Towards Automatic Software Lineage Inference

Abstract—**Software continuously evolves to reflect changing requirements, feature updates, and bug fixes. Most existing research focuses on analyzing software release histories to understand the software evolution process and to describe evolutionary relationships among programs. However, there has been little research on *inferring* software lineage from (binary) programs.**

In this paper, we take a systematic approach towards software lineage inference. We explore three fundamental questions not addressed by existing work. First, how do we measure the quality of a lineage inference algorithm? Second, given existing approaches to binary similarity analysis, how good are they for lineage both currently and in an idealized setting? Third, what are the challenging problems in software lineage inference? Towards these goals we build LIMetric—a system for automatic software lineage inference of binary programs. We evaluated LIMetric on two types of lineage—straight line lineage and directed acyclic graph (DAG) lineage. We have also extended our technique to handle multiple straight line lineages. Our experiments used large scale real-world programs, with a total of 1,777 releases spanning over a combined 110 years of development history. In order to quantify lineage quality, we propose four metrics: (i) number of inversions and (ii) edit distance to monotonicity for straight line lineage, and (iii) number of lowest common ancestor (LCA) mismatches and (iv) average pairwise distance to true LCA for DAG lineage. LIMetric effectively extracted software derivation relationships among binary programs with high accuracy. Through close case analysis, we also formulate several challenging problems in software lineage inference that need to be addressed to attain even higher accuracy.

Keywords-**software evolution, software lineage, systematic evaluation**

I. Introduction

Software evolves to adapt to changing needs, bug fixes, and feature requests [18]. The lineage of software through its evolution provides a potentially rich source of information for a number of security questions. For example, given a collection of malware variants, which malware came first? Given a collection of binaries in forensics, are any of them derived from the others? These kinds of questions are important enough that the US Defense Advanced Research Projects Agency (DARPA) is spending $43 million to study them [1]. Unfortunately, existing work has primarily focused on analyzing known lineage, not inferring lineage. For example, Belady and Lehman studied software evolution of IBM OS/360 [5], and Lehman and Ramil formulated eight laws describing software evolution process [18]: 1) continuing change, 2) increasing complexity, 3) self regulation, 4) conservation of organisational stability, 5) conservation of familiarity, 6) continuing growth, 7) declining quality, and 8) feedback system. With a wealth of release information such as release dates and program versions, researchers have analyzed histories of open source projects [28], Firefox [21], and Linux kernel [10] in order to verify Lehman's laws of software evolutions and to understand software evolution process.

The security community has studied malicious software (malware) evolution based upon the observation that the majority of incoming malware are tweaked variants of well-known malware [4, 12, 13]. With over 1.1 million malware appearing in one day [2], researchers have studied such evolutionary relationships to identify new variants of previously-analyzed malware [8], to understand the diversity of exploits used by notorious worms [20], and to generate phylogeny models to describe derivation relationships among programs as a dendrogram [14, 15].

The task of software lineage inference is to infer a temporal ordering and ancestor/descendant relationships among programs. Software lineage can be defined as follows:

Definition I.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 an arc $a = (x, y) \in A$ denotes that program y is a derivative of program x. We say that x is a predecessor of y and y is a successor of x.

We define some terminology. A *root* is a node that has no incoming arc and a *leaf* is a node that has no outgoing arc. Since a DAG can have multiple roots, we introduce a new node called the *super root* and add a new arc from it to every root. A DAG that has been augmented in this way is called a *super DAG*. Note that a root in a super DAG has exactly one incoming arc from the super root. An *ancestor* of a node n is a node that can reach n. Note that n is an ancestor of itself. A *common ancestor* of x and y is the intersection of the two sets of ancestors. In a DAG, a *lowest common ancestor* (LCA) of two nodes x and y is a common ancestor of x and y that is not an ancestor to another common ancestor of x and y [6]. Notice that there can be multiple LCAs. We denote the set of LCAs of x and y as $\textsc{slca}(x, y)$.

In this paper, we ask three basic research questions:

- **What are good metrics?** Existing research focused on building phylogeny of malware [14, 15], but has lacked *quality metrics* to scientifically measure the quality of their output.
  Good metrics are necessary to assess how good our approach is with respect to the ground truth. Good metrics also allow us to quantify the quality of our output, and to compare different approaches.
- **How well are we doing now?** We would like to understand what are the limits of existing techniques even in *ideal* cases, meaning we have 1) control over variables affecting the compilation of programs such

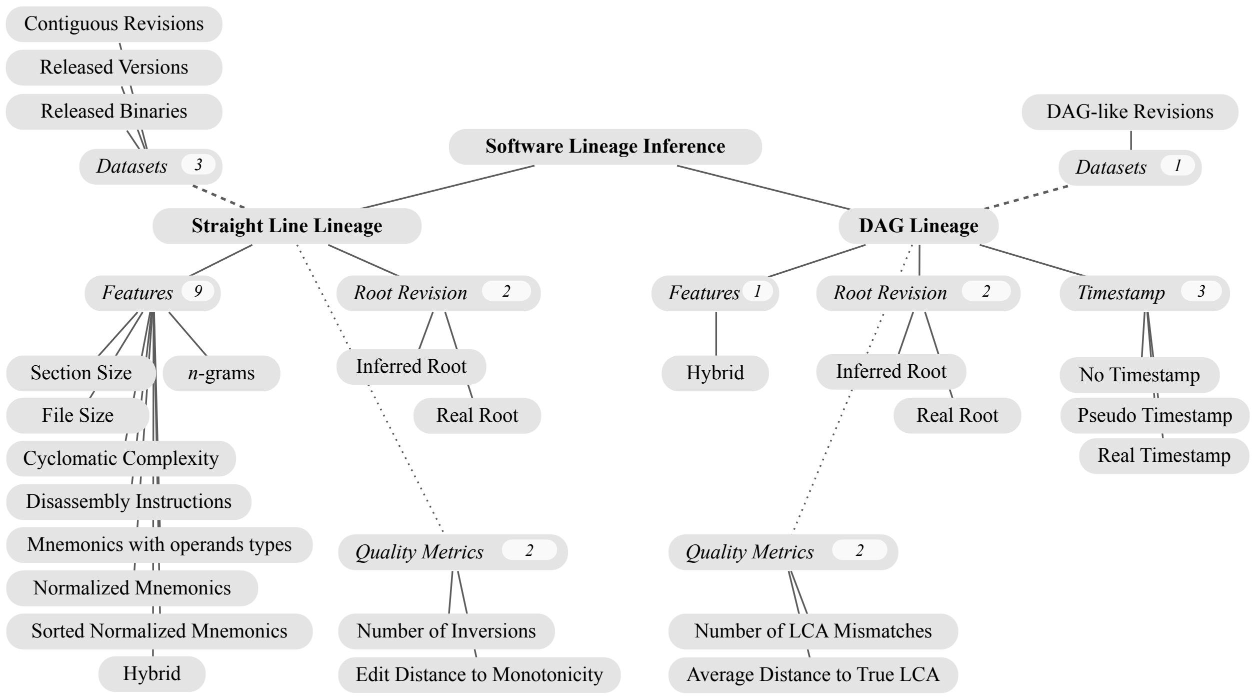


Figure 1: Design space in software lineage inference

as compiler versions and optimization options so that binary code compiled from similar source code would be similar, 2) reliable feature extraction techniques to abstract binary programs accurately and precisely, which is not always guaranteed due to the difficulty of disassembling, and 3) the ground truth with which we can compare our results to measure accuracy and to spot error cases.

Although previous approaches have focused on malware, we argue that it is necessary to first systematically validate a lineage inference technique with "goodware", e.g., open source projects. Since malware is often surreptitiously developed by adversaries, it is hard or even impossible to obtain the ground truth. Furthermore, we cannot hope to understand evolution on adversarial programs unless we first understand limits of our approach without an adversary. To the best of our knowledge, no systematic experiment addressing all the variables has been done before.

- **What are the challenging problems?** We need to identify challenging problems in order to improve software lineage inference techniques. We should evaluate with an eye towards uncovering fundamental limits or problems. Previous work has not addressed this.

In this paper, we propose techniques and metrics for the systematic investigation on software lineage as described in Figure 1. We explore two types of lineage: straight line lineage and directed acyclic graph (DAG) lineage. In addition, we extend our approach for straight line lineage to k-straight line lineage.

- **Four metrics to measure quality.** We propose two metrics for straight line lineage: *number of inversions* and *edit distance to monotonicity*. Given an inferred graph G and the ground truth G^*, the number of inversions measures how often we get the question "which one of program p_i and program p_j comes first" wrong. The edit distance to monotonicity asks "how many nodes do we need to remove in G such that the remaining nodes are in sorted order and thus respect G^*".

  We also propose two metrics for DAG lineage: *number of LCA mismatches* and *average pairwise distance to true LCA*. An LCA mismatch is a generalized version of an inversion because the LCA of two nodes in a straight line is the earlier node. In case we have a wrong LCA, we also measure the average pairwise distance between the true LCA(s) and the derived LCA(s) in G^*. This detailed measurement is desirable because it often helps us to decide which of methods is better even when their LCA scores are the same.
- **Large scale systematic evaluation.** We systematically evaluated LIMetric with goodware that we have ground truth. For straight line lineage, we collected three kinds of datasets: 1) contiguous revisions from a version control system with highly varying time gap between adjacent commits (<10 minutes to over a month)—

371 revisions from 3 programs representing 4 years of combined history, 2) released versions distributed to end users, meaning that experimental features that contiguous revisions may have are excluded—271 releases from 5 programs representing 55 years of combined history, and 3) actual released binary programs from `deb` or `rpm` package files where we do not have any control over the compiling process—355 releases from 7 programs representing 40 years of combined history. Regarding DAG lineage experiments, we downloaded revision histories that have multiple branching and merging points—780 revisions from 10 programs representing 11 years of combined history.
We also examined the effectiveness of different experiment policies: 1) whether we *infer* the root/earliest revision or use the *given real* root revision, and 2) whether we rely on *pseudo* timestamp, *real* timestamp, or *nothing*. The second policy is only for DAG lineage.
- **Challenging problems.** We investigate error cases in G constructed by LIMetric and highlight some of the difficult cases where LIMetric failed to recover the correct evolutionary relationships. We also discuss what challenging problems need to be addressed to achieve higher accuracy in software lineage inference.

II. Overview

Our goal is to systematically explore the entire design space illustrated in Figure 1 to understand the advantages and disadvantages of existing techniques for inferring software lineage. We have built LIMetric for systematic investigation on software lineage in three different scenarios: 1-straight line lineage (§II-C), k-straight line lineage (§II-D), and directed acyclic graph (DAG) lineage (§II-E).

A. Software Features for Software Lineage

We evaluate the accuracy of software lineage inference on diverse input datasets that have different characteristics: contiguous revisions, released versions, released binaries, and DAG-like revisions. Given a set of binary programs $\mathcal{P}$, various features f_i are extracted from each program $p_i \in \mathcal{P}$ to evaluate different abstractions of binary programs. 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., computer forensics, proprietary software, and malware.

We would like to evaluate existing approaches for abstracting binary programs in idealized settings, meaning that we have reliable feature extraction techniques. There are mainly three program analysis methods: syntax-based analysis, static analysis, and dynamic analysis. In this study, we leave dynamic analysis-based features out of our scope and include techniques using only syntax-based analysis and static analysis. The limitation with static analysis comes from the difficulty of getting precise disassembly outputs from binary programs [17, 19]. In order to exclude the errors introduced at the feature extraction step and focus on evaluating the performance of a software lineage inference algorithm, we also leverage assembly obtained using `gcc -S` (not source code itself) to obtain basic blocks more accurately. Note we use this to simulate what the results would be with an ideal disassembler, in line with our goal of understanding the limits of the selected approaches.

1) Using Previous Observations on Software Evolution: Previous work analyzed software release history to understand the software evolution process. It has been often observed that program size and complexity tend to increase as new revisions are released [10, 18, 28]. This observation also carries over to security scenarios, e.g., the complexity of malware is likely to grow as new variants appear [7].

We measured code section size, file size, and code complexity to assess how useful these features are in inferring lineage of binary programs.

- **Section size:** LIMetric first identifies executable sections in binary code, typically the `.text` sections, which contain executable program code, and calculates the total size.
- **File size:** Besides the section size, LIMetric also calculates the file size, including code and data.
- **Cyclomatic complexity:** Cyclomatic complexity [22] 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) Using Syntax-based Feature: Although syntax-based analysis may lack semantic understanding of a program, previous work has shown its effectiveness on classifying unpacked binary code. Indeed, n-gram analysis is widely adopted to software similarity detection, e.g., [13, 14, 16, 25]. The benefit of syntax-based analysis is that it is fast. This is because it does not require disassembling binary code.

- n**-grams:** An n-gram is a consecutive subsequence of n items in a sequence. From the identified executable sections, LIMetric extracts program code in a hexadecimal format, then n-grams are obtained by sliding a window of n-bytes over the extracted byte sequence of program code. For example, Figure 2(b) shows 4-grams extracted from Figure 2(a).

3) Using Static Features: Existing work utilized more semantically rich features by disassembling binary programs (a hexadecimal byte sequence) to produce assembly code.

```
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

```
jcc imm;mov mem,reg;test reg,reg
add imm,reg;pop reg;pop reg;pop reg;ret
add imm,reg;jmp imm;pop reg;pop reg;pop reg
```
(f) Sorted normalized mnemonics with operands type

Figure 2: Example of feature extraction

After constructing a control flow graph (CFG) of a program, each basic block can be considered as a feature [9]. In order to maximize the probability to find similar programs, they also normalized disassembly outputs such that instruction mnemonics without operands are used [15, 29] or instruction mnemonics with types of operands such as memory, a register or an immediate value are considered [24].

In our experiments, we also consider additional normalization steps: normalizing instruction mnemonics and sorting normalized instruction mnemonics. These normalization steps were motivated by the observations from our analysis on error cases in constructed lineages using the above existing techniques. Our results indicate these normalizations significantly improve lineage inference results.

- **Basic blocks consisting of *disassembly* instructions:** LIMetric disassembles programs and identifies basic blocks. Each feature indicates a sequence of disassembly instructions in a basic block. For example, in Figure 2(c), each line is a sequence of instructions in a basic block; and each line is considered as an individual feature. This feature set is potentially more semantically rich than n-grams.
- **Basic blocks consisting of instruction *mnemonics*:** For each disassembly instruction, LIMetric takes an operation mnemonic and types of operands including immediate, register and memory. For example, `add $0x2c, %esp` is transformed into `add imm, reg` in Figure 2(d). By normalizing operands, this feature set helps us to mitigate errors from syntactical differences, e.g., changes in offsets and jump target addresses, and register renaming.
- **Basic blocks consisting of *normalized* mnemonics:** LIMetric also normalizes mnemonics as well. First, mnemonics for all conditional jumps, e.g., `je`, `jne` and `jg`, are denoted as `jcc` because the same branching condition can be represented by flipped conditional jumps. For example, program p_1 uses `cmp eax, 1; jz address` while program p_2 has `cmp eax, 1; jnz address` with a different jump target address. Second, LIMetric removes `nop` since it performs no operation.
- **Basic blocks consisting of *sorted* normalized mnemonics:** LIMetric considers one more normalization step on the normalized mnemonics feature set: sorting normalized mnemonics based upon the mnemonics. This feature set helps us to remove errors caused by instruction reordering. For example, the instruction sequence `mov esi, 1; xor eax, eax; xor ebx, ebx` in program p_1 is considered as the same as the instruction sequence `xor ebx, ebx; mov esi, 1; xor eax, eax` in program p_2.

4) Using Multiple Features: Besides considering each feature set independently, LIMetric utilizes multiple feature sets to infer lineage. Normalization maximizes the chance to find similar programs by nullifying minor program changes, which means normalization can make slightly distinct programs look the same. This may be problematic when we construct lineage from a contiguous revision history of a version control system that often includes commits of minor tweaks.

- **Hybrid feature:** LIMetric utilizes multiple features in order to benefit from both normalized features and specific features. We call this a hybrid feature. Specifically, LIMetric first uses normalized features to accurately identify alike programs by reducing the distance measurement error caused by minor program changes; then utilizes specific features to distinguish fairly similar programs by capturing tiny differences.

B. Distance Metric for Software Lineage

Software evolution is a process of generating a new version of software by changing previous versions of software. Can we infer lineage of binaries by looking at syntactic features? LIMetric uses the *symmetric difference* to measure the distance between p_1 and p_2 in that the symmetric difference captures the changes made between p_1 and p_2 by measuring how many features are added or deleted.

Let f_1 and f_2 denote two feature sets extracted from p_1 and p_2, respectively. Then the symmetric difference between two feature sets f_1 and f_2 is:

$$\textsc{sd}(f_1, f_2) = |(f_1 \cup f_2) \setminus (f_1 \cap f_2)| \tag{1}$$

which denotes the cardinality of the set of features which are in either f_1 or f_2 but not in the intersection of f_1 and f_2. The symmetric difference basically measures the size of unique features in p_1 and p_2.

Other distance metrics instead of the symmetric difference may be considered for lineage inference as discussed in §V.

C. 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.

Software lineage inference in this setting is purely a problem of determining temporal ordering. Given N unlabeled revisions of a program p, the goal is to output a label "1" for the 1st revision, "2" for the 2nd revision, and so on. For example, if we are given 100 revisions of a 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}.

1) Identifying the Root Revision: In order to identify the first revision or the root 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.

LIMetric picks the root revision based upon Lehman's observation [18]. 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. Based on Lehman, 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 antivirus detection, which leads to high code complexity [7].

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

2) Inferring Order: From the selected root revision, LIMetric greedily picks the ordering by choosing the closest revision in terms of the symmetric difference as the next revision. Suppose we have three contiguous revisions: p^1, p^2, and p^3. One hypothesis is that the symmetric difference between p^1 and p^2 is smaller than the symmetric difference between p^1 and p^3. This hypothesis follows logically from Lehman's software evolution laws. In other words, the symmetric difference between two adjacent revisions would be smaller.

There may be cases where the symmetric differences between two different pairs are the same, i.e., a tie. Suppose $\textsc{sd}(p^1, p^2) = \textsc{sd}(p^1, p^3)$. Then both p^2 and p^3 become candidates for the next revision of p^1. Compared to the use of specific features like n-grams, the use of normalized features have more chances to have more ties because of the information loss.

LIMetric needs a way to break ties. LIMetric utilizes more specific features in order to break ties more correctly (see §II-A4). For example, if the symmetric differences using sorted normalized mnemonics are the same, then the symmetric differences using normalized mnemonics are used to break a tie. If two pairs still have the same symmetric difference, instruction mnemonics, disassembly instructions, and then n-grams are used to break a tie.

3) Handling Outliers: As an optional step, LIMetric handles outliers in our recovered ordering, if any. Since LIMetric constructs lineage in a greedy way, if one revision is not selected mistakenly, that revision may not be selected until the very last round. Suppose we have 5 revisions: p^1, p^2, p^3, p^4, and p^5. If LIMetric falsely selects p^3 as the next revision of p^1 ($p^1 \rightarrow p^3$) and $\textsc{sd}(p^3, p^4) < \textsc{sd}(p^3, p^2)$, then p^4 will be chosen as the next revision ($p^1 \rightarrow p^3 \rightarrow p^4$). It is likely that $\textsc{sd}(p^4, p^5) < \textsc{sd}(p^4, p^2)$ 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 detect outliers, LIMetric monitors the symmetric difference between every two adjacent pairs. An outlier can cause a peak of the symmetric difference (see Figure 8 for an example). In our example, $\textsc{sd}(p^5, p^2)$ is likely to be higher than the symmetric difference between other pairs.

If any possible outlier is identified by looking at a peak, then the possible outlier is moved either before or after the closest revision where the symmetric difference is minimized. Suppose p^3 is the closest revision to p^2. LIMetric compares $\textsc{sd}(p^1, p^3)$ (the gap *before* the closest revision) with $\textsc{sd}(p^3, p^4)$ (the gap *after* the closest revision), then insert p^2 to the bigger gap to minimize overall symmetric difference. If $\textsc{sd}(p^3, p^4)$ of neighboring revisions is smaller than $\textsc{sd}(p^1, p^3)$, we have $p^1 \rightarrow p^2 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$.

4) Lineage Quality Metrics: We use dates of commit histories and version numbers as the ground truth of ordering $G^* = (N, A^*)$, and compare the recovered ordering by LIMetric $G = (N, A)$ with the ground truth to measure

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

(a) Lineage 1

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

(b) Lineage 2

Figure 3: Inversions and edit distance to monotonicity

how close G is to G^*.

The accuracy of the constructed lineage graph G is measured by two metrics: *number of inversions* and *edit distance to monotonicity*. An inversion happens if LIMetric 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 edit distance to monotonicity 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 found in G, which is the longest (not necessarily contiguous) subsequence in sorted order. Then the edit distance to monotonicity is calculated by $|N|-|LIS|$, which depicts how many nodes are out-of-place in G.

For example, we have 5 revisions of a program and LIMetric 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 edit distance to monotonicity (delete p^2). Lineage 2 has 3 inversions ($p^3 - p^2$, $p^4 - p^2$, and $p^5 - p^2$) and 1 edit distance to monotonicity (delete p^2). As shown in both cases, the number of inversions can be different even when the edit distance to monotonicity is the same.

D. *k-Straight Line Lineage*

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

For k-straight line lineage, LIMetric first performs clustering on a given dataset $\mathcal{P}$ 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 each other, LIMetric identifies the root/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.

Given a collection of programs and revisions, previous work shows that clustering can effectively separate them [3, 12, 13, 29]. Based upon existing work, LIMetric performs clustering to find variants using the Jaccard distance. The Jaccard distance between two sets f_1 and f_2 is defined as follows:

$$\text{JD}(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}$$

For example, $\text{JD}(f_1, f_2) = 0$ when f_1 and f_2 are identical; on the other hand, $\text{JD}(f_1, f_2) = 1$ if f_1 and f_2 have no shared features.

LIMetric 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. LIMetric 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 10 for an example).

E. *Directed Acyclic Graph Lineage*

The third 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.

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 §II-C1.

2) Building Spanning Tree Lineage: LIMetric 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, LIMetric 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, LIMetric 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 LIMetric 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:

$$\begin{tikzcd} & p^1 \arrow[dl] \arrow[dr] & \\ p^2 & & p^3 \end{tikzcd}$$

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^r to be added to G is determined by the minimum symmetric difference $\texttt{min}\{\textsc{sd}(p^t, p^r) : p^t \in \hat{N}, p^r \in \hat{N}^c\}$ 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^t, p^r) is added. However, with the use of a timestamp, the revision $p^r \in \hat{N}^c$ to be inserted is determined by the earliest timestamp and an arc is drawn based upon the minimum symmetric difference. In other words, we insert nodes in the order of timestamps.

3) Adding Non-Tree Arcs: While building (directed) spanning tree lineage, LIMetric identifies *branching* points by allowing the revisions $p^t \in \hat{N}$ to have more than one outgoing arcs—revisions with multiple children. In order to pinpoint *merging* points, LIMetric adds non-tree arcs also known as cross arcs to the spanning tree lineage.

For every non-root node p^i, LIMetric 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 LIMetric examines if u^i and f^k extracted from $p^k \in N (k \neq i, j)$ have common features, and adds a non-tree arc from p^k to p^i, if any. 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, LIMetric outputs DAG lineage showing both branching and merging.

4) Lineage Quality Metrics: We propose measuring the accuracy of the constructed DAG lineage graph by two metrics: *number of LCA mismatches* and *average pairwise distance to true LCA*. Note that an inversion is 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.

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

(i) 1 point (correct) if $\textsc{slca}(x, y) = \textsc{slca}^*(x, y)$
(ii) 1 point (correct) if $\textsc{slca}(x, y) \subseteq \textsc{slca}^*(x, y)$
(iii) 1 point (correct) if $\textsc{slca}(x, y) \supseteq \textsc{slca}^*(x, y)$
(iv) $1 - \textsc{jd}(\textsc{slca}(x, y), \textsc{slca}^*(x, y))$ point

Then the number of LCA mismatches is

$$|N \times N| - \sum_{(x,y) \in N \times N} C(\textsc{slca}(x, y), \textsc{slca}^*(x, y)).$$

The 1st policy is sound and complete, i.e., we only consider exact match of $\textsc{slca}$. However, even small errors can lead to a large number of LCA mismatches. The 2nd policy is sound, i.e., every node in $\textsc{slca}$ 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., $\textsc{slca}$ 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 $\textsc{slca}$ and $\textsc{slca}^*$. In our evaluation, LIMetric followed the 4th policy since it allows us to attain a more fine-grained measure.

In the case of an LCA mismatch, i.e., $C(\textsc{slca}, \textsc{slca}^*) \neq 1$, we also propose a metric to measure the distance between the true LCA(s) and the reported LCA(s). For example, if LIMetric falsely reports p^5 as an LCA of p^6 and p^7 in Figure 4, then the pairwise distance to true LCA is 2 (= distance between p^4 and p^5). Formally, let $\text{dist}(u, v)$ represent the distance between nodes u and v in the ground truth G^*. Given $\textsc{slca}(x, y)$ and $\textsc{slca}^*(x, y)$, we define the pairwise distance to true LCA $D(\textsc{slca}(x, y), \textsc{slca}^*(x, y))$ to be

$$\sum_{(l,l^*) \in \textsc{slca}(x,y) \times \textsc{slca}^*(x,y)} \frac{\text{dist}(l, l^*)}{|\textsc{slca}(x, y) \times \textsc{slca}^*(x, y)|}$$

and the average pairwise distance to true LCA to be

$$\sum_{(x,y) \in A'} \frac{D(\textsc{slca}(x, y), \textsc{slca}^*(x, y))}{S},$$

where S equals to $|\{(x, y) \in N \times N \text{ s.t. } C(\textsc{slca}(x, y), \textsc{slca}^*(x, y)) \neq 1\}|$.

$$\begin{tikzcd}
& p^1 \arrow[dl] \arrow[dr] & \\
p^2 \arrow[d] \arrow[drr] & & p^3 \arrow[d] \arrow[dll] \\
p^4 \arrow[d] \arrow[dr] & & p^5 \\
p^6 & p^7 &
\end{tikzcd}$$

Figure 4: Lowest common ancestors

III. Implementation

LIMetric is implemented using C (2.5 KLoC) and IDAPython plugin (100 LoC). We use the IDA Pro disassembler[1] to disassemble binary programs and to identify basic blocks. As discussed in §II-A, `gcc -S` output is used to compensate the errors introduced at the disassembling step. For the scalability reason, we use feature hashing technique [13, 27] to encode extracted features into bit-vectors.

[1]http://www.hex-rays.com/products/ida/index.shtml

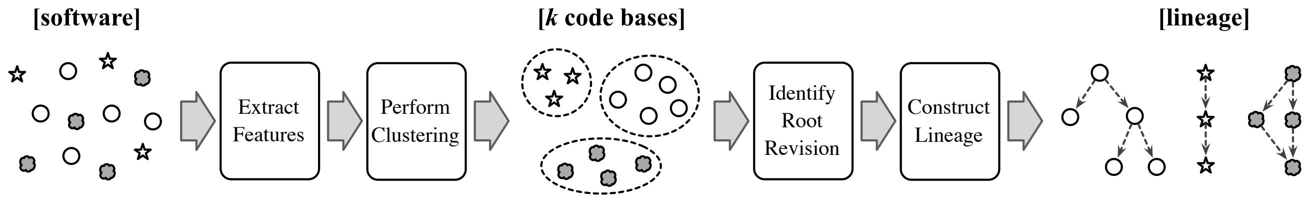


Figure 5: Software lineage inference overview

Let bv_1 and bv_2 denote two bit-vectors generated from f_1 and f_2 using feature hashing. Then the symmetric difference in Equation 1 can be calculated by:

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

where $\otimes$ denotes bitwise-XOR and $S(\cdot)$ means the number of bits set to one. The use of fast bitwise operations on bit-vectors instead of slow set operations allows LIMetric to perform experiments with a number of variables quickly.

IV. Evaluation

We systematically explored all the design spaces in Figure 1 with a variety of datasets using LIMetric as depicted in Figure 5.

A. 1-Straight Line Lineage

1) Datasets: For 1-straight line lineage experiments, we have collected three different kinds of datasets: contiguous revisions, released versions, and actual release binary files.

- **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 which changed only comments because they did not affect the resulting binary programs.

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

Table I: Datasets of contiguous revisions

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

- **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 which would be excluded in released versions. In other words, released versions are more controlled datasets. We also compiled source code with the same compiler and the same compiling options for ideal settings.

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

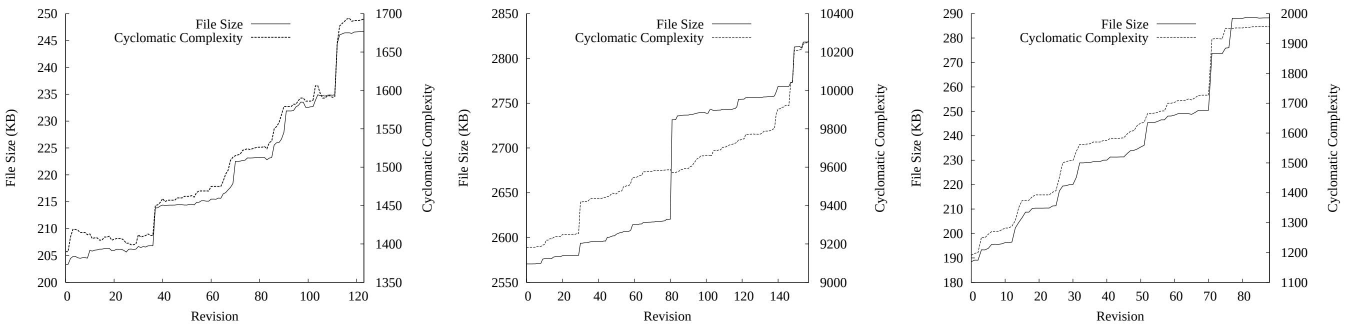
Table II: Datasets of released versions

- **Actual Release Binaries:** We collected binary files (not source code) of released versions from `rpm` or `deb` package files. 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 dataset is a representative of real-world scenarios where we do not have any information about development environments.

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

Table III: Datasets of actual released binaries

2) Results: What selection of features provides the best lineage graph with respect to the number of inversions and



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

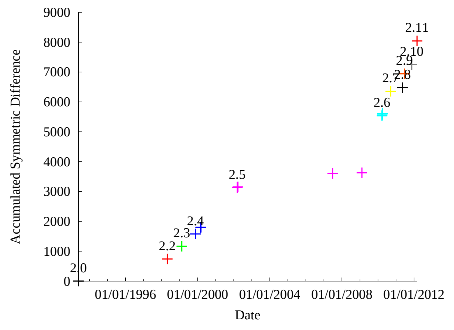
Figure 6: File size and complexity for contiguous revisions

	`memcached`			`redis`			`redislite`		
	# Inversion	ED	# Ties	# Inversion	ED	# Ties	# Inversion	ED	# Ties
Section size	357 (95.3%)	48	34 (12/22)	57 (99.5%)	28	31 (13/18)	47 (98.8%)	19	20 (11/9)
File size	171 (97.8%)	39	13 (9/4)	93 (99.3%)	36	31 (17/14)	50 (98.7%)	21	9 (3/6)
2-grams	38 (99.5%)	17	2 (2/0)	783 (93.7%)	55	21 (10/11)	134 (96.6%)	21	3 (1/2)
4-grams	24 (99.7%)	11	0 (0/0)	50 (99.6%)	25	0 (0/0)	45 (98.9%)	17	0 (0/0)
8-grams	92 (98.8%)	15	0 (0/0)	30 (99.8%)	17	1 (1/0)	46 (98.8%)	18	0 (0/0)
16-grams	60 (99.2%)	14	0 (0/0)	40 (99.7%)	21	0 (0/0)	131 (96.7%)	19	0 (0/0)
Instructions	233 (96.9%)	31	6 (2/4)	304 (97.6%)	58	6 (4/2)	208 (94.7%)	27	7 (2/5)
Mnemonics	75 (99.0%)	6	9 (4/5)	26 (99.8%)	18	29 (15/14)	13 (99.7%)	5	8 (6/2)
Normalized	75 (99.0%)	5	10 (6/4)	26 (99.8%)	17	30 (17/13)	7 (99.8%)	7	10 (5/5)
Hybrid	0 (100%)	0	0 (10/0, 0/0)	15 (99.9%)	8	0 (26/4, 0/0)	3 (99.9%)	3	0 (8/1, 0/0)

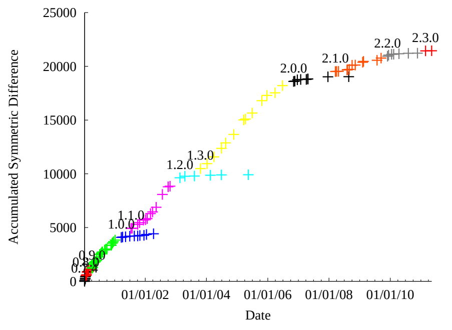
Table IV: Lineage accuracy for contiguous revisions (Percentage in inversion columns denotes accuracy.)

the edit distance to monotonicity? We evaluated different features sets on diverse datasets.

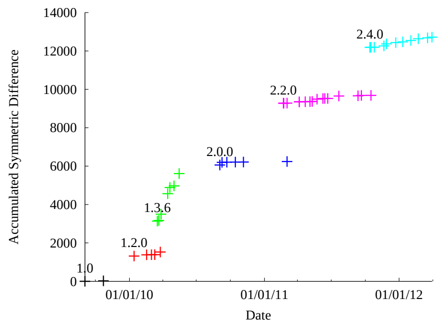
- **Contiguous Revisions:** In order to identify the first revision of each program, code complexity and code size of every revision were measured. As shown in Figure 6, both file size and cyclomatic complexity generally increased as new revisions were released. For these three datasets, the first/root revisions were correctly identified by selecting the revision that had the minimum file size and cyclomatic complexity.
  Lineage for each program was constructed as described in §II-C. The accuracy results including the number of inversions, the edit distance to monotonicity, and the number of ties are shown in Table IV. The numbers in parentheses in tie columns denote the number of correct/wrong random guessing in case of ties.
  Section/file size achieved high accuracy from 95.3% to 99.5%. However, there were many ties, which might increase/decrease the accuracy depending on random guessing choices.
  n-grams over byte sequences generally achieved better accuracy; however, 2-grams (small size of n) and 16-grams (big size of n) were relatively unreliable features, e.g., 6.3% inversion error in `redis` and 3.3% inversion error in `redislite`. In our experiments, n=4 bytes worked reasonably well for these three datasets.
  The usage of disassembly instructions had up to 5.3% 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 5.3% to 0.3%. With additional normalization, normalized instruction mnemonics with operands types achieved the same or better accuracy. Note that more normalized features can result in better or worse accuracy because there may be more ties where random guessing is involved.
  In order to break ties, more specific features were used in the hybrid feature. Regarding the hybrid feature, correct/wrong tie breaks using specific features are also presented. For example, 10/0, 0/0 at the hybrid feature row for `memcached` means 10 times of correct tie



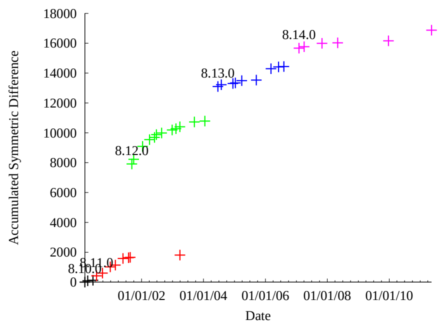
(a) `grep`



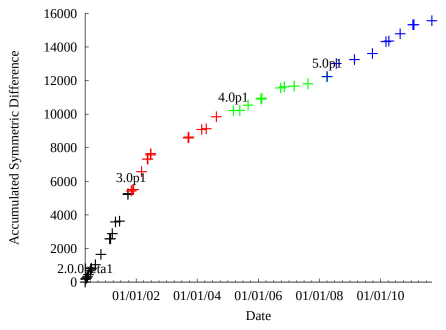
(b) `nano`



(c) `redis`



(d) `sendmail`



(e) `openssh`

Figure 7: Releases and the accumulated symmetric difference of released versions

	`grep`		`nano`		`redis`		`sendmail`		`openssh`	
	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED
Section size	21 (87.7%)	6	86 (98.7%)	26	10 (99.1%)	6	0 (100%)	0	42 (96.8%)	12
File size	4 (97.7%)	4	59 (99.1%)	23	11 (99.0%)	7	24 (96.6%)	10	18 (98.6%)	9
2-grams	29 (83.0%)	8	26 (99.6%)	9	324 (70.0%)	29	592 (15.8%)	30	695 (47.6%)	39
4-grams	3 (98.3%)	3	10 (99.8%)	6	6 (99.4%)	3	1 (99.9%)	1	5 (99.6%)	5
8-grams	2 (98.8%)	2	8 (99.9%)	5	17 (98.4%)	5	1 (99.9%)	1	5 (99.6%)	5
16-grams	2 (98.8%)	2	16 (99.8%)	7	15 (98.6%)	4	1 (99.9%)	1	5 (99.6%)	5
Instructions	38 (77.8%)	5	15 (99.8%)	10	29 (97.3%)	11	9 (98.7%)	5	14 (98.9%)	8
Mnemonics	0 (100%)	0	6 (99.9%)	5	5 (99.5%)	4	5 (99.3%)	1	2 (99.9%)	2
Normalized	0 (100%)	0	4 (99.9%)	4	5 (99.5%)	4	5 (99.3%)	1	3 (99.8%)	3
Hybrid	0 (100%)	0	1 (99.9%)	1	1 (99.9%)	1	5 (99.3%)	1	3 (99.8%)	3

Table V: Lineage accuracy for released versions (Percentage in inversion columns denotes accuracy.)

breaks using more specific features and 0 times of wrong tie breaks, and 0 times of correct guesses and 0 times of wrong guesses. This showed the effectiveness of using hybrid feature for breaking ties.

- **Released Versions:** Figure 7 shows the accumulated symmetric difference between two neighboring releases, and major release versions are marked with version numbers and in different colors.
  The first/root revisions were also correctly identified by selecting the revision that had the minimum section size. Table V shows the accuracy of lineage inference using different feature sets. Sometimes simple feature sets, e.g., section/file size achieved higher accuracy than semantically rich feature sets (requiring more expensive process), e.g., instruction sequences. For example, lineage inference with section size yielded even 100% accuracy while lineage inference with instructions got 98.7% only in `sendmail`. Like the experiments on contiguous revisions, 2-grams performed worse in the experiments on released versions, e.g., 15.8% accuracy in `sendmail`. Among various feature sets, the hybrid feature outperformed other feature sets.
- **Actual Release Binaries:** The first/root revisions for `nano` and `openssh` were correctly identified by selecting the revision that had the minimum section size. For the other five datasets, we performed the experiments both with the wrong inferred root and with the correct root given from the ground truth.
  Overall accuracy of constructed lineage was fairly high across all the datasets as shown in Table VI even though we did not control the variables of the compiling process. 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).
  It was confirmed that lineage inference can be highly affected by the root selection step. For example, LIMetric 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.

3) Case Study:

- **Outlier:** Without handling outliers, LIMetric had 70 inversions and 1 edit distance for the contiguous revisions of `memcached`. The error came because the 53rd revision was incorrectly located at the end of lineage due to the nature of greedy selection as we discussed in §II-C3. Figure 8 shows the symmetric differences between two adjacent revisions in the recovered lineage before we process outliers. The outlier caused a peak of the symmetric difference at the rightmost of the Figure 8. LIMetric identified such possible outliers by looking at peaks, then generated perfect lineage of

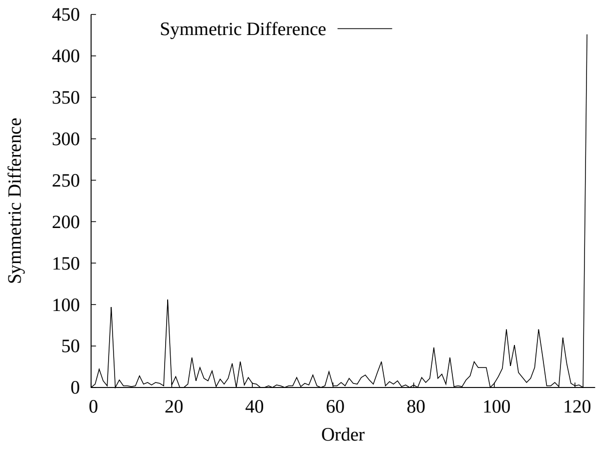


Figure 8: An outlier in `memcached`

  `memcached` after handling the outlier.
- **Reverting to the previous revision:** A revision adding new functionalities is sometimes followed by stabilizing steps including bug fixes. Bug fixes might be done by reverting to the previous revision, i.e., *undoing* the modifications of code.
  Regression of code is in fact a challenging problem in software lineage inference. For example, the errors in the constructed lineage for `redislite` occurred at the following revisions:

$$\begin{tikzcd} p^{31} \arrow[r] & p^{33} \arrow[r] & \boxed{p^{32}} \end{tikzcd}$$

(a)

$$\begin{tikzcd} p^{55} \arrow[r] & p^{57} \arrow[r] & \boxed{p^{56}} \end{tikzcd}$$

(b)

Figure 9: Inversions in `redislite`

  p^{33} fixed a bug by reverting buggy code introduced at p^{32} to the previous code at p^{31} as in Listing 1:

```
- if (page->list->right_page == 0) {
-         right->right_page = right->↩
    left_page;
- }
- else {
-         right->right_page = page->list->↩
    right_page;
- }
+ right->right_page = page->list->right_page↩
    ;
```

Listing 1: Regression of code at p^{33} in `redislite`

  p^{57} fixed a bug that sent a wrong struct at p^{56} by changing back to the code in the previous revision p^{55} as in Listing 2:

	grep				nano		redis				openssh	
	Inferred root		Real root				Inferred root		Real root			
	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED
Section size	57 (91.4%)	17	75 (88.7%)	18	104 (95.6%)	22	32 (95.7%)	12	35 (95.3%)	15	235 (91.5%)	34
File size	38 (94.3%)	13	36 (94.6%)	12	144 (93.9%)	27	33 (95.6%)	12	37 (95.0%)	12	274 (90.1%)	32
2-grams	148 (77.8%)	22	74 (88.9%)	13	411 (82.5%)	30	333 (55.1%)	25	357 (51.8%)	24	2041 (26.5%)	64
4-grams	30 (95.5%)	11	12 (98.2%)	8	19 (99.2%)	8	50 (93.3%)	11	45 (93.9%)	10	398 (85.7%)	19
8-grams	39 (94.1%)	13	24 (96.4%)	10	10 (99.6%)	5	12 (98.4%)	12	29 (96.1%)	12	146 (94.7%)	22
16-grams	62 (90.7%)	17	54 (91.9%)	15	12 (99.5%)	6	28 (96.2%)	10	23 (96.9%)	9	300 (96.9%)	25
Instructions	88 (86.8%)	18	63 (90.5%)	13	68 (97.1%)	12	30 (96.0%)	10	27 (96.4%)	10	295 (89.2%)	30
Mnemonics	32 (95.2%)	13	13 (98.1%)	9	2 (99.9%)	2	28 (96.2%)	9	24 (96.8%)	8	396 (85.7%)	22
Normalized	25 (96.3%)	10	7 (99.0%)	6	5 (99.8%)	4	28 (96.2%)	9	25 (96.6%)	9	397 (85.7%)	22
Hybrid	26 (96.1%)	10	9 (98.7%)	7	4 (99.8%)	3	30 (96.0%)	11	26 (96.5%)	10	398 (85.7%)	23

	sendmail				FileZilla				p7zip			
	Inferred root		Real root		Inferred root		Real root		Inferred root		Real root	
	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED	# Inversion	ED
Section size	121 (85.2%)	24	110 (86.6%)	21	492 (74.0%)	26	489 (74.1%)	25	208 (58.1%)	20	284 (42.7%)	21
File size	123 (85.0%)	22	131 (84.0%)	23	987 (47.8%)	35	840 (55.6%)	45	200 (59.7%)	19	292 (41.1%)	22
2-grams	298 (63.7%)	24	219 (73.3%)	24	1176 (37.8%)	52	1099 (41.9%)	51	314 (36.7%)	26	321 (35.3%)	25
4-grams	184 (77.6%)	16	141 (82.8%)	15	920 (51.4%)	28	8 (99.6%)	6	200 (59.7%)	16	12 (97.6%)	6
8-grams	131 (84.0%)	20	104 (87.3%)	18	765 (59.6%)	24	5 (99.7%)	5	197 (60.3%)	16	67 (86.5%)	9
16-grams	32 (96.1%)	15	19 (97.7%)	11	766 (59.5%)	23	3 (99.8%)	3	159 (67.9%)	12	47 (90.5%)	9
Instructions	220 (73.2%)	19	176 (78.5%)	19	768 (59.4%)	24	800 (57.7%)	27	196 (60.5%)	18	138 (72.2%)	15
Mnemonics	185 (77.4%)	20	138 (83.2%)	18	916 (51.6%)	27	5 (99.7%)	5	189 (61.9%)	12	57 (88.5%)	5
Normalized	153 (81.3%)	24	136 (83.4%)	17	916 (51.6%)	27	3 (99.8%)	3	189 (61.9%)	12	57 (88.5%)	5
Hybrid	151 (81.6%)	24	137 (83.3%)	18	915 (51.6%)	26	3 (99.8%)	3	189 (61.9%)	12	57 (88.5%)	5

Table VI: Lineage accuracy for actual release binaries (Percentage in inversion columns denotes accuracy.)

```
- status = redislite_insert_key(_cs, page->↩
    page, str, length, 1, ↩
    REDISLITE_PAGE_TYPE_FIRST);
+ status = redislite_insert_key(_cs, page, ↩
    str, length, 1, ↩
    REDISLITE_PAGE_TYPE_FIRST);
```

Listing 2: Regression of code at p^{57} in `redislite`

The code was reverted to the previous revision so that $\text{SD}(p^{31}, p^{33}) < \text{SD}(p^{31}, p^{32})$ and $\text{SD}(p^{55}, p^{57}) < \text{SD}(p^{55}, p^{56})$. As a result, inversions happened at p^{32} and p^{56}. We argue that unless we build a precise model describing the developers' reverting activity, *no* reasonable algorithm may be able to construct the same lineage as the ground truth. Rather, the constructed lineage can be a representation of more "practical" evolutionary relationships.

Our data indicates that the hybrid feature can achieve over 99% accuracy in idealized settings, and over 83% accuracy on real-world binaries. Using similar techniques, e.g., for malware, one cannot expect to do much better.

B. *k*-Straight Line Lineage

Does having multiple mixed k-straight line lineage software affect results? For 2 straight line of lineage, we mixed `memcached` and `redislite` in that both programs have the same functionality, similar code section sizes, and a reasonable number of revisions. Figure 10 shows the resulting number of clusters with various similarity threshold. From 0.5 to 0.8 similarity threshold, the resulting number of clusters were 2. This means LIMetric can first perform clustering to divide the dataset into two groups, then build straight line lineage for each group.

The resulting number of clusters of the mixed dataset of 3 programs including `memcached`, `redislite`, and `redis` became stabilized to 3 from 0.5 to 0.8 similarity threshold, which means they were successfully clustered for the subsequent straight line lineage building process.

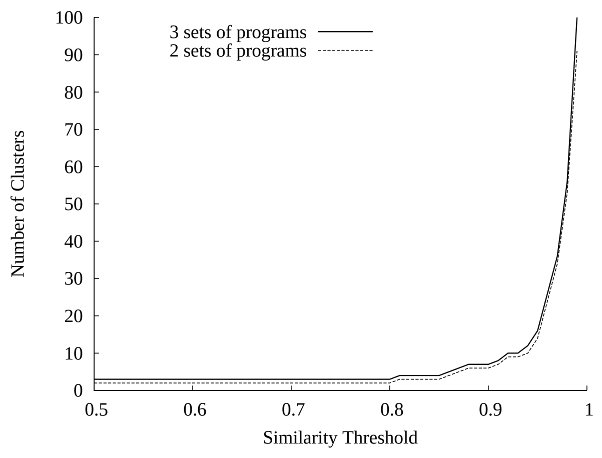


Figure 10: Number of clusters from 2 and 3 sets of programs

C. Directed Acyclic Graph Lineage

1) Datasets: We have collected 10 datasets for directed acyclic graph lineage experiments from github[2]. 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.

We downloaded DAG-like 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
`http-parser`	55	2010-11-05	2012-07-27
`libgit2`	61	2012-06-25	2012-07-17
`redis`	98	2010-04-29	2010-06-04
`rediślite`	97	2011-04-19	2011-06-12
`shell-fm`	107	2008-10-01	2012-06-26
`stud`	73	2011-06-09	2012-06-01
`tig`	58	2006-06-06	2007-06-19
`uzbl`	73	2011-08-07	2012-07-01
`webdis`	96	2011-01-01	2012-07-20
`yajl`	62	2010-07-21	2011-12-19

Table VII: Datasets of DAG lineage

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 hybrid feature sets for DAG lineage experiments in that hybrid feature sets were demonstrated to attain the best accuracy in constructing straight line lineage.

Without having any prior knowledge, as described in Table VIII, LIMetric achieved from 60.8% to 89.5% accuracy

[2]https://github.com/

$$\begin{tikzcd}
p^{17} \arrow[r] & p^{18} \arrow[r] & p^{19} \arrow[r] & p^{20} \arrow[r] & p^{21} \arrow[r] & p^{22} \arrow[r] & p^{23}
\end{tikzcd}$$

(a) Ground truth

$$\begin{tikzcd}
p^{17} \arrow[r] \arrow[dr] & p^{18} \arrow[r] & p^{19} \arrow[r] & p^{20} \\
 & p^{21} \arrow[r] & p^{22} \arrow[r] & p^{23}
\end{tikzcd}$$

(b) Constructed lineage

Figure 11: Ground truth and constructed lineage of `http-parser`

and the average pairwise distance to true LCA was from 1.3 to 3.0, which means only 3 nodes apart from the true LCA even when it was wrong. By using the real root revision, the accuracy increased to from 64.4% to 89.5%. For example, in case of `tig`, LIMetric gained about 20% increase of the accuracy.

Pseudo timestamp means LIMetric first builds straight line lineage then use the ordering as a timestamp. Since LIMetric achieved fairly high accuracy in straight line lineage, this method was expected to have good performance. Surprisingly, the accuracy was worse even with the use of real root revisions. The reason was that branching made it difficult to recover the correct ordering because each branch had been developed in parallel. This resulted in highly inaccurate (pseudo) timestamps.

By using the real timestamps, LIMetric achieved high accuracy from 73.5% to 98.3% and the average pairwise distance to true LCA was from 1.1 to 4.4. This shows that the recovered DAG lineage was very close to the true DAG lineage.

3) Case Study:

- **Reverting to the previous revision:** Figure 11 shows partial ground truth lineage and constructed lineage by LIMetric for `http-parser`. From p^{17} to p^{23}, ground truth had a straight line of evolution relationships. The reason why LIMetric generated an arc from p^{17} to p^{21} was that p^{21} reverted (removed) code for supporting non-ASCII characters to the previous revision and became $\text{SD}(p^{17}, p^{21}) < \text{SD}(p^{20}, p^{21})$. As discussed in §IV-A3, reverting is a challenging problem when software lineage is inferred from binary files.
  Figure 12 shows partial ground truth and the constructed lineage of `webdis`. The difference was that the constructed lineage had an arc from p^{17} to p^{27} instead of p^{26} to p^{27}. This also happened because some code was reverted.
- **Pseudo timestamp:** As shown in Table VIII, LIMetric with pseudo timestamp performed worse. Each branch had been developed separately, it was challenging to recover precise ordering with the straight

Policies		http-parser		libgit2		redis		redislite		shell-fm	
Time stamp	Real Root	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist
-	-	582 (60.8%)	1.8	439 (76.0%)	2.8	613 (87.1%)	1.4	524 (88.8%)	2.0	1619 (71.5%)	2.2
-	√	528 (64.4%)	1.9	439 (76.0%)	2.8	613 (87.1%)	1.4	507 (89.1%)	2.0	1524 (73.1%)	2.8
Pseudo	-	653 (56.0%)	2.2	535 (70.8%)	5.1	706 (85.2%)	1.9	978 (79.0%)	5.0	1893 (66.6%)	2.0
Pseudo	√	599 (59.7%)	2.3	535 (70.8%)	5.1	706 (85.2%)	1.9	627 (86.5%)	1.7	1555 (72.6%)	3.1
Real	√	394 (73.5%)	2.9	285 (84.4%)	2.3	485 (89.8%)	4.4	315 (93.2%)	1.1	1391 (75.5%)	2.2

Policies		stud		tig		uzbl		webdis		yajl	
Time stamp	Real Root	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist	LCA Mismatch	Avg Dist
-	-	901 (65.7%)	1.5	552 (66.6%)	2.6	584 (77.8%)	2.2	479 (89.5%)	1.3	479 (74.7%)	3.0
-	√	878 (66.6%)	1.5	225 (86.4%)	1.4	331 (87.4%)	2.6	479 (89.5%)	1.3	479 (74.7%)	3.0
Pseudo	-	1307 (50.3%)	7.2	824 (50.2%)	5.8	1342 (48.9%)	7.3	1533 (66.4%)	14.4	769 (59.3%)	5.2
Pseudo	√	1340 (49.0%)	7.1	524 (68.3%)	7.1	964 (63.3%)	8.8	1533 (66.4%)	14.4	751 (60.3%)	5.3
Real	√	389 (85.2%)	1.2	28 (98.3%)	2.0	211 (92.0%)	1.2	256 (94.4%)	1.4	325 (82.8%)	1.9

Table VIII: Lineage accuracy for directed acyclic graph lineage (Percentage in LCA Mismatch columns denotes accuracy.)

$$\begin{tikzcd}
p^{17} \arrow[r] \arrow[dr] & p^{19} \arrow[r] & p^{20} \arrow[r] & p^{21} \arrow[r] & p^{23} \arrow[r] & p^{25} \arrow[r] & p^{29} \\
 & p^{22} \arrow[r] \arrow[urrr] & p^{24} \arrow[r] \arrow[urrr] & p^{26} \arrow[r] & p^{27} \arrow[r] & p^{28} \arrow[ur] &
\end{tikzcd}$$

(a) Ground truth

$$\begin{tikzcd}
p^{17} \arrow[r] \arrow[dr] \arrow[ddr] & p^{19} \arrow[r] & p^{20} \arrow[r] & p^{21} \arrow[r] & p^{23} \arrow[r] & p^{25} \arrow[r] & p^{29} \\
 & p^{22} \arrow[r] \arrow[urrr] & p^{24} \arrow[r] \arrow[urrr] & p^{26} & & & \\
 & p^{27} \arrow[r] & p^{28} \arrow[uurrrr] & & & &
\end{tikzcd}$$

(b) Constructed lineage

Figure 12: Ground truth and constructed lineage of `webdis`

$$\begin{tikzcd}
p^{22} \arrow[r] & p^{23} \arrow[r] \arrow[dr] & p^{29} \arrow[r] & p^{30} \arrow[r] & p^{35} \arrow[r] & p^{36} \\
 & & p^{40} \arrow[r] & p^{41} \arrow[r] & p^{42} \arrow[r] & p^{43}
\end{tikzcd}$$

(a) Ground truth

$$\begin{tikzcd}
p^{22} \arrow[rrrr] \arrow[d] & & & & p^{23} \arrow[r] & p^{29} \arrow[r] & p^{30} \arrow[r] & p^{35} \arrow[r] & p^{36} \\
p^{40} \arrow[r] & p^{41} \arrow[r] & p^{42} \arrow[r] & p^{43} \arrow[ur] & & & & &
\end{tikzcd}$$

(b) Constructed lineage

Figure 13: Ground truth and constructed lineage of `uzbl`

line lineage method. For example, Figure 13 shows the partial ground truth and the constructed lineage by LIMetric for `uzbl` with pseudo timestamp. LIMetric *without* timestamp successfully recovered the ground truth lineage. However, the use of pseudo timestamp resulted in poor performance. The recovered ordering, i.e., pseudo timestamp was $p^{22}, p^{40}, p^{41}, p^{42}, p^{43}, p^{23}, p^{29}, p^{30}, p^{35}, p^{36}$. Due to the imprecise timestamp, the derivation relationships in the constructed lineage was not accurate.

V. Discussion

In many open source projects and malware, code size usually grows over time [7, 28]. In other words, addition of new code is preferred to deletion of existing code. This also holds in our datasets except for major changes followed by minor cleanups. Differentiating the costs of addition and deletion helps us to decide a direction of derivation. Suppose a deletion cost is 2 and an addition cost is 1, i.e., a deletion is a twice expensive operation. Program p_i has a feature set $f_i = \{m_1, m_2, m_3\}$, and program p_j contains a feature set $f_j = \{m_1, m_2, m_4, m_5\}$. A direction of evolution $p_i \to p_j$ has a distance of 4 (=deletion of m_3 and addition of m_4 and m_5). On the other hand, a direction of evolution $p_j \to p_i$ has a distance of 5 (=deletion of m_4 and m_5 and addition of m_3). Consequently, $p_i \to p_j$ is a more plausible scenario.

We evaluated LIMetric with a deletion cost of 2 and an addition cost of 1 with 10 DAG lineage datasets. Overall average accuracy remained almost the same, e.g., from 75.74% to 75.28%. One possible reason was that one line modification of source code would result in a deletion of a basic block (feature) and an addition of a basic block (feature), i.e., 1 modification = 1 deletion + 1 addition. We leave it as a future work to distinguish a modification with a deletion and an addition.

Correct software lineage inference on a *revision history* may not correspond with software *release date* lineage. For example, as shown in Figure 7b, a development branch of `nano-1.3` and a stable branch of `nano-1.2` are developed in parallel. In straight line lineage, LIMetric infers software lineage consistent with a revision history.

In our experiments, we used the symmetric difference as a distance metric. Other distance metrics can be considered as alternatives. For example, the Jaccard distance can be used to calculate dissimilarity between two feature sets. However, the downside of the Jaccard distance is that the same amount of code change can yield different distances depending on the size of feature sets, and the Jaccard distance does not indicate which features are added or deleted unlike the symmetric difference.

VI. Related Work

Existing research analyzed open source projects [28] and Linux kernel [10] to understand evolutionary relationship among programs, and studied the security implication of software evolution on known vulnerabilities in Firefox [21]. Empirical study was performed to evaluate the effects of branching in software development on software quality with Windows Vista and Windows 7 [26].

In order to describe evolutionary relationships among malware, empirical study on malware metadata including text descriptions and dates collected by an anti-virus vendor was performed [11], phylogeny of remote code injection exploits was constructed [20], and phylogeny models were generated using n-perms of code [14]. Researchers used derivation relationships among malware to find new variants of well known malware [8].

There have been many approaches to abstract binary programs. Syntax-based methods identifies code sections in a program and extracts n-grams features on byte sequences of program code, e.g., [13, 14, 16, 25]. Static analysis methods translates machine code into assembly code and use instructions [15, 24, 29] and basic blocks [9]. Dynamic analysis methods collect information about binary programs by monitoring program executions at run time [4, 23].

VII. Conclusion

In this paper, we systematically explored the entire design space in software lineage inference. We built LIMetric to control a number of variables and performed over 400 different experiments on large scale real-world programs—1,777 releases for a combined 110 years of development history. We built software lineage on two types of lineage: straight line lineage and directed acyclic graph (DAG) lineage. We also proposed four metrics to measure lineage quality: number of inversions and edit distance to monotonicity for straight line lineage, and number of LCA mismatches and average pairwise distance to true LCA for DAG lineage. We showed that LIMetric effectively extracted software evolutionary relationships among binary programs with high accuracy.

References

[1] DARPA-BAA-10-36, Cyber Genome Program. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html. Page checked 8/5/2012.

[2] Symantec internet security threat report. http://www.symantec.com/threatreport/. URL checked 8/5/2012.

[3] zynamics BinDiff. http://www.zynamics.com/bindiff.html. Page checked 8/5/2012.

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

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

[6] M. A. 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, Nov. 2005.

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

[8] 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.

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

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

[11] A. Gupta, P. Kuppili, A. Akella, and P. Barford. An empirical study of malware evolution. In *International Communication Systems and Networks and Workshops*, pages 1–10, Jan. 2009.

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

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

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

[15] W. M. Khoo and P. Lio. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *SysSec Workshop*, pages 3–10, July 2011.

[16] 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, Dec. 2006.

[17] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static

disassembly of obfuscated binaries. In *Proceedings of the USENIX Security Symposium*, 2004.

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

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

[20] 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*, page 53, 2006.

[21] F. Massacci, S. Neuhaus, and V. H. Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems*, pages 195–208, 2011.

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

[23] 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.

[24] 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*, page 117, 2009.

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

[26] 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.

[27] 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.

[28] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. *Proceedings of the IEEE International Conference on Software Maintenance*, pages 51–60, Sept. 2009.

[29] 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*, pages 95–104, 2010.