Towards Automatic Software Lineage Inference

Jiyong Jang, Maverick Woo, and David Brumley, *Carnegie Mellon University*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

Towards Automatic Software Lineage Inference

Jiyong Jang, Maverick Woo, and David Brumley
{jiyongj, pooh, dbrumley}@cmu.edu
Carnegie Mellon University

Abstract

Software lineage refers to the evolutionary relationship among a collection of software. The goal of software lineage inference is to recover the lineage given a set of program binaries. Software lineage can provide extremely useful information in many security scenarios such as malware triage and software vulnerability tracking.

In this paper, we systematically study software lineage inference by exploring four fundamental questions not addressed by prior work. First, how do we automatically infer software lineage from program binaries? Second, how do we measure the quality of lineage inference algorithms? Third, how useful are existing approaches to binary similarity analysis for inferring lineage in reality, and how about in an idealized setting? Fourth, what are the limitations that any software lineage inference algorithm must cope with?

Towards these goals we build iLINE, a system for automatic software lineage inference of program binaries, and also iEVAL, a system for scientific assessment of lineage quality. We evaluated iLINE on two types of lineage—straight line and directed acyclic graph—with large-scale real-world programs: 1,777 goodware spanning over a combined 110 years of development history and 114 malware with known lineage collected by the DARPA Cyber Genome program. We used iEVAL to study seven metrics to assess the diverse properties of lineage. Our results reveal that partial order mismatches and graph arc edit distance often yield the most meaningful comparisons in our experiments. Even without assuming any prior information about the data sets, iLINE proved to be effective in lineage inference—it achieves a mean accuracy of over 84% for goodware and over 72% for malware in our data sets.

1 Introduction

Software *evolves* to adapt to changing needs, bug fixes, and feature additions [28]. As such, software lineage—the evolutionary relationship among a set of software—can be a rich source of information for a number of security questions. Indeed, the literature is replete with analyses of known or manually recovered software lineages. For example, software engineering researchers have analyzed the histories of open source projects and the Linux kernel to understand software evolution [14, 45] as well as its effect on vulnerabilities in Firefox [33]. The security community has also studied malware evolution based upon the observation that the majority of newly detected malware are tweaked variants of well-known malware [2, 18, 20]. With over 1.1 million malware appearing daily [43], researchers have exploited such evolutionary relationships to identify new malware families [23, 31], create models of provenance and lineage [9], and generate phylogeny models based upon the notion of code similarity [22].

The wealth of existing research demonstrating the utility of software lineage immediately raises the question—"Can we infer software lineage *automatically*?" We foresee a large number of security-related applications once this becomes feasible. In forensics, lineage can help determine software provenance. For example, if we know that a closed-source program p_A is written by author X and another program p_B is derived from p_A, then we may deduce that the author of p_B is likely to be related to X. In malware triage [2, 18, 20], lineage can help malware analysts understand trends over time and make informed decisions about which malware to analyze first. This is particularly important since the order in which the variants of a malware are captured does not necessarily mirror its evolution. In software security, lineage can help track vulnerabilities in software of which we do not have source code. For example, if we know a vulnerability exists in an earlier version of an application, then it may also exist in applications that are derived from it. Such logic has been fruitfully applied at the source level in our previous work [19]. Indeed, these and related applications are important enough that the US Defense Advanced Research Projects Agency (DARPA) is funding a $43-million Cyber Genome program [6] to study them.

Having established that automatically and accurately infer software lineage is an important open problem, let us look at how to formalize it. Software lineage inference is the task of inferring a temporal ordering and ancestor/descendant relationships among programs. We model software lineage by a *lineage graph*:

Definition 1.1. A lineage graph $G = (N, A)$ is a directed acyclic graph (DAG) comprising a set of nodes N and a set of arcs A. A node $n \in N$ represents a program, and

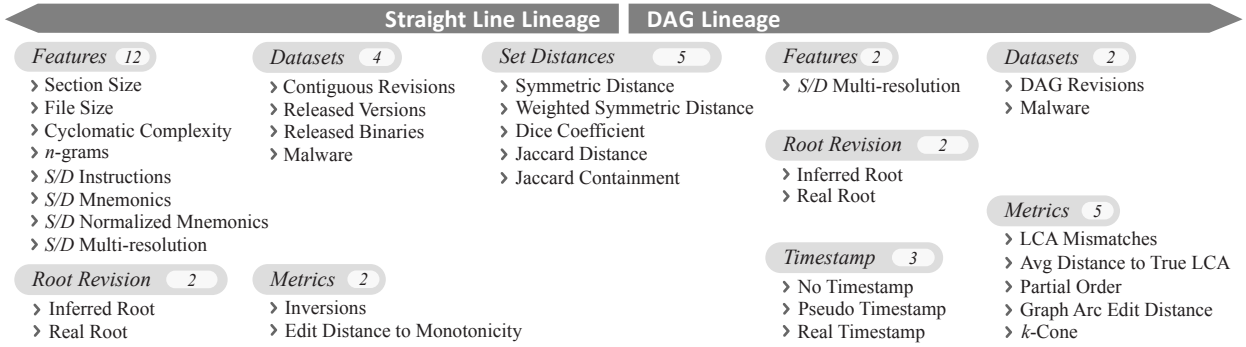


Figure 1: Design space in software lineage inference (*S/D* represents static/dynamic analysis-based features.)

an arc $(x, y) \in A$ denotes that program y is a derivative of program x. We say that x is a parent of y and y is a child of x.

A *root* is a node that has no incoming arc and a *leaf* is a node that has no outgoing arc. The set of *ancestors* of a node n is the set of nodes that can reach n. Note that n is an ancestor of itself. The set of *common ancestors* of x and y is the intersection of the two sets of ancestors. The set of *lowest common ancestors* (LCAs) of x and y is the set of common ancestors of x and y that are not ancestors of other common ancestors of x and y [4]. Notice that in a tree each pair of nodes must have a unique LCA, but in a DAG some pair of nodes can have multiple LCAs.

In this paper, we ask four basic research questions:

1. Can we automatically infer software lineage? Existing research focused on studying known software history and lineage [14, 33, 45], not creating lineage. Creating lineage is different from building a dendrogram based upon similarity [22, 23, 31]. A dendrogram can be used to identify families; however it does not provide any information about a temporal ordering, e.g., root identification.

In order to infer a temporal ordering and evolutionary relationships among programs, we develop new algorithms to automatically infer lineage of programs for two types of lineage: straight line lineage (§4.1) and directed acyclic graph (DAG) lineage (§4.2). In addition, we extend our approach for straight line lineage to k-straight line lineage (§4.1.4). We build ILINE to systematically evaluate the effectiveness of our lineage inference algorithms using twelve software feature sets (§2), five distance measures between feature sets (§3), two policies on the root identification (§4.1.1), and three policies on the use of timestamps (§4.2.2).

Without any prior information about data sets, for straight line linage, the mean accuracies of ILINE are 95.8% for goodware and 97.8% for malware. For DAG lineage, the mean accuracies are 84.0% for goodware and 72.0% for malware.

2. What are good metrics? Existing research focused on building a phylogenetic tree of malware [22, 23], but did not provide *quantitative* metrics to scientifically measure the quality of their output. Good metrics are necessary to quantify how good our approach is with respect to the ground truth. Good metrics also allow us to compare different approaches. To this end, we build IEVAL to assess our lineage inference algorithms using multiple metrics, each of which represents a different perspective of lineage.

IEVAL uses two metrics for straight line lineage (§5.1). Given an inferred lineage graph G and the ground truth G^*, the *number of inversions* measures how often we make a mistake when answering the question "which one of programs p_i and p_j comes first". The *edit distance to monotonicity* asks "how many nodes do we need to remove in G so that the remaining nodes are in the sorted order (and thus respect G^*)".

IEVAL also utilizes five metrics to measure the accuracy of DAG lineage (§5.2). An *LCA mismatch* is a generalized version of an inversion because the LCA of two nodes in a straight line is the earlier node. We also measure the *average pairwise distance between true LCA(s) and derived LCA(s) in* G^*. The *partial order mismatches* in a DAG asks the same question as inversions in a straight line. The *graph arc edit distance* for (labeled) graphs measures "what is the minimum number of arcs we need to delete from G and G^* to make both graphs equivalent". A *k-Cone mismatch* asks "how many nodes have the correct set of descendants counting up to depth k".

Among the above seven metrics, we recommend two metrics—partial order mismatches and graph arc edit distance. In §5.3, we discuss how the metrics are related, which metric is useful to measure which aspect of a lineage graph, which metric is efficient to compute, and which metric is deducible from other metrics.

3. How well are we doing now? We would like to understand the limits of our techniques even in *ideal* cases, meaning we have *(i)* control over variables affecting the compilation of programs, *(ii)* reliable feature extraction techniques to abstract program binaries accurately and precisely, and *(iii)* the ground truth with which we can compare our results to measure accuracy and to spot error cases. We discuss the effectiveness of different feature

sets and distance measures on lineage inference in §8.

We argue that it is necessary to also systematically validate a lineage inference technique with "goodware", e.g., open source projects. Since malware is often surreptitiously developed by adversaries, it is typically hard or even impossible to obtain the ground truth. More fundamentally, we simply cannot hope to understand the evolution of adversarial programs unless we first understand the limits of our approach in our idealized setting.

We systematically evaluated ILINE with both goodware and malware that we have the ground truth on: 1,777 goodware spanning over a combined 110 years of development history and 114 malware collected by the DARPA Cyber Genome program.

4. What are the limitations? We investigate error cases in G constructed by ILINE and highlight some of the difficult cases where ILINE failed to recover the correct evolutionary relationships. Since some of our experiments are conducted on goodware with access to source code, we are able to pinpoint challenging issues that must be addressed before we can improve the accuracy in software lineage inference. We discuss such challenging issues including reverting/refactoring, root identification, clustering, and feature extraction in §9. This is important because we may not be able to understand malware evolution without understanding limits of our approach with goodware.

2 Software Features for Lineage

In this study, we use three program analysis methods: syntax-based analysis, static analysis, and dynamic analysis. Given a set of program binaries $\mathcal{P}$, various features f_i are extracted from each $p_i \in \mathcal{P}$ to evaluate different abstractions of program binaries. Source code or metadata such as comments, commit messages or debugging information is not used as we are interested in results in security scenarios where source code is typically not available, e.g., forensics, proprietary software, and malware.

2.1 Using Previous Observations

Previous work analyzed software release histories to understand a software evolution process. It has been often observed that program size and complexity tend to increase as new revisions are released [14, 28, 45]. This observation also carries over to security scenarios, e.g., the complexity of malware is likely to grow as new variants appear [8]. We measured code section size, file size, and code complexity to assess how useful these features are in inferring lineage of program binaries.

- **Section size:** ILINE first identifies executable sections in binary code, e.g., `.text` section, which contain executable program code, and calculates the size.
- **File size:** Besides the section size, ILINE also calculates the file size, including code and data.

```
8b5dd485db750783c42c5b5e5dc383c42c5b5e5de9adf8ffff
```

(a) Byte sequence of program code

```
8b5dd485   5dd485db   d485db75   85db7507   db750783
750783c4   0783c42c   83c42c5b   c42c5b5e   2c5b5e5d
5b5e5dc3   5e5dc383   5dc383c4   c383c42c   5b5e5de9
5e5de91d   5de9adf8   e9adf8ff   adf8ffff
```

(b) 4-grams

```
mov -0x2c(%ebp),%ebx;test %ebx,%ebx;jne 805e198
add $0x2c,%esp;pop %ebx;pop %esi;pop %ebp;ret
add $0x2c,%esp;pop %ebx;pop %esi;pop %ebp;jmp 805da50
```

(c) Disassembled instructions

```
mov mem,reg;test reg,reg;jne imm
add imm,reg;pop reg;pop reg;pop reg;ret
add imm,reg;pop reg;pop reg;pop reg;jmp imm
```

(d) Instructions mnemonics with operands type

```
mov mem,reg;test reg,reg;jcc imm
add imm,reg;pop reg;pop reg;pop reg;ret
add imm,reg;pop reg;pop reg;pop reg;jmp imm
```

(e) Normalized mnemonics with operands type

Figure 2: Example of feature extraction

- **Cyclomatic complexity:** Cyclomatic complexity [34] is a common metric that indicates code complexity by measuring the number of linearly independent paths. From the control flow graph (CFG) of a program, the complexity M is defined as $M = E - N + 2P$ where E is the number of edges, N is the number of nodes, and P is the number of connected components of the CFG.

2.2 Using Syntax-Based Feature

While syntax-based analysis may lack semantic understanding of a program, previous work has shown its effectiveness on classifying unpacked programs. Indeed, n-gram analysis is widely adopted in software similarity detection, e.g., [20, 22, 26, 40]. The benefit of syntax-based analysis is that it is fast because it does not require disassembling.

- **n-grams:** An n-gram is a consecutive subsequence of length n in a sequence. From the identified executable sections, ILINE extracts program code into a hexadecimal sequence. Then, n-grams are obtained by sliding a window of n bytes over it. For example, Figure 2b shows 4-grams extracted from Figure 2a.

2.3 Using Static Features

Existing work utilized semantically richer features by first disassembling a binary. After reconstructing a control flow graph for each function, each basic block can be considered as a feature [12]. In order to maximize the probability of identifying similar programs, previous work also normalized disassembly outputs by considering instruction mnemonics without operands [23, 46] or instruction mnemonics with only the types of each operand (such as memory, a register or an immediate value) [39].

In our experiments, we introduce an additional normalization step of normalizing the instruction mnemonics themselves. This was motivated by our observations when

we analyzed the error cases in the lineages constructed using the above techniques. Our results indicate that this normalization notably improves lineage inference quality.

We also evaluate binary abstraction methods in an idealized setting in which we can deploy reliable feature extraction techniques. The limitation with static analysis comes from the difficulty of getting precise disassembly outputs from program binaries [27, 30]. In order to exclude the errors introduced at the feature extraction step and focus on evaluating the performance of software lineage inference algorithms, we also leverage assembly generated using `gcc -S` (not source code itself) to obtain basic blocks more accurately. Note that we use this to simulate what the results would be with ideal disassembling, which is in line with our goal of understanding the limits of the selected approaches.

• **Basic blocks comprising *disassembly* instructions:** ILINE disassembles a binary and identifies its basic blocks. Each feature is a sequence of instructions in a basic block. For example, in Figure 2c, each line is a series of instructions in a basic block; and each line is considered as an individual feature. This feature set is semantically richer than *n*-grams.

• **Basic blocks comprising instruction *mnemonics*:** For each disassembled instruction, ILINE retains only its mnemonic and the types of its operands (immediate, register, and memory). For example, `add $0x2c, %esp` is transformed into `add imm, reg` in Figure 2d. By normalizing the operands, this feature set helps us mitigate errors from syntactical differences, e.g., changes in offsets and jump target addresses, and register renaming.

• **Basic blocks comprising *normalized* mnemonics:** ILINE also normalizes mnemonics. First, mnemonics for all conditional jumps, e.g., `je`, `jne` and `jg`, are normalized into `jcc` because the same branching condition can be represented by flipped conditional jumps. For example, program p_1 uses `cmp eax, 1; jz` $\texttt{addr}_1$ while program p_2 has `cmp eax, 1; jnz` $\texttt{addr}_2$. Second, ILINE removes the `nop` instruction.

2.4 Using Dynamic Features

Modern malware is often found in a packed binary format [15, 21, 32, 38] and it is often not easy to analyze such packed/obfuscated programs with static analysis tools. In order to mitigate such difficulties, dynamic analysis has been proposed to monitor program executions and changes made to a system at run time [1, 2, 13, 35]. The idea of dynamic analysis is to run a program to make it disclose its "behaviors". Dynamic analysis on malware is typically performed in controlled environments such as virtual machines and isolated networks to prevent infections spreading to other machines [37].

• **Instructions executed at run time:** For malware specifically, ILINE traces an execution using a binary instrumentation tool and collects a set of instruction traces. Similar to static features, ILINE also generates additional sets of features by normalizing operands and mnemonics.

2.5 Using Multi-Resolution Features

Besides considering each feature set individually, ILINE utilizes multiple feature sets to benefit from normalized and specific features. Specifically, ILINE first uses the most normalized feature set to detect similar programs and gradually employs less-normalized feature sets to distinguish highly similar programs. This ensures that less similar programs (e.g., major version changes) will be connected only after more similar programs (e.g., only changes of constant values) have been connected.

3 Distance Measures Between Feature Sets

To measure the distance between two programs p_1 and p_2, ILINE uses the symmetric difference between their feature sets, which captures both additions and deletions made between p_1 and p_2. Let f_1 and f_2 denote the two feature sets extracted from p_1 and p_2, respectively. The *symmetric distance* between f_1 and f_2 is defined to be

$$\text{SD}(f_1, f_2) = |f_1 \setminus f_2| + |f_2 \setminus f_1|, \tag{1}$$

which denotes the cardinality of the set of features that are in f_1 or f_2 but not both. The symmetric distance basically measures the number of unique features in p_1 and p_2.

Distance metrics other than symmetric distance may be used for lineage inference as well. For example, the *Dice coefficient distance* $\text{DC}(f_1, f_2) = 1 - \frac{2|f_1 \cap f_2|}{|f_1| + |f_2|}$, the *Jaccard distance* $\text{JD}(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}$, and the *Jaccard containment distance* $\text{JC}(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{\texttt{min}(|f_1|, |f_2|)}$ can all be used to calculate the dissimilarity between two sets.

Besides the above four distance measures, which are all symmetric, i.e., $\texttt{distance}(f_1, f_2) = \texttt{distance}(f_2, f_1)$, we have also evaluated an *asymmetric* distance measure to determine the direction of derivation between p_1 and p_2. We call it the *weighted symmetric distance*, denoted $\text{WSD}(f_1, f_2) = |f_1 \setminus f_2| \times \texttt{C}_{\texttt{del}} + |f_2 \setminus f_1| \times \texttt{C}_{\texttt{add}}$ where $\texttt{C}_{\texttt{del}}$ and $\texttt{C}_{\texttt{add}}$ denote the cost for each deletion and each addition, respectively. Note that $\text{WSD}(f_1, f_2) = \text{SD}(f_1, f_2)$ when $\texttt{C}_{\texttt{del}} = \texttt{C}_{\texttt{add}} = 1$.

Our hypothesis is that additions and deletions should have different costs in a software evolution process, and we should be able to infer the derivative direction between two programs more accurately using the weighted symmetric distance. For example, in many open source projects and malware, code size usually grows over time [8, 45]. In other words, addition of new code is preferred to deletion of existing code. Differentiating $\texttt{C}_{\texttt{del}}$ and $\texttt{C}_{\texttt{add}}$ can help us to decide a direction of derivation. In this paper, we set $\texttt{C}_{\texttt{del}} = 2$ and $\texttt{C}_{\texttt{add}} = 1$. (We leave it as

future work to investigate the effect of these values.) Suppose program p_i has feature set $f_i = \{m_1, m_2, m_3\}$, and program p_j contains feature set $f_j = \{m_1, m_2, m_4, m_5\}$. By introducing asymmetry, evolving from p_i to p_j has a distance of 4 (deletion of m_3 and addition of m_4 and m_5), while the opposite direction has a distance of 5 (deletion of m_4 and m_5 and addition of m_3). Since $p_i \rightarrow p_j$ has a smaller distance, we conclude that it is the more plausible scenario.

For the rest of our paper, we use SD as a representative distance metric when we explain our lineage inference algorithms. We evaluated the effectiveness of all five distance measures on inferring lineage using SD as a baseline (see §8). Regarding metric-based features, e.g., section size, we measure the distance between two samples as the difference of their metric values.

4 Software Lineage Inference

Our goal is to automatically infer software lineage of program binaries. We build ILINE to systematically explore the design space illustrated in Figure 1 to understand advantages and disadvantages of our algorithms for inferring software lineage. We applied our algorithms to two types of lineage: straight line lineage (§4.1) and directed acyclic graph (DAG) lineage (§4.2). In particular, this is motivated by the observation that there are two common development models: serial/mainline development and parallel development. In serial development, every developer makes a series of check-ins on a single branch; and this forms straight line lineage. In parallel development, several branches are created for different tasks and are merged when needed, which results in DAG lineage.

4.1 Straight Line Lineage

The first scenario that we have investigated is 1-straight line lineage, i.e., a program source tree that has no branching/merging history. This is a common development history for smaller programs. We have also extended our technique to handle multiple straight line lineages (§4.1.4).

Software lineage inference in this setting is a problem of determining a temporal ordering. Given N unlabeled revisions of program p, the goal is to output label "1" for the 1st revision, "2" for the 2nd revision, and so on. For example, if we are given 100 revisions of program p and we have no timestamp of the revisions (or 100 revisions are randomly permuted), we want to rearrange them in the correct order starting from the 1st revision p^1 to the 100th revision p^{100}.

4.1.1 Identifying the Root Revision

In order to identify the root/first revision that has no parent in lineage, we explore two different choices: *(i)* inferring the root/earliest revision, and *(ii)* using the real root revision from the ground truth.

ILINE picks the root revision based upon Lehman's observation [28]. The revision that has the minimum code complexity (the 2nd software evolution law) and the minimum size (the 6th software evolution law) is selected as the root revision. The hypothesis is that developers are likely to add more code to previous revisions rather than delete other developers' code, which can increase code complexity and/or code size. This is also reflected in security scenarios, e.g., malware authors are also likely to add more modules to make it look different to bypass anti-virus detection, which leads to high code complexity [8]. In addition, provenance information such as first seen date [10] and tool-chain components [36] can be leveraged to infer the root.

We also evaluate ILINE with the real root revision given from the ground truth in case the inferred root revision was not correct. By comparing the accuracy of the lineage with the real root revision to the accuracy of the lineage with the inferred root revision, we can assess the importance of identifying the correct root revision.

4.1.2 Inferring Order

From the selected root revision, ILINE greedily picks the closest revision in terms of the symmetric distance as the next revision. Suppose we have three contiguous revisions: p^1, p^2, and p^3. One hypothesis is $\text{SD}(p^1, p^2) < \text{SD}(p^1, p^3)$, i.e., the symmetric distance between two adjacent revisions would be smaller. This hypothesis follows logically from Lehman's software evolution laws.

There may be cases where the symmetric distance between two different pairs are the same, i.e., a tie. Suppose $\text{SD}(p^1, p^2) = \text{SD}(p^1, p^3)$. Then both p^2 and p^3 become candidates for the next revision of p^1. Using normalized features can cause more ties than using specific features because of the information loss.

ILINE utilizes more specific features in order to break ties more correctly (see §2.5). For example, if the symmetric distances using normalized mnemonics are the same, then the symmetric distances using instruction mnemonics are used to break ties. ILINE gradually reduces normalization strength to break ties.

4.1.3 Handling Outliers

As an optional step, ILINE handles outliers in our recovered ordering, if any. Since ILINE constructs lineage in a greedy way, if one revision is not selected mistakenly, the revision may not be selected until the very last round. To see this, suppose we have 5 revisions p^1, p^2, p^3, p^4, and p^5. If ILINE falsely selects p^3 as the next revision of p^1 ($p^1 \rightarrow p^3$) and $\text{SD}(p^3, p^4) < \text{SD}(p^3, p^2)$, then p^4 will be chosen as the next revision ($p^1 \rightarrow p^3 \rightarrow p^4$). It is likely that $\text{SD}(p^4, p^5) < \text{SD}(p^4, p^2)$ holds because

p^4 and p^5 are neighboring revisions, and then p^5 will be selected ($p^1 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$). The probability of selecting p^2 is getting lower and lower if we have more revisions. At last p^2 is added as the last revision ($p^1 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5 \rightarrow p^2$) and becomes an outlier.

In order to handle such outliers, ILINE monitors the symmetric distance between every adjacent pair in the constructed lineage G. Since the symmetric distance at an outlier is the accumulation of changes from multiple revisions, it would be much larger than the difference between two contiguous revisions. (See Figure 10 for a real life example.) ILINE detects outliers by detecting peaks among the symmetric distances between consecutive pairs by means of a user-configurable threshold.

Once an outlier r has been identified, ILINE eliminates it in two steps. First, ILINE locates the revision y that has the minimum distance with r. Then, ILINE places r immediately next to y, favoring the side with a gap that has a larger symmetric distance. In our example, suppose p^3 is the closest revision to p^2. ILINE will compare $\text{SD}(p^1, p^3)$ (*before*) with $\text{SD}(p^3, p^4)$ (*after*) and then insert p^2 into the bigger of the two gaps. Therefore, in the case when $\text{SD}(p^1, p^3)$ is larger than $\text{SD}(p^3, p^4)$, we will recover the correct lineage, i.e., $p^1 \rightarrow p^2 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$.

4.1.4 k-Straight Line Lineage

We consider k-straight line lineage where we have a mixed data set of k different programs instead of a single program, and each program has straight line lineage.

For k-straight line lineage, ILINE first performs clustering on a given data set $\mathcal{P}$ to group the same (similar) programs into the same cluster $P_k \subseteq \mathcal{P}$. Programs are similar if $D(p_i, p_j) \leqq t$ where $D(\cdot)$ means a distance measurement between two programs and t is a distance threshold to be considered as a group. After we isolate distinct program groups between each other, ILINE identifies the earliest revision p_k^1 and infers straight line lineage for each program group P_k using the straight line lineage method. We denote the r-th revision of the program k as p_k^r. One caveat with the use of clustering as a preprocessing step is that more precise clustering may require reliable "components" extraction from program binaries, which is out of our scope.

Given a collection of programs and revisions, previous work shows that clustering can effectively separate them [5, 18, 20, 46]. ILINE uses hierarchical clustering because the number of variants k is not determined in advance. Other clustering methods like k-means clustering require that k is set at the beginning. ILINE groups two programs if $\text{JD}(f_1, f_2) \leqq t$ where t is a distance threshold ($0 \leqq t \leqq 1$). In order to decide an appropriate distance threshold t, we explore entire range of t and find the value where the resulting number of clusters becomes stable (see Figure 7 for an example).

4.2 Directed Acyclic Graph Lineage

The second scenario we studied is directed acyclic graph (DAG) lineage. This generalizes straight line lineage to include branching and merging histories. Branching and merging are common in large scale software development because branches allow developers to modify and test code without affecting others.

In a lineage graph G, branching is represented by a node with more than one *outgoing* arcs, i.e., a revision with multiple children. Merging is denoted by a node with more than one *incoming* arcs, i.e., a revision with multiple parents.

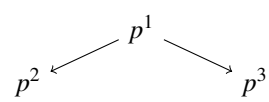
4.2.1 Identifying the Root Revision

In order to identify the root revision in lineage, we explore two different choices: *(i)* inferring the root/earliest revision and *(ii)* using the real root revision from the ground truth as discussed in §4.1.1.

4.2.2 Building Spanning Tree Lineage

ILINE builds (directed) spanning tree lineage by greedy selection. This step is similar to, but different from the ordering recovery step of the straight line lineage method. In order to recover an ordering, ILINE only allows the *last revision* in the recovered lineage G to have an outgoing arc so that the lineage graph becomes a straight line. For DAG lineage, however, ILINE allows *all revisions* in the recovered lineage G to have an outgoing arc so that a revision can have multiple children.

For example, given three revisions p^1, p^2, and p^3, if p^1 is selected as a root and $\text{SD}(p^1, p^2) < \text{SD}(p^1, p^3)$, then ILINE connects p^1 and p^2 ($p^1 \rightarrow p^2$). If $\text{SD}(p^1, p^3) < \text{SD}(p^2, p^3)$ holds, p^1 will have another child p^3 and a lineage graph looks like the following:



We evaluate three different policies on the use of a timestamp in DAG lineage: *no* timestamp, the *pseudo* timestamp from the recovered straight line lineage, and the *real* timestamp from the ground truth. Without a timestamp, the revision p^j to be added to G is determined by the minimum symmetric distance $\texttt{min}\{\text{SD}(p^i, p^j) : p^i \in \hat{N}, p^j \in \hat{N}^c\}$ where $\hat{N} \subseteq N$ represents a set of nodes already inserted into G and $\hat{N}^c$ denotes a complement of $\hat{N}$; and an arc (p^i, p^j) is added. However, with the use of a timestamp, the revision $p^j \in \hat{N}^c$ to be inserted is determined by the earliest timestamp and an arc is drawn based upon the minimum symmetric distance. In other words, we insert nodes in the order of timestamps.

4.2.3 Adding Non-Tree Arcs

While building (directed) spanning tree lineage, ILINE identifies *branching* points by allowing the revisions $p^i \in$

$\hat{N}$ to have more than one outgoing arcs—revisions with multiple children. In order to pinpoint *merging* points, iLINE adds non-tree arcs also known as cross arcs to spanning tree lineage.

For every non-root node p^i, iLINE identifies a unique feature set u^i that does not come from its parent p^j, i.e., $u^i = \{x : x \in f^i \text{ and } x \notin f^j\}$. Then iLINE identifies possible parents $p^k \in N$ as follows:

i) if real/pseudo timestamps are given, p^k with earlier timestamps than the timestamp of p^i
ii) for symmetric distance measures such as SD, DC, JD, and JC, non-ancestors p^k added to G before p^i
iii) for the asymmetric distance measure WSD, non-ancestors p^k satisfying $\text{WSD}(p^k, p^i) < \text{WSD}(p^i, p^k)$

become possible parents. Among the identified possible parents p^k, if u^i and f^k extracted from p^k have common features, then iLINE adds a non-tree arc from p^k to p^i. Consequently, p^i becomes a merging point of p^j and p^k and a lineage graph looks like the following:

$$\begin{tikzcd} p^j \arrow[dr] & & p^k \arrow[dl] \\ & p^i & \end{tikzcd}$$

After adding non-tree arcs, iLINE outputs DAG lineage showing both branching and merging.

5 Software Lineage Metrics

We build iEVAL to scientifically measure the quality of our constructed lineage with respect to the ground truth.

5.1 Straight Line Lineage

We use dates of commit histories and version numbers as the ground truth of ordering $G^* = (N, A^*)$, and compare the recovered ordering by iLINE $G = (N, A)$ with the ground truth to measure how close G is to G^*.

iEVAL measures the accuracy of the constructed lineage graph G using two metrics: *number of inversions* and *edit distance to monotonicity* (EDTM). An inversion happens if iLINE gives a wrong ordering for a chosen pair of revisions. The total number of inversions is the number of wrong ordering for all $\binom{|N|}{2}$ pairs. The EDTM is the minimum number of revisions that need to be removed to make the remaining nodes in the lineage graph G in the correct order. The longest increasing subsequence (LIS) can be computed in G, which is the longest (not necessarily contiguous) subsequence in the sorted order. Then the EDTM is calculated by $|N| - |LIS|$, which depicts how many nodes are out-of-place in G.

$p^1 \to p^3 \to p^2 \to p^4 \to p^5$ (a) Lineage 1

$p^1 \to p^3 \to p^4 \to p^5 \to p^2$ (b) Lineage 2

Figure 3: Inversions and edit distance to monotonicity

For example, we have 5 revisions of a program and iLINE outputs lineage 1 in Figure 3a and lineage 2 in Figure 3b. Lineage 1 has 1 inversion (a pair of $p^3 - p^2$) and 1 EDTM (delete p^2). Lineage 2 has 3 inversions ($p^3 - p^2$, $p^4 - p^2$, and $p^5 - p^2$) and 1 EDTM (delete p^2). As shown in both cases, the number of inversions can be different even when the EDTM is the same.

5.2 Directed Acyclic Graph Lineage

We evaluate the practical use of five metrics for measuring the accuracy of the constructed DAG lineage: *number of LCA mismatches*, *average pairwise distance to true LCA*, *partial order mismatches*, *graph arc edit distance*, and *k-Cone mismatches*.

$$\begin{tikzcd} p^1 \arrow[r] \arrow[dr] & p^2 \arrow[r] \arrow[dr] & p^4 \arrow[r] \arrow[dr] & p^6 \\ & p^3 \arrow[r] \arrow[ur] & p^5 & p^7 \end{tikzcd}$$

Figure 4: Lowest common ancestors

We define $\text{SLCA}(x, y)$ to be the set of LCAs of x and y because there can be multiple LCAs. For example, in Figure 4, $\text{SLCA}(p^4, p^5) = \{p^2, p^3\}$, while $\text{SLCA}(p^6, p^7) = \{p^4\}$. Given $\text{SLCA}(x, y)$ in G and the true $\text{SLCA}^*(x, y)$ in G^*, we can evaluate the correct LCA score of (x, y) $L(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ in the following four ways.

i) 1 point (correct) if $\text{SLCA}(x, y) = \text{SLCA}^*(x, y)$
ii) 1 point (correct) if $\text{SLCA}(x, y) \subseteq \text{SLCA}^*(x, y)$
iii) 1 point (correct) if $\text{SLCA}(x, y) \supseteq \text{SLCA}^*(x, y)$
iv) $1 - \text{JD}(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ point

Then the number of LCA mismatches is

$$|N \times N| - \sum_{(x,y) \in N \times N} L(\text{SLCA}(x, y), \text{SLCA}^*(x, y)).$$

The 1st policy is sound and complete, i.e., we only consider an exact match of SLCA. However, even small errors can lead to a large number of LCA mismatches. The 2nd policy is sound, i.e., every node in SLCA is indeed a true LCA (no false positive). Nonetheless, including any extra node will result in a mismatch. The 3rd policy is complete, i.e., SLCA must contain all true LCAs (no false negative). However, missing any true LCA will result in a mismatch. The 4th policy uses the Jaccard distance to measure dissimilarity between SLCA and SLCA^*. In our evaluation, iLINE followed the 4th policy since it allows us to attain a more fine-grained measure.

We also measure the distance between the true LCA(s) and reported LCA(s). For example, if iLINE falsely reports p^5 as an LCA of p^6 and p^7 in Figure 4, then the pairwise distance to the true LCA is 2 (=distance between p^4 and p^5). Formally, let $D(u, v)$ represent the distance between nodes u and v in the ground truth G^*. Given $\text{SLCA}(x, y)$ and $\text{SLCA}^*(x, y)$, we define the pairwise distance to true LCA $T(\text{SLCA}(x, y), \text{SLCA}^*(x, y))$ to be

SPECIAL CASE ← · · · · · · · · · · · · · · · → GENERAL CASE			Property measured
Straight Line	DAG		
Inversions	**PO**	SLCA	Order/Topology
EDTM		**GAED**	Out-of-place nodes/arcs
		k-Cone	Descendants within depth k

Table 1: Relationships among metrics

$$\sum_{(l,l^*)\in \text{SLCA}(x,y)\times \text{SLCA}^*(x,y)} \frac{D(l,l^*)}{|\text{SLCA}(x,y)\times \text{SLCA}^*(x,y)|}$$

and the average pairwise distance to true LCA to be

$$\sum_{(x,y)\in N\times N} \frac{T(\text{SLCA}(x,y),\text{SLCA}^*(x,y))}{|N\times N|}.$$

A partial order (PO) of x and y is to identify which one of x and y comes first: either x or y, or incomparable if they are not each other's ancestors. For example, in Figure 4, the PO of p^3 and p^7 is p^3, while the PO of p^6 and p^7 is incomparable. The total number of PO mismatches is the number of wrong ordering for all $\binom{|N|}{2}$ pairs.

A graph arc edit distance (GAED) measures how many arcs need to be deleted from G and G^* to make both G and G^* identical. For every node x, we calculate $E(x) = \text{SD}(\texttt{Adj}(x),\texttt{Adj}^*(x))$ where $\texttt{Adj}(x)$ and $\texttt{Adj}^*(x)$ denotes the adjacency list of x in G and G^* respectively. Then GAED becomes $\sum_{x\in N} E(x)$.

We define k-CONE(x) to be the set of descendants within depth k from node x. For example, in Figure 4, 2-CONE of p^1 is $\{p^2,p^3,p^4,p^5\}$. Then the given k-CONE(x) in G and the true k-CONE$^*(x)$ in G^*, we can evaluate the correct k-CONE score of x $R(k\text{-CONE}(x))$ using four different ways of set comparisons: an exact match, a subset match, a superset match, or the Jaccard index. In our evaluation, ILINE used the Jaccard index for a more fine-grained measure. Then the number of k-CONE mismatches is $|N| - \sum_{x\in N} R(k\text{-CONE}(x))$. With smaller k, we can measure the accuracy of nearest descendants.

5.3 Relationships among Metrics

Table 1 shows the relationships among different metrics and a property measured by each metric. A PO mismatch is a special case of an LCA mismatch because when x and y are in different branches, an LCA mismatch measures the accuracy of SLCA while a PO mismatch just says two nodes are incomparable. An inversion is also a special case of an LCA mismatch because querying the LCA of x and y in a straight line is the same as asking which one of x and y comes first. Essentially, a PO mismatch in a DAG is equal to an inversion in a straight line.

EDTM is a special case of GAED and an upper bound of GAED in a straight line is GAED $\leq$ EDTM$\times$6. One out-of-place node can cause up to six arcs errors. For example, $p^1 \to p^2 \to p^4 \to p^3 \to p^5$ has 1 EDTM (delete p^3 or p^4) and 6 GAED (delete $p^2 \to p^4$, $p^4 \to p^3$, and $p^3 \to p^5$ in G and $p^2 \to p^3$, $p^3 \to p^4$, and $p^4 \to p^5$ in G^*).

A k-Cone mismatch is a *local* metric to assess the correctness of nearest descendants of nodes while the other six metrics are *global* metrics to evaluate the correctness of the order of nodes and to count out-of-place nodes/arcs.

What are good metrics? Among the seven metrics, we recommend two metrics—partial order mismatches and graph arc edit distance. PO mismatches and GAED are both desirable because they evaluate different properties of lineage and are not deducible from each other.

To see this, observe that PO mismatches and SLCA mismatches measure the same property of lineage and have similar accuracy results in our evaluation. However, PO mismatches are more efficient to compute than SLCA mismatches; moreover, PO gives an answer for a more intuitive question, "which one of these two programs comes first". Thus, PO mismatches are preferred. Average distance to true LCA is supplementary to SLCA mismatches and so this metric is not necessary if we exclude SLCA mismatches. The number of inversions and edit distance to monotonicity can be respectively seen as special cases of PO mismatches and GAED in the case of straight line lineages. k-Cone mismatches can be extremely useful to an analyst during manual analysis, but it can be difficult to pick the right value of k automatically.

6 Implementation

ILINE is implemented using C (2.5 KLoC) and IDAPython plugin (100 LoC). We use the IDA Pro disassembler[1] to disassemble program binaries and to identify basic blocks. As discussed in §2.3, `gcc -S` output is used to compensate the errors introduced at the disassembling step. We utilize Cuckoo Sandbox[2] to monitor native functions, API calls and network activities of malware. On top of Cuckoo Sandbox, we use malwasm[3] with pintool[4], which allows us to obtain more fine-grained instruction level of traces. Since some kinds of malicious activities require "live" connections, we also employ INetSim[5] to simulate various network services, e.g., web,

[1] http://www.hex-rays.com/products/ida/index.shtml
[2] http://cuckoosandbox.org/
[3] http://code.google.com/p/malwasm/
[4] http://software.intel.com/en-us/articles/pintool
[5] http://www.inetsim.org/

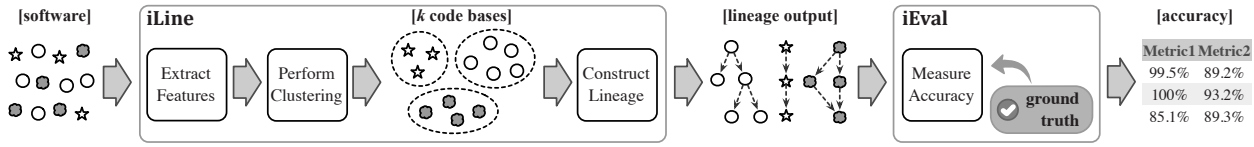


Figure 5: Software lineage inference overview

email, DNS, FTP, IRC, and so on. For example, BlasterWorm in our data set sent exploit packets and propagated itself via TFTP only when there were (simulated) live vulnerable hosts.

For the scalability reason, we use the feature hashing technique [20, 44] to encode extracted features into bit-vectors. For example, let bv_1 and bv_2 denote two bit-vectors generated from f_1 and f_2 using feature hashing. Then the symmetric distance in Equation 1 can be calculated by:

$$\text{SD}_{bv}(bv_1, bv_2) = S(bv_1 \otimes bv_2) \quad (2)$$

where $\otimes$ denotes bitwise-XOR and $S(\cdot)$ means the number of bits set to one.

7 Evaluation

As depicted in Figure 5, we systematically evaluated our lineage inference algorithms using *(i)* iLINE to explore all the design spaces described in Figure 1 with a variety of data sets and *(ii)* iEVAL to measure the accuracy of our outputs with respect to the ground truth.

7.1 Straight Line Lineage

7.1.1 Data sets

For straight line lineage experiments, we have collected three different kinds of goodware data sets, e.g., contiguous revisions, released versions, and actual release binaries, and malware data sets.

i) **Contiguous Revisions:** Using a commit history from a version control system, e.g., subversion and git, we downloaded contiguous revisions of a program. The time gap between two adjacent commits varies a lot, from <10 minutes to more than a month. We excluded some revisions that changed only comments because they did not affect the resulting program binaries.

Programs	# revisions	First rev	Last rev	Period
memcached	124	2008-10-14	2012-02-02	3.3 yr
redis	158	2011-09-29	2012-03-28	0.5 yr
redislite	89	2011-06-02	2012-01-18	0.6 yr

Table 2: Data sets of contiguous revisions

In order to set up idealized experiment environments, we compiled every revision with the same compiler and the same compiling options. We excluded variations that can come from the use of different compilers.

ii) **Released Versions:** We downloaded only released versions of a program meant to be distributed to end users. For example, subversion maintains them under the `tags` folder. The difference with contiguous revisions is that contiguous revisions may have program bugs (committed before testing) or experimental functionalities that would be excluded in released versions. In other words, released versions are more controlled data sets. We compiled source code with the same compiler and the same compiling options for ideal settings.

Programs	# releases	First release		Last release		Period
		Ver	Date	Ver	Date	
grep	19	2.0	1993-05-22	2.11	2012-03-02	18.8 yr
nano	114	0.7.4	2000-01-09	2.3.1	2011-05-10	11.3 yr
redis	48	1.0	2009-09-03	2.4.10	2012-03-30	2.6 yr
sendmail	38	8.10.0	2000-03-03	8.14.5	2011-05-15	11.2 yr
openssh	52	2.0.0	2000-05-02	5.9p1	2011-09-06	11.4 yr

Table 3: Data sets of released versions

iii) **Actual Release Binaries:** We collected binaries (not source code) of released versions from `rpm` or `deb` package files.

Programs	# files	First release		Last release		Period
		Ver	Date	Ver	Date	
grep	37	2.0-3	2009-08-02	2.11-3	2012-04-17	2.7 yr
nano	69	0.7.9-1	2000-01-24	2.2.6-1	2010-11-22	10.8 yr
redis	39	0.094-1	2009-05-06	2.4.9-1	2012-03-26	2.9 yr
sendmail	41	8.13.3-6	2005-03-12	8.14.4-2	2011-04-21	6.1 yr
openssh	75	3.9p1-2	2005-03-12	5.9p1-5	2012-04-02	7.1 yr
FileZilla	62	3.0.0	2007-09-13	3.5.3	2012-01-08	4.3 yr
p7zip	32	0.91	2004-08-21	9.20.1	2011-03-16	6.6 yr

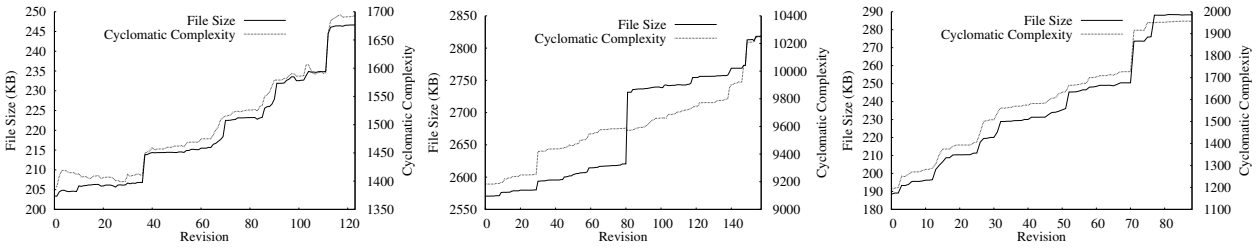
Table 4: Data sets of actual release binaries

The difference is that we did not have any control over the compiling process of the program, i.e., different programs may be compiled with different versions of compilers and/or optimization options. This data set is a representative of real-world scenarios where we do not have any information about development environments.

iv) **Malware:** We used 84 samples with known lineage collected by the Cyber Genome program. The data set includes bots, worms, and Trojan horses and contains 7 clusters.

Cluster	# samples	Family	Cluster	# samples	Family
MC1	10	KBot	MC5	10	CleanRoom.B
MC2	17	BlasterWorm	MC6	15	MiniPanzer.B
MC3	15	MiniPanzer.A	MC7	10	CleanRoom.C
MC4	7	CleanRoom.A			

Table 5: Data sets of malware



(a) memcached (b) redis (c) redislite

Figure 6: File size and complexity for contiguous revisions

7.1.2 Results

What selection of features provides the best lineage graph with respect to the ground truth? We evaluated different feature sets on diverse data sets.

i) **Contiguous Revisions:** In order to identify the first revision of each program, code complexity and code size of every revision were measured. As shown in Figure 6, both file size and cyclomatic complexity generally increased as new revisions were released. For these three data sets, the first revisions were correctly identified by selecting the revision that had the minimum file size and cyclomatic complexity.

A lineage for each program was constructed as described in §4.1. Although section/file size achieved high accuracies, e.g., 95.5%–99.5%, they are not reliable features because many ties can decrease/increase the accuracies depending on random guesses. *n*-grams over byte sequences generally achieved better accuracies; however, 2-grams (small size of *n*) were relatively unreliable features, e.g., 6.3% inversion error in `redis`. In our experiments, *n*=4 bytes worked reasonably well for these three data sets.

The use of disassembly instructions had up to 5% inversion error in `redislite`. Most errors came from syntactical differences, e.g., changes in offsets and jump target addresses. After normalizing operands, instruction mnemonics with operands types decreased the errors substantially, e.g., from 5% to 0.4%. With additional normalization, normalized instruction mnemonics with operands types achieved the same or better accuracies. Note that more normalized features can result in better or worse accuracies because there may be more ties where random guesses are involved.

In order to break ties, more specific features were used in multi-resolution features. For example, all 10 tie cases in `memcached` were correctly resolved by using more specific features. This demonstrated the effectiveness of using multi-resolution features for breaking ties.

ii) **Released Versions:**The first/root revisions were also correctly identified by selecting the revision that had the minimum code size. In some cases, simple feature sets, e.g., section/file size, could achieve higher accuracies than semantically rich feature sets (requiring more expensive process), e.g., instruction sequences. For example, ILINE with section size yielded 88.3% accuracy, while ILINE with instructions achieved 77.8% accuracy in `grep`. This, however, was improved to 100% with normalization. Like the experiments on contiguous revisions, 2-grams performed worse in the experiments on released versions, e.g., 18.9% accuracy in `sendmail`. Among various feature sets, multi-resolution features outperformed the other feature sets, e.g., 99.3%–100%.

iii) **Actual Release Binaries:** The first/root revisions for `nano` and `openssh` were correctly identified by selecting the revision that had the minimum code size. For the other five data sets, we performed the experiments both with the wrong inferred root and with the correct root given from the ground truth.

Overall accuracy of the constructed lineage was fairly high across all the data sets even though we did not control the variables of the compiling process, e.g., 83.3%–99.8% accuracy with the correct root. One possible explanation is that closer revisions (developed around the same time) might be compiled with the same version of compiler (available around the same time), which can make neighboring revisions look related to each other at the binary code level.

It was confirmed that lineage inference can be improved with the knowledge of the correct root. For example, ILINE picked a wrong revision as the first revision in `FileZilla`, which resulted in 51.6% accuracy; in contrast, the accuracy increased to 99.8% with the correct root revision.

iv) **Malware:** The first/root samples for all seven clusters were correctly identified by selecting the sample that had the minimum code size. Section size achieved high accuracies, e.g., 93.3–100%, which showed new variants were likely to add more code to previous malware. File size was not a good feature to infer a lineage of `MC2` because all samples in `MC2` had the same file size. The multi-resolution feature yielded 94.9–100% accuracy.

Dynamic instrumentations at the instruction level enabled us to catch minor updates between two adjacent variants. For example, subsequent BlasterWorm samples add more checks for virtual environments to hide its malicious activities if it is being monitored, e.g., examin-

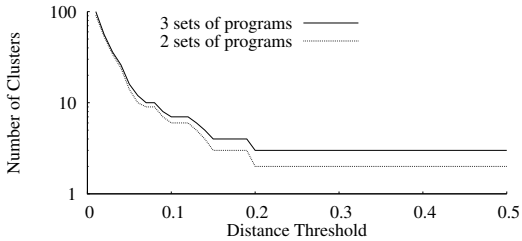


Figure 7: Clustering mixed data set of 2 and 3 programs

ing user names (sandbox, vmware, honey), running processes (VBoxService.exe, joeboxserver.exe), and current file names (C:\sample.exe). Dynamic feature sets yielded worse accuracy in `MC1`, `MC2`, `MC3`, `MC5`, and `MC6` while achieving the same accuracy in `MC4` and better accuracy in `MC7`. One main reason of the differences in accuracy is that dynamic analysis followed a specific execution path depending on the context. For example, in `MC2`, some variants exited immediately when they detected a VirtualBox service process, and produced limited execution traces.

v) ***k*-Straight Line Lineage:** We evaluated ILINE on mixed data sets including k different programs. For 2-straight line lineage, we mixed `memcached` and `redislite` in that both programs have the same functionality and similar code section sizes. Figure 7 shows the resulting number of clusters with various distance threshold values. From 0.2 to 0.5 distance threshold, the resulting number of clusters was 2. This means ILINE can first perform clustering to divide the data set into two groups, then build a straight line lineage for each group. The resulting number of clusters of the mixed data set of 3 programs including `memcached`, `redislite`, and `redis` became stabilized to 3 from 0.2 to 0.5 distance threshold, which means they were successfully clustered for the subsequent straight line lineage building process. We have also evaluated ILINE on three mixed malware data sets, each of which is a combination of different clusters in Table 5: {`MC2+MC5`}, {`MC4+MC6`}, and {`MC2+MC3+MC7`}. For each mixed data set, ILINE also clustered malware samples correctly for the subsequent straight line lineage inference. We discuss inferring lineage on incorrect clusters in §9.

7.2 Directed Acyclic Graph Lineage

7.2.1 Data sets

For DAG lineage experiments, we also evaluated ILINE on both goodware and malware.

i) **Goodware:** We have collected 10 data sets for directed acyclic graph lineage experiments from github[6]. We used github because we know *when* a project is forked from a *network graph* showing the development history as a graph including branching and merging.

[6]https://github.com/

We downloaded DAG revisions that had multiple times of branching and merging histories, and compiled with the same compilers and optimization options.

Programs	# revisions	First rev	Last rev	Period
`http-parser`	55	2010-11-05	2012-07-27	1.7 yr
`libgit2`	61	2012-06-25	2012-07-17	0.1 yr
`redis`	98	2010-04-29	2010-06-04	0.1 yr
`redislite`	97	2011-04-19	2011-06-12	0.1 yr
`shell-fm`	107	2008-10-01	2012-06-26	3.7 yr
`stud`	73	2011-06-09	2012-06-01	1.0 yr
`tig`	58	2006-06-06	2007-06-19	1.0 yr
`uzbl`	73	2011-08-07	2012-07-01	0.9 yr
`webdis`	96	2011-01-01	2012-07-20	1.6 yr
`yajl`	62	2010-07-21	2011-12-19	1.4 yr

Table 6: Goodware data sets for DAG lineage

ii) **Malware:** We used two malware families with known DAG lineage collected by the Cyber Genome program. They contain 30 samples in total.

Cluster	# samples	Family
`MC8`	21	WormBot
`MC9`	9	MinBot

Table 7: Malware data sets for DAG lineage

7.2.2 Results

We set two policies for DAG lineage experiments: the use of timestamp (none/pseudo/real) and the use of the real root (none/real). The real timestamp implies the real root so that we explored $3 \times 2 - 1 = 5$ different setups. We used multi-resolution feature sets for DAG lineage experiments because multi-resolution feature sets attained the best accuracy in constructing straight line lineage.

i) **Goodware:** Without having any prior knowledge, ILINE achieved 71.5%–94.1% PO accuracies. By using the real root revision, the accuracies increased to 71.5%–96.1%. For example, in case of `tig`, ILINE gained about 20% increase in the accuracy.
With pseudo timestamps, accuracies were worse even with the real root revisions for most of data sets, e.g., 64.0%–90.9% (see §8). By using the real timestamps, ILINE achieved higher accuracies of 84.1%–96.7%. This means that the recovered DAG lineages were very close to the true DAG lineages.

ii) **Malware:** ILINE achieved 68.6%–75.0% accuracies without any prior knowledge. Using the correct timestamps, the accuracies increased notably to 86.2%–91.7%. While we obtained the real timestamps from the ground truth in our experiments, we can also leverage first seen date of malware, e.g., Symantec's Worldwide Intelligence Network Environment [10].
With dynamic features, ILINE achieved 59.0%–75.0% accuracies without any prior knowledge, and 68.6%–80.6% accuracies with real timestamps, which is a bit lower than the accuracies based upon static features.

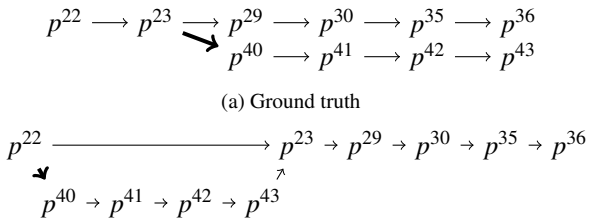
7.3 Performance

Given N binaries with their features already extracted, the complexity of constructing lineage is $O(N^2)$ due to the computation of the $\binom{|N|}{2}$ pairwise distances. To give concrete values, we measured the time to construct lineage with multi-resolution features, SD, and 32 KB of bit-vectors on a Linux 3.2.0 machine with a 3.40 GHz i7 CPU utilizing a single core. Depending on the size of the data sets, it took 0.002–1.431s for straight line lineage and 0.005–0.385s for DAG lineage with the help of feature hashing. On average, this translates to 146 samples/s and 180 samples/s for straight line lineage and DAG lineage, respectively. As a comparison, our BitShred malware clustering system [20], which represents the state of the art at the time of its publication in 2011, can process 257 samples per second using a single core on the same machine. Since the running times of malware clustering and lineage inference are both dominated by distance comparisons, and since ILINE needs to resolve ties using multi-resolution features whereas BitShred needs not, we conclude that our current implementation of ILINE is competitive in terms of performance.

8 Discussion & Findings

Features. File/section size features yielded 94.6–95.5% mean accuracy in straight line lineage on goodware. Such high accuracy supports Lehman's laws of software evolution, e.g., continuing growth. However, size is not a reliable feature to infer malware lineage where malware authors can obfuscate a feature, e.g., samples with the same file size in `MC2`. As simple syntactic features, 4/8/16-grams achieved 95.3–96.3% mean accuracy in straight line lineage on goodware, whereas 2-grams achieved only 82.4% mean accuracy. This is because 2-grams are not distinctive enough to differentiate between samples and cause too many ties. Basic blocks as semantic features achieved 94.0–95.6% mean accuracy in straight line lineage on goodware. This slightly lower accuracy when compared to n-grams was due to ties. Multi-resolution features performed best, e.g., it achieved 95.8–98.4% mean accuracy in straight line lineage on goodware. This is due to its use of both syntactic and semantic features.

Distance Metrics. Our evaluation indicates that our lineage inference algorithms perform similarly regardless of the distance metrics except for the Jaccard containment (JC) distance. JC turns out to be inappropriate for lineage inference because it cannot capture evolutionary changes effectively. Suppose there are three contiguous revisions p^1, p^2, and p^3; and p^2 adds 10 lines of code to p^1 and p^3 adds 10 lines of code to p^2. Then, $\text{JC}(p^1,p^2) = \text{JC}(p^1,p^3) = \text{JC}(p^2,p^3) = 0$ because one revision is a subset of another revision. Such ties result in low accuracy. For example, JC yielded 74.5% mean accuracy, whereas SD yielded 84.0% mean accuracy in DAG lineage on goodware.



(a) Ground truth

(b) Constructed lineage with the use of pseudo timestamps

Figure 8: Error caused by pseudo timestamps in `uzbl`

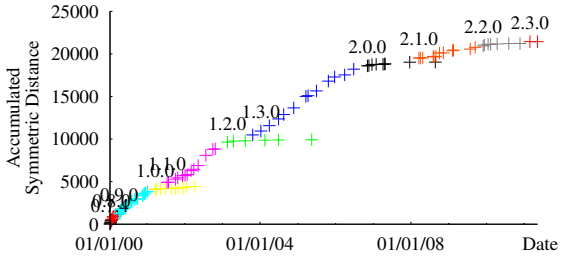


Figure 9: Development history of `nano`

Pseudo Timestamp. ILINE computes pseudo timestamps by first building a straight line lineage and then use the recovered ordering as timestamps. Since ILINE achieved fairly high accuracy in straight line lineage, at first we expected this approach to do well in DAG lineage. To our initial surprise, ILINE with pseudo timestamps actually performed worse. In retrospect, we observed that since each branch had been developed separately, it is challenging to determine the precise ordering between samples from different branches. For example, Figure 8 shows the partial ground truth and the constructed lineage by ILINE for `uzbl` with pseudo timestamps. Although ILINE *without* pseudo timestamps successfully recovered the ground truth lineage, the use of pseudo timestamps resulted in poor performance. The recovered ordering, i.e., pseudo timestamps were $p^{22}, p^{40}, p^{41}, p^{42}, p^{43}, p^{23}, p^{29}, p^{30}, p^{35}, p^{36}$. Due to the imprecise timestamps, the derivative relationships in the constructed lineage were not accurate.

Revision History vs. Release Date. Correct software lineage inference on a *revision history* may not correspond with software *release date* lineage. For example, Figure 9 shows the accumulated symmetric distance between two neighboring releases where a development branch of `nano-1.3` and a stable branch of `nano-1.2` are developed in parallel. ILINE infers software lineage consistent with a revision history.

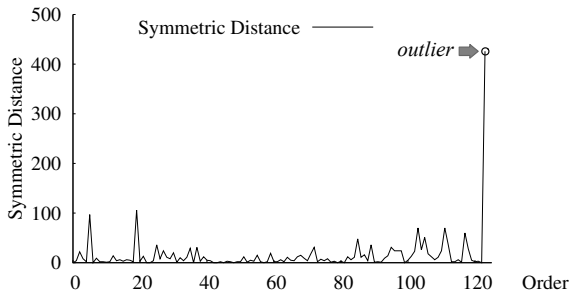


Figure 10: An outlier in `memcached`

Threats to Validity. Our malware experiments were performed on a relatively small data set because of difficulties in obtaining the ground truth. Although it is hard to indicate a representative of modern malware due to its surreptitious nature, we evaluated our methods on common malware categories such as bots, worms, and Trojan horses. To the best of our knowledge, we are the first to take a systematic approach towards software lineage inference to provide scientific evidence instead of speculative remarks.

9 Limitations

Reverting/Refactoring. Regression of code is a challenging problem in software lineage inference. A revision adding new functionalities is sometimes followed by stabilizing phases including bug fixes. Bug fixes might be done by reverting to the previous revision, i.e., undoing the modifications of the code.

Some revisions can become outliers because of ILINE's greedy construction and reverting/refactoring issues. In §4.1.3, we propose a technique to detect and process outliers by looking for peaks of the distance between two contiguous revisions. For example, ILINE had 70 inversions and 1 EDTM for the contiguous revisions of `memcached`. The error came from the 53rd revision that was incorrectly located at the end of the lineage. Figure 10 shows the symmetric distance between two adjacent revisions in the recovered lineage before we process outliers. The outlier caused an exceptional peak of the symmetric distance at the rightmost of the Figure 10. ILINE identified such possible outliers by looking for peaks, then generated the perfect lineage of `memcached` after handling the outlier.

There can also be false positives among detected outliers, i.e., a peak is identified even revisions are in the correct order. For example, a peak can be identified between two contiguous revisions when there is a huge update like major version changes. However, such false positives do not affect overall accuracy of ILINE because the original (correct) position will be chosen again when minimizing the overall distance.

Although our technique improves lineage inference, it may not be able to resolve every case. Unless we design a precise model describing the developers' reverting/refactoring activity, *no* reasonable algorithm may be able to recover the same lineage as the ground truth. Rather, the constructed lineage can be considered as a more practical/pragmatic representation of the truth.

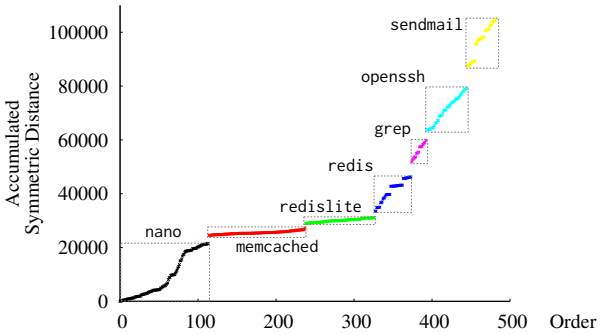


Figure 11: Recovered ordering of mixed data set

Root Identification. It is a challenging problem to identify the correct roots of data sets where we do not have any knowledge about the compilation process. ILINE successfully identified the correct roots based upon code size and complexity in all data sets except for some data sets of actual release binaries. This shows that the Lehman's laws of software evolution are generally applicable to root identification, but with a caveat. For example, with actual release binaries data sets, ILINE achieved 77.8% mean accuracy with the inferred roots. The accuracy increased to 91.8% with the knowledge of the correct first revision.

In order to improve lineage inference, we can leverage "first seen" date of malware, e.g., Symantec's Worldwide Intelligence Network Environment [10] or toolchain provenance such as compilers and compilation options [36].

Clustering. Clustering may not be able to group program accurately due to noise or algorithmic limitations. In order to simulate cases where clustering failed, we mixed binaries from seven programs including `memcached`, `redis`, `redislite`, `grep`, `nano`, `sendmail`, and `openssh` into one set and ran our lineage inference algorithm on it. As shown in Figure 11, revisions from each program group located next to each other in the recovered order (each program is marked in a different color). This shows ILINE can identify close relationships within the same program group even with high noise in a data set. There are multiple *intra*-program gaps and *inter*-program gaps. Relatively big intra-program gaps corresponded to major version changes of a program where the Jaccard distances were 0.28–0.66. The Jaccard distances at the inter-program gaps were much higher, e.g., 0.9–0.95. This means we can separate the mixed data set into different program groups based on the inter-program gaps.

Feature Extraction. Although iLINE achieved an overall 95.8% mean accuracy in straight line lineage of goodware, iLINE achieved only 77.8% mean accuracy with actual released binaries. In order to improve lineage inference, future work may choose to leverage better features. For example, we may use recovered high-level abstraction of program binaries [41], or we may detect similar code that was compiled with different compilers and optimization options [24].

10 Related Work

While previous research focuses on studying known software lineage or development history, our focus is on designing algorithms to create lineage and evaluating metrics to assess the quality of constructed lineage.

Belady and Lehman studied software evolution of IBM OS/360 [3], and Lehman and Ramil formulated eight laws describing software evolution process [28]. Xie et al. analyzed histories of open source projects in order to verify Lehman's laws of software evolution [45], and Godfrey and Tu investigated the Linux kernel to understand a software evolution process in open source development systems [14]. Shihab et al. evaluated the effects of branching in software development on software quality with Windows Vista and Windows 7 [42]. Kim et al. studied the history of code clones to evaluate the effectiveness of refactoring on software improvement with respect to clones [25].

Massacci et al. studied the effect of software evolution, e.g., patching and releasing new versions, on vulnerabilities in Firefox [33], and Jang et al. proposed a method to track known vulnerabilities in modern OS distributions [19]. Edwards and Chen statistically verified that an increase of security issues identified by a source code analyzer in a new release may indicate an increase of exploitable bugs in a release [11]. Davies et al. proposed a signature-based matching of a binary against a known library repository to identify library version information, which can be potentially used for security vulnerabilities scans [7].

Gupta et al. studied malware metadata collected by an anti-virus vendor to describe evolutionary relationships among malware [16]. Dumitras and Neamtiu studied malware evolution to find new variants of well-known malware [9]. Karim et al. generated phylogeny models based upon code similarity to understand how new malware related to previously seen malware [22]. Khoo and Lio investigated FakeAV-DO and Skyhoo malware families using phylogenetic methods to understand statistical relationships and to identify families [23]. Ma et al. studied diversity of exploits used by notorious worms and constructed dendrograms to identify families and found non-trivial code sharing among different families [31]. Lindorfer et al. investigated the malware evolution process by comparing subsequent versions of malware samples that were collected by exploiting embedded auto-update functionality [29]. Hayes et al. pointed out the necessity of systematic evaluation in malware phylogeny systems and proposed two models to artificially generate reference sets of samples: mutation-based model and feature accretion-based model [17].

11 Conclusion

In this paper, we proposed new algorithms to infer software lineage of program binaries for two types of lineage: straight line lineage and directed acyclic graph (DAG) lineage. We built iLINE to systematically explore the entire design space depicted in Figure 1 for software lineage inference and performed over 2,000 different experiments on large scale real-world programs—1,777 goodware spanning over a combined 110 years of development history and 114 malware with known lineage. We also built iEVAL to scientifically measure lineage quality with respect to the ground truth. Using iEVAL, we evaluated seven different metrics to assess diverse properties of lineage, and recommended two metrics—partial order mismatches and graph arc edit distance. We showed iLINE effectively extracted evolutionary relationships among program binaries with over 84% mean accuracy for goodware and over 72% for malware.

12 Acknowledgment

We would like to thank our shepherd Fabian Monrose for his support in finalizing this paper. We also would like to thank the anonymous reviewers for their insightful comments. This material is based upon work supported by Lockheed Martin and DARPA under the Cyber Genome Project grant FA975010C0170. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Lockheed Martin or DARPA.

References

[1] M. Bailey, J. Oberheide, J. Andersen, F. J. Z. Morley Mao, and J. Nazario. Automated classification and analysis of internet malware. In *International Symposium on Recent Advances in Intrusion Detection*, September 2007.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium*, 2009.

[3] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[4] M. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[5] zynamics BinDiff. http://www.zynamics.com/bindiff.html. Page checked 5/23/2013.

[6] DARPA-BAA-10-36, Cyber Genome Program. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html. Page checked 5/23/2013.

[7] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle. Software bertillonage: finding the provenance of an entity. In *Working Conference on Mining Software Repositories*, New York, New York, USA, 2011.

[8] F. de la Cuadra. The geneology of malware. *Network Security*, 2007(4):17–20, 2007.

[9] T. Dumitras and I. Neamtiu. Experimental challenges in cyber security: a story of provenance and lineage for malware. In *Cyber Security Experimentation and Test*, 2011.

[10] T. Dumitras and D. Shou. Toward a standard benchmark for computer security research: the worldwide intelligence network environment (WINE). In *Building Analysis Datasets and Gathering Experience Returns for Security*, 2011.

[11] N. Edwards and L. Chen. An historical examination of open source releases and their vulnerabilities. In *ACM Conference on Computer and Communications Security*, 2012.

[12] H. Flake. Structural comparison of executable objects. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2004.

[13] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors. In *IEEE Symposium on Security and Privacy*, 2010.

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

[15] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *International Symposium on Recent Advances in Intrusion Detection*, 2008.

[16] A. Gupta, P. Kuppili, A. Akella, and P. Barford. An empirical study of malware evolution. In *International Communication Systems and Networks and Workshops*, 2009.

[17] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology*, 5(4):335–343, July 2008.

[18] X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In *ACM Conference on Computer and Communications Security*, 2009.

[19] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy*, 2012.

[20] J. Jang, D. Brumley, and S. Venkataraman. BitShred: feature hashing malware for scalable triage and semantic analysis. In *ACM Conference on Computer and Communications Security*, 2011.

[21] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *ACM Workshop on Rapid Malcode*, 2007.

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

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

[24] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Working Conference on Mining Software Repositories*, 2013.

[25] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *European software engineering conference - Foundations of software engineering*, 2005.

[26] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, 2006.

[27] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[28] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[29] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero. Lines of malicious code: insights into the malicious software industry. In *Annual Computer Security Applications Conference*, 2012.

[30] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, 2003.

[31] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *ACM SIGCOMM on Internet Measurement*, 2006.

[32] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference*, 2007.

[33] F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *International Conference on Engineering Secure Software and Systems*, 2011.

[34] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[35] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[36] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, 2011.

[37] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. V. Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *IEEE Symposium on Security and Privacy*, 2012.

[38] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference*, December 2006.

[39] A. Sæ bjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis*, 2009.

[40] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD/PODS Conference*, 2003.

[41] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, 2013.

[42] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012.

[43] Symantec. Symantec internet security threat report, volume 17. http://www.symantec.com/threatreport/, 2012. Page checked 5/23/2013.

[44] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *International Conference on Machine Learning*, 2009.

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

[46] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010.

A Appendix

A.1 Straight Line Lineage

		Mean accuracy with the *inferred* root		Mean accuracy with the *real* root	
Distance Metric	Features	Inversion Accuracy	ED	Inversion Accuracy	ED
SD	Multi	95.8%	8.6	98.4%	6.0
WSD		95.4%	9.0	98.1%	6.7
DC		93.7%	9.7	97.1%	8.4
JD		93.7%	9.7	97.1%	8.4
JC		93.0%	12.2	97.1%	9.1

Table 8: Mean accuracy for straight line lineage on goodware

		Mean accuracy with the *inferred* root (=*real* root)	
Distance Metric	Features	Inversion Accuracy	ED
SD	Static Multi	97.8%	0.9
WSD		94.2%	1.3
DC		98.2%	0.9
JD		98.2%	0.9
JC		84.3%	3.1
SD	Dynamic Multi	86.7%	2.6
WSD		80.0%	2.9
DC		85.5%	2.9
JD		85.5%	2.9
JC		70.9%	4.1

Table 9: Mean accuracy for straight line lineage on malware

A.2 DAG Lineage

		Mean accuracy with *no* prior information		Mean accuracy with *real* timestamp	
Distance Metric	Features	PO Accuracy	GAED	PO Accuracy	GAED
SD	Multi	84.0%	52.4	91.1%	20.3
WSD		82.6%	57.3	90.0%	23.0
DC		83.8%	56.1	91.1%	20.0
JD		83.8%	56.1	91.1%	20.0
JC		74.5%	90.0	90.6%	35.0

Table 10: Mean accuracy for DAG lineage on goodware

		Mean accuracy with *no* prior information		Mean accuracy with *real* timestamp	
Distance Metric	Features	PO Accuracy	GAED	PO Accuracy	GAED
SD	Static Multi	69.5%	8.5	87.0%	6.0
WSD		72.0%	8.5	90.2%	5.5
DC		69.5%	8.5	87.0%	6.0
JD		69.5%	8.5	87.0%	6.0
JC		50.8%	19.5	86.6%	9.5
SD	Dynamic Multi	61.4%	17.0	70.3%	13.0
WSD		62.2%	17.0	76.4%	12.5
DC		59.8%	19.0	72.8%	12.5
JD		59.8%	19.0	72.8%	12.5
JC		55.3%	17.5	72.8%	12.5

Table 11: Mean accuracy for DAG lineage on malware