

Inferring Accurate Histories of Malware Evolution from Structural Evidence

Craig Darnetko, Steven Jilcott, John Everett

{cdarnetk, sjilcott, jeverett}@bbn.com, Raytheon BBN Technologies

Abstract

An important problem in malware forensics is generating a partial ordering of a collection of variants of a malware program, reflecting a history of the malware's evolution as it is adapted by the original or new authors. Frequently the only temporal clue to which variants were developed earlier is the date on which they were first observed in the wild. In the absence of reliable temporal clues, our approach leverages heuristic evidence based on common trends in the evolution of software structure over time. We extract structural features from each variant binary executable and generate from them three different forms of evidence that one variant is a likely ancestor of another. We then combine this evidence using a truth maintenance system to create a family tree of malware variants.

Introduction

An important problem in malware forensics is generating a partial ordering of a collection of variants of a malware program. Such an ordering provides a “family tree” tracking the malware's evolution (henceforth called a *lineage*), helping the analyst understand malware trends in multiple ways. First, the analyst can see how malware has evolved as the original author or new authors adapted the malware to deal with new defensive measures or new targets. Second, analysts can see where one malware program has copied or imported code from a completely different program. Finally, all of these insights help analysts speculate about the origins of malware and, in some cases, contributes to attribution.

Malware binaries recovered in the wild lack explicit temporal clues, making temporal ordering a challenging problem. Timestamps produced during compilation are easy to falsify or remove, and most malware authors do so.

About the only piece of temporal evidence available is the date of sample collection, which is an unreliable guide to whether one variant was derived from another.

Recently, security researchers have begun to look toward classical AI techniques to reconstruct lineage. Gupta et al. [2] used metadata (such as time of collection and analyst notations) compiled by McAfee in a knowledge-based approach to lineage reconstruction. Lineage reconstruction using only features from malware binaries is essentially a new problem. In work sponsored by DARPA, Dumitras and Neamtiu [1] proposed tracking individual ‘traits’ (e.g., small code subsets) across samples, and training a classifier to properly identify lineage relationships. Their work led to the formation of the DARPA Cyber Genome program, of which our effort is one part. DARPA Cyber Genome sponsors four efforts examining the lineage problem; our effort called DECODE, MalLET (from Lockheed Martin Advanced Technology Labs), MAAGI (from Charles River Analytics), and MATCH (from Sentar, Inc.). Our effort is the first to publish a lineage approach; thus we are presently unable to provide summaries of the alternative technologies, although we summarize our performance relative to the alternatives in the concluding section.

Our solution uses evidence of structural change to determine how malware has evolved. Lehman articulated a set of “Laws of Software Evolution” [4], observations about how traditional, proprietary software evolves over time, and the International Organization for Standards has documented various techniques and strategies that are often used during software evolution [3]. As software is maintained, functionality is added or subtracted, code is restructured and bugs are fixed. Certain structural changes, then, provide evidence that one variant is an ancestor of another.

Our paper makes the following contributions:

- We describe how we extract and represent three types of structural change evidence from malware binaries.
- We present an approach for combining evidence using a truth maintenance system, and deduce a lineage from a collection of samples.
- We evaluate our approach on a set of malware lineages provided by another organization for use in Cyber Genome for which precise ground truth lineage was known.

We first overview the process of extracting structural information from binary executables, identifying evidence of structural change, and then reconstructing a lineage based on that evidence. We then give a detailed description of each of the three types of lineage evidence. Next we describe our use of the Jess rule engine (<http://herzberg.ca.sandia.gov/>) to implement a truth maintenance system and generate a lineage. We conclude with an evaluation of our approach on several sets of malware variants and summarize our performance relative to the alternative approaches in Cyber Genome.

Overview

A lineage is a partial ordering of malware samples, and can be represented as a graph where each vertex represents a sample, and a directed edge connects a sample to another sample immediately derived from it. We assume that the lineage is connected; each sample in the set is the source of or is derived from at least one other sample in the set. Given the structural change heuristics we describe below, it is sufficient if each sample shares a majority of its code with at least one other sample.

Most malware samples are executable binaries. Much of the potential evidence present in source code is obscured or lost during compilation to machine code. To assess structural change, we first extract features from the binary. Static binary analysis usually relies on a disassembly of the machine code into assembly language containing instructions for the particular processor architecture, which can be obtained with a commercial tool such as IDA Pro™. From the disassembly, static analysis techniques can derive a control flow graph for the program; identify the subroutines, their arguments and sometimes their data types, construct a call graph between subroutines, identify string data in the binary's data section, and detect other structural features.

We use these features to derive lineage evidence. Ideal features have three characteristics:

- Present in most, if not all samples of malware

- Difficult to fake or obfuscate
- Increases the likelihood that one sample is an ancestor of another

For example, different compiler versions or the use of x86 instructions from newer instruction sets satisfies characteristics 2 and 3, but these examples do not satisfy 1 due to their scarcity of appearance. Similarly, timestamps in the binary satisfy 1 and 3, but can be easily faked and thus do not satisfy characteristic 2.

We compute three types of evidence from structural features: functional point complexity, similarity of string data, code subsets. Each type of evidence provides a likelihood that some samples are ancestors of other samples.

We insert ancestry evidence into a truth maintenance system (TMS) built on top of the Jess rule engine. We propagate the evidence in the TMS to determine the most likely of three relationships for each pair of samples: ancestor, descendant, or unrelated. We then extract from the final Jess state a lineage graph with the maximum likelihood.

Extracting Lineage Evidence

Functional Point Complexity

One of the observations in the so-called “Lehman’s Laws of Software Evolution” [4] is that unless refactored, software will increase in complexity over time. We compute the complexity of a malware sample using a Functional Point Complexity metric adapted from [7]. The metric normally applies to source code, but we adapted the metric to analogous features in binary code.

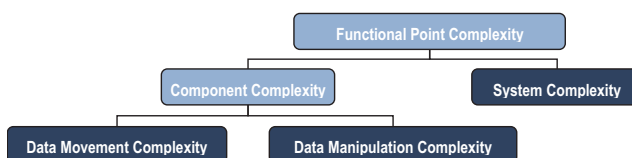


Figure 1. Factors contributing to the functional point complexity metric.

The Functional Point Complexity metric (Figure 1) is a linear combination of three sub metrics: data movement, data manipulation, and overall system complexity. The data movement complexity is defined as the count of reads, writes, and function entries or exits. Data manipulation complexity is the count of conditionals and loops in the code base. System complexity is calculated using the cyclomatic complexity of the module dependency graph.

We adapted these source code characteristics to related features extracted by static analysis. Data movement attributes map to data access instructions and control flow jumps, such as *call*, *ret*, and unconditional jumps in x86 assembly. Data manipulation attributes map to conditional instructions. Lastly, we compute cyclomatic complexity over the subroutine call graph instead of the module dependency graph.

Once the complexity is calculated for each sample, all are ordered by increasing complexity. Next, we insert an evidence fact into Jess for each ancestor/descendant pair in the linear ordering, where the more complex sample is marked as the descendant and the less complex as its ancestor. Except for the case where two samples have equal complexity scores, each fact includes the same likelihood that greater complexity implies descent. This likelihood was empirically estimated. In cases where the complexity values of the samples are equal, the likelihood is set so that neither direction of descent is more likely.

Similarity of String Data

Our next type of evidence draws on the intuition that if samples belong to a totally ordered lineage (i.e., there are no branches of descent), then similar samples ought to be closer to one another. We compute similarity of samples by computing the Jaccard distance [6] between sets of moderately-sized strings found in the data section of the binary. We compute the distance between each pair of samples and then attempt to place samples on a line such that their distances on the line reproduce as closely as possible the matrix of their pairwise distances (Figure 2). Once the samples are placed on the line, we have an estimate for a total ordering of the samples.

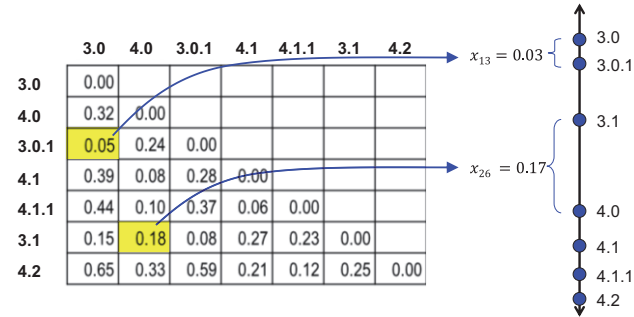


Figure 2. Using multi-dimensional scaling, versions of a program are placed on a line with distances closely approximating the distances in the similarity matrix.

Mathematically, this technique is known as multi-dimensional scaling (MDS). In the general case, MDS attempts to map n points in a high-dimensional space $\mathbb{R}^d$ (where d may be unknown) to $\mathbb{R}^m$ using only distances δ_{ij} between points in order to minimize the stress criterion:

$$\sigma = \sum_{i < j \leq n} (|x_i - x_j| - \delta_{ij})^2$$

In our application, $m = 1$.

The likelihood of ordering relationships for a particular sample depends on the stress of the linearization with and without the sample. When the stress of the MDS ordering is high, the sample violates the total ordering assumption, thus it is less likely that the MDS ordering reflects the actual ordering for that sample.

MDS allows us to place the samples in an ordering, but does not tell us which end of the order has the sample that came first. In order to discover the correct temporal direction for the ordering, we rely on combination with the functional complexity evidence.

Code and String Subset

Our next type of evidence examines code sharing across samples. Predecessor samples often have a subset (or a near subset) of the functionality of samples derived from them, since functionality tends to increase.

In other work, we have developed algorithms for matching subroutines (extracted with static analysis) across samples, even if the subroutines have been modified (due to code changes) or if the samples were compiled with different compilers or under different optimization settings. Using these algorithms, we can form sets of subroutines and identify equivalent elements in those sets.

Sets of subroutines from each pair of samples can be compared by taking a set difference. The set difference between sets A and B, denoted $A \setminus B$, is the set of elements in A not appearing in B. When there are relatively few elements in A that are not in B, it is said that A is a near subset of B, i.e., given a threshold ϵ ,

$$\frac{|A \setminus B|}{|A \cup B|} < \epsilon.$$

When A is a near subset of B, but B is also a near subset of A, this is very weak evidence that A might be an ancestor of B (it might be vice versa). However, when A is a near subset of B, but $|B \setminus A|/|A \cup B|$ is large, this is good evidence that A might be an ancestor of B. We assert the fact with a likelihood that is a decreasing function of $|A \setminus B|/|B \setminus A|$.

Evidential Reasoning and Lineage Construction

Jess Engine and Truth Maintenance

Jess is a rule production system derived from CLIPS. Upon matching an asserted fact with the head of an

inference rule (using the Rete algorithm), it asserts the clause in the fact store, repeating until no new facts match inference rules.

We implemented a probabilistic truth maintenance system (TMS) in Jess by adding parameters for truth valence and belief (as a probability) to facts, as well as likelihoods for supporting evidence statements. We also added production rules responsible for updating beliefs based on likelihoods associated with supporting evidence.

TMS functionality is important for doing the probabilistic reasoning and evidence combination needed for the lineage problem. The semantics of a rule engine (such as Jess) are that anything in working memory is true. A logic-based truth maintenance system (LTMS), however, explicitly represents truth as a label associated with each working memory proposition [5]. Propositions may be simple declarative statements or logical relationships that hold among particular propositions. The most common relationship is implication, of the form PropA implies PropB. Propositions may enter the database as immutable assertions, retractable assumptions, or as the consequents of logical statements. The core mechanism of an LTMS is belief propagation, which determines the ramifications of a retraction. On a retraction, the LTMS updates the truth values of all propositions, accounting for any alternative logical support that may exist. The LTMS manages a logical constraint network, through which it propagates truth values when propositions are assumed or retracted.

Fact Representation and Evidence Propagation

Our Jess-based TMS uses three types of predicates. Ordering relationship *conclusions* are the basic lineage facts from which we will construct a lineage graph. *Supporting evidence* predicates represent the different types of structural change evidence described earlier. *Evidence-for* predicates bridge supporting evidence and conclusions in a manner that allows us to update probabilities of belief for conclusions.

Ordering relationship *conclusions* are represented using these three-argument predicates:

(*descendent-of* <spec-1> <spec-2> <truth-label>) means that the fact that specimen-1 is a descendent of specimen-2 has a given truth value. When <truth-label> = :true, this relation places the two specimens into a lineage and specifies relative temporal ordering. One can extract a given lineage from working memory by accumulating all descendent-of expressions.

(*child-of* <spec-1> <spec-2> <truth-label>) means that the fact that specimen-1 is a direct descendant of specimen-2 has a given truth value. This fact helps us extract the lineage graph directly from the fact store.

(*unrelated* <spec-1> <spec-2> <truth-label>) means that specimen-1 and specimen-2 are not ordered with respect to one another in the lineage. This occurs, for example, when two samples are siblings in different branches of a lineage.

In addition to the truth-label, each conclusion predicate has a *belief*, represented as a probability. Each conclusion predicate is associated with an *evidence-for* predicate that includes a likelihood. For example,

(*evidence-for* *descendent-of* <spec-1> <spec-2> <truth-label> <likelihood>)

is such a statement. When a supporting evidence statement is asserted, a rule fires that asserts an evidence-for statement with some likelihood. These evidence-for statements cause rules to fire that update the probability of conclusions based on their likelihoods.

The inference algorithm has three stages: Rule Propagation, Probability Updating, and Truth Maintenance. Rule Propagation asserts possible evidentiary statements that might support one of the three types of conclusions. Probability Updating uses the likelihoods associated with the evidence to update the probabilities of the conclusions. Finally, Truth Maintenance propagates the truth values of conclusion statements to generate additional conclusions. The algorithm cycles through these stages until likelihoods converge (within a pre-set threshold).

Stage 1—Rule Propagation

Rule propagation fires a set of evidence-marshaling rules that add propositions concerning evidence for or against a given lineage inference into working memory. The structural change evidence described earlier causes *evidence-for* statements to be generated with appropriate likelihoods.

Stage 2—Probability Updating

The first stage of inference will cease once all rules have fired. Rules fire in arbitrary order, and while they do so the state of working memory is in constant flux. Once no more evidence rules fire, an additional set of rules acts to update the likelihood values.

Let P be the current probability estimate of a conclusion, and λ the likelihood of an *evidence-for* statement. Then updates to P will be in the form of a likelihood ratio:

$$P' = (P \times \lambda) / [(P \times \lambda) + (1 - P)]$$

Likelihood ranges from zero to infinity, although in practice a range of 1/1000 to 1000 is sufficient. Likelihood greater than 1 represents confirming evidence; less than 1 represents disconfirming evidence.

The purpose of likelihoods is to express a relative change in the current estimation of probability, based on new

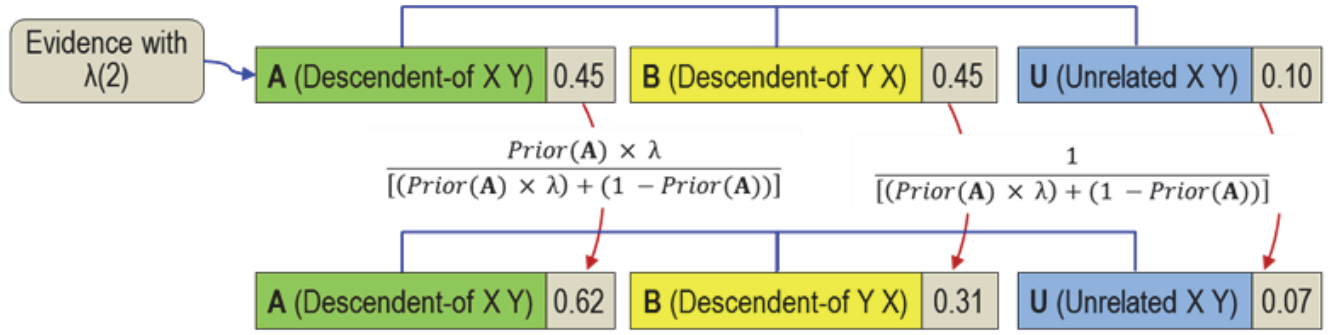


Figure 3. Example propagation of lineage evidence for likelihood=2.

evidence. Early medical diagnostic expert systems used absolute probabilities, so they had rules of the form *If symptom(cough) then $P(sick) = 0.6$* , which lead to illogical conclusions in corner cases (e.g., the system would reduce $P(sick)$ of a terminal cancer patient who suddenly exhibits a cough, from 0.99 to 0.6). Using likelihoods, the rule would be *If symptom(cough) then $\lambda(sick) = 5$* (where the lambda symbol λ indicates likelihood). Using the above formula, the patient's $P(sick)$ would increase from 0.99 to 0.9997, a more sensible result.

The second stage of inference, Probability Updating, collects all evidence propositions that bear on a given conclusion and sequentially applies the likelihood factors of each statement to update the probability of the inference. If the evidence set has not changed from the prior cycle, then no update is necessary. If there is new evidence, then an incremental update is sufficient. However, if some previously believed evidence now lacks support, probability updating must start from scratch.

Initially, we assume that the prior probability of an unrelated specimen is low, around 0.1. Absent any other information, we assume that the probability of a given specimen descending from another is equal to its being the others ancestor, and therefore assign a prior of 0.45 to each such state.

Stage 3—Truth Maintenance

The TMS updating process assumes that the most probable inference is true and then repeats the inference cycle until there is no change in the truth of any inference. If conclusions depended solely on direct evidence, then the inference algorithm would halt after one cycle, because Stage 1 would produce all the ramifications from the evidence, and there would be no possibility of a change in belief. Evidence statements never become false, but are assumed to be true for the duration of a given analysis. However, the inference algorithm will iterate for more than one cycle because it will build on inferences from prior cycles. Specifically, some of the rules will have one or more conclusions as one of their triggers. For example,

child-of statements can be inferred directly from multiple *descendant-of* statements.

Truth Maintenance checks the probability of each statement in the evidence sets for each conclusion. If there has been no change in the most probable statement, then no update to truth values is necessary. If, however, the most probable statement has changed, then the system must correctly update the truth labels of all affected inferences.

The Truth Maintenance step's most important function is that it allows us to use ancestor and descendant relationships to infer the direct parent-child relationships belonging to the final lineage graph.

Results

We evaluated our lineage extraction approach on 13 sets of malware variants, comprising a total of 120 distinct samples. Ground truth lineage information for evaluation purposes is extremely difficult to obtain from malware collected from the wild. The Cyber Genome evaluation team provided the samples, for which they knew precise lineage ground truth. Basic obfuscations like timestamp modification and executable packers were used to increase the difficulty of some tests.

For the packed samples, our analysis system ran them through an unpacker in order to generate an unpacked binary prior to lineage analysis. Lineages generated from unpacked samples were generally comparable with results from non-packed samples. When the unpacker could not handle a specific type of packer (e.g., VMProtect), then the resulting static analysis and lineage were much worse.

We report our results on the 13 lineage cases in Table 1. Test cases with “Y” in the “Tot Ord” column have ground truth lineages that are totally ordered (lineages do not branch into separate lines of evolution). When “Tot Ord” is “N”, the ground truth lineages are partially ordered, including branches and merges in functionality. We provide measures of performance for our recovery of two different relationships: “PC” for parent-child relationships, and “AD” for ancestor-descendant relationships. The

suffix “-A” represents our accuracy, the percentage of relationships we identified that were present in the ground truth lineage. The suffix “-C” represents our coverage, the percentage of the relationships in the ground truth that we successfully recovered.

ID	size	Tot Ord	AD-A	AD-C	PC-A	PC-C
S1	10	Y	0.93	0.93	0.78	0.78
S1a	10	Y	1	1	1	1
S2	8	Y	1	1	1	1
S2a	17	Y	0.95	0.95	0.5	0.5
S3	12	N	0.82	0.98	0.7	0.7
S3a	23	N	0.65	0.87	0.57	0.55
S5	15	Y	1	0.52	1	0.93
S9	10	N	0.38	0.85	0.11	0.08
S10	10	Y	1	1	1	1
S12	4	N	0.5	0.6	0	0
S13	5	N	0.4	0.57	0	0
S14	4	N	0.17	0.25	0.33	0.33
S15	4	N	0.5	1	0.33	0.33

When working with totally ordered lineages, we added additional evidence rules to Jess to reflect total ordering constraints and performed significantly better. Under these circumstances, we correctly identify 77% of the direct parent-child relationships across test cases. Out of all ancestor/descendant pairings, we correctly identify 91%. Thus, even when we sometimes place a sample incorrectly, most of the ordering relationships with other samples are preserved, meaning that the placement is “close” to the correct position in the lineage.

In the case of partially ordered lineages, our performance was less successful. We are still working on finding the proper rules for propagating code subset evidence to reach correct conclusions about branching and merging lineage events. However, our results on partially ordered lineages show that, even when we report a total ordering on a partially ordered lineage, we tend to correctly order the individual lineage paths. Our current evidence propagation can still correctly identify ancestors and descendants within branches, but has trouble separating samples appearing in distinct branches.

Of the four participants in Cyber Genome lineage tests (DECODE, MalLET, MAAGI, and MATCH), our system (DECODE) was scored first on totally-ordered lineages and on packed lineages, and third on partially ordered lineages.

Conclusion

Lineage reconstruction is important for malware forensics, since analysts can more easily observe trends in malware families. We have presented a technique for reconstructing lineages of malware using truth maintenance system techniques to combine structural change evidence extracted via static binary analysis. Our solution showed a remarkable ability to correctly identify ancestor/descendant pairs and, in cases where ground truth lineages were totally ordered, to correct identify a large number of direct parent-child relationships.

We have also run our system on malware we collected from the wild. Where source code was available, we could independently establish an estimated ground truth lineage. The results on the wild malware were comparable with those from the Cyber Genome experiment, although we noticed that wild malware appears to exhibit a denser “web” of code-sharing relationships than the straightforward tree models in the Cyber Genome sets.

We are investigating extensions to the way we use structural evidence that may help mitigate weaknesses in our approach. One current handicap is recognizing when a branch or merge event occurs in a lineage, a relatively common occurrence in the wild. Our work also needs to be extended to handle cases where two related samples are compiled under different compilers or employ obfuscations that alter complexity estimates.

This work was supported by DARPA through contract FA8750-10-C-0173. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

- [1] Dumitras, T., Neamtiu, I. “Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware,” *Cyber Security Experimentation and Test*, 2011.
- [2] Gupta, A., Kuppili, P., Akella, A., Barford, P., “An Empirical Study of Malware Evolution,” *Conference on Communication Systems and Networks*, 2009.
- [3] ISO/IEC 14764:2006, 2006.
- [4] Lehman, M. “Programs, Life Cycles, and Laws of Software Evolution,” *Proceedings, IEEE* 68, pp. 1060- 1076, 1980.
- [5] McAllester, D., “A three-valued truth maintenance system,” S.B. thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1978. also McAllester, D., “Truth Maintenance,” in *Proceedings of AAAI-90*, 1990, 1109-1116.
- [6] Tan, P.-N., Steinbach, M., and Kumar, V. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2006.
- [7] Tran-Cao, D., Levesque, G., Abran, A. “Measuring software functional size: towards an effective measurement of complexity,” *Proceedings, International Conference on Software Maintenance 2002*, pp. 370- 376, 2002.