Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

An Evolutionary Trace Algorithm for Constructing Malware Lineages

Alex Heinricher and Steven Jilcott
Raytheon BBN Technologies
{aheinric, sjilcott}@bbn.com

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. We present new work extending our results on the malware lineage problem originally presented at FLAIRS 2013. We provide a new algorithm for reconstructing malware lineages with and without branch and merge events. This algorithm incorporates two innovations – the evaluation of candidate evolutionary traces based on candidate sets of feature accretion events and a machine-learning inspired approach to reducing overexplanation in the final lineage. The evolutionary trace algorithm is evaluated on several small families of malware whose ground truth lineage is known.

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, speculate about the origins of malware, and ultimately, attribute the malware to a specific actor. 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.

A *malware 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. Ideally, malware sample A is ordered before malware sample B just in the case where A was actually compiled before sample B. The *malware lineage reconstruction problem* is to construct such a partial ordering from an input collection of samples based on "features" extracted from the malware binaries.

The collection of samples provided as input is assumed to be sufficiently closely related so that the notion that one sample is derived from another makes sense; thus, the lineage graph derived from the input samples is connected. Such a notion can be made precise by defining a minimum percentage of shared code between any pair of samples in the input; in our research, we assumed that the input binary samples shared at least 70% of their original source code.

The quality of any solution to the malware lineage problem depends on the "features" extracted from the malware binaries. All our binaries were x86 Windows PE executables. We extracted two kinds of features from binaries. First, all external function calls are features (calls both to the Windows operating system and to any external libraries). If two binaries both make an external call, they share that feature. Second, using standard static program analysis techniques, we identified all subroutines internal to a binary. Each subroutine is a feature. If two samples share a variant of a subroutine, they both share that feature. Detecting whether two subroutines are variants is itself a challenging research problem; the outlines of our technique can be found in (Jilcott, 2015).

In a FLAIRS 2013 paper (Darmetko, Jilcott, and Everett 1997), Darmetko et al presented a probabilistic truth maintenance system approach for producing a malware lineage. The algorithms for assessing evidence, however, could only produce lineages which were a total ordering of the samples – a straight-line lineage. The technique produced good results only when the ground truth lineage for the input samples had no branch or merge events. As a continuation of this research, we present a new algorithm, the *evolutionary trace algorithm*, which produces high-quality lineages with branching and merging events.

The approach in Darmetko et al used, as part of its evidence, a feature accretion model – the notion that software generally increases in complexity over time, so that sam-

ples containing a feature are more likely to be later than samples that do not contain it. The evolutionary trace algorithm relies wholly on this model, but incorporates two innovations. First, rather than a coarse summarization of all the feature evidence into evidence for a single parent-child relationship, the algorithm assesses feature evidence more finely. The algorithm constructs candidate evolutionary traces for many different feature subsets and then compares the quality of these traces for inclusion in the lineage graph. This collection of traces allows us to reconstruct branch and merge events. Second, inspired by a technique for learning the structure of Bayesian Network models, the algorithm minimizes overexplanation by pruning excess parent-child relationships. This pruning helps ensure that remaining branch and merge events are more likely to match the ground truth.

We present the results of using the evolutionary trace algorithm to generate lineages for a collection of nine malware families. These families, and their precise ground truth lineage, were provided to us by another organization for use in the DARPA Cyber Genome program.

Related Work

A number of researchers have studied collections of both goodware and malware variants to derive models of software evolution (Godfrey and Tu 2000, Karim, Walenstein, and Lakhotia 2005, Khoo and Lio 2011, Lindorfer et al 2012, Xie, Chen, and Neamtiu 2012). Such models can inform approaches to the lineage reconstruction problem; our work is an algorithm for actually computing a reconstruction.

(Gupta et al, 2009) used metadata (such as time of collection and analyst notations) compiled by McAfee in a knowledge-based approach to lineage reconstruction. (Dumitras and Neamtiu, 2011) proposed tracking individual 'traits' (e.g., small code subsets) across samples, and training a classifier to properly identify lineage relationships.

The work most directly related to ours is (Jang, Woo, and Brumley, 2013), by colleagues on the DARPA Cyber Genome Program. Jang et al makes a couple of significant contributions. First, Jang et al describe a collection of different metrics which can be computed on a lineage graph to measure its overall quality versus the ground truth. Second, Jang et al provide the first algorithm that constructs lineage graphs (containing branch and merge events). Jang et al present separate algorithms for constructing straight-line and directed acyclic graph lineages; our algorithm can reconstruct either kind of lineage depending on the best approximation to the ground truth. Furthermore, Jang et al leverage only the symmetric distance between feature sets, relying on other heuristics to introduce directionality in the lineage. Our approach uses traces constructed from finer-grained assessment of the presence of individual features in samples. This approach can better solve the problem of root sample identification, and can compensate for some reversion or refactoring events that might otherwise reduce overall code complexity.

The Evolutionary Trace Algorithm

Algorithm Overview

The Evolutionary Trace Algorithm (ETA) builds a partial parent-child ordering – a lineage graph – from evidence for ancestor-descendant orderings. Pseudocode for the main body of the algorithm is shown in Figure 1.

createLineage():

1) Initialize directed graph G with a single node containing all samples.
2) For each node *n* in G containing more than one sample:
    a. Build the set of all possible traces T out of the features contained in the programs of *n*.
    b. Find *t* in T with the highest result for scoreTrace(*t*, *n*).
    c. If scoreTrace(*t*, *n*) > threshold:
        i. Let *r* be the root node of G.
        ii. Execute applyDivisions(*f*, *r*, G) for every feature *f* in *t*.
3) If at least one trace was applied in step 2, and there is at least one node in G containing more than one program, repeat step 2.
4) Execute pruneEdges(G).

Figure 1. Pseudocode for createLineage(), the ETA algorithm.

The algorithm progresses by iteratively refining the lineage graph from a single node labeled with all samples, to a final graph where each node is labeled uniquely by one sample, and the edges reflect parent-child relationships.

ETA reframes the ancestor-descendant ordering problem as a binary classification problem. If you consider each program in a lineage to be defined by a vector of features that are either present or absent in that binary, then a single feature will partition the set of programs into two: the programs that possess that feature, and the programs which do not. If the assumption of feature accretion holds, then the programs with the feature must descend in some way from the programs that lack it. A single feature can be used to split a node into two nodes; one containing all the supposed ancestor samples, and one containing all the supposed descendant samples.

It follows that it is possible to build a candidate partial ordering of ancestors and descendants by selecting a series

of features where each feature is present in at least one more program than the previous feature. Each adjacent pair of features fixes the location of at least one program within the ordering; those programs contain at least one feature that their ancestors lack, but lack at least one feature that their descendants possess. Each series of features thus induces one candidate *evolutionary trace*; each presents evidence for one ancestor-descendant path through the set of programs. We enumerate these traces in step 2a. Some series of features produce evolutionary traces that are better than others, in the sense that the trace best satisfies the feature accretion trend over all features, not just those in the series (step 2b, illustrated in Figure 2).

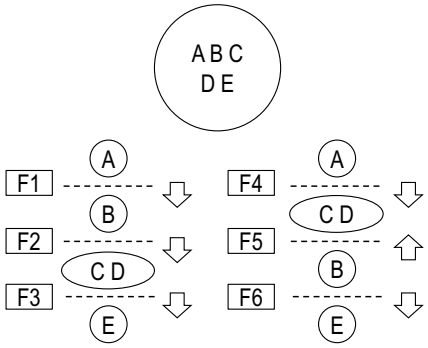

Figure 2. The series of features F1,F2,F3 induces a different evolutionary trace than F4,F5,F6. The left trace is superior because it does not contradict other feature evidence.

Scoring a trace is accomplished by evaluating the trace on two metrics:

- How well does the trace fit the overall data for the unpartitioned programs?
- How well does the trace partition those programs?

To score a trace, ETA examines the features in each adjacent pair of programs in the partial ordering defined by the trace. The score of each "split" is computed as

$$S(c,p) = \frac{sig(|F_c \backslash F_p| - |F_p \backslash F_c|) * (|F_c \backslash F_p| - |F_p \backslash F_c|)^2}{|F_p \cap F_c|}$$

where F_p is the set of features in the parent, F_c is the set of features in the child, and sig() is the signum function. This function was designed to reward splits where the features in the child are close to a proper superset of the features in the parent. The total score for the trace is computed as the sum of the individual split scores, thus rewarding more higher-quality splits.

By considering only the best single series of features, we can characterize lineages that do not branch or merge functionality; in such lineages, some set of features induces an ancestor-descendant ordering that is the same as the parent-child ordering. In order to expand the algorithm to handle more complex partial orderings – in particular, directed-acyclic graphs – it is necessary to look at features outside the best series. These features fall into three categories:

Type A. A feature is present in exactly the same set of samples as some feature already included in the trace, so it is superfluous when describing the trace.

Type B. A feature contradicts the ordering defined by the candidate trace because the feature does not obey the feature accretion model. If one were to examine the ground truth, it would be present in a malware sample, but absent in something defined as a descendant. Type B features introduce "noise" into our lineage reconstruction algorithm.

Type C. A feature contradicts the ordering defined by the candidate trace because a branch or merge event has occurred and the feature fits an ordering defined by a parallel branch of development. Type C features are considered in step 2c. If there is such a parallel branch of development, then we attempt to find a second evolutionary trace that characterizes it and add that path to the lineage graph.

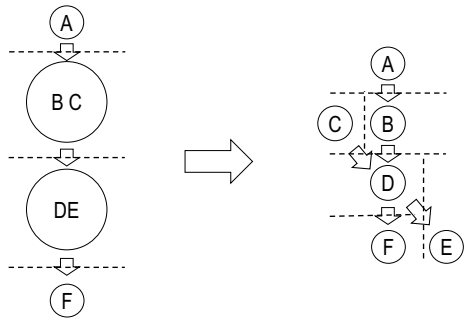

Figure 3. If one series of features does not recover a total ordering, ETA finds a second series of features that best orders the remaining samples, thus refining the lineage graph.

Figure 3 shows an example. B and C cannot be totally ordered because there is some feature f_B that doesn't appear in C and a feature f_C that doesn't appear in B. However, f_C appears in alternative feature series that induces a trace that orders samples C, D, and E.

ETA constructs a lineage graph by recovering and distinguishing the principal evolutionary traces from the feature data, ideally recovering those representing all true paths of development while ignoring (as much as possible) traces induced by Type B "noise" features.

Finally, for reasons that will become clear when describing applyDivisions(), splitting off new traces in this way introduces too many parent-child relationships. This approach ends up including all the parent-child relations possible given the ancestor-descendant data, but this overexplains the origins of a descendant. ETA borrows the notion of "graph thinning" from another partial-ordering reconstruction algorithm, the Bayes Network construction algorithm from (Cheng, Bell, and Liu 1997), to identify which relations are true parent-child relations, and which

are ancestor-descendant links already explained by parallel multi-edge paths.

Building and Merging Traces

ETA builds the set of evolutionary traces greedily. It starts with a single, unordered set of programs, and attempts to find the single trace that best characterizes the feature data. If the lineage is a straight-line lineage, then this series of features should completely partition that initial set of samples into a total ordering. If it does not, then one of two cases must hold: there were fewer features than there were samples (this is unlikely when using subroutines as features, but possible for extremely small programs), or there must exist Type B or C features that weren't used in the original trace.

In order to find unused Type C features and build a new trace from them, the algorithm repeatedly consideres each unpartitioned set of samples and attempts to find the best alternative candidate trace that characterizes that set. If all the unused features are Type A, then it won't find any traces that characterize the unpartitioned set more than it already has been. If a trace that partitions the set is found, the algorithm must differentiate the Type B features from the Type C features. This is accomplished by comparing the score of the best-fit trace to a threshold. If the best-scoring trace is good enough, then at least one of the features in it is a Type C feature, and we should partition the set by adding the trace to the lineage graph. If the best-scoring trace does not characterize the overall feature data for the programs, then all the unused features are Type B. We then conclude that this part of the data set is not well-described by a feature accretion model. We iterate until all nodes are partitioned, or no traces pass the threshold test.

Once the best-fit trace has been found for a particular unpartitioned node, the next step is to refine the graph by splitting nodes to introduce the new ancestor-descendant relationships (Figure 4).

applyDivisions(*f*, *n*, G):

1) Separate *n* into two nodes *a* and *d*, where *a* contains all of the programs that do not contain *f*, and *d* contains all of the programs that do.
2) If neither *a* nor *d* is empty, replace *n* with *d*, and add an edge from *a* to *d*.
3) For every child *c* of *n* in G:
   a. Compute (*a*', *d*') = applyDivisions(f, c, G)
   b. If *a*, *d*, *a*' and *d*' are all non-empty, add edges from *a* to *a*' and *d* to *d*'.
   c. Otherwise, fully connect the remaining non-empty nodes.
4) Return (*a*, *d*).

Figure 4. Pseudocode for introducing a new trace into the graph.

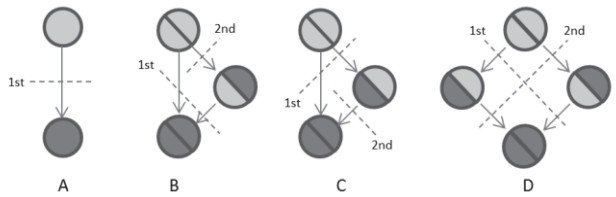

Figure 5. The results of applying two evolutionary traces to a set of programs.

Figure 5 shows the different possible results of applying two traces to a single node in the graph. Figure 5A shows the results of applying a single trace; one node is separated into one set of ancestors and one set of descendants. Figures 5B, 5C, and 5D show the possible results when a second trace is added. In 5B and 5C, only one of the two nodes was divided by the presence of a second feature; in 5B, only the ancestor set was partitioned, while in 5C only the descendant set was partitioned. Whenever one introduces a new path by applying a new trace in this way, it isn't possible to determine conclusively whether the three nodes form two parallel branches or a single linear path of evolution. It is conceivable that the two paths represent coincidental, convergent development paths. A similar pattern holds in figure 5D when both the ancestor and descendant sets are separated. As such, when a node is partitioned, all possible parent-child relations are preserved. This method introduces the potential for overexplaining the origins of a child sample.

pruneEdges(G):

For each node *n*:

For each node *a* child of *n*:

a) If rateSubset(F_n, F_a) < threshold, remove (*n*, *a*) from G.

b) Otherwise, consider every node *b* that is also a child of *n* in G.

i) If there is no path from *a* to *b* in G, or if rateSubset(F_a, F_b) < threshold, continue.

ii) Otherwise, let *test* = *b*, *next* = *a*, P = path(*a*, *b*), F = F_{test}

iii) Let F = F\F_{next}.

iv) If rateSubset(F_n, F) < threshold, remove (*n*, *b*) from G.

v) Otherwise, let *next* = child of next on P. If next is not null, go to step iii.

$$\text{rateSubset}(a,b) = \frac{|a \backslash b| - |b \backslash a|}{\max(|a \backslash b|, |b \backslash a|)}$$

Figure 6. Pseudocode for pruning overexplaining parent-child edges from the final lineage graph.

Reducing Overexplanation

Once the algorithm completes this process of applying traces, the remaining step is to prune the potential parent-

child relationships down to the set of probable ones. As stated above, this algorithm borrows directly from the third phase of the algorithm from (Cheng, Bell, and Liu 1997). The intuition behind (Cheng, Bell, and Liu 1997) is that an edge is only valid if it carries information that isn't already carried by a parallel path. The ETA applies this intuition to each node in the graph iteratively using the process in Figure 6.

The function rateSubset determines the relative strength of a subset relationship between two sets of features, with a negative score indicating that the relationship is actually a superset relationship. An edge from a node *n* to one of its children is invalidated given one of two conditions (Figure 7):

- The relation from *n* to its child isn't a subset relationship – it actually violates the feature accretion model.
- The relation from *n* to a particular child *b* is explained by the relation from *n* to another of its children *a*, which is also an ancestor of *b*.

If neither of these conditions hold, then there must be some features contributed from *n* to its child that cannot be explained by other relationships.

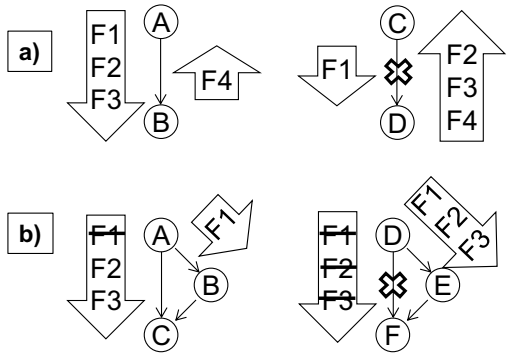

Figure 7. Two sources of overexplanation that the ETA algorithm can eliminate.

Results

We applied the ETA algorithm to nine malware families for which we had ground-truth lineage graphs provided by the cyber genome program. The ground truth lineages were a mix of straight-line and directed acyclic graph lineages.

For each of these lineages, we computed precision (1 – false positive rate) and recall (1 – false negative rate) for parent/child and ancestor/descendant relationships. Our precision metric for ancestor/descendant relationships is equivalent to the accuracy metric found in (Jang, Woo, and Brumley, 2013).

Label	Name	# Samples
S1	Cleanroom.B	10
S1.C2	Cleanroom.C	10
S2.1	BlasterWorm.A	8
S2.2	BlasterWorm.B	17
S3	KnightBot	11
S5.1	MiniPanzer.A	15
S5.2	MiniPanzer.B	15
S6	Cleanroom.A	7
S10	Kbot	10

Table 1. Malware lineages to which we applied ETA. The labels are used in results Table 2.

ETA is effective across-the-board at correctly finding ancestor-descendant relationships for both straight-line and complex lineages. ETA still has trouble with overexplanation, causing a drop in precision for parent-child edges on complex lineages.

For example, consider the lineage S3, a complex graph lineage shown in Figure 8.

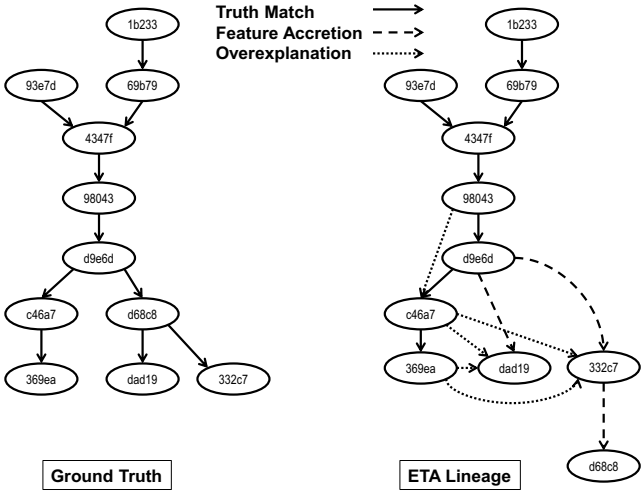

Figure 8. Comparison of Knightbot (S3) ground truth with our reconstructed lineage.

Encouragingly, the overall structure of the lineage matches up fairly well, as would be expected given the relatively high precision of ancestor-descendant relationship recovery. There are several edges predicted by the feature accretion model that do not fit the ground truth. In these cases we were able to verify that a refactoring or code deletion event occurred that foiled feature accretion assumptions.

	S1	S1.C2	S2.1	S2.2	S3	S5.1	S5.2	S6	S10	mean
parent/child relationships(precision)	67%	100%	67%	73%	47%	100%	100%	100%	100%	84%
parent/child relationships(recall)	67%	100%	57%	69%	70%	93%	93%	67%	87%	78%
ancestor/descendant relationships(precision)	100%	100%	100%	100%	92%	100%	100%	100%	100%	99%
ancestor/descendant relationships(recall)	85%	100%	88%	91%	97%	100%	100%	87%	99%	94%

Table 2. Results of the evolutionary trace algorithm on several malware lineages

The drop in parent-child precision is due to overexplaining relationships that the graph thinning phase of ETA was unable to invalidate. We have found that in most lineages we examine, these extraneous relationships tend to crop up in the later part of the lineage, whereas we obtain high accuracy earlier in the lineage. There are several hypotheses about why this occurs that we are still investigating:

- Longer evolutionary traces might capture lineage relationships with higher confidence, so features with a longer history of attestation give us more accuracy in the early lineage
- Samples later in a lineage represent more mature software; minor software changes introduce Type C features that might be more difficult to discriminate from Type B features
- Software is more likely to be refactored late in the lineage to accommodate new functionality, introducing more Type B features relative to Type C features

To improve the evolutionary trace algorithm, we will need to understand better the causes of late-lineage overexplanation and devise compensating mechanisms.

The evolutionary trace algorithm represents a promising approach to the malware lineage reconstruction problem that performs well on both straight-line and complex lineages, when the assumption of feature accretion holds for the input samples. Our scoring of evolutionary traces provides a good estimate for how strongly feature accretion appears to hold overall, and provides a good indicator for when an alternative model of software evolution might need to be substituted.

Acknowledgement

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). 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

Cheng, J., Bell, D., and Liu, W. "Learning belief networks from data: an information theory based approach", *Proceedings of the sixth international conference on Information and knowledge management (CIKM '97)*. ACM, New York, NY, USA, 325-331.

Darmetko, C., Jilcott, S., Everett, J. "Inferring accurate histories of malware evolution from structural evidence", *Twenty-Sixth International FLAIRS Conference (FLAIRS 2013)*.

Dumitras, T., Neamtiu, I. "Experimental Challenges in Cyber Security: A Story of Provenance and Lineage for Malware," *Cyber Security Experimentation and Test*, 2011.

Godfrey, M.W. and Tu, Q. "Evolution in open source software: A case study", *International Conference on Software Maintenance*, 2000.

Gupta, A., Kuppili, P., Akella, A., Barford, P. "An Empyrical Study of Malware Evolution," *Conference on Communication Systems and Networks*, 2009.

Jang, J., Woo, M., and Brumley, D. "Towards automatic software lineage inference", *Proceedings of the 22nd USENIX conference on Security (SEC'13)*, USENIX Association, Berkeley, CA, USA, 81-96.

Jilcott, S. "Scalable malware forensics using phylogenetic analysis", *2015 IEEE International Symposium on Homeland Security Technologies (IEEE HST 2015)*.

Karim, M.E., Walenstein, A., Lakhotia, A., and Parida, L. "Malware phylogeny generation using permutations of code", *Journal in Computer Virology*, 1:13–23, 2005.

Khoo, W.M. and Lio, P. "Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families", *SysSec Workshop*, 2011.

Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., and Zanero, S. "Lines of malicious code: insights into the malicious software industry", *2012 Annual Computer Security Applications Conference (ACSAC 2012)*.

Xie, G., Chen, J., and Neamtiu, I. "Towards a better understanding of software evolution: An empirical study on open source software", *IEEE International Conference on Software Maintenance*, 2009.