Risk Reduction Dataset” in the figure). We measured the precision and recall of our parent-child predictions. If we define TP as the fraction of parent-child relationships in the ground truth lineage that were correctly identified by the algorithm, FN as the fraction of correct parent-child relationships that were not identified by our algorithm, and FP as the fraction of parent-child relationships identified by the algorithm, then the precision is defined by $\frac{TP}{TP+FP}$, while recall is $\frac{TP}{TP+FN}$. There is a natural tradeoff between false positive and false negative rate that is captured by precision and recall. To summarize precision and recall, we report the F-measure, which is defined to be $2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$.

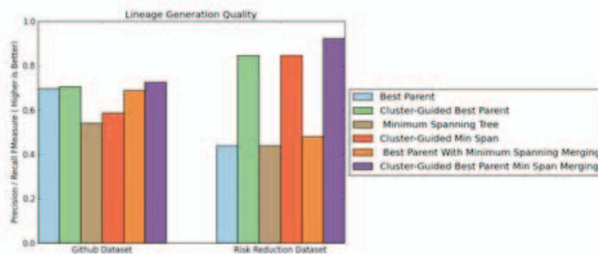

Figure 4: Lineage results on two datasets

The figure compares three algorithms with two versions each. The three algorithms are best parent alone, minimum spanning tree, and best parent with minimum spanning merging. For each algorithm, we show both the basic version that operates on all clusters simultaneously and the version that builds lineages on individual clusters separately before merging them as appropriate. We see that the cluster guided versions of the algorithms do better in all cases, and especially on the malware dataset. An explanation for the significant difference in the malware dataset is that that dataset contained a large number of noise samples that could interfere with the lineage; by focusing the lineage constructions on identified clusters, we avoid mixing the noise samples in the lineage. Of the three algorithms, we see that best parent with minimum spanning merging is slightly better than the others. Overall, the algorithm achieved a precision-recall F-measure of 73% on the Github data and 92% on the malware.

7. Conclusion

We have described our Malware Analysis and Attribution through Genetic Information (MAAGI) effort. MAAGI uses reverse engineering, including semantic and functional analysis, and evolutionary analysis to determine the lineage of malware and characterize its source. Our approach of using multiple reverse engineering analysis components to produce “genetic” features appears to be successful. Our current results on determining malware similarity, clustering malware, order determination, lineage construction, and identifying behaviors and purposes, are promising. We plan to continue working on this project to improve these results, as well as to apply the lineage, together with our semantic and functional analysis, to assist in the source characterization and attribution of malware.

Acknowledgements

This work was supported by US Air Force contract FA8750-10-C-0171, with thanks to Dr. Michael VanPutte and Mr. Timothy Fraser. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] M. R. Chouchane, A. Walenstein, and A. Lakhotia, "Metamorphic Authorship Recognition Using Markov Models," *Virus Bulletin*, pp. 8-11, May2008.

[2] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," 2007.

[3] G. Erdelyi and E. Carrera, "Digital Genome Mapping: Advanced Binary Malware Analysis," Chicago, IL: 2004, pp. 187-197.

[4] R. W. Lo, K. N. Levitt, and R. A. Olsson, "MCF: A Malicious Code Filter," *Computers & Security*, vol. 14, no. 6, pp. 541-566, 1995.

[5] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, "Constructing Computer Virus Phylogenies," *Journal of Algorithms*, vol. 26, pp. 188-208, 1998.

[6] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," 2006, pp. 289-300.

[7] L. Martignoni, M. Christodeorescu, and S. Jha, "OmniUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," 2007, pp. 431-441.

[8] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis Via Hardware Virtualization Extensions," 2008, pp. 51-62.

[9] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic Reverse Engineering of Malware Emulators," 2009.

[10] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic Analysis of Malware Behavior Using Machine Learning," *Technical Report*, vol. 18, no. 209 2009.

[11] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast Unfolding of Communities in Large Networks,"2008.

[12] M. Girvan and M. E. J. Newman, "Community Structure in Social and Biological Networks," *Proceedings of the National Academy of Sciences*, vol. 99, pp. 7821-7826, 2002.

[13] A. Pfeffer, "Creating and Manipulating Probabilistic Programs with Figaro," 2012.