Scaling Software Security Analysis to Millions of Malicious Programs and Billions of Lines of Code

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Jiyong Jang
B.S., Computer Science and Industrial System Engineering, Yonsei University
M.S., Computer Science, Yonsei University

Carnegie Mellon University
Pittsburgh, PA

August, 2013

Thesis Committee:

Prof. David Brumley, Chair (Carnegie Mellon University)

Prof. Adrian Perrig (Carnegie Mellon University and ETH Zürich)

Prof. Dawn Song (University of California, Berkeley)

Prof. Heejo Lee (Korea University)

This research is supported by Lockheed Martin and Defense Advanced Research Projects Agency under the Cyber Genome Project grant FA975010C0170, and by the Graduate Student Fellowship from Symantec, Inc.

The views and conclusions contained here are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Carnegie Mellon University, Lockheed Martin, Defense Advanced Research Projects Agency, or Symantec.

Keywords: Malware, Triage, Feature Hashing, Co-clustering, Hadoop, Unpatched Code Clone, Bloom Filter, Lineage, Binary Analysis, Code Reuse, Big Data

Abstract

Software security is a big data problem. The volume of new software artifacts created far outpaces the current capacity of software analysis. This gap has brought an urgent challenge to our security community—scalability. If our techniques cannot cope with an ever increasing volume of software, we will always be one step behind attackers. Thus developing scalable analysis to bridge the gap is essential.

In this dissertation, we argue that automatic code reuse detection enables an efficient data reduction of a high volume of incoming malware for downstream analysis and enhances software security by efficiently finding known vulnerabilities across large code bases. In order to demonstrate the benefits of automatic software similarity detection, we discuss two representative problems that are remedied by scalable analysis: malware triage and unpatched code clone detection.

First, we tackle the onslaught of malware. Although over one million new malware are reported each day, existing research shows that most malware are not written from scratch; instead, they are automatically generated variants of existing malware. When groups of highly similar variants are clustered together, new malware more easily stands out. Unfortunately, current systems struggle with handling this high volume of malware. We scale clustering using feature hashing and perform semantic analysis using co-clustering. Our evaluation demonstrates that these techniques are an order of magnitude faster than previous systems and automatically discover highly correlated features and malware groups. Furthermore, we design algorithms to infer evolutionary relationships among malware, which helps analysts understand trends over time and make informed decisions about which malware to analyze first.

Second, we address the problem of detecting unpatched code clones at scale. When buggy code gets copied from project to project, eventually all projects will need to be patched. We call clones of buggy code that have been fixed in only a subset of projects unpatched code clones. Unfortunately, code copying is usually ad-hoc and is often not

tracked, which makes it challenging to identify all unpatched vulnerabilities in code bases at the scale of entire OS distributions. We scale unpatched code clone detection to spot over 15,000 latent security vulnerabilities in 2.1 billion lines of code from the Linux kernel, all Debian and Ubuntu packages, and all C/C++ projects in SourceForge in three hours on a single machine. To the best of our knowledge, this is the largest set of bugs ever reported in a single paper.

Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Professor David Brumley, for his help and support throughout my Ph.D. journey. His enthusiasm and creativity inspired me, and he helped me develop scientific research and presentation skills. I also would like to convey my appreciation to my committee members, Professor Adrian Perrig, Professor Dawn Song, and Professor Heejo Lee for dedicating their time and effort to improving my dissertation.

I deeply appreciate all my coauthors and colleagues for all of the great work and constructive feedback. I would like to thank Shobha Venkataraman, Maverick Woo, and Tudor Dumitraş for energetic discussions, extensive advice, and guidance. I also thank Sang Kil Cha, Iulian Moraru, and Abeer Agrawal for great ideas and hard work. I would like to give my thanks to Thanassis Avgerinos, Edward Schwartz, JongHyup Lee, Brian Pak, Manuel Egele, Matthew Maurer, Alexandre Rebert, Tiffany Bao, Samantha Gottlieb, and Peter Chapman for insightful feedback and encouragement.

I owe an immense debt of gratitude to my parents, Dong Il and Jung Ja, my sister, Jisun, my brother-in-law, Jaeyoung, my parents-in-law, Suk Jeong and Nam Sook, and sister-in-law, Sujung. I would not be able to complete my Ph.D. without their continued love, faith, and support.

Lastly, I cannot even try to express in words how much I appreciate my lovely wife and my soulmate, Sumi. Her unconditional love and encouragement supported me to get over all the difficulties.

There are too many people around me who have influenced my research to enumerate them all. Although many people have contributed to completing this dissertation, all defects contained here are my own.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Improving Software Security

People use many types of software everyday, from operating systems and web browsers to word processors and email clients. With this extensive use, however, comes a pressing and ongoing need for software security. According to the National Vulnerabilities Database, 5,289 new software vulnerabilities were reported in 2012 [7]. Ubuntu [9] and Debian [4] also released 364 and 220 security advisories in 2012, respectively. These security reports cover various kinds of software vulnerabilities, such as buffer overflow, integer overflow, denial of service, information disclosure, and so on. Some of these vulnerabilities can be exploited to install malware. For example, the 2011 attack against RSA started from an email with a spreadsheet attachment. The spreadsheet contained an exploit for Adobe Flash vulnerability (CVE-2011-0609) that was used to install a backdoor for gathering credential information [144].

Malware becomes increasingly prevalent with the advance of technologies that allow more users to be connected to each other, including malicious users. According to Symantec, 5.5 billion malicious attacks were detected in 2011, which was an increase of more than 81% from 2010 [154]. McAfee, similarly, received hundreds of thousands of new malware per day in 2012 [110], a number that is likely to keep increasing. In this hostile environment, it is important to ensure the security of software so that we can safely use software for sensitive tasks, e.g., online banking and editing confidential documents.

Given limited resources of time, hardware, money, and human ability, it is necessary to have techniques that can examine an ever greater amount of software. If our current techniques can handle only part of the incoming malware, then the remaining malware cannot

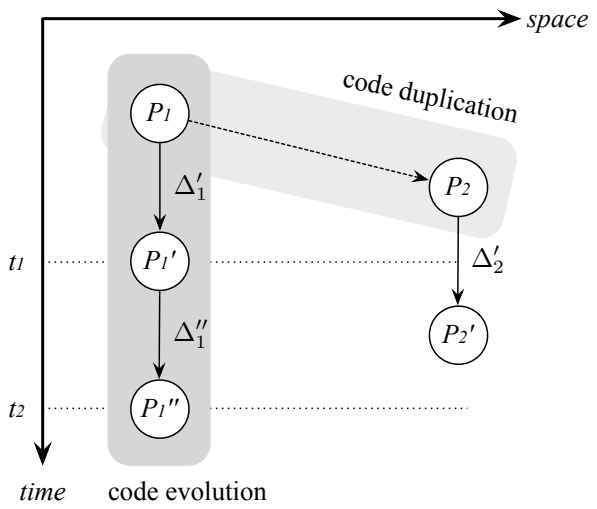


Figure 1.1: Code reuse

be processed and we may miss security-critical malware, e.g., next variants of Stuxnet.

My vision is to bridge the gap between the volume of new software created and the current capacity of software analysis by developing scalable analysis for software security. In this dissertation, we focus on scalability issues posed by extensive code reuse, or the reuse of existing code to develop new software.

1.2 Code Reuse in Software Development

Code reuse is not uncommon in software development, including open source projects and malware. For example, previous research found that 12% of all code in the Linux file system code was copied [80, 83] and that 49 bugs in Linux were due to developers not fixing buggy code that was copied from one project to another [101]. Symantec also discovered that Duqu shared a significant portion of code with Stuxnet [153], and previous research showed that most incoming malware were minor variants of existing malware [22, 69].

There are two types of code reuse: code duplication and code evolution. Code duplication involves copying and pasting some or all of the code from an existing program to a new program. This is sharing of code among different programs called spatial code reuse, as shown in Figure 1.1, where code is copied from P_1 to P_2. A software library for handling common file formats (like an XML parser library or a JPEG image library) is a real-world example of spatial code reuse. Developers can reuse a software library to reduce the costs of software development and maintenance and to benefit from well-designed code [54].

Code evolution, on the other hand, refers to the process of developing a new version of software by updating previous versions. In the evolution of program source code, subsequent versions are derived from previous versions so that a substantial amount of code is typically shared among them. This is called temporal code reuse, e.g., updating code from P_1 to P_1', as shown in Figure 1.1. Software is constantly evolving to adapt to changing needs, bug fixes, and feature additions [99]. For example, new variants of malware can be equipped with additional modules to see if the malware is being run on a virtual environment, which malware analysts often do for safety reasons. If such a virtual environment is detected, malware may stop executing or hide malicious activities to avoid detection.

1.3 Scalability as a Security Problem

Extensive code reuse has brought an urgent challenge to our security community—scalability. The community is faced with questions about whether or not our current techniques will work on very large data sets, and whether we can handle a large amount of data efficiently.

According to the Symantec Internet Security Threat Report [154], Symantec received over 403 million new variants of malware in 2011, which is a 41% increase over 2010. McAfee also collected over 35 million new malware in 2012 [110], which is a 52% increase over 2011. It is unlikely that 1.1 million new malware are written from scratch every day. It is more likely that malware authors slightly modify existing code to produce "new" malware variants. Reusing existing code to produce new variants saves both time and money. For example, simple tweaks such as reordering instructions, inserting dummy instructions, changing file headers, and modifying null alignment parts are enough to change the MD5 hash value of a file. When an anti-virus signature for a specific malware is based upon MD5 checksums, e.g., `hdb` for a file and `mdb` for a Portable Executable (PE) section in ClamAV [40], such tweaks can bypass anti-virus detection.

In order to handle such a huge volume of incoming malware fueled by extensive code reuse, the security community proposed to perform clustering on malware [22, 69, 84, 131, 140], with the idea that if two malware samples shared a significant amount of identical code, they were likely to be part of the same overall family. Based on the clustering results, malware triage can be done to determine which malware should be analyzed with a high priority. By grouping similar malware, analysts can also determine how much malware is really unique and how much malware is a minor variant of existing malware.

However, current techniques are not scalable enough to handle such a huge volume of incoming malware. For example, if we used one of the existing similarity metric libraries,

such as SimMetrics v1.6.2 [36], to measure pairwise similarity of all possible pairs, we would need more than 200,000 CPUs to handle 1.1 million new malware per day. We need faster and more scalable techniques to efficiently handle this volume of malware.

Extensive code reuse also causes problems for goodware when the original code contains bugs. When buggy code from one project is copied to a new project, both the original and new project will eventually need to be patched. For example, if a patch Δ_1' fixes a bug found in the code that is copied from P_1 to P_2 (as shown in Figure 1.1), then the duplicated buggy code in P_2 also needs to be fixed. Unfortunately, code copying is usually ad-hoc and is often not recorded, which makes it challenging to identify and correct every copy of the duplicated code. Unless the developer of P_2 addresses the known bug with Δ_2' or copies the updated code from P_1', P_2' still contains the bug which has been public since t_1. For example, consider the HFS and HFS+ filesystems, which both had the same buffer overflow vulnerability due to code duplication. The vulnerability in HFS filesystem was fixed in December 2009, but it wasn't until May 2012 that the same vulnerability was fixed in HFS+ filesystem—more than 2 years later [156]. Existing research also shows that 17–45% of software bugs are recurring bugs [128], meaning that resources are wasted in repeatedly diagnosing the same problems. In order to mitigate such recurring problems, we need a method to find all duplicated code of a known buggy version. For example, when a patch is released to fix a problem or to add a new feature, we can search for the same *pre*-patched original code that might be duplicated in other locations of OS distributions.

This raises the question of whether our current techniques can efficiently manage code bases at the scale of entire OS distributions, e.g., the stable version of Debian Squeeze containing over 16 GB of non-empty and non-comment code and spanning more than 348 million lines. In order to analyze the Linux kernel, all Debian and Ubuntu packages, and all C/C++ projects in SourceForge, we need techniques scaling to billions of lines of code. Scalable techniques would also enable us to promptly check for copies of known vulnerabilities in day-to-day development to improve the security of code.

1.4 Automatically Detecting Code Reuse

The thesis of this work is that *automatic code reuse detection enables an efficient data reduction of a high volume of incoming malware for downstream analysis and enhances software security by efficiently finding known vulnerabilities across large code bases.*

Code reuse detection is the process of finding common code sequences between programs. Automatic code reuse detection is a core component in many security scenarios and

has the most value when used on very large data sets. For example, security companies receive more than 403 million malware samples that they would like to analyze to determine how much malware is really unique. Similarly, we want to search entire OS distributions, e.g., Debian Squeeze consisting of 348 million lines of non-empty and non-comment code, for recurring software bugs.

We discuss how automatic code reuse detection is used in two security scenarios: malware triage and software vulnerability tracking at scale.

1.4.1 Malware Triage

In order to handle the exponentially increasing volume of malware, analysts need to allocate limited resources effectively, e.g., by prioritizing incoming malware. Given a set of malware, triage is performed to determine which malware needs to be analyzed first.

- Common triage tasks include identifying malware families via clustering [22] and identifying the neighbors nearest to a particular malware sample [69]. This requires measuring the similarity/distance between malware samples, e.g., two malware samples are similar if they have a significant amount of shared code.

- Semantic correlation between malware families allows analysts to reason about correlated malware groups and feature groups, e.g., common and distinguishing features between malware samples.

- A temporal ordering of malware samples provides a clue to derivative relationships among malware, e.g., malware `y` is a successor of malware `x`.

1.4.1.1 What Malware Are Related?

Given a set of malware samples, clustering is the process of grouping similar malware in such a way that malware in the same group are more similar to each other than to malware in other groups. In order to obtain an accurate clustering result, we need to calculate the pairwise similarity between malware samples and then group similar malware samples into the same cluster. The resulting clusters indicate that malware samples in the same group are more closely related to each other than to malware samples in other groups.

It is challenging to perform clustering at a large scale due to performance bottleneck, e.g., $\frac{N(N-1)}{2}$ times of pairwise similarity calculation where N is the number of malware samples. As we discuss in Chapter 2, one way to address this issue is to reduce the required

number of pairwise computations using locality sensitive hashing (LSH) [12] for probabilistic dimension reduction. At a high level, LSH hashes input malware samples such that similar malware samples are mapped into the same hash value with a high probability. However, LSH may have false positives, in which dissimilar malware samples are grouped in the same bucket, and false negatives, in which similar malware samples are separated into different buckets. For a more accurate clustering result, exact clustering can be performed within each bucket that has a much smaller number of malware samples; thus, the total number of pairwise comparisons will decrease. Another way to address this performance bottleneck is to make pairwise comparison faster, which can be done by approximating similarity calculation instead of reducing the number of pairwise computations. Minhashing [30] and Winnowing [146] are well-known algorithms to generate compact "fingerprints" from large documents while preserving the expected similarity of document pairs. These fingerprinting algorithms provide fast approximate similarity computation for finding similar malware groups quickly.

In Chapter 3, we propose a new fingerprinting algorithm using feature hashing [150, 159] with bit vectors for code resemblance detection. At a high level, feature hashing allows us to reduce a high dimensional input space into a low dimensional feature space; moreover, the use of compact bit vectors and bitwise operations significantly decreases cache misses. Therefore, similarity comparisons can be performed an order of magnitude faster than existing approaches, and we prove theoretically and empirically that our algorithm well-approximates the true similarity. This fast and accurate similarity calculation enables us to handle a large scale of input data efficiently. Furthermore, our algorithm has only one-sided error, i.e., false positives, while minhashing and winnowing can have both false positives and false negatives.

We implement BitShred [75], a system for fast similarity detection and clustering, based on our fingerprinting algorithm. We demonstrate BitShred's scalability and accuracy by evaluating it with a large number of real-world malware data sets. Our approach makes inter-malware comparisons in typical large-scale triage tasks, such as clustering and finding nearest neighbors, an order of magnitude faster than existing methods while using less memory. As a result, BitShred scales to current and future malware volumes where previous approaches do not. We have also developed a distributed version of BitShred on the Hadoop [1] where double the hardware gives almost double the performance. In our experiments, we demonstrate that BitShred can scale to clustering over 1.9 million malware per day.

1.4.1.2 Why Are They Related?

Given a set of malware samples, clustering only answers the question, "What malware are related?" In other words, clustering produces a set of malware groups where each group includes malware samples that are similar to each other in terms of similarity metrics, e.g., amount of shared code. For example, when malware `x` and malware `y` are grouped into the same cluster, we know that `x` and `y` share a significant amount of code. From the clustering results, we can tell which malware samples are similar and which malware samples are dissimilar.

However, clustering provides no semantic meaning of grouping. For example, we only know that malware `x` and malware `y` from the same cluster share a large amount of code. We do not know which features caused `x` and `y` to be grouped. From the malware analysts' perspective, it is valuable to understand what the common and distinguishing features are between `x` and `y`.

In Chapter 4, we propose a new technique for semantic analysis on the clustering results to answer the question, "Why are they related?" Traditional clustering alone acts like a blackbox, telling us only that malware samples grouped in the same cluster are similar. We move one step beyond traditional clustering to identify common and distinguishing features within the same group or among different groups. Common features can be main functionalities of a specific group or shared functionalities among multiple groups, whereas distinguishing features can be clues to understand how variants are generated within a group.

We extend BitShred to perform semantic analysis on the clustering results—BitShred-Semantic [75]. We utilize a co-clustering [35, 129] algorithm adapted to our fingerprinting algorithm to automatically mine correlated features and malware groups. The identified correlation enables analysts to reason about shared functionalities and unique operations presented by a few variants.

1.4.1.3 How Are They Related?

Given a set of malware samples, clustering groups similar samples into the same group and separates dissimilar samples into different groups. Co-clustering then identifies the semantic meaning of the grouping results.

However, neither clustering nor co-clustering consider temporal code reuse, i.e., code evolution for adapting to changing needs, bug fixes, and feature additions [99]. For example, if malware `y` is derived from malware `x`, the similarity between `x` and `y` can be high. Clustering will group both `x` and `y` into the same cluster, and co-clustering will report

the main parts of the malware as common features and updated parts of the malware as distinguishing features.

In Chapter 5, we propose new techniques to infer a temporal ordering of and derivative relationship between malware samples, which answers the question, "How are they related?" Software lineage—the evolutionary relationship among a set of software—provides a potentially rich source of information for a number of security questions. From the program analysts' perspective, it is very useful to know which program is the earliest version and how subsequent versions are derived in a cluster. For example, this information can provide a timeline of a collection of programs for forensics and help malware analysts understand trends over time.

We implement iLINE to automatically infer software lineage from program binaries and build iEVAL to scientifically measure the quality of a lineage inference algorithm [77]. We evaluate iLINE on two types of lineage—straight line lineage and directed acyclic graph (DAG) lineage—with large-scale real-world programs: 1,777 goodware spanning a combined 110 years of development history and 114 malware with known lineage collected by the DARPA Cyber Genome program [3]. Since there has been little study of software lineage inference algorithms and proper metrics, we also evaluate 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. Our evaluation shows that we can effectively identify software derivative relationships among program binaries for which we have no prior information with over 84% mean accuracy for goodware and over 72% mean accuracy for malware.

1.4.2 Software Vulnerability Tracking

Code duplication across different projects and different branches must be followed by coordination when a bug is fixed in one location. However, it becomes more complex to track all code changes, including bug fixes, as code and development teams grow in size. For example, parallel branching development enables developers to maintain multiple customized versions of software; however, this requires developers to ensure that bug fixes are in sync across all branches.

In Chapter 6, we propose a new technique using Bloom filters [29] for software vulnerability tracking based on code containment detection. Bloom filters are designed to efficiently perform a set membership test. At a high level, we use Bloom filters to approach a code containment problem (i.e., does program $\mathtt{p}$ contain code sequence $\mathtt{c}$?) as a

set membership problem (i.e., is code sequence c a member of program p?).

We implement ReDeBug [74], an architecture designed for unpatched code clone detection. ReDeBug is designed to scale to entire OS distributions, to support many different languages, and to guarantee zero false detections. Using ReDeBug, we examine over 2.1 billion lines of code from all packages in Debian Lenny/Squeeze, Ubuntu Maverick-/Oneiric, all C/C++ projects in SourceForge, and the Linux kernel on a commodity desktop, and identify 15,546 unpatched vulnerabilities in three hours. To the best of our knowledge, this is the largest set of bugs ever reported in a single paper.

1.5 Contributions

We focus on developing scalable and accurate techniques for automatic code reuse detection to examine a myriad of software for code security. This dissertation makes the following high-level contributions:

- We propose new scalable techniques for code resemblance detection and code containment detection. Our approaches enable code reuse detection that is an order of magnitude faster than existing techniques.
- We prove the correctness of our approximate code resemblance calculation theoretically and verify its accuracy empirically.
- We demonstrate the practical usages of code reuse detection in two security scenarios: malware triage and software vulnerability tracking.

Chapter 2

Background in Code Reuse Detection

Code reuse is not uncommon in software development. In an open source community, the same code, e.g., library code, is often reused in multiple places. In order to understand the current situation of code duplication, we performed a large-scale experiment to measure the overall amount of copied code in OS distributions. We measured this at two different granularities: the function level and the token (n-lines of source code) level.

First, for all C/C++ source files in the Debian Lenny code base, we roughly identified functions using the following Perl regular expression:

```
/^    \w+?\s[^;]*?   \(   [^;]*?\)\s*({   (?:[^{}]++|(?1))*})/xgsm
```

We realize that a regex may not be able to recognize all functions—that would require a complete parser. However, for our evaluation this is sufficient to provide an estimate of code duplication at the function level. We identified a total of 3,230,554 functions and measured their pairwise similarity using the Jaccard index. As shown in Figure 2.1, most of the function pairs had very low similarity (below 0.1), which is natural because different packages would have dissimilar code for different functionality. However, surprisingly, 694,883,223 pairs of functions had more than 0.5 similarities, and 172,360,750 of them were more than 90% similar. The result clearly shows a significant amount of code cloning, and this suggests that code duplication will continue to be important and relevant in the future.

Second, we calculated the total fraction of shared tokens in each file for the SourceForge data set. As shown in Figure 2.2, about 30% of files were almost unique (0–10% shared tokens). In contrast, more than 50% of files shared more than 90% of tokens with other files, which shows that code cloning is common within the SourceForge community as

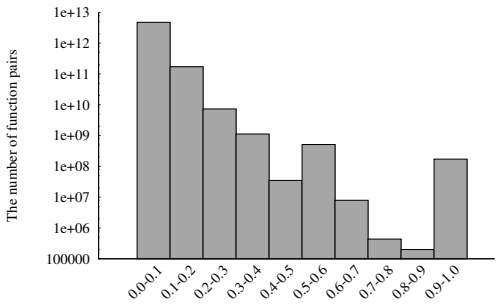


Figure 2.1: Similarity among functions

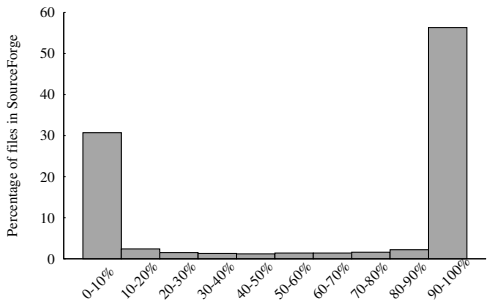


Figure 2.2: Fraction of shared tokens

well. Note that 100% of shared tokens in a file does not necessarily mean it is copied from another file as a whole. For example, this could also happen when a file consists of small fractions from multiple files.

Code reuse is also common among malware authors, who reuse existing malware to generate "new" variants [22, 69, 75].

2.1 What Features Do We Extract From Software?

2.1.1 Binary Abstraction

In many cases we may not have access to source code for the programs we want to analyze for inferring software relationships. Malware authors are unlikely to provide source code in order to aid malware analysis. It is also unlikely that proprietary software is provided with its source code, since the code can be used to study and modify the program. Most people typically do not have source code to the programs they run; however, they have an access to binary, i.e., executable code. For example, malware authors only publish the executable code. Commodity operating systems ship as binary code. Thus, it is often necessary to abstract binary code for subsequent analysis steps.

There are three primary binary program analysis methods: syntax-based analysis, static analysis, and dynamic analysis.

Syntax-Based Analysis. While syntax-based analysis may lack semantic understanding of a program, previous work has shown its effectiveness in classifying unpacked programs. Indeed, n-gram analysis is widely adopted in software similarity detection, e.g., [75, 84, 91, 146, 155]. The benefit of syntax-based analysis is that it is fast. This is because it does not require disassembling.

```
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 2.3: Example of feature extraction

• **n-grams.** An n-gram (or a shingle) is a consecutive subsequence of length n in a sequence. In order to abstract binary code, n-grams are obtained by sliding a window of n bytes over the extracted byte sequence of program code. For example, Figure 2.3b shows 4-grams extracted from Figure 2.3a. The size of n should be large enough that the probability of any given n-gram appearing in binary code is low. If n is too small, the similarity between binary code may be overestimated.

• **n-perms.** An n-perm is a variant of an n-gram such that an n-perm represents every possible permutation of n items in an n-gram. Since the order of n items in n-perms does not affect the matching process, n-perms analysis is expected to be more tolerant of reordering and to yield higher similarity scores. Karim et al. [84] proposed a malware phylogeny generation technique using n-perms to match every possible permuted code.

Static Analysis. Existing work utilized semantically richer features by first disassembling a binary. After reconstructing a control flow graph (CFG) of a program, each basic block can be considered as a feature [53, 59]. In order to maximize the probability of identi-

fying similar programs, previous work also normalized disassembly outputs by considering instruction mnemonics without operands [85, 87, 165] or instruction mnemonics with only the types of each operand (such as memory, a register, or an immediate value) [145]. The limitation with static analysis comes from the difficulty of getting precise disassembly outputs from obfuscated program binaries [103], e.g., inserting junk bytes at unreachable locations. In order to disassemble such obfuscated binaries, Kruegel et al. [96] proposed techniques that disassemble every address as the beginning of a new instruction, build a supergraph of the real CFG by invoking a recursive disassembler at jump target candidates, and prune invalid instructions.

- **Instructions.** Instruction sets can be obtained by disassembling the raw code in a hexadecimal byte sequence. Instructions represent computational operations performed by binary programs, and they include opcodes specifying types of operations and operands specifying arguments.

- **Basic blocks.** A basic block is a sequence of instructions that has one entry point and one exit point. In other words, no instruction within the basic block is a target of a jump instruction, and only the last instruction can change the execution flow to other basic blocks. Therefore, once the first instruction in a basic block is executed, the rest of the instructions in the basic block are executed sequentially. Gheorghescu [59] proposed a malware classification system based upon common basic blocks.

- **Call graphs.** A call graph represents the control flow between procedures in a directed graph where a node indicates a procedure and an edge (f_i, f_j) indicates f_i calls f_j. For example, Hu et al. [69] presented a function-call graph-based malware detection system where each malware is represented by its function-call graph and similar malware is detected using graph isomorphism. Call graph matching is expected to be less susceptible to deception by polymorphism than syntax-based matching.

- **Control flow graphs.** A control flow graph represents possible execution flows within a procedure. It can be represented as a directed graph consisting of nodes indicating basic blocks and edges (b_i, b_j) indicating a jump from b_i to b_j. For example, Zynamics BinDiff [167] is a tool that uses a similarity metric based on isomorphism between control flow graphs. Kruegel et al. [95] constructed a control flow graph from a network stream and identified structural similarities to detect a polymorphic worm.

Dynamic Analysis. The main difference between dynamic analysis and syntax/static analysis is that dynamic analysis runs programs while monitoring program execution and changes made to a system. Modern malware is often found in a packed binary format [63, 81, 107, 143], 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 [16, 22, 55, 140, 149]. 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 from spreading to other machines [142]. Well-known malware analysis systems are CWSandbox [5, 161], Cuckoo sandbox [62], TTAnalyze [23], and Anubis [71]. The limitation with dynamic analysis is that we can see a specific execution path instead of entire execution paths depending on the context. Attackers can even blacklist public malware analysis systems by tracking the domains and IPs of those systems[1]. This will make it more difficult to monitor malware behaviors.

Rossow et al. [142] discussed guidelines for designing malware experiments, including the compilation of correct data sets, transparency of experimental setups, ensuring realism of experiments, and safety or containment of experiments. Graziano et al. [61] proposed techniques to model network traffic of malware execution, which can mimic the network environment for the repeatability and the containment of malware execution. Kolbitsch et al. [90] presented an approach to mitigate stalling code problems, which are repeated "slow" operations causing significant slowdown on a monitoring system.

2.1.2 Source Code Representation

If we have access to source code, we can directly extract textual feature sets from source code or parse source code to obtain higher-level representations.

String. A program is divided into strings, typically lines. Similarity between two code fragments is then measured based on the amount of matched strings [17, 18, 48].

Token. A program is split into a sequence of tokens, e.g., variables, operators, keywords, and so on. A token-based approach is more robust against simple modifications, such as identifier renaming [80, 101].

[1] http://avtracker.info/

Tree. A program is parsed to build a parse tree or abstract syntax tree (AST), a tree-based representation of the tokens contained in the source code [21, 93, 109]. Similar code fragments can be identified by comparing subtrees [57, 112, 163].

Program Dependency Graph (PDG). A program is represented as a graph where a node denotes program statement/predicate and an edge means data/control dependence [56, 92, 94]. PDG-based similarity detection uses some variant of subgraph isomorphism to detect similar code fragments.

2.2 How Do We Measure Distance Between Software?

Once we obtain feature sets from programs through binary analysis, we can measure distance/similarity between two sets using set distance metrics. There are several distance metrics:

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:

$$\text{SD}(f_1, f_2) = |f_1 \setminus f_2| + |f_2 \setminus f_1|, \tag{2.1}$$

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.

The *Dice coefficient distance* is defined as:

$$\text{DC}(f_1, f_2) = 1 - \frac{2|f_1 \cap f_2|}{|f_1| + |f_2|}. \tag{2.2}$$

The *Jaccard distance* is defined as:

$$\text{JD}(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|}. \tag{2.3}$$

The *Jaccard containment distance* is defined as:

$$\text{JC}(f_1, f_2) = 1 - \frac{|f_1 \cap f_2|}{\texttt{min}(|f_1|, |f_2|)}. \tag{2.4}$$

In addition to these four distance measures, which are all symmetric, i.e., $\texttt{distance}(f_1, f_2) = \texttt{distance}(f_2, f_1)$, there is also an *asymmetric* distance measure, which can be used to cal-

culate dissimilarity between two sets. We call it the *weighted symmetric distance*, denoted

$$\text{WSD}(f_1, f_2) = |f_1 \setminus f_2| \times \mathtt{C_{del}} + |f_2 \setminus f_1| \times \mathtt{C_{add}} \tag{2.5}$$

where $\mathtt{C_{del}}$ and $\mathtt{C_{add}}$ denote a cost for deleting and adding a feature, respectively. Note that $\text{WSD}(f_1, f_2) = \text{SD}(f_1, f_2)$ when $\mathtt{C_{del}} = \mathtt{C_{add}} = 1$.

2.3 How Do We Group Similar Software?

Clustering is a task of grouping similar objects in such a way that objects in the same group are more similar to each other than to objects in other groups. Clustering is an unsupervised learning process where we do not have labeled data. Since we deal with unlabeled data, we need ground truth with which we can compare our clustering outputs and measure the accuracy of our clustering algorithm. There are two primary types of clustering algorithms: hierarchical algorithms and partitioning algorithms.

Hierarchical Clustering. Hierarchical clustering or agglomerative clustering algorithms start with each object in its own cluster. Then it combines the most similar pair of clusters and updates the similarity between the newly combined cluster with other clusters. The clustering stops when there remains only one cluster or when the similarity of remaining clusters is below a predetermined threshold.

Since a cluster consists of multiple objects, there are different ways of determining the similarity between clusters: single-linkage, complete-linkage, and average-linkage. In single-linkage clustering, the similarity between two clusters is defined as the similarity between their closest objects.

$$D(X, Y) = \max\{D(x, y) : x \in X, y \in Y\}$$

In complete-linkage clustering, the similarity between two clusters is defined as the similarity between their furthest objects.

$$D(X, Y) = \min\{D(x, y) : x \in X, y \in Y\}$$

In average-linkage clustering, the similarity between two clusters is calculated as the aver-

	f_1	f_2	f_3	f_4	f_5
s_1	0	1	0	1	0
s_2	1	1	0	0	0
s_3	0	1	1	1	1
s_4	1	0	0	0	1
s_5	1	1	0	0	0

Figure 2.4: Sample/feature matrix (Fingerprinting reduces $|F|$ and LSH reduces $|S|$.)

age similarity between objects from one cluster and objects from another cluster.

$$D(X, Y) = \sum_{(x,y)\in X\times Y} \frac{D(x, y)}{|X \times Y|}$$

Partitioning Clustering. Partitioning clustering constructs a partition of objects by assigning an object to the cluster which it best fits. A popular algorithm is k-means where we predetermine a desired number of resulting clusters k. k-means starts with k randomly selected cluster centroids. Then it assigns every object to the closest cluster centroid and adjusts the k cluster centroids accordingly. It stops when none of objects are moved to other clusters. The resulting clusters of the k-means algorithm are sensitive to the initial assignment of k cluster centroids. Therefore, k-means clustering typically runs with different seeds.

2.4 Can We Make Clustering Faster?

Bottleneck in Clustering As we discussed in §2.3, k-means algorithm requires a desired number of clusters k. However, when we cluster malware, it is not straightforward to determine the resulting number of clusters at the beginning. For this reason, hierarchical clustering is widely used for malware clustering [22, 75, 132].

Calculating the similarity between two malware samples is typically much slower than comparing two similarity values for sorting in hierarchical clustering algorithms. The main performance bottleneck in clustering is $\frac{n(n-1)}{2}$ comparisons. Suppose that we have a sample/feature matrix where a row represents a sample and a column represents a feature, as shown in Figure 2.4. In order to make clustering faster, we can choose one of two approximation methods.

First, if we can reduce the number of columns (features), we can speed up clustering

by having faster similarity calculation. Fingerprinting or feature dimension reduction algorithms, e.g., minhashing [30] and Winnowing [146], construct a compact representation of a sample. With smaller-sized signatures, similarity calculation can be performed more quickly.

Second, if we can reduce the number of rows (samples), we can speed up clustering by decreasing the required number of comparisons. Similar objects can be mapped to the same bucket with high probability by performing locality-sensitive hashing (LSH) [12]. Then exact clustering can be performed with a much smaller number of objects within each bucket. Bayer et al. [22] proposed the use of LSH for scalable malware clustering.

2.4.1 Fingerprinting Methods

Typically the feature sets extracted from binary code are large sets. For example, in the case of n-gram analysis with $n = 4$ bytes, there are $2^{32} = 4,294,967,296$ possible 4-grams. For more efficient set similarity detection, we can generate smaller-sized "signatures" by "fingerprinting" large feature sets. Such signatures allow us to more quickly perform similarity detection between sets by estimating the true similarity. There is a trade-off between efficiency and accuracy, i.e., the larger the signatures, the more accurate the similarity detection.

Minhashing. One popular approach is minhashing [30]. Suppose we have 4 programs $\{p_0, p_1, p_2, p_3\}$ and each program has features as described in Table 2.1. We first permute columns of the matrix, then the minhash of each program is the order of the first column that has a 1. For example, with the permutation $perm_1$, the minhash of p_0 is 1. Similarly, $h_1(p_1) = 0$, $h_1(p_2) = 1$, and $h_1(p_3) = 0$.

In order to construct a minhash signature, we select n number of random permutations of the columns. In Table 2.4, we have two random permutations. From these permutations, we compute the minhash signature matrix. The matrix initially consists of all ∞'s. In Table 2.2, $h_1(p_0) = 1$, $h_1(p_1) = 0$, $h_1(p_2) = 1$, and $h_1(p_3) = 0$, and we can fill the first column of the signature matrix using these values. In Table 2.3, $h_2(p_0) = 1$, $h_2(p_1) = 3$, $h_2(p_2) = 1$, and $h_2(p_3) = 0$, and we can fill the second column of the signature matrix accordingly. Then we have the minhash signature matrix, as shown in Table 2.4.

The signature matrix typically has fewer columns than the original matrix. We can estimate the Jaccard similarity from the signature matrix because the probability that the minhash values for two sets are the same as the Jaccard similarity of two sets. Given two sets p_1 and p_2, there are three types of columns: 1) T_1 having 1 in both columns, 2) T_2

	f_0	f_1	f_2	f_3	f_4
p_0	1	0	0	0	1
p_1	0	1	0	0	0
p_2	1	0	1	0	1
p_3	0	1	1	1	0

Table 2.1: Matrix representing feature sets

	f_1	f_4	f_3	f_0	f_2
p_0	0	1	0	1	0
p_1	1	0	0	0	0
p_2	0	1	0	1	1
p_3	1	0	1	0	1

Table 2.2: A permutation of columns (h_1)

	f_3	f_4	f_2	f_1	f_0
p_0	0	1	0	0	1
p_1	0	0	0	1	0
p_2	0	1	1	0	1
p_3	1	0	1	1	0

Table 2.3: A permutation of columns (h_2)

	h_1	h_2
p_0	1	1
p_1	0	3
p_2	1	1
p_3	0	0

Table 2.4: Minhash signature

having 1 in one column and 0 in another column, and 3) T_3 having 0 in both columns. The Jaccard similarity can be calculated as $\frac{|T_1|}{|T_1|+|T_2|}$. $|T_1|$ is the size of $p_1 \cap p_2$ and $|T_1| + |T_2|$ is the size of $p_1 \cup p_2$. The probability that we first meet T_1 column prior to T_2 column is $\frac{|T_1|}{|T_1|+|T_2|}$, which is the probability that $h(p_1) = h(p_2)$. If we first meet T_2 column before T_1, then $h(p_1) \neq h(p_2)$.

Therefore, using this compressed-sized signature matrix, we can estimate the Jaccard similarity between sets. For example, since p_0 and p_1 have no common minhash values in the signature matrix, we can estimate that the similarity between p_0 and p_1 is 0. When we compute the true Jaccard similarity from the original matrix, the similarity of p_0 and p_1 is 0 (correct). The estimated similarity between p_0 and p_2 is 2/2=1 from the signature matrix while the true Jaccard similarity is 2/3 (overestimated). The similarity of p_2 and p_3 is estimated to be 0 from the signature matrix while the true Jaccard similarity is 1/5 (underestimated). In order to acquire close-to-true Jaccard similarity, a large number of minhash values are required.

Winnowing. Schleimer et al. [146] presented a new document fingerprinting algorithm, Winnowing, to detect local matches. Winnowing has previously been widely used in the MOSS closed-source plagiarism detection tool [6].

Winnowing first divides a document into n-grams. All n-grams are hashed, and the selected subset of the hashes becomes the fingerprint to represent the corresponding document. Winnowing guarantees that any match of longer than or equal to t is detected, and

32 12 55 7 45 25 7 65 19 41 53 12
(a) Hashes of the 5-grams

(32 12 55 **7**) (12 55 7 45) (55 7 45 25) (7 45 25 **7**) (45 25 7 65) (25 7 65 19)
(7 65 19 41) (65 **19** 41 53) (19 41 53 **12**)
(b) Windows of hashes when $w = 4$

7 7 19 12
(c) Fingerprints selected by Winnowing

Figure 2.5: Document fingerprinting using Winnowing

any match of shorter than n is not detected. A user determines two parameters t and $n \leq t$.

Let $w = t - n + 1$ be a window size, and $h_1, h_2, ..., h_k$ the sequence of hashes from the document. Winnowing selects the minimum hash value in every window of $h_i, h_{i+1}, ..., h_{i+w-1}$. If there is more than one minimum hash value in a window, Winnowing selects the rightmost hash value. These selected hash sequences become the fingerprints of the document. Figure 2.5 shows how winnowing selects hashes when $w = 4$.

The expected fingerprints density of winnowing is $d = \frac{2}{w+1}$. This equation tells us that there is a trade-off between w and d or between t and the size of fingerprints. In other words, small size of fingerprints can detect only a long match.

2.4.2 Feature Hashing

In this thesis, we present a new efficient fingerprinting algorithm which allows us to reduce a high dimensional input space into a low dimensional feature space and utilizes a cache-friendly data structure and operations (§3.1). Our fingerprinting algorithm is based on feature hashing, which is an effective strategy for dimensionality reduction and practical nonparametric estimation [150, 159]. We employ feature hashing to increase scalability in malware comparison and to enable data mining on co-occurring features.

Our use of feature hashing was motivated by the observation that we observed less than 1% of the total features from 1,000 Windows XP binary files when n=4 bytes, as shown in Figure 2.6. We utilize feature hashing as an effective way of encoding features as a bit vector. Most existing implementations of bit vector Jaccard, e.g., the one found in python, assume that the feature space is completely encoded using index variables where feature 1 corresponds to bit 1, feature 2 to bit 2, feature 3 to bit 3, and so on. This scheme is impractical when the feature space is large, such as in our case where such an encoding

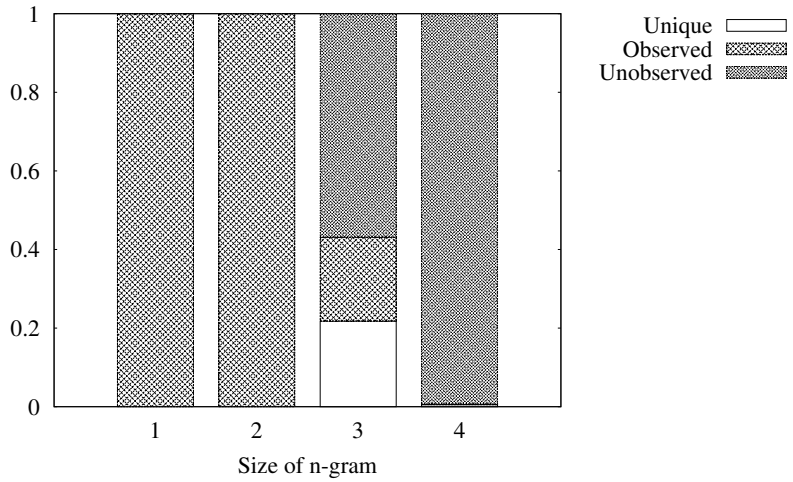


Figure 2.6: Fraction of unique, observed, and unobserved features

would result in a per-program data structure that is gigabytes in size.

Applications. Attenberg et al. [15] proposed a collaborative spam filtering system using a hashing trick to build a personalized global classifier. Hanna et al. [65] built a system to calculate similarity between Android applications based on feature hashing for identifying known buggy code, malware, and pirated applications. Asiaee T. et al. [14] used a hashing trick for Twitter sentiment analysis to compress a sparse feature vector into a lower-dimensional space to gain computational efficiency without significant loss of classification accuracy. Response prediction using collaborative filtering [113] and collaborative competitive filtering [164] utilized feature hashing to effectively handle large-scale sparse data for optimization.

2.4.3 Locality-Sensitive Hashing (LSH)

The main idea in LSH is to define a hash function h such that $h(s_1) = h(s_2)$ if the two programs p_1 and p_2 are similar [12]. The hash is run over all programs, and only those with unique hash values are compared. LSH is complementary to feature hashing; while LSH reduces the number of items, feature hashing reduces the number of features. Despite this complementarity, our evaluation shows that the exclusive use of feature hashing outperforms the exclusive use of LSH by a factor of 2 to 1. Previous work has shown that this bears on theoretical analysis as well [150].

The size of feature space ($2^{128} \approx 10^{38}$ when n=16 bytes) is typically much larger than

the number of malware samples (10^6 per day according to [154]) in malware analysis. Thus reducing feature space is preferable to reducing sample size. While in our evaluation we focus on the effects of each algorithm independently, both feature hashing and locality-sensitive hashing could be combined in a real system.

2.5 How Do We Semantically Group Software?

Clustering alone acts like a blackbox, telling us only that programs are grouped in the same cluster because they are similar. From the analysts' perspectives, it is very useful to understand *why* programs are grouped into the same cluster or separated into different clusters.

2.5.1 Co-clustering

Co-clustering [35, 129] (aka bi-clustering) goes one step further and tells us why programs are similar by simultaneously clustering features as well as programs. For example, co-clustering allows us to group two programs and to identify the feature sets that explain why they are similar (e.g., a significant amount of shared code) and why they are different (e.g., contacting different command and control servers).

Co-clustering provides more general cross-program and cross-feature correlation than the above approaches. Note also that co-clustering allows for a substantially richer analysis than the nearest-neighbor algorithms. Nearest-neighbor algorithms typically only provide a set of "neighboring" programs that is similar to an input sample (and may provide some information on the amount of structural similarity between the "neighboring" programs). Co-clustering results, on the other hand, provide information on the structural similarity across *all* programs analyzed, i.e., they illustrate which groups of features co-occur with which groups of programs for the entire data set.

Formally, n programs $\{s_0, s_1, ..., s_n\}$ that have m features $\{f_0, f_1, ..., f_m\}$ can be described as $n \times m$ matrix M where each row represents a program and each column represents a specific feature. Let S and F be discrete random variables that take values from $\{s_0, s_1, ..., s_n\}$ and $\{f_0, f_1, ..., f_m\}$, respectively; then $n \times m$ matrix can be described as the joint probability distribution between S and F, $p(S, F)$. Given the desired number of disjointed row groups k and the desired number of disjointed column groups l, co-clustering finds mapping $\hat{S} : \{s_0, s_1, ..., s_n\} \rightarrow \{\hat{s_0}, \hat{s_1}, ..., \hat{s_k}\}$ and $\hat{F} : \{f_0, f_1, ..., f_m\} \rightarrow \{\hat{f}_0, \hat{f}_1, ..., \hat{f}_l\}$

Cost Functions & Distributed Approach. The goal of co-clustering is to find the "best" row/column group assignment such that a cost function becomes minimized. Since finding a globally optimal co-clustering is NP-hard, algorithms typically perform a local optimal search [46].

Dhillon et al. proposed information-theoretic co-clustering by considering co-clustering as a problem of maximizing the mutual information between the clustered random variables—row-clusters and column-clusters [46]. Ramanathan et al. proposed a parallel information-theoretic co-clustering utilizing framework for rapid implementation of data mining engines (FREERIDE) [137]. Huang et al. presented a scalable ensemble information-theoretic co-clustering algorithm to improve the accuracy of co-clustering results [70]. They first generate multiple different co-clusters using a MapReduce framework and then combine them to produce a final co-cluster with an ensemble clustering method. In information-theoretic co-clustering, an optimal co-clustering minimizes $I(S;F) - I(\hat{S};\hat{F})$ where $I(S;F)$ denotes the mutual information. For a fixed distribution p, $I(S;F)$ is fixed so that minimizing the loss in mutual information is maximizing $I(\hat{S};\hat{F})$. The loss in mutual information can be calculated as $D_{KL}(p(S,F)||q(S,F))$ where $D_{KL}(\cdot||\cdot)$ denotes the Kullback-Leibler divergence (aka relative entropy or information gain) and $q(s,f) = p(\hat{s},\hat{f})p(s|\hat{s})p(f|\hat{f})$.

Chakrabarti et al. proposed a cross-association algorithm where co-clustering finds homogeneous rectangular regions by minimizing code description length [35]. Papadimitriou and Sun designed a distributed co-clustering algorithm based on the cross-association algorithm using a MapReduce framework [129]. In a cross-association algorithm, the total code length is defined as the sum of $log^*k + log^*l$ (number of desired groups), $\sum_{i=1}^{k-1}\lceil log\bar{a_i}\rceil + \sum_{j=1}^{l-1}\lceil log\bar{b_j}\rceil$ (number of rows/columns in each row/column group), $\sum_{i=1}^{k}\sum_{j=1}^{l}\lceil log(a_ib_j+1)\rceil$ (number of ones in the matrix), and $\sum_{i=1}^{k}\sum_{j=1}^{l}C(D_{i,j})$ (number of bits to encode). The total number of bits sent for each cross-association is calculated as $C(D) = \sum_{i=0}^{1} n_i(D)\ log(\frac{n(D)}{n_i(D)})$ where $n_0(D)$ denotes the number of nonzero entries, $n_1(D)$ denotes the number of zero entries, and $n(D) = n_0(D)+n_1(D)$ in D. In addition, $\bar{a_i} = (\sum_{t=i}^{k} a_t) - k + i, 1 \leq i \leq k-1$ and $\bar{b_j} = (\sum_{t=j}^{l} b_t) - l + j, 1 \leq j \leq l-1$ where a_i and b_j denote dimensions of $D_{i,j}$.

Cho et al. defined co-clustering as minimizing the total squared residue which is the sum of squared distance between each point and the mean (or row mean and column mean) of a co-cluster [38]. Zhou and Khokhar proposed a parallel co-clustering algorithm based upon a sum-squared distance using MPI [166]. Let m_{ij} be the (i,j)-th element of matrix M. Two types of residues of an element m_{ij} are defined by co-cluster index I and J: $h_{ij} = m_{ij} - m_{IJ}$ and $h_{ij} = m_{ij} - m_{iJ} - m_{Ij} + m_{IJ}$ where $m_{IJ} = \frac{\sum_{i\in I, j\in J} m_{ij}}{|I|\cdot|J|}$ is

the mean of all the entries in the co-cluster, $m_{iJ} = \frac{\sum_{j \in J} m_{ij}}{|J|}$ is the mean of the entries in row i of the co-cluster, and $m_{Ij} = \frac{\sum_{i \in I} m_{ij}}{|I|}$ is the mean of the entries in column j of the co-cluster. Then the goal of co-clustering is to minimize the total squared residue $||H||^2 = \sum_{I,J} ||H_{IJ}||^2 = \sum_{IJ} \sum_{i \in I, j \in J} h_{ij}^2$ where H_{IJ} means the residue of the co-cluster determined by I and J. In other words, to goal is to minimize the Frobenius norm of the residue matrix H.

Angiulli et al. proposed a greedy search approach to co-clustering where the quality of homogeneity is measured based on squared rows and columns means together with the size of the co-cluster [13]. The quality of each co-cluster indexed by I and J is calculated as $Q(I, J) = \frac{\sum_{i \in I} (m_{iJ})^2 + \sum_{j \in J} (m_{Ij})^2}{|I| + |J|} \times \sum_{i \in I, j \in J} m_{ij}$.

Applications. Co-clustering has been studied in many applications, e.g., text mining to find similar documents with related word clusters [13, 46, 70], graph mining to summarize the underlying structure of object associations [35], analysis of gene expression data to capture the trends of genes over a subset of experimental conditions [38], and collaborative filtering to predict a user's preferences based on other users' preferences in recommendation systems [58].

We can benefit from co-clustering even if we want to cluster one dimension of a matrix with sparse and high-dimensional data [46]. For example, Dhillon et al. demonstrate that co-clustering is more effective than one-sided clustering even if we want cluster documents with a word-document matrix [46]. This is because co-clustering implicitly performs dimensionality reduction while using word clusters instead of individual words as underlying features. A more detailed discussion on co-clustering can be found in survey [106].

2.6 How Do We Detect Temporal Code Reuse?

Software *evolves* to adapt to changing needs, bug fixes, and feature additions [99]. 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 [60, 162] as well as its effect on vulnerabilities in Firefox [108]. The security community has also studied malware evolution based on the observation that the majority of newly detected malware are tweaked variants of well-known malware [22, 69, 75]. With over 1.1 million malware appearing daily [154], researchers

have exploited such evolutionary relationships to identify new malware families [87, 105], create models of provenance and lineage [49], and generate phylogeny models based upon the notion of code similarity [84].

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 [22, 69, 75], 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 their evolution. In software security, lineage can help track vulnerabilities in software for which we do not have source code. For example, if we know that a vulnerability exists in an earlier version of an application, then it may also exist in applications that are derived from the earlier version. Such logic has been fruitfully applied at the source level in our previous work [74]. 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 [3] to study them.

We have established that the need for the automatic and accurate inference of software lineage is an important open problem. Now 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 2.6.1. 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 $(x, y) \in A$ 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 [26]. Notice that in a tree each pair of nodes must have a unique LCA, but in a DAG some pairs of nodes can have multiple LCAs.

2.7 Applications

2.7.1 Malware Classification & Clustering

Software similarity detection can be used to identify copied or related code among malware samples. Once copied code (and unique code) is identified, there are many subsequent uses, e.g., using the code as feature vectors in machine learning algorithms that recognize new malware [91]. Similarly, software similarity detection can be utilized to cluster malware: if two malware samples contain shared code, they are likely to be part of the same overall family [84]. We can analyze malware samples to determine how much malware is really unique.

There has been extensive research that proposes the need for large-scale malware analysis and triage, e.g., [22, 69, 84, 131, 140]. The main difference in our work is scalability and co-clustering to automatically identify semantic features. Our work builds on per-sample analysis and feature extraction, which is an extremely active area of research.

Syntactic Feature-based Analysis. Schultz et al. [147] proposed a data mining method to detect malware based on PE header information such as DLLs, strings, and byte sequences. Abou-Assaleh et al. [11] applied n-gram analysis to detect malware. Kolter and Maloof [91] suggested a classification method based upon 4-grams. Karim et al. [84] proposed a malware phylogeny generation technique using n-perms to match every possible permuted code. Wicherski [160] proposed peHash to calculate a per-binary specific hash value based on structural information of binary and to cluster polymorphic malware.

Static Feature-based Analysis. Zynamics BinDiff tool [167] and Hu et al. [69] proposed malware similarity methods that are based upon isomorphism between control flow and function call graphs. Although graph-based isomorphism is expensive, it is less susceptible to being fooled by polymorphism [69]. Ye et al. [165] designed a system to cluster malware based on instruction frequencies and function-level instruction sequences. They utilized a cluster ensemble to aggregate hybrid hierarchical clustering (hierarchical clustering and k-medoids clustering at each iteration) on instruction frequencies and weighted k-medoids clustering on instruction sequences. Neugschwandtner et al. [127] proposed a malware sample selection method to maximize information gain when the sample is analyzed. Their method is based on static features, such as peHash [160], static cluster from [72], PE header information, Antivirus labels, and sample submitter information.

Dynamic Feature-based Analysis. Bailey et al. [16] proposed a behavior-based malware classification and clustering technique. They define the behavior of malware in terms of system state changes, i.e., abstraction of system calls, and use normalized compress distance as a distance metric. Rieck et al. [138–140] proposed a system for malware clustering and classification based on behaviors observed by CWSandbox [161]. They encoded observed behaviors into malware instruction sets, extracted n-grams over the encoded instruction sets, and embedded the n-grams into a vector space. Prototype-based and incremental clustering were used for scalability. Bayer et al. [22] performed large-scale malware clustering based on dynamic behavior profiles, including system calls and network activities [71] and LSH.

Model-based Analysis. Christodorescu et al. [39] proposed a malware detection algorithm based on a formal semantic-aware template to detect variants of malware by mitigating common obfuscations. Fredrikson et al. [55] proposed a technique to extract discriminative specifications to describe unique properties for a class of malware based upon dynamic behavior analysis. Lanzi et al. [98] proposed an anomaly behavior-based malware detection system based on a system-centric approach to model interactions of malware with persistent OS resources, such as file systems and registries.

Reputation-based Analysis. Chau et al. [37] proposed a technique to detect malware based on file reputation using the belief propagation algorithm. File reputation is related to the prevalence of the file and the reputation of hosting machines. Rajab et al. [136] designed a reputation-based malware detection approach integrated into a web browser where the reputation of a domain or signer is determined by its history of maliciousness for a certain period.

Structure-based Analysis. Perdisci et al. [132] explored clustering malware based on similar network behavior interacting with the Web. They utilized both statistical similarity and structural similarity of HTTP traffic. Šrndić and Laskov [152] proposed a static method to detect malicious PDF files based on differences in the structural properties of malicious and benign PDF files.

Textual Feature-based Analysis. Rahman et al. [135] designed a system to detect fake, annoying, and possibly damaging posts on social media (a.k.a. socware) using a Machine

Learning classifier based on spam keyword scores, message similarity, number of wall/news feed posts, and number of likes/comments for a URL embedded in a post.

Clustering Quality. Li et al. [100] pointed out difficulties in evaluating malware clustering results. Along with [142], they argued that balanced and carefully designed data sets are important for validating clustering accuracy. Perdisci and U [133] proposed a method to assess the quality of malware clustering results by building a more representative reference set. In order to handle the inconsistencies among different antivirus (AV) labels, they built an AV label graph, an undirected weighted graph where a node denotes an AV label of an AV scanner, an edge denotes two nodes (AV labels) assigned to at least one sample, and a weight is calculated by the frequency of co-occurrences.

Packing and Obfuscation. In order to cope with packed malware, unpacking has been actively studied [47, 63, 81, 107, 143, 149]. For example, Perdisci et al. [131] presented a classification method between packed and non-packed PE files exploiting PE header information. Portable Executable Identifier (PEiD) [79] is used to detect packed binary code. PEiD uses a signature database to detect packing and encrypting methods so that a signature database has to be updated to identify new packing methods. After PEiD identifies the packing methods, this information can be used to unpack binary code. Royal et al. [143] presented a behavior-based hidden-code extraction technique from malware samples, and implemented a tool called PolyUnpack. PolyUnpack takes a malware sample as an input and performs static analysis to generate a static code model for each sample; then PolyUnpack carries out dynamic analysis while running a malware sample in an isolated environment. By comparing the runtime behavior of a malware sample with its static code model, PolyUnpack is able to extract hidden code of a malware sample. Jacob et al. [72] proposed a method to identify similar malware samples that might be packed. According to Jacob et al., current packers typically employ compression and weak encryption schemes so that certain properties of binary files are preserved even after packing is applied. They utilized PE structural characteristics, code signal, and bigram distribution over raw byte sequences. Coogan et al. [42] proposed a method to approximate original code from virtualization-obfuscated code. From execution traces, they identify relevant instructions that affect system calls, e.g., values of arguments and conditional control flow.

2.7.2 Code Clone & Clone-related Bug Detection

Code similarity detection has been utilized by many researchers to detect code clones. Well-known examples include CCFinder [80], CP-Miner [101], Deckard [78], and DejaVu [57]. This line of research uses a variety of matching heuristics based on high-level code representations such as CFGs and parse trees. For example, CCFinder [80] generates a token sequence from a program using a lexer and transforms the token sequence based on language-dependent rules. A suffix-tree-based matching algorithm is then used to determine similar code. CP-Miner [101] detects copy-paste related bugs which can be caused by inconsistent modification on copy-pasted code fragments. It parses a program, hashes its tokens into numeric values, and then runs the frequent subsequence mining algorithm to detect clone-related bugs. CP-Miner found 49 bugs in Linux that were due to developers not fixing copied buggy code [101]. Deckard [78] and DejaVu [57] both build parse trees and represent structural information of a parse tree as a vector, then cluster the vectors with respect to the Euclidean distance. An advanced heuristic matching, however, can suffer from a higher false detection rate. For example, 73% of bug reports from CP-Miner and 37% of bug reports from DejaVu were false code clones. Furthermore, implementing good parsers is a difficult problem with which even professional software assurance companies struggle [28]. In order to process a large-scale of data sets, DejaVu utilized a cluster with 5 nodes to examine 75 million lines of commercial code. Livieri et al. [104] designed a distributed CCFinder where they analyzed FreeBSD spanning 400 million lines of code in two days using 80 workstations.

Programmers do not often recognize some duplicated code when performing modifications. Propagating modifications is necessary in order to synchronize changes in all of the clones [97]. Software similarity detection can be used to identify code similar to a known buggy version. Pham et al. [134] studied vulnerability reports and categorized three types of recurring problems: reusing implementation code, sharing common APIs/libraries, and reusing algorithm or design. They proposed a method to extract object usage models showing a set of related entities from vulnerable code to find matches. Nguyen et al. [128] conducted an empirical study where seven experienced programmers inspected several thousands of fixing changes and 17–45% of total fixing changes were recurring. They proposed a method to identify code peers with similar object interactions and to recommend fixes to other peers. For example, when a patch is released, programmers can search for the same buggy code in entire OS distributions, such as Debian Squeeze consisting of 348 million lines of non-empty and non-comment code. Code Assurance (aka Patch Miner) [130] is a

commercial tool to find recurring bugs.

MOSS [146] is a well-known syntactic similarity detection tool using n-grams. MOSS is based on an algorithm called Winnowing [146], a fuzzy hashing technique that selects a subset of n-grams to find similar code. SYDIT [112] is a program transformation tool that characterizes edits as AST node modifications and generates context-aware edit scripts from sample edits. It was tested on a data set of 56 pairs of sample edits from open source projects in Java, and perfectly mimicked developer edits on 70% of the targets. Comprehensive comparisons of code clone detection tools are presented at [25], and the pros and cons of code clones are discussed at [82].

2.7.3 Software Lineage Inference

Existing work has primarily focused on analyzing known lineage, not inferring lineage. For example, Belady and Lehman [24] studied the software evolution of IBM OS/360, and Lehman and Ramil [99] formulated eight laws describing the software evolution process: 1) continuing change, 2) increasing complexity, 3) self-regulation, 4) conservation of organizational stability, 5) conservation of familiarity, 6) continuing growth, 7) declining quality, and 8) feedback system. Xie et al. [162] analyzed histories of open source projects including Samba, Bind 9, OpenSSH, SQLite, Vsftpd, Sendmail, and Quagga in order to verify Lehman's laws of software evolution and confirmed that continuing change, increasing complexity, self-regulation, and continuing growth are still valid in today's open source projects. However, the other four laws were not confirmed because of either lack of data or imprecise definitions. Godfrey and Tu [60] investigated the Linux kernel to understand a software evolution process in open source development systems. They found that Linux has grown at a super-linear rate despite its huge size and its development model, which was possible because most of the code can be developed independently, e.g., device drivers. Kim et al. [89] studied the history of code clones to evaluate the effectiveness of refactoring on software improvement with respect to clones. Shihab et al. [151] evaluated the effects of branching in software development on software quality with Windows Vista and Windows 7.

Massacci et al. [108] studied the effects of software evolution, such as patching and releasing new versions, on vulnerabilities in six major versions of Firefox. They found that inherited vulnerabilities from the previous versions constituted almost half of the existing vulnerabilities in Firefox due to a large fraction of code reuse. This indicates that software evolution may not be able to resolve all known vulnerabilities. Davies et al. [43] proposed

a signature-based matching of a binary against a known library repository to identify library version information, which can potentially be used for security vulnerabilities scans. Edwards and Chen [51] statistically verified that an increase of security issues identified by a source code analyzer may indicate an increase of exploitable bugs while examining histories of Sendmail, Postfix, Apache httpd, and OpenSSL.

Gupta et al. [64] studied malware metadata including text descriptions and dates collected by an anti-virus vendor to describe evolutionary relationships among malware. Dumitras and Neamtiu [49] studied malware evolution to find new variants of well-known malware. Karim et al. [84] generated phylogeny models based on code similarity to understand how new malware were related to previously seen malware. Khoo and Lio [87] investigated FakeAV-DO and Skyhoo malware families using a phylogenetic method to understand evolution and to identify families. Ma et al. [105] studied the diversity of exploits used by notorious worms to identify families by constructing dendrograms. Ma et al. also found non-trivial code sharing among different families. Lindorfer et al. [102] investigated malware evolution by comparing subsequent versions of malware samples that were collected by exploiting embedded auto-update functionality. By measuring the number of added/deleted/shared basic blocks between behavioral components from two versions, they observed incremental updates that reused most of the code. In addition, they found a correlation between the release of new versions and detection by a larger number of AV engines.

Hayes et al. [67] pointed out the necessity of systematic evaluation in malware phylogeny systems and proposed two models to artificially generate reference sets of samples: a mutation-based model and a feature accretion-based model. They generated 19 sets with 15 samples from Agobot. The accuracy is measured based on nodal distance, which is the sum of the differences in path lengths between two graphs.

Part I

Code Reuse Detection at the Binary Code Level

Chapter 3

Malware Triage via Code Resemblance Detection

The volume of new malware, fueled by easy-to-use malware morphing engines, is growing at an exponential pace [154]. In 2011 Symantec received over 403 million unique malware samples, which means over 1.1 million unique variants of malware are created every day [154]. The sheer volume of malware means we need automatic methods for large-scale malware triage techniques and systems.

At a high level, triage has two steps. First, per-sample malware analysis is run on each sample to extract a set of features. Second, malware are compared in a pairwise fashion to determine similarity, e.g., by using the Jaccard distance. Once we determine which malware are similar, and what similarities and differences they have relative to known malware cases, we can use triage to make informed decisions. For example, triage may be used to perform further in-depth analysis on one representative malware sample per family in cases where it would be cost-prohibitive to do analysis on the entire data set.

In this section, we present BitShred [75], a system for large-scale code similarity analysis and clustering. BitShred's key feature is its agnosticism to the particular per-software analysis routine, even when the extracted feature set has a very large feature space. This property is useful in malware analysis because malware authors and defenders are caught in a cyclic battle in which defenders invent ever-more advanced and accurate per-malware analyses for feature extraction, which are then defeated by new malware obfuscation algorithms. This cyclic battle underscores the need for malware triage techniques that allow us to plug in the latest or most appropriate analysis for feature extraction. We empirically show that BitShred meets the desired requirements by demonstrating BitShred on two previously

proposed per-sample analyses: dynamic behavior analysis from Bayer et al. [22] where the feature space is 2^{17}, and static code reuse detection as proposed in [11, 91, 158] where the feature space is 2^{128}.

The main issues for handling large volumes of malware are (a) efficiently representing malware features (so we can fit more data in main memory without paging), (b) comparing feature sets between malware, and (c) determining which features are correlated for malware groups. For a sense of scale, consider that currently over 1.1 million new malware are observed per day, which require about 605 billion comparisons to find families using hierarchical clustering. If we perform n-item analysis when $n = 16$ bytes, an exact representation of the features would require 2^{128} (2^{95} *gigabytes*) per sample. We could not perform all 605 billion comparisons on previous data structures in 24 hours on a single CPU.

The central idea in BitShred is to use feature hashing [150, 159]. Feature hashing allows for dramatic dimensionality reduction, so the hashed representation takes less room in memory, and is also L1/L2 cache efficient. However, feature hashing introduces collisions in the reduced feature space. For example, if we use a hash function that compresses the 2^{128} feature space down to 2^{18}, there will be an enormous number of collisions in the feature space. The surprising thing with feature hashing is that we need just a single hash function for the dimensionality reduction of the feature space. BitShred's advantages—the requirement of a single hash function and the dimensionality reduction—have immediate performance implications. Our result is backed by theory and experimentation which show that pairwise comparison and algorithms built on top like hierarchical clustering will be close to exact.

Contributions. Our main contribution is a system for performing the triage tasks described above that scales to data sets which are orders of magnitude larger than what existing approaches are able to handle. We present a theoretical analysis showing that feature hashing with the Jaccard offers near optimal results, and build a real system called BitShred that is independent of the particular per-malware analysis engine and works even for high-dimensional feature sets. We extensively evaluate BitShred's scalability, speed, and accuracy using two different per-sample analyses: code similarity and dynamic behaviors. Our performance evaluation shows that BitShred can cluster over 116,000 malware per day on a single node, and over 1.9 million per day on a Hadoop cluster where we develop an optimal schedule that minimizes communication overhead and provides uniform node work.

3.1 Fingerprinting for Resemblance Detection

We focus on any analysis that outputs a set of features that are Boolean, or that can be encoded as Boolean variables. For example, in code reuse detection the features are the presence or absence of a code fragment. Real-value features can be encoded via bucketizing, where a Boolean feature is true if the feature falls within a particular bucket. This allows us to plug in many types of analyses. As a new analysis is developed, we can continue to use BitShred by simply defining a hash function over the particular features extracted by the analysis.

We compute code similarity using the Jaccard similarity metric. The Jaccard calculates the percentage of common features, with the idea that the more features the malware share, the more alike the malware are, and the Jaccard is used extensively in previous work [22, 132]. More formally, given two feature sets g_a and g_b for programs p_a and p_b respectively, the Jaccard similarity (i.e., index) is:

$$J(g_a, g_b) = \frac{|g_a \cap g_b|}{|g_a \cup g_b|}. \tag{3.1}$$

In order to motivate feature hashing, consider first using a standard implementation of Jaccard, e.g., as found in SimMetrics [36]. The advantage of this approach is that the size of the feature data structure is linear in the number of features a program actually presents. For example, if our feature space is in size 2^{128} but a particular program only has 2^{30} features, the data structure is still only 2^{30} in size. Unfortunately, the set union and intersection operations themselves are a bottleneck in similarity calculation. In our experiments, we could only cluster about 2,388 malware per day using this approach (§3.5, labeled as exact Jaccard).

We take an approach of encoding features as a bit vector so that the Jaccard similarity can be calculated by fast CPU-friendly logic operations:

$$J_{bv}(f_a, f_b) = \frac{S(f_a \wedge f_b)}{S(f_a \vee f_b)} \tag{3.2}$$

where f_i is the bit vector representation of the feature set for program p_i and $S(\cdot)$ counts the number of set bits.

Feature hashing [150, 159] is a specific way of encoding features as a bit vector. Most existing implementations of bit vector Jaccard, e.g., the one found in Python, assume that the feature space is completely encoded using index variables where feature 1 corresponds

to bit 1, feature 2 to bit 2, feature 3 to bit 3, and so on. This scheme is impractical when the feature space is large, such as in our case where such an encoding would result in a per-program data structure that is gigabytes in size.

Others have proposed malware similarity methods that do not use boolean features. For example, the Zynamics BinDiff tool [167] and Hu et al. [69] use a similarity metric based on isomorphism between control flow and function call graphs. While we can compute call graph similarity based upon features, e.g., how many basic blocks are in common, our approach cannot readily be adapted to actually compute the isomorphism.

Classification vs. Clustering. Classification uses *labeled* samples to learn a rule for assigning labels to new samples. Feature hashing was previously used to build an efficient spam classifier [15], which is trained with labeled (i.e., spam/not-spam) emails and then determines whether an incoming email is spam or not. Clustering, on the other hand, groups *unlabeled* samples based on given similarity metrics. In our setting, (unlabeled) malware are grouped based on similar features, such as static code or dynamic behaviors. To the best of our knowledge, ours is the first study to introduce clustering techniques that combine feature hashing with the Jaccard and to present a theoretical proof of correctness.

Security. Feature hashing, like LSH, uses a hash function which must be kept secret. If the hash function is known, then an attacker may be able to "fool" the algorithm into an atypical number of collisions, thus potentially reducing the overall accuracy. This problem is mitigated by using a keyed hash function, e.g., picking a secret key k and computing $h(k||s_a||k)$ for item s_a. For simplicity, in the rest of this paper we refer to the hash as simply h instead of a keyed variant, with the expectation that it is used as a keyed function for security.

3.2 BitShred Architecture

In this section we describe BitShred, our system for fast similarity analysis and malware triage on very large malware data sets. We focus on the performance of our hash feature approach in comparison to previously proposed methods, such as straight set-based analysis, Winnowing, and locality-sensitive hashing. While there are data reduction techniques that reduce the size of s, e.g., locality-sensitive hashing, we can expect that even after data reduction the number of malware we need to cluster will continue to increase rapidly. Another triage task is automatically identifying the nearest neighbors, which requires that

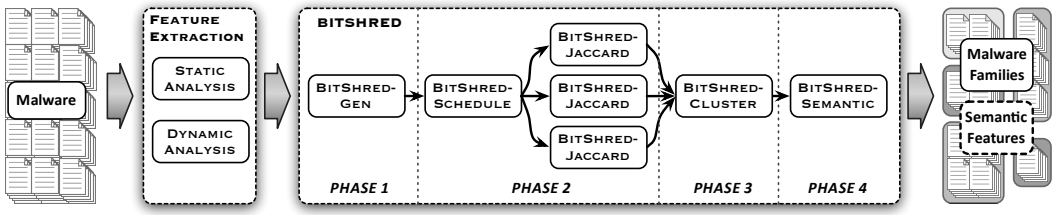


Figure 3.1: BitShred Overview

given a sample m, we compute its distance to all other malware. An exacerbating issue is that we want analysis which extracts many features, which in turn creates extremely high-dimensional feature sets, and this, in turn makes each comparison more expensive. If we can make each comparison as efficient as possible, then we will scale better even in worst case scenarios.

Our main conclusion is that BitShred provides the same accuracy but better performance. Better performance means that we can scale to much larger malware volumes and deal with current volumes much more quickly. We also show that BitShred can be parallelized, which allows us to take advantage of infrastructures like supercomputers and Hadoop as performance requirements exceed that which can be provided by a single CPU. Finally, we show BitShred in the context of an end-to-end system for malware triage. In our data set, BitShred is more than 90% accurate in automatically identifying malware families. While malware authors can always add more obfuscation and make analysis harder, thereby decreasing the accuracy of any system, the core concepts in BitShred can "plug in" any malware analysis that outputs Boolean or (binary-encoded) integer-valued values, and speed it up while retaining similar accuracy to the exact Jaccard.

Throughout the rest of this paper, we use s_i to denote malware sample i, G to denote the set of all features, and g_i to denote the subset of all features G present in s_i.

3.2.1 BitShred Overview

At a high level, BitShred takes in a set of malware, runs per-malware analysis, and then performs inter-malware comparison, correlation, and feature analysis, as shown in Figure 3.1. BitShred's job is to speed up subsequent correlation after using existing techniques to perform per-sample feature extraction. In our implementation, we experiment with using n-grams as proposed in [11, 91, 158] because the feature space is extremely large, and dynamic behavior analysis from Bayer et al. [22] because it has been shown to be effective.

BitShred's key role is the use of feature hashing to compactly represent even high-

dimensional feature sets in a bit vector. We call the bit vector the malware *fingerprint*. The algorithm that calculates the malware fingerprint using feature hashing is called BITSHRED-GEN in Figure 3.1. We then replace the existing exact inter-malware feature set comparison (called the Jaccard index) with an approximation algorithm called BITSHRED-COMPARE that is just as accurate (with high probability), yet is significantly faster. In particular, the main bottleneck with the Jaccard distance computation is that it requires a set intersection and union operation with the entire feature space. BitShred's algorithm replaces set operations with bit vector operations, which are orders of magnitude more efficient. We then perform clustering using BITSHRED-CLUSTER to identify families.

We use full hierarchical clustering as a representative computationally expensive triage task. Hierarchical clustering has a lower bound of $s(s-1)/2$ comparisons for s malware to cluster [52]. Other problems, such as incremental clustering and finding the nearest neighbor, are algorithmically less expensive. For example, incremental clustering, or comparing incoming newly reported s' malware against s malware in a database, requires $s' \times s$ comparisons where $s' \ll s$. The nearest neighbors to malware s_i can be identified by comparing it to all other samples which are linear in s.

Without loss of generality, we focus on the representative setting where incoming malware is clustered by similarity, as shown in Figure 3.1. Other typical modifications that encompass the same overall workflow tasks include finding the nearest neighbor for all incoming malware [69], clustering new malware with respect to previously known samples [22], etc. Post-triage analysis can then be used to make choices based on the clustering, e.g., focus on the largest clusters of malware first, compare identified clusters with previously seen malware, or any number of other options.

3.2.2 Single Node BitShred

In this section we describe the core components of BitShred: BITSHRED-GEN, BITSHRED-JACCARD, and BITSHRED-CLUSTER. In §3.2.3, we show how the algorithm can be parallelized, e.g., to run on top of Hadoop or multi-core systems.

- **BITSHRED-GEN ($G \rightarrow F$):** BITSHRED-GEN is an algorithm from the extracted feature set $g_i \in G$ to fingerprints $f_i \in F$ for each malware sample s_i. A BitShred fingerprint f_i is a bit vector of length m, initially set to 0. BITSHRED-GEN performs feature hashing to represent feature sets g_i in fingerprints f_i. More formally, for a particular feature set we define a hash function $h : \chi \rightarrow \{0,1\}^m$ where the domain χ is the domain of possible features and m is the length of the bit vector. We use djb2 [27] and reduce the

result modulo m. (Data reduction techniques such as locality-sensitive hashing [22] and Winnowing [146] can be used to pare down the data set for which we call BITSHRED-GEN and perform subsequent steps.)

• **BITSHRED-JACCARD** $(F \times F \to \mathbb{R})$**:** BITSHRED-JACCARD computes the similarity $d \in [0, 1]$ between fingerprints f_a and f_b using the bit vector Jaccard from Equation 3.2. A similarity value of 1 means that the two samples are identical, while a similarity of 0 means that the two samples have nothing in common (in our setting, this means they share no features in G). Formally, Theorem 1 states that BITSHRED-JACCARD well-approximates the Jaccard index.

Theorem 1. Let g_a, g_b denote two sets of size N with c common elements, and f_a, f_b denote their respective fingerprints with bit vectors of length m and k hash functions. Let Y denote $\frac{S(f_a \wedge f_b)}{S(f_a \vee f_b)}$. Then, for $m \gg N$, $\epsilon, \epsilon_2 \in (0, 1)$,

$$Pr[Y \leq \frac{c(1+\epsilon_2)}{2N - c - m\epsilon}] \geq 1 - e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$$

and

$$Pr[Y \geq \frac{c(1-\epsilon_2)}{(2N - c) + m\epsilon}] \geq 1 - e^{-mq\epsilon_2^2/2} - 2e^{-2\epsilon^2 m^2/Nk}$$

for $q = 1 - 2\left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}$.

Although BITSHRED-COMPARE is probabilistic, Theorem 1 proves that it closely approximates the Jaccard index. While attackers could certainly try to manipulate the per-sample analysis, they cannot affect the accuracy of BitShred's feature hashing as long as the hash function is either unknown or collision-resistant.

We show a full proof in §3.3. Note that because the goal of feature hashing is different than the goal of Bloom filters, our guarantees are not in terms of the false positive rate for standard Bloom filters, but of how well our feature hashing data structure lets us approximate the Jaccard index. The intuition behind the difference with traditional Bloom filters is that we are measuring similarity distance, rather than false positive probability, and the overestimate in this distance grows with the number of hash functions.

• **BITSHRED-CLUSTER** $((F \times F \times \mathbb{R}$ **list**$) \times \mathbb{R} \to C)$**:** BITSHRED-CLUSTER takes the list containing the similarity between each pair of malware samples and a threshold t, and outputs a clustering C for the malware. BITSHRED-CLUSTER groups two malware if their similarity d is greater than or equal to t: $d \geq t$. The threshold t is set by the desired

precision trade-off based on past experience. While a smaller t divides malware into a few general families, a larger t discovers specific variants of a family. See §3.5 for our experiments for different values of t.

BitShred currently uses an agglomerative hierarchical clustering algorithm to produce clusters in that the number of clusters is typically not known in advance. Initially each malware sample s_i is assigned to its own cluster c_i. The closest pair is selected and merged into a cluster. We iterate the merging process until there is no pair whose similarity exceeds the input threshold t. When there are multiple samples in a cluster, we define the similarity between cluster c_A and cluster c_B as the maximum similarity between all possible pairs, i.e., BITSHRED-JACCARD$(c_A, c_B) = \max\{$BITSHRED-JACCARD$(f_i, f_j) | f_i \in c_A, f_j \in c_B\}$ (single-linkage). We chose a single-linkage approach as it is efficient and accurate in practice.

3.2.3 Distributed BitShred

BitShred's throughput, as well as any clustering algorithm, is bottlenecked by how quickly fingerprints can be compared. In addition to improved single-node performance, we have also developed a distributed version of BitShred based upon Hadoop [1]. The distributed version performs equal work on each node and does not perform cross-node communication other than returning the result of Jaccard.

In order to improve performance linearly as we add more hardware resources, we need to address two challenges. First, can we design an algorithm that does not require cross-node communication? Second, can we develop an algorithm where no node is a bottleneck, i.e., all nodes do the same amount of work? In this section we describe how the BITSHRED-SCHEDULE algorithm optimally parallelizes BitShred to achieve both goals, as well as how the parallelization can be implemented in the MapReduce framework.

We want methods that are parallelizable so that they are not bottlenecked by the resources of a single node and so that we can improve performance by adding more hardware resources. We utilize the Apache open-source Hadoop Map/Reduce framework by transforming our sequential BitShred algorithm into a parallel BitShred algorithm. MapReduce [45] is a software framework that processes huge data sets in parallel on a large number of nodes. The input data sets are divided into smaller data sets that are separately processed by map tasks. The outputs from map tasks are gathered, sorted, and handled by reduce tasks. Each map and reduce task can be performed in parallel so that we can solve a big problem (i.e., n^2 comparison problem) by solving smaller sub-problems in a distributed

manner.

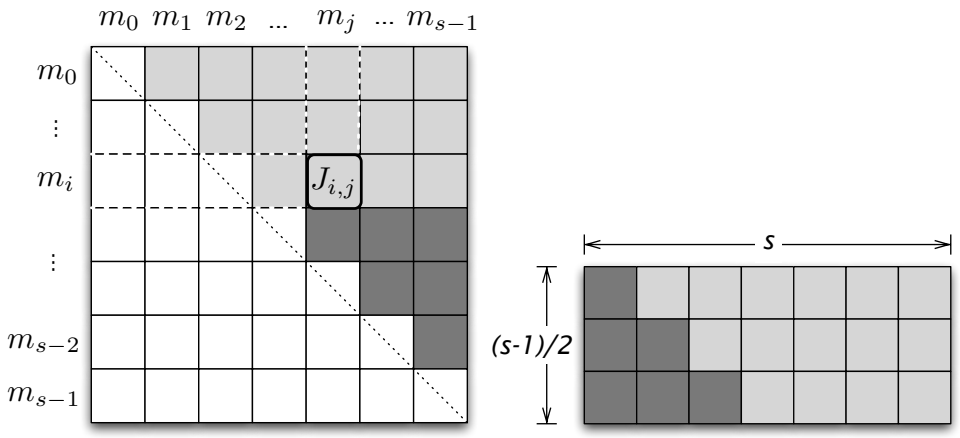
- **BITSHRED-SCHEDULE:** We parallelize two steps in BitShred: fingerprint generation in Phase 1, and the $s(s-1)/2$ fingerprint comparisons in Phase 2 during clustering. Parallelizing fingerprint generation is straightforward: given s malware samples and r resources, we assign s/r malware to each node and run BITSHRED-GEN on each assigned sample.

  Parallelizing BITSHRED-JACCARD in a resource- and communication-efficient manner requires more thought. There are $s(s-1)/2$ comparisons and every comparison takes the same fixed time; so if every node does $s(s-1)/2r$ comparisons, all nodes do equal work. Unlike BITSHRED-JACCARD, comparisons between variable lengths of two sets take time (computation) depending on the length, thus distributing uniform node work is not simple.

  To accomplish this, we first observe that while the first malware needs to be compared against all other malware (i.e., $s-1$ fingerprint comparisons), each of the remaining malware require fewer than $s-1$ comparisons each. In particular, malware i requires only $s-i$ comparisons, and malware $s-i$ requires $s-(s-i)$ comparisons.

  A naive approach is to round-robin each set of comparisons on the available nodes, e.g., node 1 does $s-1$ amount of work performing $1 \rightarrow \langle 2, s\rangle$, node 2 does $s-2$ work initially performing $2 \rightarrow \langle 3, s\rangle$, and so on. The problem with a round-robin approach is that we must interactively communicate with each node to assign the next comparison work, which causes a significant communication overhead. Furthermore, the amount of work at each node is unbalanced.

  Instead, we want to be able to assign similar amounts of work to each node, and to do this while incurring minimal communication overhead between the nodes. Our main insight is to pair the comparisons for malware i with $s-i$, so that the total comparisons for each pair is $s-i+s-(s-i)=s$. If we pair the comparisons for malware i with $s-i$, the total comparisons for each pair is $s-i+s-(s-i)=s$. Thus, for each node to do uniform work, BITSHRED-SCHEDULE ensures that the $s-i$ comparisons for malware i are scheduled on the same node as the $s-(s-i)$ comparisons for malware $s-i$. BITSHRED-SCHEDULE then simply divides the pairs among the r nodes. Since each pair requires s comparisons, and there are $(s-1)/2$ pairs, we can split the jobs to $(s-1)/2$ nodes where each node performs exactly s comparisons. Since each comparison takes constant time, the total load is distributed equally across all nodes using the fixed schedule.



(a) Similarity matrix

(b) Similarity rectangle

Figure 3.2: Similarity matrix

Although there may be other protocols for distributing computations, we note that this approach is simple, and optimal in the sense that all nodes do equal work and there is no inter-node communication during the s^2 Jaccard calculations.

Our algorithm can be understood as the computation of a similarity matrix where cell i, j corresponds to the distance between malware i and j, as shown in Figure 3.2a. Since the similarity between i, j is the same as j, i, we only need to compute the upper triangle of the matrix. BITSHRED-SCHEDULE essentially turns the triangle in Figure 3.2a into the rectangle shown in Figure 3.2b. Finally, we note that BITSHRED-SCHEDULE is optimal for distributing comparison operations evenly across all nodes in a communication-efficient manner.

3.2.4 BitShred on Hadoop

Our distributed implementation uses the Hadoop implementation of MapReduce [1, 45]. MapReduce is a distributed computing technique for taking advantage of large computer nodes to carry out large data analysis tasks. In MapReduce, functions are defined with respect to ⟨key,value⟩ pairs. MapReduce takes a list of ⟨key,value⟩ pairs and returns a list of values. MapReduce is implemented by defining two functions:

1. **MAP:** $\langle K_i, V_i \rangle \rightarrow \langle K_o, V_o \rangle$ **list.** In the MAP step, the master Hadoop node takes the input pair of type $\langle K_i, V_i \rangle$ and partitions it into a list of independent chunks of work. Each chunk of work is then distributed to a node, which may in turn apply MAP to

further delegate or partition the set of work to complete. The process of mapping forms a multi-level tree structure where leaf nodes are individual units of work, each of which can be completed in parallel. When a node completes a unit of work, the output $\langle K_o, V_o \rangle$ is passed to REDUCE.

2. **REDUCE:** $\langle K_o, V_o \rangle$ **list** $\rightarrow V_f$ **list.** In the REDUCE step, the lists of answers from the partitioned work units are combined and assembled into a list of answers of type V_f.

We also take advantage of the Hadoop distributed file system (HDFS) to share common data among nodes.

In Phase 1, distributed BitShred produces fingerprints using the Hadoop by defining the following MapReduce functions:

1. **MAP:** $\langle K_i, s_i \rangle$ **list** $\rightarrow \langle K_i, f_i \rangle$ **list.** Each MAP task is assigned the subset of malware samples s_i and creates fingerprints f_i to be stored on HDFS. Fingerprint files are named as K_i representing the index of the corresponding malware samples. The outputs of map tasks are fingerprints for input samples. For example, if we process 2,048 samples, we have 2,048 32KB fingerprint files where the size of a fingerprint is 32KB.

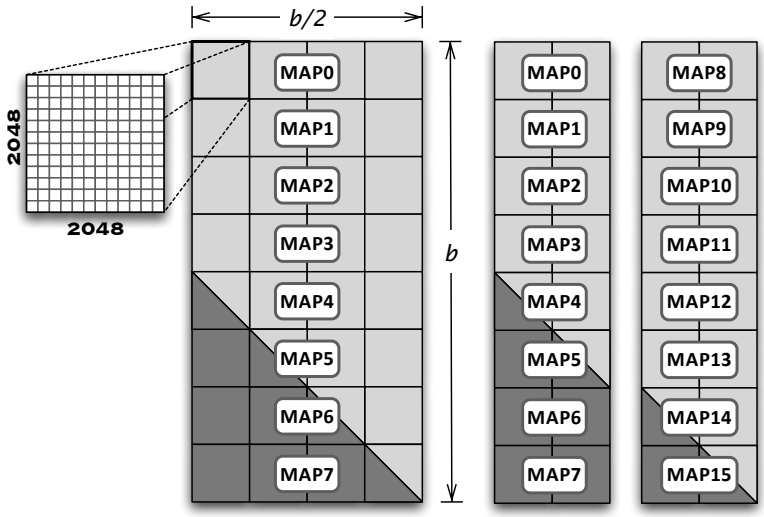
   However, accessing lots of small files normally causes significant disk I/O overhead. For instance, 2,096,128 disk accesses are required to compare every pair of 2,048 samples. Furthermore, HDFS is primarily designed for processing large streaming data, so it is not efficient for accessing lots of small files.

   To reduce disk I/O overhead, we divide input data sets into chunks of 2,048 samples and assign those to map tasks. That is, each map task processes given 2,048 samples and generate a big file containing 2,048 fingerprints. We assemble 2,048 fingerprints because the default block size of HDFS is 64MB (=32KB$\times$2,048). Similarly, we can assemble multiples of 2,048 samples at once, e.g., 4,096, 8,192, etc.

2. REDUCE. In this step, no REDUCE step is needed. In our implementation, we will optionally return the list of fingerprint files K_i.

In Phase 2, distributed BitShred runs BITSHRED-JACCARD across all Hadoop nodes by defining the following functions:

1. MAP: $\langle K_i, f_i \rangle$ list $\rightarrow \langle \mathbb{R}, (s_a, s_b) \rangle$ list MAP tasks read fingerprint data files created during Phase 1 and runs BITSHRED-JACCARD on each fingerprint pair, outputting the similarity $d \in \mathbb{R}$.



(a) Arranging grids (b) Splitting grids

Figure 3.3: Chunking MAP tasks for optimized I/O.

2. REDUCE: $\langle \mathbb{R}, (s_a, s_b) \rangle$ list $\rightarrow$ sorted $\langle \mathbb{R}, (s_a, s_b) \rangle$ list REDUCE gathers the list of the similarity values for each pair and returns a sorted list of pairs based upon similarity.

We have also optimized this phase to use HDFS-sized blocks for fingerprints, where each block corresponds to 2,048 malware fingerprints. BITSHRED-SCHEDULE creates a schedule to break up MAP tasks into 2,048$\times$2,048-sized grids, as shown in Figure 3.3a. Given b blocks, we assign $b/2$ grid comparisons to up to b MAP tasks. BITSHRED-SCHEDULE also optimizes the order in which blocks are accessed so as to minimize the number of reads. For example, MAP0 in Figure 3.3a needs to read 1 vertical fingerprint block and 4 horizontal fingerprint blocks to compare, which requires 5 disk reads. This schedule prevents less optimal orderings; for example, when MAP0 is assigned 4 grid comparisons chosen at random, 8 disk accesses are needed (4 times for the vertical blocks and 4 times for the horizontal blocks). If we want to utilize more MAP tasks, we can further divide the $b/4$ grid comparisons up to $2b$ MAP tasks by splitting as in Figure 3.3b.

This phase returns a list of malware pairs sorted by similarity using standard Hadoop sorting. The sorted list is essential for the agglomerative single-linkage clustering. In particular, malware s_i's family is defined as the set of malware whose distance is less than θ; thus all malware in the sorted list with similarity $> \theta$ are in the cluster. Given the sorted list, it is straightforward to apply an agglomerative hierarchical clustering algorithm to produce malware clusters.

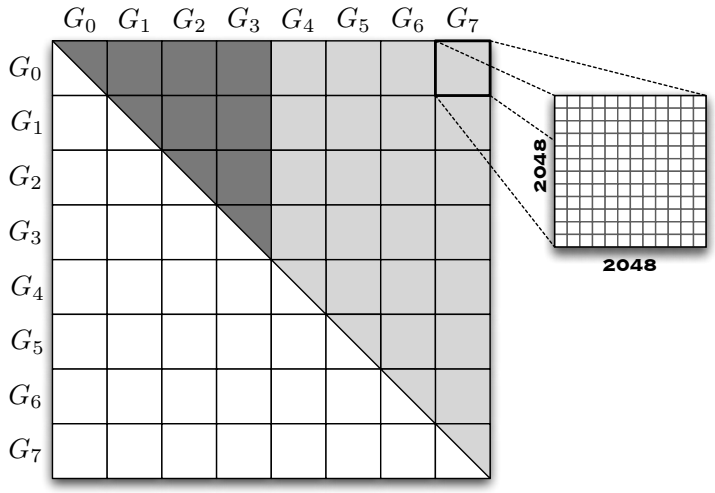


Figure 3.4: Similarity matrix consisting of grids

We assemble 2,048 fingerprints in a file in order that we divide the matrix into small grids whose size is 2,048×2,048 to access fingerprint data files in an efficient manner. For example, to complete $G_{0,7}$ in Figure 3.4, we need to read 2,048 vertical fingerprints from G_0 and 2,048 horizontal fingerprints from G_7, which can be done by 2 disk reads.

3.3 Proof of Similarity Approximation

Our analysis shows that, with high probability, the Jaccard index $\frac{|g_i \cap g_j|}{|g_i \cup g_j|}$ is well approximated by the $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$, where f_i and f_j are the fingerprints of g_i and g_j. Throughout this analysis, we let c denote the number of shared elements between sets g_i and g_j (note that the Jaccard index $\frac{|g_i \cap g_j|}{|g_i \cup g_j|}$ is then $\frac{c}{2N-c}$). The focus of our analysis is to show that the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$ is close to $\frac{c}{2N-c}$ with high probability (unlike other analyses [73] that restrict their focus to computing the expected value of $S(f_i \wedge f_j)$). We make the usual assumption that the hash functions used are k-wise independent.

We first consider the union $g_i \cup g_j$. We note that the bit vector obtained by computing the bitwise-OR of the two fingerprints f_i and f_j is equivalent to the bit vector that would be obtained by directly inserting all the elements in $g_i \cup g_j$ if the same k hash functions are used on a bit vector of the same size.

Let the random variable U denote the number of bits set to 1 in $f_i \vee f_j$. Note that the set $g_i \cup g_j$ contains $2N - c$ elements. If these elements are inserted into a bit vector of size

m with k hash functions, the probability q_u that a bit is set to 1 is: $1 - \left(1 - \frac{1}{m}\right)^{k(2N-c)}$. We can use this to compute the expected value of U:

$$E[U] = mq_u = m\left(1 - \left(1 - \frac{1}{m}\right)^{k(2N-c)}\right) \tag{3.3}$$

As U is tightly concentrated around its expectation [31], we get:

$$Pr[|U - E[U]| \geq \epsilon m] \leq 2e^{-2\epsilon^2 m^2/(2N-c)k} \leq 2e^{-2\epsilon^2 m^2/Nk}.$$

Next, we consider the intersection $g_i \cap g_j$. Let the random variable I denote the number of bits set to 1 in $f_i \wedge f_j$. A bit z is set in $f_i \wedge f_j$ in one of two ways: (1) it may be set by some element in $g_i \cap g_j$, or (2) it may be set by some element in $g_i - (g_i \cap g_j)$ and by some element $g_j - (g_i \cap g_j)$. Let I_z denote the indicator variable for bit z in $f_i \wedge f_j$. Then,

$$\begin{aligned} Pr[I_z = 1] \quad = \quad & \left(1 - \left(1 - \frac{1}{m}\right)^{kc}\right) + \\ & \left(1 - \frac{1}{m}\right)^{kc}\left(1 - \left(1 - \frac{1}{m}\right)^{k(|g_i|-c)}\right) \\ & \cdot \left(1 - \left(1 - \frac{1}{m}\right)^{k(|g_j|-c)}\right) \end{aligned}$$

which may be simplified as:

$$1 - \left(1 - \frac{1}{m}\right)^{kN} - \left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}.$$

With linearity of expectation, we can compute $E[I]$ as $\sum_z Pr[I_z = 1]$, which reduces to:

$$E[I] = m\left(1 - 2\left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}\right). \tag{3.4}$$

Note that the random variables $I_1, I_2 \ldots I_m$ are negatively dependent, so we can apply Chernoff-Hoeffding bounds to compute the probability that I deviates significantly from $E[I]$: e.g., $Pr[I \geq E[I](1+\epsilon_2)] \leq e^{-mq\epsilon_2^2/3}$, where $q = 1 - \left(1 - \frac{1}{m}\right)^{kN} - \left(1 - \frac{1}{m}\right)^{kN} + \left(1 - \frac{1}{m}\right)^{k(2N-c)}$.

We now turn to the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$; let the random variable Y denote this ratio. We have just shown that U and I are both likely to remain close to their expected values, and we can

use this to compute upper and lower bounds on Y – since U and I lie within an additive or multiplicative factor of their expectations with a probability of at least $1 - 2e^{-mq\epsilon_2^2/3}$ and $1 - 2e^{-2\epsilon^2 m^2/Nk}$, respectively. We can derive upper and lower bounds on Y that hold with probability at least $1 - 2e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$.

To do this, we first simplify the quantities $E[U]$ and $E[I]$. Assuming that $m \gg 2kN$, we can approximate $E[U]$ and $E[I]$ by discarding the higher-order terms in each of binomials in 3.3 and 3.4:

$$\begin{aligned} E[U] &\geq m\left(1 - \left(1 - \frac{k(2N-c)}{m}\right)\right) \\ &= mk\left(\frac{2N-c}{m}\right) = k(2N-c). \end{aligned}$$

Likewise, we can approximate $E[I]$ as:

$$\begin{aligned} E[I] &\leq m\left(1 - 2\left(1 - \frac{kN}{m}\right) + \left(1 - \frac{k(2N-c)}{m}\right)\right) \\ &= mk\left(\frac{c}{m}\right) = ck. \end{aligned}$$

Using these approximations for $E[I]$ & $E[U]$, we see that $Y \leq \frac{c(1+\epsilon_2)}{2N-c-m\epsilon}$, with probability at least $1 - e^{-mq\epsilon_2^2/3} - 2e^{-2\epsilon^2 m^2/Nk}$. We can compute a similar lower bound for Y, i.e., $Y \geq \frac{c(1-\epsilon_2)}{(2N-c)+m\epsilon}$, with probability at least $1 - e^{-mq\epsilon_2^2/2} - 2e^{-2\epsilon^2 m^2/Nk}$. This shows that with high probability, the ratio $\frac{S(f_i \wedge f_j)}{S(f_i \vee f_j)}$ is close to the Jaccard index $\frac{c}{2N-c}$ for appropriately chosen values of m and k. We have thereby proven our Theorem 1.

Lastly, we give an example to illustrate our bounds in our application scenario. Suppose we set $\epsilon m \geq 5$, $m \approx 1000N$, $k = 6$. Then, our analysis shows us that with probability at least 95%, $Y \in \left(\frac{c(1-\frac{1}{\sqrt{2c}})}{2N-c+5}, \frac{c(1+\frac{4}{\sqrt{c}})}{2N-c-5}\right)$, i.e., that ratio of the bits set to the union is very close to the Jaccard index.

3.4 Implementation

We have implemented single-node BitShred in 2,000 lines of C code. Since BitShred is agnostic to the particular per-malware analysis methods, we only need individualized routines for extracting raw input features before converting them into fingerprints. In the case

of static code analysis, BitShred divides an executable code section identified by `GNU BFD` library into n-grams and hashes each n-gram to create fingerprints. For dynamic behavior analysis, BitShred simply parses input behavior profile logs and hashes every behavior profile to generate fingerprints. We use `berkeley DB` to store and manage the fingerprints database. After building the database, BitShred retrieves fingerprints from the database to calculate the Jaccard similarity between fingerprints. After applying an agglomerative hierarchical clustering algorithm, malware families are formed. We use graphviz and Cluto [86] for visualizing the clustering and family trees generated, as shown in Figure 3.11, 3.12.

Distributed BitShred is implemented in 500 lines of Java code. We implement a parser for extracting section information from Portable Executable header information because there is no `BFD` library for Java. In our implementation, we perform a further optimization that groups several fingerprints into a single HDFS disk block in order to optimize I/O. In the Hadoop infrastructure we use, the HDFS block size is 64MB. We optimize for this block size by dividing the input malware set so that each node works on 2,048 malware samples at a time (because 64MB = 32KB $\times$ 2,048). That is, each MAP task is given 2,048 samples ($s_i, s_{i+1}, \cdots s_{i+2047}$) and generates a single file containing all fingerprints. We can similarly optimize for other block sizes and different bit vector lengths, e.g., 64KB bit vectors result in batching 1,024 malware samples per node.

3.5 Evaluation

We have evaluated BitShred for speed and accuracy using two types of per-sample analysis for features. First, we use a static code reuse detection approach where features are code fragments, and two malware are considered similar if they share common code fragments. Second, we use a dynamic analysis feature set where features are displayed behaviors, and two malware are considered similar if they have similar behaviors. Note that similarity is a set comparison, so order does not matter (e.g., re-ordering basic blocks is unlikely to affect the results). We stress that we are not advocating a particular approach such as static or dynamic analysis but are instead demonstrating how BitShred could be used once an analysis is selected.

3.5.1 Experimental Setup

System Environment. All single-node experiments were performed on a Linux 2.6.32-23 machine (Intel Core2 6600 / 4GB memory) using only a single core. The distributed

experiments were performed on a Hadoop using 64 worker nodes, each with 8 cores, 16 GB DRAM, 4 1TB disks, and 10GbE connectivity between nodes [2]. 53 nodes had a 2.83GhZ E5440 processor, and 11 had a 3GhZ E5450 processor. Each node was configured to allow up to 6 map tasks and up to 4 reduce tasks at a time.

Malware Data Set. We performed our experiments on a malware data set collected from a variety of open repositories such as Malware Analysis System (aka CW-Sandbox) [5], Offensive Computing [8], and our Universities' security infrastructure inbetween 2009-2010. Our total data set consists of 655,360 unique samples by MD5 hash.

When evaluating accuracy, we used unpacked malware samples as identified by PEiD [79] and ClamAV. Recall that we assume an effective unpacking strategy, e.g., previous work has shown that up to 92% of malware can be automatically unpacked [63, 107, 143]. Since the goal of this paper is not unpacking, we simply use ClamAV to unpack since it is public, though we also have support for the research unpacking tool Renovo [81].

3.5.2 BitShred with Code Reuse as Features

Setup. Our static experiments are based on reports that malware authors reuse code as they invent new malware samples [11, 91, 158]. Since malware is traditionally a binary-level analysis, not a source analysis, our implementation uses n-grams to represent binary code fragments. Malware similarity is determined by the percentage of n-grams shared.

We chose n-grams-based analysis because it is a previously proposed approach that demonstrates a high-dimensionality feature space. We set $n = 16$ bytes, so there are 2^{128} possible n-gram features. We chose 16 bytes based on experiments that show that it would cover at least a few instructions. If n is too small, then it is hard to catch meaningful byte sequences. For example, if $n = 1$, every n-gram only represents 1 byte binary code, which is not enough to capture the meaning of an instruction. On the other hand, if n is too big, then it is not resilient to code reordering. For example, when n is equal to the size of the entire program, we can only tell whether or not two programs are exactly the same.

In static code analysis, each n-gram can roughly be thought of as an n-length instruction in the binary code without actually performing the disassembly. Since n-gram analysis rolls over the code, our analysis is conceptually similar to the analysis of all possible n-length disassemblies of the code. In order to determine an appropriate value of n to match instructions, we disassembled 9,000 malware samples using IDA Pro and counted the distribution on instruction lengths. As shown in Figure 3.5, more than 98% of code sections consisted of instructions whose length was equal to or less than 8 bytes. We set $n = 16$ bytes from

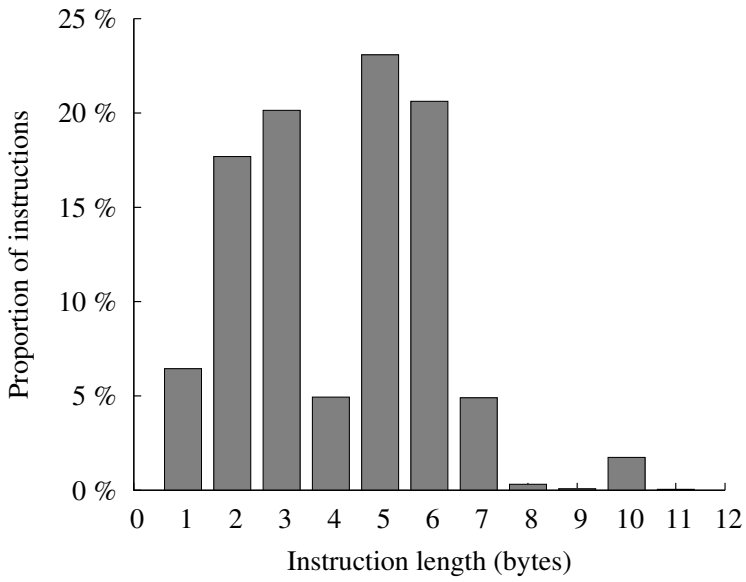


Figure 3.5: Proportion of instructions of each instruction length

this experiment because it is highly likely that each n-gram covers at least 2 contiguous instructions.

We can extend BitShred to use other features, such as basic blocks [53] by first building the appropriate feature and then defining a hash function on it; applying all possible extensions of the per-sample analysis is out of scope for this work. Surprisingly, even this simple analysis had over 90% accuracy when the malware was unpacked using off-the-shelf unpackers. Pragmatically, n-gram analysis also has the advantage of not requiring disassembling or building a control flow graph, which are known hard problems on malware.

Single Node Performance. Table 3.1 shows BitShred's performance using a single node in terms of speed, memory consumed, and the resulting error rate. We limited our experiment to clustering 1,000 malware samples (which requires 499,500 pairwise comparisons) in order to keep the exact Jaccard time reasonable. The "exact Jaccard" row shows the overall performance when computing the set operations as shown in Equation 3.1 using the SimMetrics library [36]. Clustering using exact Jaccard took more than 4 hours and required 644.13MB of memory. This works out to about 33 malware comparisons per second and 2,388 malware clustered per day.

We performed two performance measurements with BitShred: one with 64KB fingerprints and one with 32KB fingerprints. With 64KB fingerprints, BitShred ran about 317

	Size of fingerprints	Time to compare every pair	Average error on all pairs	Average error on similar (>0.5) pairs	Malware comparisons per second	Malware clustered per day
EXACT JACCARD	644.13MB	4h 12m 16s	-	-	33	2,388
BS64K	62.50MB	48s	0.0199	0.0017	10,472	42,538
BS32K	31.25MB	24s	0.0403	0.0050	20,812	59,970
WINNOW (W4)	66.97MB	41m 5s	0.0019	0.0109	203	5,918
WINNOW (W12)	30.16MB	20m 35s	0.0081	0.0128	404	8,360
BS32K (W4)	31.25MB	24s	0.0159	0.0009	20,812	59,970
BS32K (W12)	31.25MB	24s	0.0062	0.0039	20,812	59,970
BS8K (W4)	7.81MB	6s	0.0649	0.0086	78,047	116,131
BS8K (W12)	7.81MB	6s	0.0247	0.0016	78,047	116,131

Table 3.1: BitShred (BS) vs. Jaccard vs. Winnowing. We show BitShred with several different fingerprint sizes.

times faster than exact Jaccard. With 32KB fingerprints, BitShred ran about 2 times faster than with 64KB fingerprints, and ran about 631 times faster than exact Jaccard.

Since BitShred uses feature hashing, hash collisions may impact the accuracy of the Jaccard distance computations. The overall error rate in the distance computations is a function of the fingerprint length, the size of the feature space, and the percentage of code that is similar. The statement in Theorem 1 formally expresses this trade-off. We also made two empirical measurements. First, we computed the average error on all pairs, which worked out to be about 2% with 64KB fingerprints and 4% with 32KB fingerprints. The error goes up as the fingerprint size shrinks due to a higher chance of collisions. We also computed the average error on pairs with a similarity of at least 50% and found the error to be less than 1% of the true Jaccard. Note that the second metric (i.e., average error on pairs with higher similarity) is the more important metric – these are the numbers with the most impact on the accuracy, as these are the numbers that will primarily decide which family a malware sample belongs to. Thus BitShred is a very close approximation indeed.

BitShred vs. Winnowing. So far we have considered techniques that provide an exact ranking between all pairs of malware. Nonetheless, malware practitioners are constantly facing the difficult choice of how much time to spend given finite computing resources, and therefore may prefer "approximate but faster" clustering over "theoretically correct but slower" clustering. LSH is one type of data reduction technique that improves performance. Here we discuss another called *Winnowing*.

In addition to comparing BitShred's probabilistic algorithm to exact Jaccard, we also compare BitShred to Winnowing [146]. Recall that Winnowing is a feature reduction technique used for code reuse detection. Winnowing is interesting because its performance is guaranteed to be within 33% of the lower bound [146], and is currently the fastest (from a theoretic sense) technique that we are aware of. We compare two settings: BitShred vs. Winnowing as in previous work, and BitShred extended to include Winnowing.

Winnowing, the algorithm used by the MOSS plagiarism detection tool, is a fuzzing hashing technique that selects a subset of features from a sample for analysis [146]. Let w be a window measured in some way, e.g., w statements, w consecutive n-grams, w behaviors, etc. Winnowing guarantees that at least one shared unit in any window of length at least $w+n-1$ will be included in the feature set [146]. In our evaluation, we measure Winnowing because a) MOSS is well-known, b) it corresponds to similarity detection based on code as proposed in previous work [11, 91, 158], and thus is directly related to our approach, and c) it is guaranteed to be within 33% of optimal for similarity detection [146]. We compared BitShred and Winnowing in two settings: BitShred vs. Winnowing as in previous work, and BitShred extended to include Winnowing. Table 3.1 also shows these results for window sizes 4 (denoted as W4) and 12 (denoted as W12).

We observed that BitShred beats straight Winnowing. We reimplemented Winnowing as detailed in [146] using a 32-bit hash function, as the original implementation is not public. For the purpose of performance comparison, we computed the similarity using SimMetrics library. BitShred is anywhere from 26 to 102 times faster while requiring less memory. Winnowing does have a slightly better error rate, though none of the error rates are very high. A more interesting case to consider is pre-processing the feature set with Winnowing and then applying BitShred. With Winnowing applied, we can reduce the BitShred fingerprint size down to 8KB, allowing all 1,000 samples to be clustered in 6 seconds.

Figure 3.6 relates all experiments with respect to the total number of malware clustered per day. BitShred can efficiently handle incoming malware. Figure 3.6 also shows on the right-hand y-axis one reason why BitShred is faster. Recall that exact Jaccard computations are slow in part because they use set operations. These, in turn, are not efficient on real architectures. BitShred's bit vector fingerprints, on the other hand, are L1/L2 cache friendly.

Distributed BitShred. We have implemented a distributed version of BitShred on the Hadoop and performed several experiments to measure overall scalability and throughput. In our large-scale experiments, we used 131,072 UPX-unpacked samples to measure Bit-

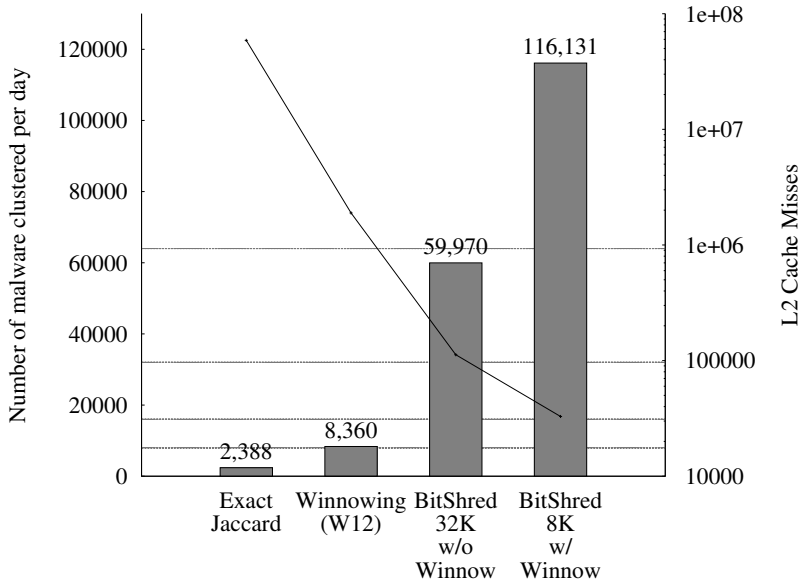


Figure 3.6: Overall malware clustered-per-day capabilities. We also report relative L1/L2 cache misses.

Shred's clustering quality and 20,480 to 655,360 samples[1] to evaluate BitShred's scalability, with each intermediate point doubling the total number of malware.

Figure 3.7 shows the BITSHRED-GEN fingerprint generation time. In this experiment, we utilized 80 map tasks for small data sets (20,480 $\sim$ 81,920) and 320 map tasks for large data sets (163,840 $\sim$ 655,360). The total time to create fingerprints for all samples was 5m 45s with BS8K (W12) and 4m 40s with BS32K (W1). The graph also shows a linear trend in the fingerprint generation time, e.g., halving the total number of samples to 327,680 samples approximately halves the generation time to about 2m 54s and 2m 25s, respectively. BITSHRED-GEN performance slightly dropped at 163,840 samples because the startup and shutdown overhead of each map dominates the benefit of utilizing more maps.

Figure 3.8 shows the amount of time for computing the pairwise distance for the same sample set. We utilized 200 map tasks for small data sets and 320 map tasks for large data sets. Given the values in the graph, we can work out the number of comparisons per second. For example, 163,840 samples require approximately 1.3×10^{10} comparisons and take 10m 15s with BS8K (W12), which works out to 21,823,888 comparisons per second. 327,680

[1]These samples were not unpacked since public implementations of unpackers are not readily available. The scalability numbers should not be affected since generation time is dwarfed by comparison time.

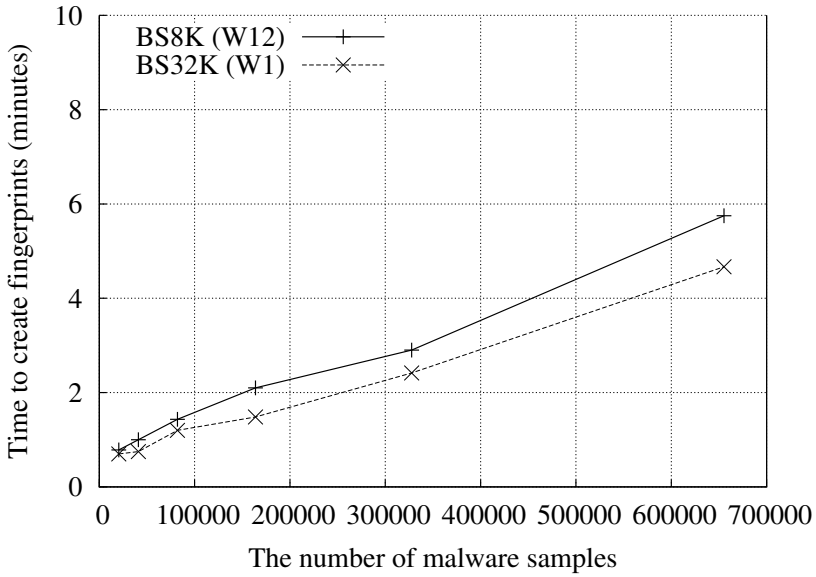


Figure 3.7: Performance of Distributed BITSHRED-GEN

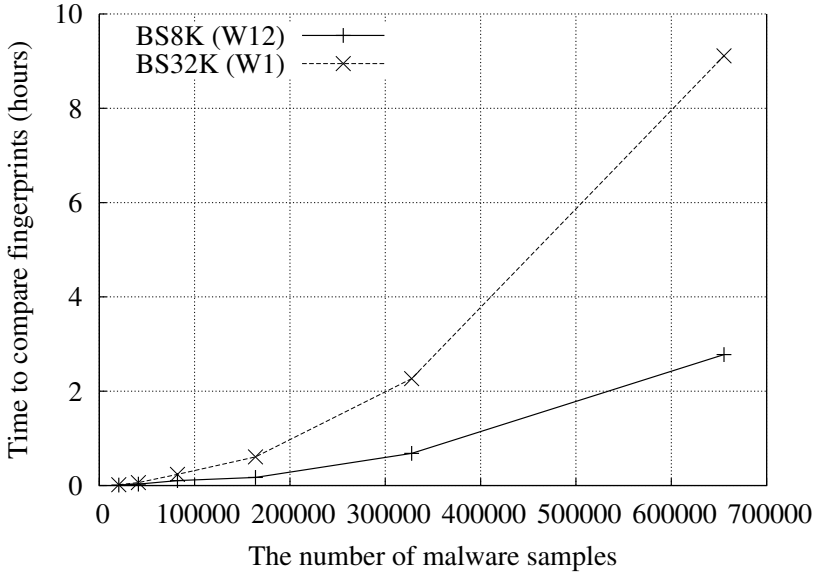


Figure 3.8: Performance of Distributed BITSHRED-JACCARD

samples require about 5.4×10^{10} comparisons and take 40m 55s with BS8K (W12), which works out to a similar 21,868,402 comparisons per second.

Overall, the distributed version achieved a pairwise comparison throughput of about 1.9×10^{12} per day. This works out to *full* hierarchical clustering over 1.9 million malware per day. In the case of *incremental* clustering, this works out to comparing over 190,000 malware per day against 10 million known malware that are already in a database.

Triage Tasks. Three common triage tasks are automatically identifying malware families via clustering [22], identifying the nearest neighbors to a particular malware sample [69], and visualizing malware by creating phylogenetic trees [84]. In this experiment, we explore using BitShred with n-grams as the extracted features. While we stress that we are not advocating n-gram analysis, we also note that it is interesting to see what the actual quality would be in such a system. We repeat these analyses in §3.5.3 using dynamic behavior features.

- **Clustering.** We define the *quality* of a clustering as how close a particular clustering is to the "correct" clustering with respect to labeled data set. Overall quality will heavily depend on the feature extraction tool (e.g., static or dynamic), the particular data set (because malware analysis often relies on undecidable questions), and the quality of the reference data set.

  To create a reference clustering data set, we used 30∼40 different antivirus labels provided by VirusTotal [10]. First, we chose samples that were detected as malware by at least 20 antivirus programs to get more reliable labels. We normalized and tokenized all the labels; then, we assigned the family name based upon only the tokens occurring at the majority of the detecting anti-virus programs. As a result, we had 3,935 samples. The malware data sets we used were collected in 2008 and 2009. We believe AV vendors could have enough time to analyze such malware samples and to assign more reliable AV labels to samples, which could help us to prepare a more reliable reference clustering data set.

  The overall clustering quality is measured with respect to two metrics: precision and recall. Precision measures how well malware in separate families are put in different clusters, and recall measures how well malware within the same family are put into the same cluster. Formally, precision and recall are defined as:

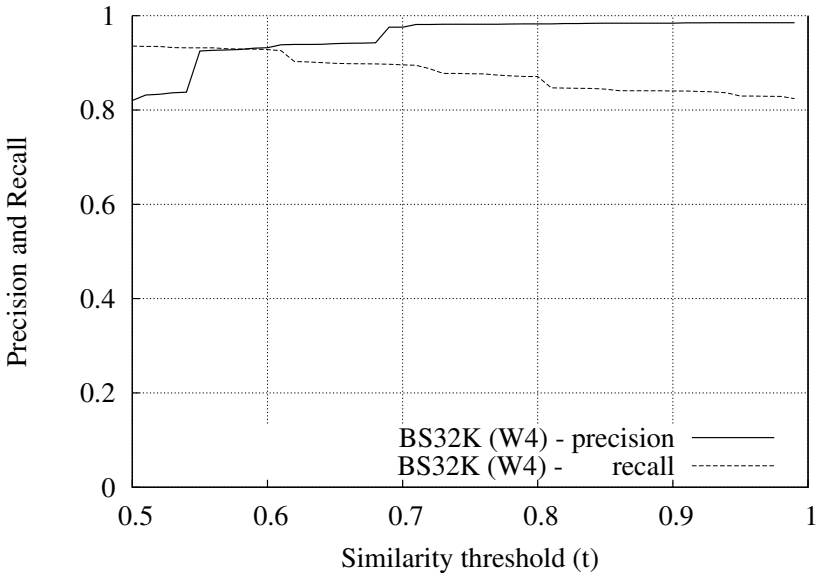


Figure 3.9: Precision and Recall (3,935 samples)

$$\text{Precision} = \frac{1}{s}\sum_{i=1}^{c}\max(|C_i \cap R_1|, ..., |C_i \cap R_r|)$$

$$\text{Recall} = \frac{1}{s}\sum_{i=1}^{r}\max(|C_1 \cap R_i|, ..., |C_n \cap R_i|)$$

Using n-gram analysis, we clustered the reference data set of 3,935 samples. Figure 3.9 shows the overall quality of BitShred with Winnowing (BS32K (W4)). Surprisingly, simple n-gram analysis did quite well. When $t = 0.6$, BS32K (W4) clustering produced 200 clusters with a precision of 0.932 and a recall of 0.928 in 8 minutes.

For a larger-scale experiment, we unpacked 131,072 malware samples using off-the-shelf unpackers. We then clustered the malware and compared the identified families to a reference clustering using ClamAV labels[2]. Figure 3.10 shows the overall results of BitShred with Winnowing (BS32K (W12)). When $t = 0.57$, BS32K (W12) clustering produced 7,073 clusters with a precision of 0.942 and a recall of 0.922. It took about 27m with 256 map tasks.

[2]A considerable amount of manual work is required to prepare a reference data set. For this reason, we simply used ClamAV as a reference for 131,072 samples.

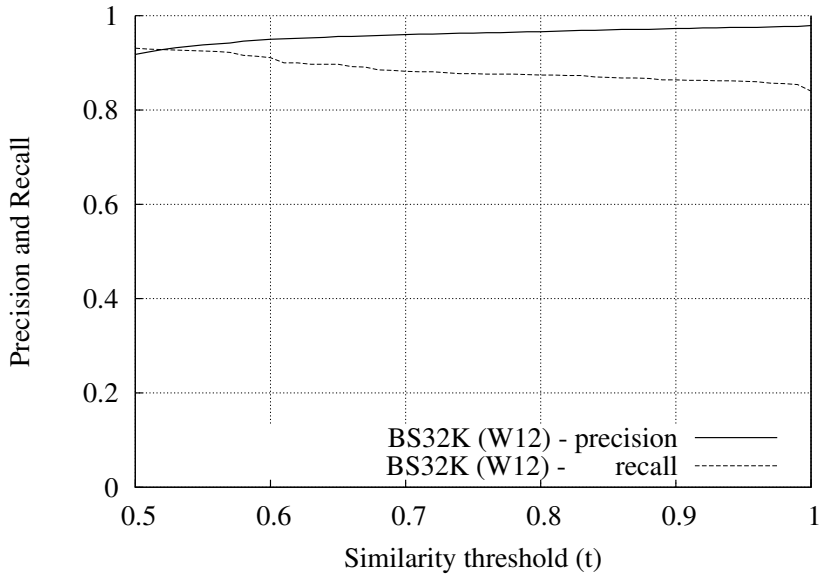


Figure 3.10: Precision and Recall (131,072 samples)

In some cases, BitShred grouped two samples whose ClamAV virus names were incorrectly different, which reduced the precision of BitShred clustering. We manually spot-checked the differences and found that in many cases, the ClamAV label was wrong and BitShred's clustering was correct. For example, ClamAV detected Trojan.Agent-65787 and Trojan.Agent-87221 based on MD5 hash of `.text` section. However, the difference is that Trojan.Agent-87221 just filled the null part of `.text` section with some dummy instructions. (Null bytes are filled at the end of each section of a PE file due to the section alignment.) As a result, both samples had different MD5 values of `.text` section while sharing the common malicious code. Similarly, there were only 7 different bytes in code sections between Trojan.Tibia-162 and Trojan.Tibia-179 and only 8 different bytes between Trojan.Spy-62099 and Trojan.Spy-62214.

Polymorphic malware changed the code pattern in every variant so that the malware produced low similarity in BitShred and dropped the recall of BitShred clustering. For example, every variant of W32.Mabezat-2 had a different code section (except for the decryption routine) because they were encrypted with different keys. ClamAV identified W32.Mabezat-2 by detecting the invariant decryption routine. Since a encrypting/decrypting procedure is similar to a packing/unpacking procedure, we can leverage a generic unpacker to effectively handle the polymorphic malware.

• **Nearest Neighbor.** Hu et al. describe finding the nearest k-neighbors to a given sample as a common triage task [69]. In their implementation, Hu et al. return the 5 nearest neighbors and achieve an 80% success rate in returning at least one malware from the correct family among 5 nearest neighbors on a data set of 102,391 samples. The query time was between 0.015s to 872s, with an average of 21s using 100MB of memory.

We have implemented similar functionality in BitShred by comparing the given malware to all other malware. We did not have access to Hu et al.'s data set; we performed experiments finding the 5 nearest neighbors to randomly chosen malware samples on the same size of our 102,391 malware data set. We achieved the same 94.2% precision and 92.2% recall as above. The average time to find the neighbors was 6.8s (w/ BS8K) and 27s (w/ BS32K), using 25MB memory, with variance always under 1s.

• **Visualization.** We also have implemented several ways to visualize clustering within BitShred. First, we can create boxed malware graphs where each square represents a malware family and each circle represents an individual sample. Figure 3.11 shows a clustering of 20,000 malware samples when $t = 0.57$[3]. In the figure we can see larger families with many malware in the center, with the size of the family decreasing as we move towards the edges. At the very edge are malware samples that cluster with no family.

Another way to visualize the results using BitShred is to create phylogenetic family trees based on similarity [84]. The more alike two malware samples are, the closer they are to each other on the tree. Figure 3.12 depicts a sample tree created from our data set, labeled with ClamAV nodes. It is interesting to note that ClamAV labels the malware as coming from three families: Spy, Dropper, and Ardamax. We manually confirmed that all three were indeed extremely similar and should be considered to be of the same family, e.g., Trojan.Ardamax-305 and Trojan.Spy-42659 are in different ClamAV families, but only differ by 1 byte.

3.5.3 BitShred with Dynamic Behaviors as Features

Static analysis may be fooled by advanced obfuscation techniques, which has led researchers to propose a variety of dynamic behavior-based malware analysis approaches, e.g., [16, 22, 115, 116, 143, 149]. One popular variant of this approach is to load the malware into a

[3]We pick 20,000 samples because larger numbers created graphs that hung our PDF reader, which could potentially happen to others as well.

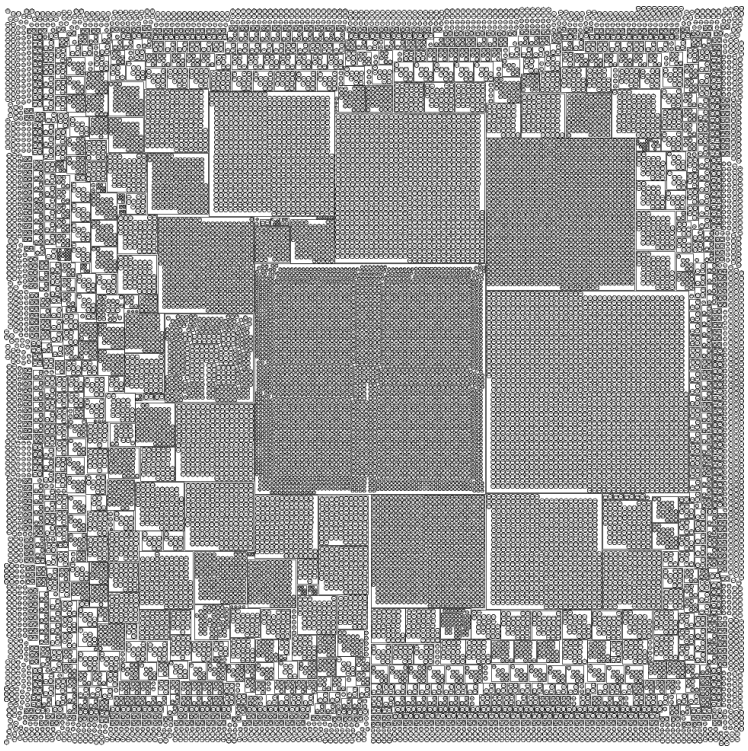


Figure 3.11: Clustering graph when $t = 0.57$

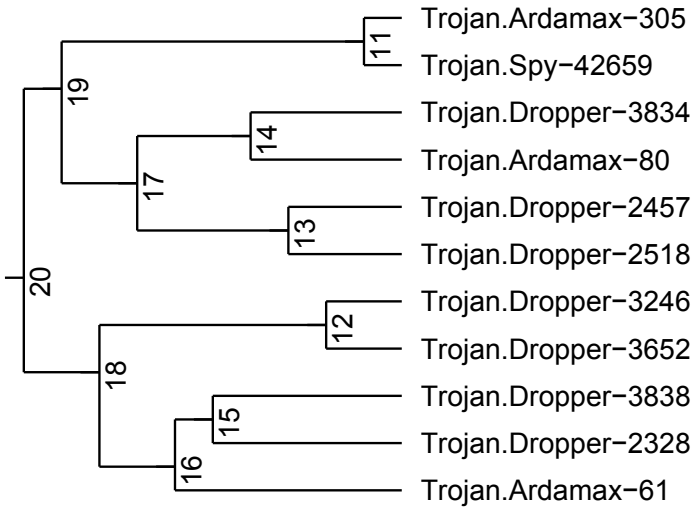


Figure 3.12: Lineage tree for a single malware family

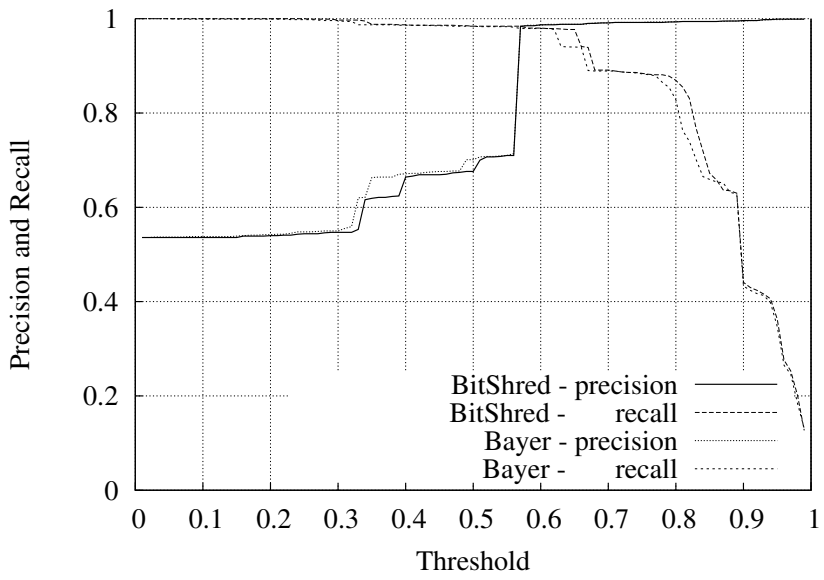


Figure 3.13: Clustering quality based upon behavior profiles

# of profiles	Clustering	Elapsed Time	Required Memory (max)
2,658	BAYER-EXACT	16s	86MB
	BITSHRED-EXACT	4s	12MB
75,692	BAYER-LSH	2h 25m 44s	4.3GB
	BITSHRED-SETBITS	24m 35s	89MB
	BITSHRED-EXACT	1h 2m 51s	89MB

Table 3.2: Scalability of systems

clean virtual machine. The VM is started, and observed behaviors such as system calls, conditional checks, etc. are recorded as features.

Bayer et al. provided us with their implementation of clustering as well as the 2,658 behavior profiles they used to measure accuracy from their paper [22]. In this data set, each behavior profile is a list of feature index numbers. The total number of features was 172,260. In our experiments, we used only a 1KB fingerprint size since the number of features was relatively small.

As shown in Table 3.2, an exact clustering took 16s and 86MB of memory using the code from Bayer et al. BitShred took 4s (4x as fast) and used 12MB of memory (7x less memory). The average error was 2% using the 1KB fingerprint. Figure 3.13 depicts the exact clustering vs. BitShred as a function of precision and recall. Both had the same

precision of 0.99 and recall of 0.98 when $t = 0.61$. Overall, BitShred was faster and used less memory, while not sacrificing accuracy for dynamic analysis feature sets.

Although Bayer et al. made the 2,658 profiles they used for accuracy, they did not provide all 75,692 profiles they used when measuring performance. In order to measure performance on this size of data, we synthetically generated 73,034 variants using the 2,658 profiles as a basis. We then ran the code from Bayer et al. on 75,692 profiles using the same parameters as described in [22]: $k = 10$, $l = 90$, and $t = 0.7$. BAYER-LSH took 2h 25m 44s using 4.3GB of memory and performed 236,132,556 distance computations. BITSHRED-SETBITS[4] took 24m 35s (5.9x as fast) using 89MB of memory (49x less memory) and computed the similarity of 1,021,322,219 pairs when $t = 0.7$. (Note that BITSHRED-SETBITS performed 4.3x more distance computations.) Even BITSHRED-EXACT at 1h 2m 51s outperformed (2.3x as fast) BAYER-LSH.

3.6 Extension of BitShred

Normalization. Our preprocessing and normalization process currently only unpacks malware. There are a variety of other techniques that can be used to maximize the probability of finding similar code. For example, renaming all general purpose-registers to some constant would prevent simple register renaming to thwart similarity analysis. Such techniques would be performed as a pre-processing step independent of the core BitShred algorithm. We leave an investigation into such extensions as future work.

Additional Applications. There are a number of other security applications for automatic code similarity detection. For example, BitShred can be used for plagiarism detection, similar to MOSS [146]. One immediate application is for finding copyright violations, e.g., by compiling all GPL libraries and then using BitShred to check for GPL violations, as done in [68]. We leave the exploration of these scenarios as future work.

Recent research shows that there is extensive code reuse in open source software [114]. Based on this observation, it is possible that a bug already fixed in one project may still exist in other projects if those projects shared their code. One good approach for finding this copy-pasted buggy code on a binary code level would be to compile the buggy source code with different compiler versions and options to generate various possible buggy binary code. BitShred allows us to quickly check code for possible binary code sequences of

[4]We sorted the samples based on the number of set bits in fingerprints. Then each sample only needed to be compared to the samples whose numbers of set bits were within the input threshold.

known vulnerabilities.

3.7 Summary

In this chapter, we presented BitShred, a system for large-scale malware triage and similarity detection. The key idea behind BitShred is the use of feature hashing to reduce the high-dimensional feature space in malware analysis, which is supported by theoretical and empirical analysis. Our approach makes inter-malware comparisons in typical large-scale triage tasks, such as clustering and finding nearest neighbors, up to an order of magnitude faster than existing methods while using less memory. As a result, BitShred scales to current and future malware volumes where previous approaches do not. We have also developed a distributed version of BitShred where $2x$ the hardware gives $2x$ the performance. In our evaluation, we show that we can scale to clustering over 1.9 million malware per day.

Chapter 4

Semantic Correlation Identification Between Malware

Clustering based exclusively on code similarity detection (§3) acts like a blackbox, telling us only that programs are grouped because they are similar. In other words, clustering itself does not tell us why the programs are grouped, i.e., what the distinguishing or common characteristics are within the program group. For example, large-scale malware analysis has so far focused primarily on grouping malware into families, or finding the malware samples most similar to an input malware sample. However, these analyses offer very little insight into *why* a certain set of malware samples are clustered into a family, and why other samples are clustered into a different family. Typically, an analyst still will need to invest significant effort into manual post-analysis in order to understand the similarities and differences between malware families.

Co-clustering (§2.5) goes one step further than clustering and tells us why programs are similar by simultaneously clustering features as well as programs. For example, co-clustering allows us to group two programs and to name the features that explain why they are similar (e.g., a significant amount of shared code) and why they are different (e.g., contacting different command and control hosts).

In this chapter, we enhance BitShred, our system for fast similarity analysis and malware triage, with co-clustering algorithms that can not only cluster malware, but also perform semantic analysis to determine which features distinguish identified malware families. These ideas are based on the idea of co-clustering, where we cluster together features and malware to identify the features that matter for a particular family. Why do we do both clustering and co-clustering? Co-clustering is more expensive because it must consider

both what features malware pairs have in common, as well as what features are important. Hierarchical clustering is faster since it only needs to determine whether or not malware is similar. Thus we run hierarchical clustering to identify families and co-clustering to identify inter-family and intra-family semantic features. For example, in our experiments, co-clustering automatically identifies the particular IP address contacted as a distinguishing feature within the Allaple malware family.

4.1 Mining Software Fingerprints

We extend BitShred to automatically mine the structure of the programs and program groups to provide insight into the structural relationships between different programs. Recall that the basis of our clustering (and indeed, the success of nearly all malware clustering approaches) is the existence of similar fingerprints such as behaviors or code sections. If we could discover fingerprints co-occurring (and not co-occurring) in programs across different groups, we could learn what features distinguish one program group from another. The resulting code fingerprint clusters reveal relationships between different programs, both within and across program groups, e.g., a single large code fingerprint cluster within the same program group along with many smaller fingerprint clusters.

Given n programs, the m length of BitShred fingerprints can be represented as $n \times m$ binary matrix $\mathbf{M}$ where each row is a program and each column is a particular feature. Then the binary matrix $\mathbf{M}$ completely encodes the features in fingerprints where an entry i, j is 1 if (hashed) feature g_i appears in program p_i and 0 otherwise. This intuition led us to the idea of using co-clustering to auto-correlate both features and programs simultaneously. Within the matrix, co-clustering auto-correlates by creating sub-matrices among columns (features) and rows (programs), where each sub-matrix is a highly correlated program/feature pair.

Co-clustering allows us to discover substantial, non-trivial structural relationships between programs, many of which would not be discovered with simpler approaches. For example, consider how the following simple approaches for mining features between two programs would be limited:

- Identify all common features between clusters. This can be accomplished by taking the bitwise-and ($\wedge$) of the bit vectors. However, we would miss identifying code that is present in 99% of a cluster, e.g., features that are shared by most, but not all, of the likely programs of a cluster.

- Identify all distinctive features in a group of programs. This can be accomplished

$$\mathbf{M} = \left|\begin{array}{ccccc} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{array}\right| \quad \Rightarrow \quad \left|\begin{array}{ccc|cc} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{array}\right| = \mathbf{M}'$$

Figure 4.1: $\mathbf{M}$ is co-clustered to identify the checkerboard sub-matrix $\mathbf{M}'$ of highly correlated malware/feature pairs.

with bitwise xor ($\oplus$) of the bit vectors. This, however, would have limited value for the same reasons as above.

- Cluster features either *before* or *after* programs have been clustered. Note, however, this approach would also result in misleading information, e.g., clustering the features *after* clustering the programs would not reveal structural similarity in different clusters, and clustering the features *before* clustering the programs may result in poor program clusters if there are many feature clusters that are common to multiple groups of program clusters.

We introduce some terminology to make co-clustering more precise. A matrix is *homogeneous* if the entries of the matrix are similar, e.g., they are mostly 0 or mostly 1, and define the *homogeneity* of a matrix to be the (larger) fraction of entries that have the same value. We define a *row-cluster* to be a subset of the rows M (i.e., programs) that are grouped together, and a *column-cluster* to be a subset of the columns (i.e., the features) that are grouped together. The goal of co-clustering is to create a pair of row- and column-labeling vectors:

$$\mathbf{r} \in \{1, 2, ..., k\}^n \quad \text{and} \quad \mathbf{c} \in \{1, 2, ..., \ell\}^m$$

The sub-matrices created are homogeneous, rectangular regions. The number of rectangular regions is either given as input to the algorithm or determined by the algorithm with a penalty function that trades off between the number of rectangles and the homogeneity achieved by these rectangles[1].

For example, Figure 4.1 shows a list of 5 programs where there are 5 possible features. It illustrates a co-clustering operation highlighting how a matrix with a low homogeneity can

[1]The goal is to make the minimum number of rectangles which achieve the maximum homogeneity. For this reason, co-clustering algorithms ensure that the homogeneity of the rectangles is penalized by the number of rectangles if they need to automatically determine k and ℓ.

be partitioned into a number of sub-matrices with high homogeneity. The result is the 5×5 matrix $\mathbf{M}$. Co-clustering automatically identifies the clustering to produce sub-matrices, as shown by the checkerboard $\mathbf{M}'$. The sub-matrices are homogeneous, indicating highly correlated feature/program pairs. In this case, the labeling vectors are $\mathbf{r} = (12122)^T$ and $\mathbf{c} = (21121)^T$. These vectors indicate that row 1 in **M** mapped to row cluster 1 (above the horizontal bar) in **M'**, row 2 mapped to row cluster 2 (below the horizontal bar), etc.; the same occurs for the column vectors for features. We can reach two clustering conclusions. First, the row clusters indicate that programs s_1 and s_3 are in one family, and s_2, s_4, and s_5 are in another family. Second, the column clusters indicate that the distinguishing features between the two clusters are features 2, 3, and 5.

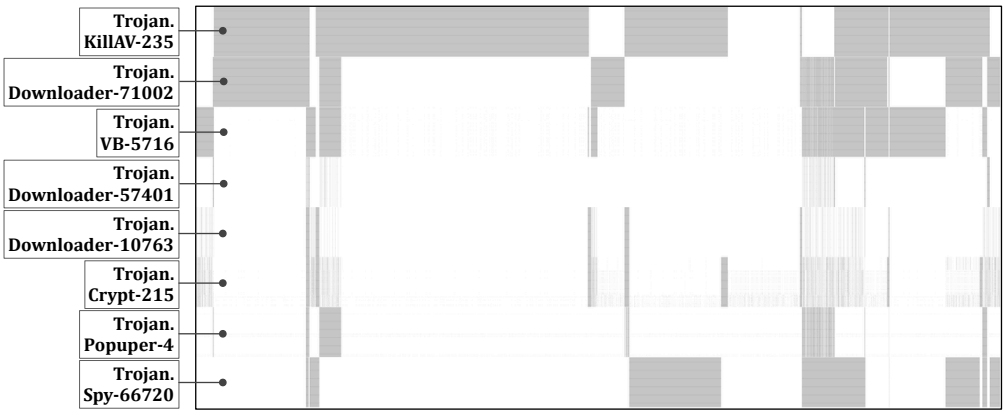
4.2 BitShred Semantic Architecture

We have adapted the cross-associations algorithm [35], redesigned for the Map-Reduce framework [129], to BitShred fingerprints. The basic steps are row iterations and column iterations. A row iteration fixes current column groups and iterates over each row, updating r to find the "best" grouping. In our algorithm, we seek to assign each row to a row group that would maximize the homogeneity of the resulting rectangles. The same occurs for columns while row groups are fixed. The algorithm performs a local optimal search (finding a globally optimal co-clustering is NP-hard [129]).

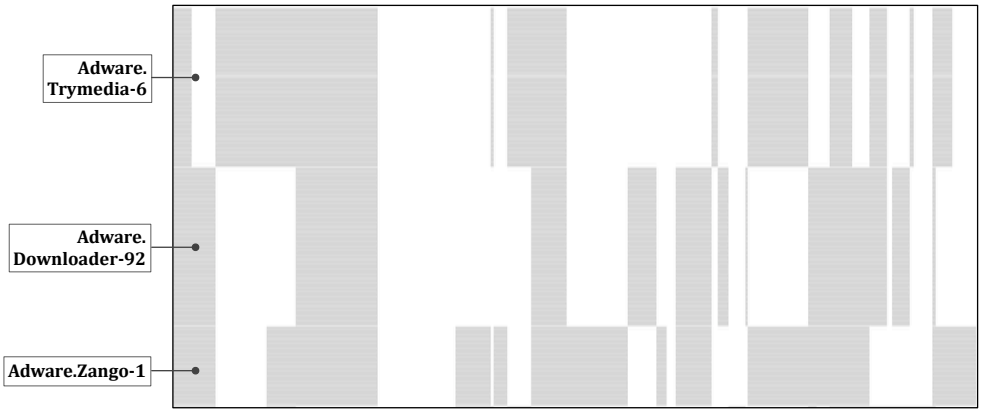
- **BITSHRED-SEMANTIC ($C \times F \rightarrow G'$):** Based on the BITSHRED-CLUSTER results, BITSHRED-SEMANTIC performs co-clustering on a subset of fingerprints to cluster features as well as malware samples. Co-clustering yields correlated features-malware subgroups G' which show the common or distinct features among malware samples.

Unlike typical co-clustering problems, co-clustering in BitShred needs to operate on hashed features, i.e., our fingerprints are not the features themselves, but hashes of these features. However, because our feature hashing is designed to approximately preserve structural similarities and differences between malware samples, we can apply co-clustering on our hashed features just as if they were regular features and still extract the structural relationships between the malware samples, with the increased computational efficiency that comes from feature hashing. To our knowledge, our results are the first to demonstrate that if co-clustering algorithms can be combined with the appropriate feature hashing functions, they can still extract the underlying structure of the data accurately.

(a) A typical matrix before co-clustering



(b) 8 different kinds of Trojans



(c) 3 different kinds of Adware

Figure 4.2: Semantic feature information (Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.)

Figure 4.2a shows a matrix before co-clustering where each row is a malware fingerprint and each column is a particular feature. Figure 4.2b and Figure 4.2c graphically depict the results of co-clustering on 8 different kinds of Trojans and 3 different kinds of Adware, respectively. Co-clustering reveals semantic feature information, i.e., the checkerboard patterns which describe distinguishing or common features across the malware families. I discuss both inter- and intra-family feature extraction in detail in §4.4.

There have been a number of co-clustering algorithms developed recently by machine learning and data mining researchers trying to discover structure in large sparse binary matrices [19, 35, 46, 58, 66]. The scale of our data volume, however, implies that we need an algorithm that can be easily used in a distributed framework. We chose the cross-associations algorithm [35], re-designed for the Map-Reduce framework [129], for two reasons: (1) it is fully automatic, requiring no parameters to be pre-specified, and (2) it is specially designed to discover structure in large, sparse, binary matrices, unlike algorithms that are designed for more generic matrices, and are therefore less efficient for our problem.

For completeness, we now briefly describe the cross-associations algorithm and how it may be used in a Map-Reduce framework. First assume that the number of row-clusters k and the number of column-clusters l are fixed. The algorithm alternates between a row-clustering phase and a column-clustering phase—in each phase, it re-arranges rows (and likewise, columns) into appropriate row-clusters (and likewise, column-clusters) so that the partitioning of the matrix M into sub-matrices (based upon the row-clusters and column-clusters) is strictly improved. Intuitively, the algorithm moves a row into a new row-cluster if the homogeneity of the affected sub-matrices increases by this movement, and in each phase it selects the best row-cluster for each of the rows. This is repeated until no further improvement in the sub-matrices, homogeneity can be achieved by a movement of a row or column cluster. The algorithm then increases k and l to see if the quality of the clustering could be improved. If the quality of the clustering cannot be improved, the algorithm stops.

Iterative Co-clustering. We also explore a new usage of co-clustering in malware analysis called *iterative co-clustering*. Iterative co-clustering would be expected to have at least three advantages. First, it can be used to perform malware clustering without choosing a fixed global similarity threshold, which is often not a trivial task. Second, it does not suffer from a local minimum, i.e., co-clustering may fail to find (near-) optimal results due to some dominant sample groups. Finally, it can be used to identify "components" or meaningful feature groups.

Iterative co-clustering is motivated by our analysis of co-clustering results. Co-clustering

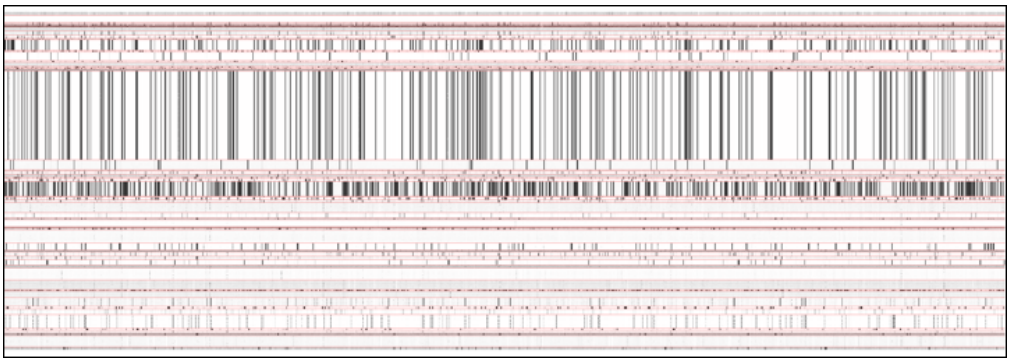


Figure 4.3: Co-clustering on 1,000 malware samples

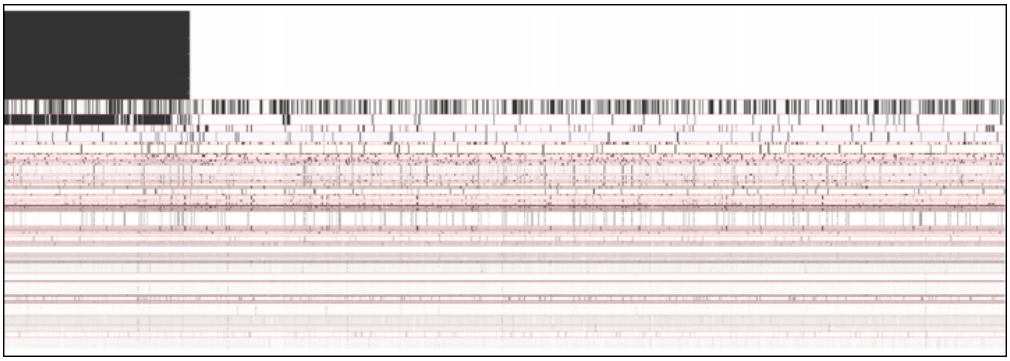


Figure 4.4: Results after permuting row/column groups to place the most dominant row group at the top

can suffer from a local minimum which prevents the algorithm from finding better correlated row-column intersections. For example, Figure 4.3 shows co-clustering results on 1,000 malware samples and Figure 4.4 shows the same co-clustering results after permuting row and column groups to place the most dominant (largest) row group at the top. As depicted in Figure 4.4, the first dominant row group prevented co-clustering from identifying better-correlated sub-matrices.

In order to mitigate this local minimum problem, we divide (intermediate) co-clustering results into two groups: the most dominant group and the rest of groups. We then re-apply co-clustering on each group. Our hypothesis is that co-clustering can find further correlation among the rest of the groups by excluding the dominant group causing a local minimum.

Cost Functions for Co-clustering. The current cost function for co-clustering is to minimize the description length, i.e., to maximize the homogeneity of the matrix. Co-clustering

has been widely used in bioinformatics and text mining, and different methods have been suggested to identify relevant clusters (§2.5.1). As future work, I will explore different cost functions to find a more efficient cost function for semantic analysis.

4.3 Implementation

BITSHRED-SEMANTIC was designed for the Map-Reduce framework and implemented in 1200 lines of Java. We implemented a Python wrapper to iterate row and column operations to find an optimal co-clustering. We also implemented BITSHRED-SEMANTIC using 3,000 lines of C and OpenMP API[2] to take advantage of multiprocessor platforms.

4.4 Evaluation

We used BITSHRED-SEMANTIC to identify semantically distinguishing features among malware families. We performed a variety of experiments and found that, overall, co-clustering automatically identified both inter-family and intra-family semantic features. Typical features identified included distinguishing register keys set and internet hosts contacted.

4.4.1 Co-clustering of Behavior-Based Profiles

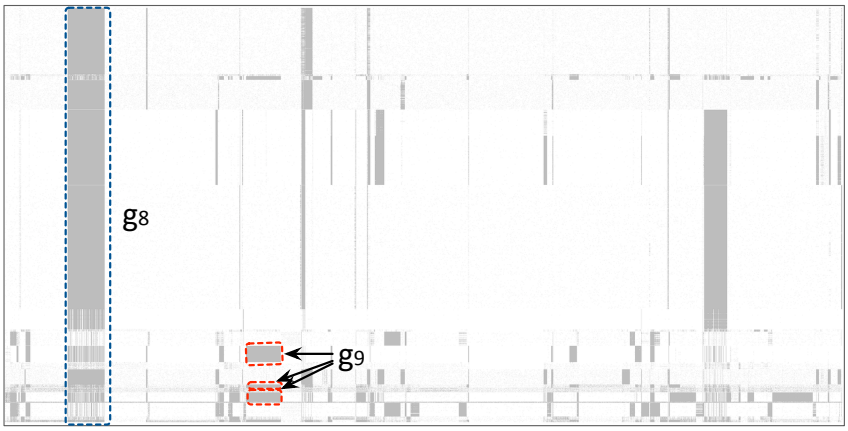
We performed a full co-clustering on the 2,658 behavior profiles from Bayer et al. that were used for their paper [22]. Figure 4.5a depicts the malware/feature matrix before co-clustering. We then co-clustered the profiles, which took 15 minutes.

Figure 4.5b shows the complete results. The checkerboard pattern corresponds to the sub-matrices identified as being homogeneous, i.e., corresponding malware/feature pairs that are highly correlated. For example, the large dark sub-matrix labeled g_8 corresponds to the fact that most malware had the same memory-mapped files including WS2HELP.dll, icmp.dll, and ws2_32.dll. The sub-matrix g_9 shows a commonality between two families but no others. The commonality corresponds to opening the file \Device\KsecDD.

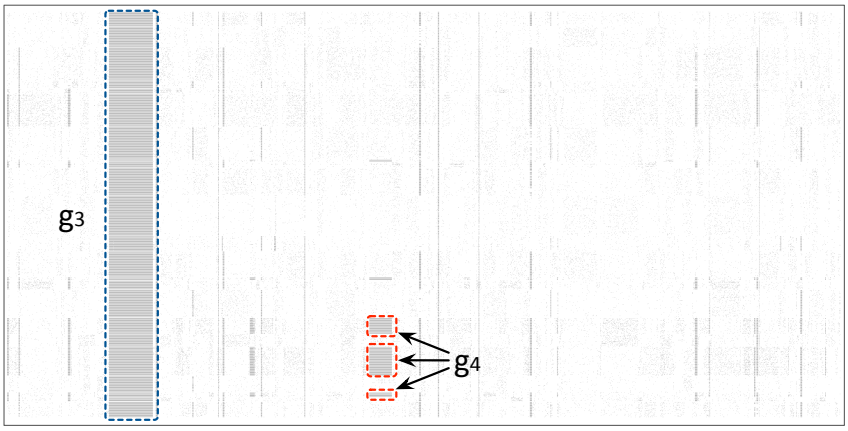
Figure 4.5c focuses on the 717 samples in the Allaple malware family. One semantic feature, labeled g_3, is that almost all samples use the same memory-mapped files such as winrnr.dll, WS2HELP.dll, icmp.dll, and ws2_32.dll. More importantly, we also found that many family members were distinguished by the register entry they created (e.g., HKLM\

[2] http://openmp.org/wp/

(a) A typical matrix before co-clustering

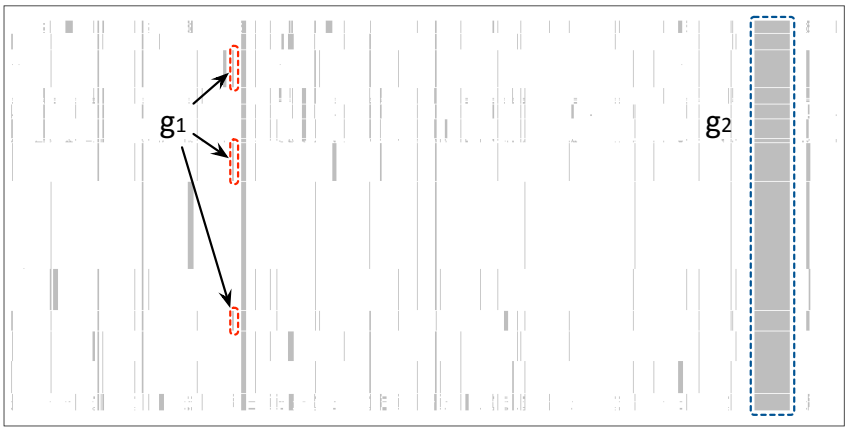


(b) Inter-family analysis based on dynamic behavior profile

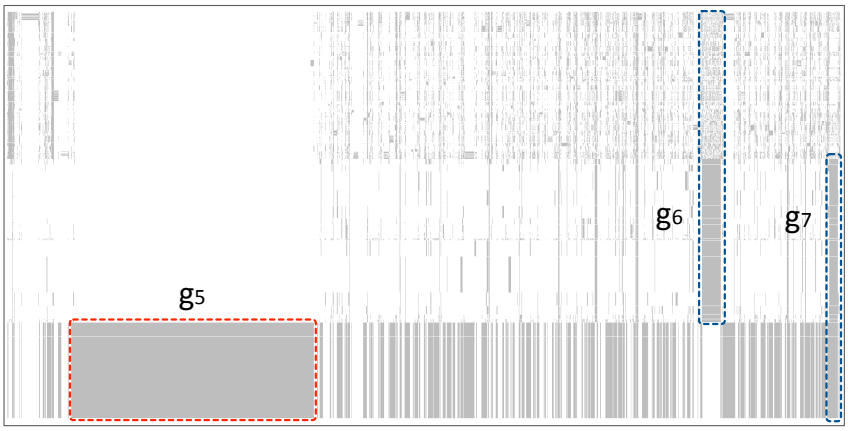


(c) Intra-family analysis based on dynamic behavior profile

Figure 4.5: Feature extraction by co-clustering (Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.)



(a) Intra-family analysis based on static code analysis



(b) Inter-family analysis based on static code analysis

Figure 4.6: Feature extraction by co-clustering (Grey dots represent 1 in the binary matrix, i.e., the presence of a feature.)

SOFTWARE\CLASSES\CLSID\ {7BDAB28A-B77E-2A87-868A-C8DD2D3C52D3} in one sample) and the IP address they connected to, e.g., one sample connected to 24.249.139.x while another connected to 24.249.150.y (shown as g_4).

4.4.2 Co-clustering of n-gram Features

We also experimented with co-clustering using the n-gram features. Figure 4.6a shows intra-family co-clustering for the Trojan.OnlineGames malware family. The features labeled g_2 indicate all code from the code entry to a particular point that overlaps. The feature set g_1 corresponds to new functionality in a few variants that makes tcp connections

to a new host not found in previous variants.

We also performed inter-family analysis. In this set of experiments, we envision that an analyst would use co-clustering to mine similarities and differences between malware family members or between malware families. We picked the Trojan.Dropper, Trojan.Spy, Trojan.OnlineGames, and Adware.Downloader families, which have 1,280 total members. The total time taken by co-clustering was 10 minutes (about 2s per sample), with about 1 minute for each column and row iteration. We used 10 maps for each row iteration and 64 maps for each column iteration.

Figure 4.6b shows the resulting co-clustering. Trojan.Dropper and Trojan.Spy were grouped together by co-clustering, which is accurate: we manually confirmed that the samples we have from those families are not well distinguished. The sub-matrix labeled g_5 is one distinguishing feature that corresponds to Adware.Downloader connecting to a particular host on the Internet. The sub-matrix labeled g_6 corresponds to data section fragments shared among members of the Trojan family, but not present in Adware. The sub-matrix labeled g_7 corresponds to shared code for comparing memory locations. This code is shared between Adware.Downloader and Trojan.OnlineGames, but not Trojan.Spy/ Trojan.Downloader.

4.4.3 Iterative Co-clustering

Iterative co-clustering can allow us to identify better-correlated sub-matrix patterns. For example, the samples in Figure 4.4 were divided into two groups: the first dominant group and the remaining group. Co-clustering is then run on each group separately. Figure 4.7 depicts the co-clustering result for the first dominant group, and the zoomed-in result of Figure 4.8 shows that iterative co-clustering identifies more fine-grained correlated patterns. In addition, iterative co-clustering on the remaining group identifies the second dominant row group, as shown in Figure 4.9.

Figure 4.10 shows the merging of the co-clustering results for the first and second largest row groups where the positions of the features from the first largest row group are fixed first, and the features from the second largest row group are aligned. Figure 4.11 shows the result after the 4th iterative co-clustering is done.

Iterative co-clustering can be used to find an effective similarity threshold value for clustering, which is often not a trivial task. In other words, without pre-defining a threshold value, we can perform iterative co-clustering on the samples to identify the dominant row groups. A similarity threshold for clustering can then be chosen in the range between the

Figure 4.7: Co-clustering on the first dominant row group

Figure 4.8: Co-clustering on the first dominant row group (zoomed in)

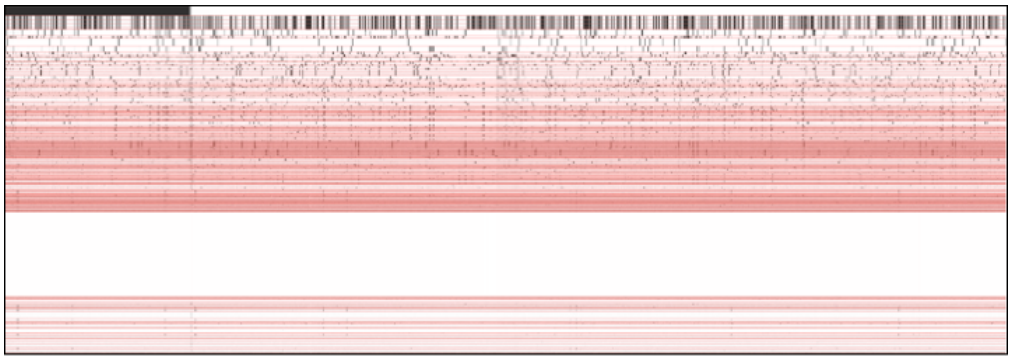


Figure 4.9: Co-clustering on the remaining samples after excluding the first dominant row groups

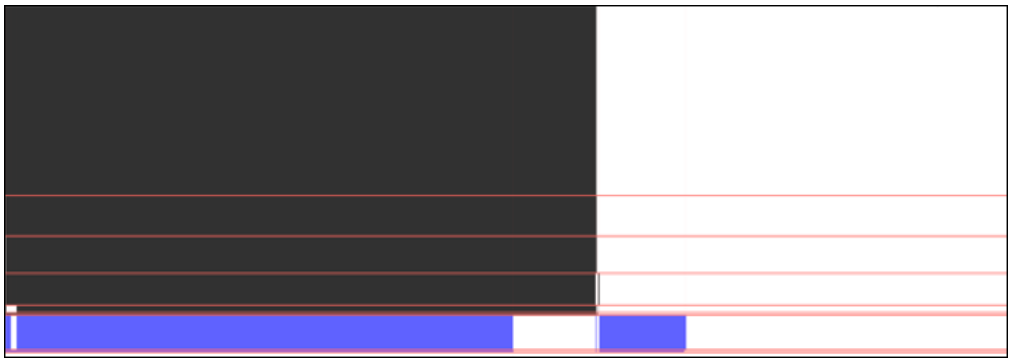


Figure 4.10: Arranging the first and the second largest row groups

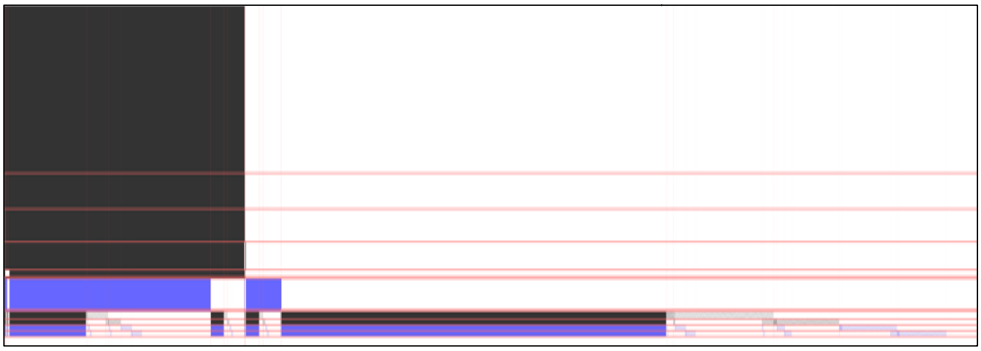


Figure 4.11: After 4th iterative co-clustering

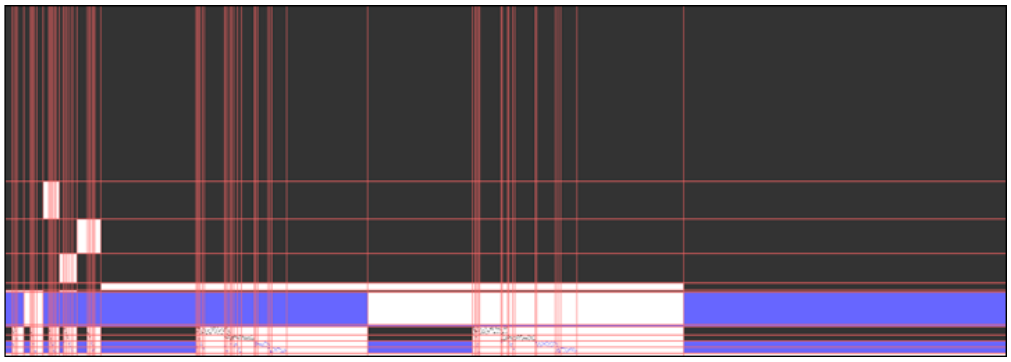


Figure 4.12: After 4th iterative co-clustering (zoomed in)

lowest similarity within the dominant row group and the highest similarity between the dominant row group and the remaining group.

Iterative co-clustering also can be utilized to identify "components". For example, as shown in Figure 4.12, a large column group in the first dominant row group was divided into several smaller column groups by adding the next iterative co-clustering results. The smaller column groups could be meaningful feature groups or "components".

4.5 Summary

In this chapter, we proposed novel techniques for performing semantic analysis using co-clustering. While simultaneously clustering rows (samples) and columns (features), co-clustering can extract semantic features between malware samples and families. The extracted features provide insight into the fundamental similarities and differences between and within malware data sets.

Chapter 5

Evolutionary Relationship Inference Between Malware

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 chapter, we propose automatic lineage inference algorithms and metrics for the systematic investigation of software lineage as described in Figure 5.1, and ask four basic research questions:

1. **Can we automatically infer software lineage?** Previous research focused on studying known software history and lineage [60, 108, 162], not creating lineage. Creating lineage is different from building a dendrogram based upon similarity [84, 87, 105].

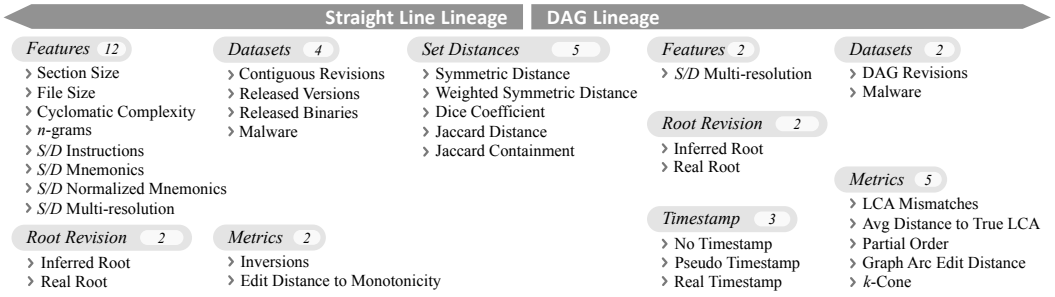


Figure 5.1: Design space in software lineage inference (*S/D* represents static/dynamic analysis-based features.)

A dendrogram can be used to identify families, but 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 (§5.2.1) and directed acyclic graph (DAG) lineage (§5.2.2). In addition, we extend our approach for straight line lineage to k-straight line lineage (§5.2.1.4). We build iLINE to systematically evaluate the effectiveness of our lineage inference algorithms using twelve software feature sets (§5.1.1), five distance measures between feature sets (§5.1.2), two policies on the root identification (§5.2.1.1), and three policies on the use of timestamps (§5.2.2.2).

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?** Previous research focused on building a phylogenetic tree of malware [84, 87], but did not provide *quantitative* metrics to scientifically measure the quality of their output. Good metrics are necessary to quantify the quality of our approach 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 (§5.3.1). 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 program, p_i or 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 (§5.3.2): 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 asks, "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 §5.3.3, we discuss how the metrics are related, which metric is useful for measuring 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, i.e., cases in which we have *(i)* control over the 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 §5.5.4.

   We argue that it is also necessary to systematically validate a lineage inference technique with "goodware", e.g., open source projects. Since malware is often surreptitiously developed by adversaries, it is typically difficult 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 for which we have the ground truth on. For straight line lineage, we collected three kinds of goodware data sets and also used malware data sets: *(i)* contiguous revisions from a version control system—371 revisions from 3 programs representing 4 years of combined history, *(ii)* released versions distributed to end users—271 releases from 5 programs representing 55 years of combined history, *(iii)* actual released program binaries from `deb` or `rpm` package files—355 releases from 7 programs representing 40 years of combined history, and *(iv)* malware data sets collected by the Cyber Genome program—84 samples with known lineage in 7 clusters, which include bots, worms, and Trojan horses. Regarding DAG lineage experiments, we downloaded revision histories of goodware that have multiple branching and merging points—780 revisions from 10 programs representing 11 years of combined history. We also used two malware families collected by the Cyber Genome program. Each family has a DAG evolution history, and there are 30 samples in total.

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 §5.5.5. This is important because we may not be able to understand malware evolution without understanding the limits of our approach with goodware.

5.1 Fingerprinting for Lineage Inference

5.1.1 Binary Abstraction

In order to fingerprint software, we use three program analysis methods: syntax-based analysis, static analysis, and dynamic analysis as described in §2.1.1. 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 are not used as we are interested in results in security scenarios where source code is not typically available, e.g., forensics, proprietary software, and malware.

Using Previous Observations. Previous work analyzed software release histories to understand a software evolution process. It has often been observed that program size and complexity tend to increase as new revisions are released [60, 99, 162]. This observation also carries over to security scenarios, e.g., the complexity of malware is likely to grow as new variants appear [44]. 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:** iLINE also calculates the file size, including code and data.

- **Cyclomatic complexity:** Cyclomatic complexity [111] 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.

```
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 5.2: Example of feature extraction

Using Syntax-based Features.

- **n-grams.** iLINE slides a window of n bytes over the identified byte sequence of program code to obtain n-grams.

Using Static Features. In our experiments, we employ normalization steps, e.g., normalizing operands and/or instruction mnemonics. This was motivated by our observations when we analyzed the error cases in the lineages constructed. 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 [96, 103]. 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 5.2c, 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 5.2d. By normalizing the 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 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.

Using Dynamic Features. For malware specifically, iLINE traces an execution using a binary instrumentation tool and collects a set of instruction traces. As with static features, iLINE also generates additional sets of features by normalizing operands and mnemonics.

Using Multi-resolution Features. Besides considering each feature set individually, iLINE also 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.

5.1.2 Distance Measures Between Feature Sets

When feature sets are extracted from programs, we can measure the distance between two feature sets using various distance metrics as discussed in §2.2. 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. Distance metrics

other than symmetric distance may be used for lineage inference as well, e.g., the *Dice coefficient distance*, the *Jaccard distance*, and the *Jaccard containment distance* can all be used to calculate the dissimilarity between two sets.

Besides the four distance measures above, which are all symmetric, i.e., $\texttt{distance}(f_1, f_2) = \texttt{distance}(f_2, f_1)$, we also evaluated an *asymmetric* distance measure to determine the direction of derivation between p_1 and p_2 using the *weighted symmetric distance*.

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 [44, 162]. 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 choose a direction of derivation. In this paper, we set $\texttt{C}_{\texttt{del}} = 2$ and $\texttt{C}_{\texttt{add}} = 1$. (We leave the investigation of the effect of these values as future work.) 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.

In this chapter, we use SD as a representative distance metric when we explain our lineage inference methods. We evaluated the effectiveness of all five distance measures on inferring lineage using SD as a baseline (see §5.5.4). Regarding metric-based features, e.g., section size, we measure the distance between two samples as the difference of their metric values.

5.2 iLINE Architecture

Our goal is to automatically infer the software lineage of program binaries. We build iLINE to systematically explore the design space illustrated in Figure 5.1 to understand the advantages and disadvantages of our algorithms for inferring software lineage. We applied our algorithms to two types of lineage: straight line lineage (§5.2.1) and directed acyclic graph (DAG) lineage (§5.2.2). 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; 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.

5.2.1 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 (§5.2.1.4).

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, from the 1st revision p^1 to the 100th revision p^{100}.

5.2.1.1 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 on Lehman's observation [99]. 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 instead of deleting 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 higher code complexity [44]. In addition, provenance information such as first seen date [50] and tool-chain components [141] 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.

5.2.1.2 Inferring Order

From the selected root revision, iLINE greedily picks the closest revision in terms of the symmetric distance as the next revision. Suppose that 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 in which 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 §5.1.1). 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.

5.2.1.3 Handling Outliers

As an optional step, iLINE handles outliers (if any) in our recovered ordering. Since iLINE constructs lineage in a greedy way, if one revision is not selected, 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 ($p^1 \rightarrow p^3 \rightarrow p^4$). p^4 and p^5 are neighboring revisions; then it is likely that $\text{SD}(p^4, p^5) < \text{SD}(p^4, p^2)$ and p^5 will be selected ($p^1 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$). The probability of selecting p^2 gets increasingly lower as more revisions are added. 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 5.10 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., $p^1 \rightarrow p^2 \rightarrow p^3 \rightarrow p^4 \rightarrow p^5$.

5.2.1.4 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 [69, 75, 165, 167]. 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 determine an appropriate distance threshold t, we explore the entire range of t and find the value where the resulting number of clusters becomes stable (see Figure 5.7 for an example).

5.2.2 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* arc, i.e., a revision with multiple children. Merging is denoted by a node with more than one *incoming* arc, i.e., a revision with multiple parents.

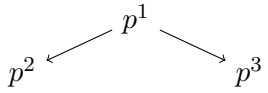
5.2.2.1 Identifying the Root Revision

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 §5.2.1.1.

5.2.2.2 Building Spanning Tree Lineage

iLINE builds (directed) spanning tree lineage by greedy selection. This step is similar to 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 will look 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, $\hat{N}^c$ 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 on the minimum symmetric distance. In other words, we insert nodes in the order of timestamps.

5.2.2.3 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 arc—revisions with multiple children. In order to pinpoint *merging* points, iLINE adds non-tree arcs (also known as cross arcs) to spanning tree lineage.

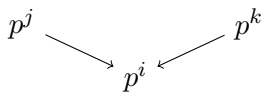
For every non-root node p^i, iLINE identifies a unique feature set u^i that does not come from its parent p^j, i.e., $u^i = \{x : x \in f^i \text{ and } x \notin f^j\}$. Then iLINE identifies possible parents $p^k \in N$ as follows:

i) if real/pseudo timestamps are given, p^k with earlier timestamps than the timestamp of p^i

ii) if symmetric distance measures such as SD, DC, JD, and JC are used, non-ancestors p^k added to G before p^i

iii) and if the asymmetric distance measure WSD is used, non-ancestors p^k satisfying $\text{WSD}(p^k, p^i) < \text{WSD}(p^i, p^k)$.

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:



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

5.3 iEVAL Architecture

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

5.3.1 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 orderings for all $\binom{|N|}{2}$ pairs. The EDTM is the minimum number of revisions that need to be removed for the remaining nodes in the lineage graph G to be 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 misplaced in G.

For example, we have 5 revisions of a program and iLINE outputs lineage 1 in Figure 5.3a and lineage 2 in Figure 5.3b. 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

(a) Lineage 1 (b) Lineage 2

Figure 5.3: Inversions and edit distance to monotonicity

(delete p^2). As shown in both cases, the number of inversions can be different even when the EDTM is the same.

5.3.2 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*.

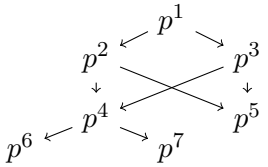


Figure 5.4: Lowest common ancestors

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 5.4, $\text{SLCA}(p^4, p^5) = \{p^2, p^3\}$, 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 follows 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 5.4, 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

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

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 identifies which one of x and y comes first: either x or y, or they are incomparable if they are not each other's ancestors. For example, in Figure 5.4, 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 orderings 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 $\texttt{Adj}^*(x)$ denote 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 5.4, 2-CONE of p^1 is $\{p^2, p^3, p^4, p^5\}$. Then given the 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 types 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

SPECIAL CASE ⟵ · · · · · · · · · · · · · · · ⟶ GENERAL CASE			Property measured
Straight Line	DAG		
Inversions	**PO**	SLCA	Order/Topology
EDTM		**GAED**	Misplaced nodes/arcs
		k-Cone	Descendants within depth k

Table 5.1: Relationships among metrics

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.

5.3.3 Relationships among Metrics

Table 5.1 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 that 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 misplaced 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? We recommend two metrics among the seven: 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.

Observe that PO mismatches and SLCA mismatches measure the same property of lineage and have similar accuracy results in our evaluation. However, it is more efficient to

compute PO mismatches 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.

5.4 Implementation

iLINE 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 §5.1.1, `gcc -S` output is used to compensate for 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 traces. Since some kinds of malicious activities require "live" connections, we also employ INetSim[5] to simulate various network services, e.g., web, 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 scalability reasons, we encode extracted features into bit vectors using the feature hashing technique as described in §3.1. Feature hashing allows to represent high-dimensional feature sets as cache-friendly 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 2.1 can be calculated efficiently by:

$$\text{SD}_{bv}(bv_1, bv_2) = S(bv_1 \otimes bv_2) \tag{5.1}$$

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 iLINE to perform

[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/

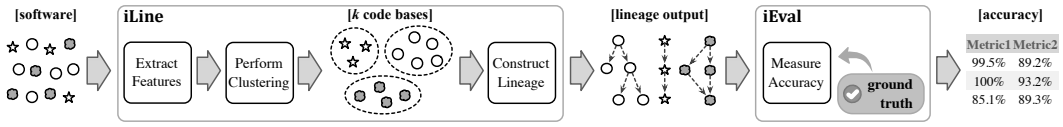


Figure 5.5: Software lineage inference overview

experiments with a number of variables quickly.

5.5 Evaluation

As depicted in Figure 5.5, we systematically evaluated our lineage inference algorithms using *(i)* ILINE to explore all of the design spaces described in Figure 5.1 with a variety of data sets and *(ii)* IEVAL to measure the accuracy of our outputs with respect to the ground truth.

5.5.1 Straight Line Lineage

Data sets. For straight line lineage experiments, we 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 considerably, from less than 10 minutes to more than a month. We excluded some revisions that only changed 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 5.2: 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 could 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 they 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 5.3: Data sets of released versions

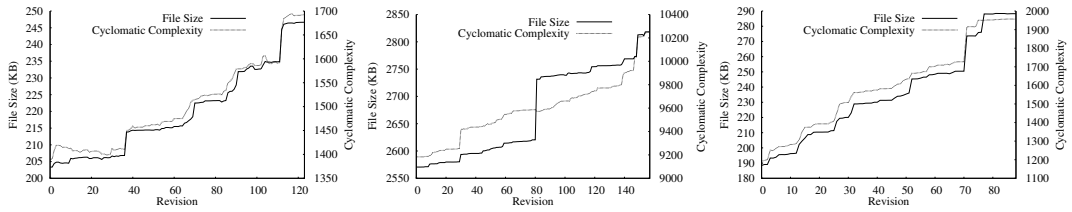
iii) **Actual Release Binaries.** We collected binary files (not source code) of released versions from `rpm` or `deb` package files.

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 5.4: Data sets of actual release binaries

The difference with contiguous revisions and released versions 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 representative of real-world scenarios in which we lack information about development environments.

iv) **Malware:** We used 84 samples with known lineage collected by the Cyber Genome



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

Figure 5.6: File size and complexity for contiguous revisions

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 5.5: Data sets of malware

Results. To determine which features provide the best lineage graph with respect to the ground truth, we evaluated different feature sets on diverse data sets.

		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 5.6: Mean accuracy for straight line lineage on goodware

i) **Contiguous Revisions:** In order to identify the first revision of each program, we measured the code complexity and code size of every revision. As shown in Figure 5.6, both file size and cyclomatic complexity generally increased as new revisions were released. For these three data sets, selecting the revision that had the minimum file size and cyclomatic complexity correctly identified the first revisions.

A lineage for each program was constructed as described in §5.2.1. Although section and file size achieved high accuracies, e.g., 95.5%–99.5%, these are not reliable features because many ties can decrease or increase the accuracies depending on random guesses. n-grams over byte sequences generally achieved better accuracies with an exception of 2-grams (small size of n). 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 demonstrates the effectiveness of using multi-resolution features for breaking ties.

ii) **Released Versions:** By selecting the revision that had the minimum code size, we correctly identified the first/root revisions. In some cases, simple feature sets, e.g., section/file size, could achieve higher accuracies than semantically rich feature sets (i.e., those requiring more expensive processes), such as instruction sequences. For example, iLINE with section size yielded 88.3% accuracy, while iLINE with instructions achieved 77.8% accuracy in `grep`. iLINE with instructions, 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:** Selecting the revision that had the minimum code size correctly identified the first/root revisions for `nano` and `openssh`. For the other five data sets, we performed the experiments with both the incorrect inferred root and 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 an incorrect revision as the first revision in `FileZilla`, which resulted in 51.6% accuracy; the accuracy increased to 99.8% with the correct root revision.

		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 5.7: Mean accuracy for straight line lineage on malware

iv) **Malware:** Selecting the sample that had the minimum code size correctly identified the first/root samples for all seven clusters. Section size achieved high accuracies, e.g., 93.3–100%, which showed that 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 their malicious activities if they are being monitored, e.g., examining 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 key reason for the differences in accuracy is that dynamic analysis followed a spe-

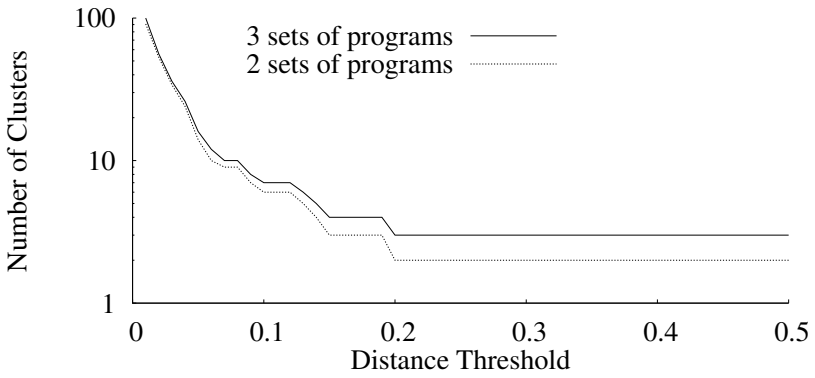


Figure 5.7: Clustering mixed data sets of 2 and 3 programs

cific execution path depending on the context. In `MC2`, for example, 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` because both programs had the same functionality and similar code section sizes. Figure 5.7 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 that 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—`memcached`, `redislite`, and `redis`—became stabilized to 3 from 0.2 to 0.5 distance threshold; this means they were successfully clustered for the subsequent straight-line lineage building process.

We also evaluated iLINE on three mixed malware data sets, each of which is a combination of the different clusters in Table 5.5: {`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 §5.5.5.

5.5.2 Directed Acyclic Graph Lineage

Data sets. For DAG lineage experiments, we evaluated iLINE on both goodware and malware.

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

We downloaded DAG revisions that had multiple branching and merging histories and compiled them 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 5.8: 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 5.9: Malware data sets for DAG lineage

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 implied 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.

[6] https://github.com/

		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 5.10: Mean accuracy for DAG lineage on goodware

i) **Goodware.** Without having any prior knowledge, ILINE achieved 71.5%–94.1% PO accuracies. By using the real root revision, ILINE increased accuracies to 71.5%–96.1%. For example, in case of `tig`, ILINE gained about 20% accuracy.

With pseudo timestamps, accuracies were low for most of the data sets, even with the real root revisions, e.g., 64.0%–90.9% (see §5.5.4). 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.

		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 5.11: Mean accuracy for DAG lineage on malware

ii) **Malware.** ILINE achieved 68.6%–75.0% accuracies without any prior knowledge. Using the correct timestamps, the accuracies notably increased to 86.2%–91.7%. While we obtained the real timestamps from the ground truth in our experiments, we can also leverage the first-seen date of malware, e.g., Symantec's Worldwide Intelligence Network Environment [50].

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 on static features.

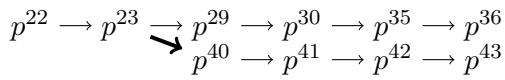
5.5.3 Performance

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 per second and 180 samples per second for straight line lineage and DAG lineage, respectively. As a comparison, our BitShred malware clustering system [75]—which was 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 does not, we conclude that our current implementation of iLINE is competitive in terms of performance.

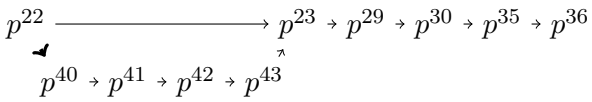
5.5.4 Discussion & Findings

Features. File and 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 for inferring malware lineage when malware authors are able to 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 (when compared to n-grams) accuracy was due to ties. Multi-resolution features performed the best, achieving 95.8–98.4% mean accuracy in straight line lineage on goodware. This is due to their use of both syntactic and semantic features.

Distance Metrics. Our evaluation indicates that our lineage inference algorithms perform similarly regardless of distance metrics except for the Jaccard containment (JC) distance. JC turns out to be inappropriate for lineage inference because it cannot capture



(a) Ground truth



(b) Constructed lineage with the use of pseudo timestamps

Figure 5.8: Error caused by pseudo timestamps in `uzbl`

evolutionary changes effectively. Suppose that there are three contiguous revisions p^1, p^2, and p^3. 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.

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 was challenging to determine the precise ordering between samples from different branches. For example, Figure 5.8 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) 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 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 5.9 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.

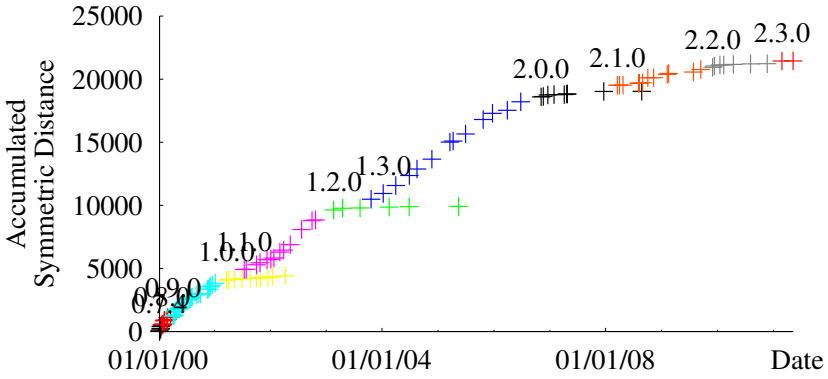


Figure 5.9: Development history of `nano`

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 difficult 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.

5.5.5 Limitations

Reverting and Refactoring. Regression of code is a challenging problem in software lineage inference. A revision that adds new functionalities is sometimes followed by stabilizing phases, including bug fixes. Bug fixes might be performed 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 issues with reverting and refactoring. In §5.2.1.3, we proposed 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 5.10 shows the symmetric distance between two adjacent revisions in the recovered lineage before we processed outliers. The outlier caused an exceptional peak of the symmetric distance at the rightmost side of Figure 5.10. 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

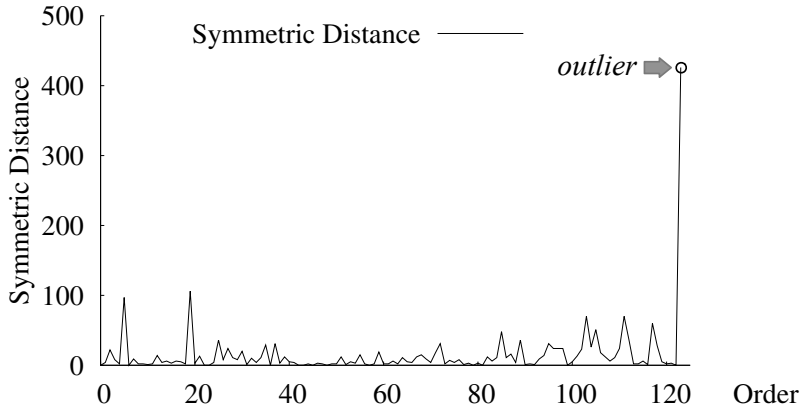


Figure 5.10: An outlier in `memcached`

if revisions are in the correct order. For example, a peak can be identified between two contiguous revisions when there is a huge update, such as a major version change. However, such false positives do not affect the 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 and 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 and pragmatic representation of the truth.

Root Identification. It is challenging to identify the correct roots of data sets when 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 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 the "first-seen" date of malware, e.g., Symantec's Worldwide Intelligence Network Environment [50] or tool-chain provenance, such as compilers and compilation options [141].

Clustering. Clustering may not be able to group programs accurately due to noise or algorithmic limitations. In order to simulate cases where clustering failed, we mixed bi-

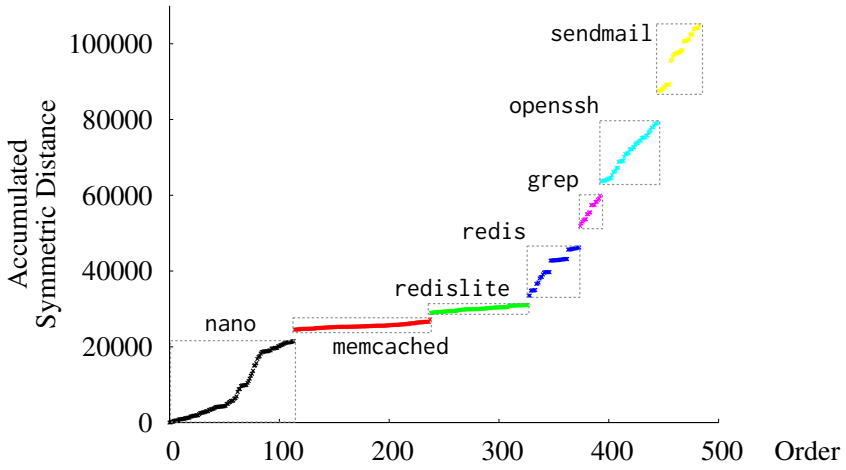


Figure 5.11: Recovered ordering of mixed data set

naries 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 5.11, revisions from each program group are located next to each other in the recovered order (each program is marked in a different color). This shows that 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 large 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 that 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 [148], or we may detect similar code that was compiled with different compilers and optimization options [88].

5.6 Summary

In this chapter, we proposed new algorithms to infer software lineage of binary programs 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 5.1 for software lineage inference and performed over 2,000 different experiments on large-scale real-world programs—1,777 goodware spanning 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 that iLINE effectively extracted evolutionary relationships among program binaries with over 84% mean accuracy for goodware and over 72% for malware.

Part II

Code Reuse Detection at the Source Code Level

Chapter 6

Software Vulnerability Tracking via Code Containment Detection

Programmers should never fix the same bug twice. Unfortunately, buggy code often gets copied from project to project and the programmer fixes the same bug on each project independently. This means resources are wasted in repeatedly diagnosing the same bug. We call clones of buggy code that have been fixed in only a subset of projects *unpatched code clones* because they are code clones to which patches have yet not been applied. Unpatched code clones are latent bugs which are likely to be vulnerable and can cause a serious vulnerability window, i.e., the time frame between when a vulnerability is disclosed and when a project containing the vulnerable code clone is fixed. It is therefore important to detect such latent vulnerabilities early before an adversary exploits them.

For example, the patch presented in Listing 6.1 was issued in July 2009 to fix a heap overflow bug in `libvorbis`. The patched vulnerability could cause a program crash or arbitrary code execution via a maliciously-crafted OGG file [119]. Unfortunately, we found 93 unpatched code clones of this bug in our November 2011 data set. Projects including `mplayer` and `libtritonus-java` in Debian, `mednafen` and `libvorbisidec` in Ubuntu, and `ffdshow` and `guliverkli` in SourceForge all had the same vulnerable unpatched code. In this case, a total of 93 packages were exposed to this known vulnerability for more than 800 days past the initial patch date.

```
--- a/lib/res0.c
+++ b/lib/res0.c
@@ -208,10 +208,18 @@
    info->partitions=oggpack_read(opb,6)+1;
    info->groupbook=oggpack_read(opb,8);
```

```
+   /* check for premature EOP */
+   if(info->groupbook<0)goto errout;
+
    for(j=0;j<info->partitions;j++){
        int cascade=oggpack_read(opb,3);
-       if(oggpack_read(opb,1))
-           cascade|=(oggpack_read(opb,5)<<3);
+       int cflag=oggpack_read(opb,1);
+       if(cflag<0) goto errout;
+       if(cflag){
+           int c=oggpack_read(opb,5);
+           if(c<0) goto errout;
+           cascade|=(c<<3);
+       }
        info->secondstages[j]=cascade;

        acc+=icount(cascade);
```

Listing 6.1: Patch in `libvorbis` for CVE-2009-3379

As a second example, consider the patch presented in Listing 6.2, which was issued in August 2010 to fix a bug in `FreeType` that used incorrect integer data types before version 2.4.2 was released. The original vulnerability could be exploited to cause a program crash or possibly arbitrary code execution [121]. Unfortunately, we found 185 unpatched code clones of this bug in our November 2011 data set. Projects including `vtk` and `paraview` in Debian, `qt4-x11` and `texlive-bin` in Ubuntu, and `MobileXpdf` and `TomPlayer` in SourceForge all had the same vulnerable unpatched code. In this case, a total of 185 packages were exposed to this known vulnerability for more than 400 days past the initial patch date.

```
--- a/src/smooth/ftsmooth.c
+++ b/src/smooth/ftsmooth.c
@@ -140,8 +140,26 @@
     cbox.xMax = FT_PIX_CEIL( cbox.xMax );
     cbox.yMax = FT_PIX_CEIL( cbox.yMax );
-    width  = (FT_UInt)( ( cbox.xMax - cbox.xMin ) >> 6 );
-    height = (FT_UInt)( ( cbox.yMax - cbox.yMin ) >> 6 );
+    if ( cbox.xMin < 0 && cbox.xMax > FT_INT_MAX + cbox.xMin )
+    {
+      FT_ERROR(( "ft_smooth_render_generic: glyph too large:"
+                 " xMin = %d, xMax = %d\n",
+                 cbox.xMin >> 6, cbox.xMax >> 6 ));
+      return Smooth_Err_Raster_Overflow;
+    }
+    else
+      width  = (FT_UInt)( ( cbox.xMax - cbox.xMin ) >> 6 );
+
```

```
+     if ( cbox.yMin < 0 && cbox.yMax > FT_INT_MAX + cbox.yMin )
+     {
+       FT_ERROR(( "ft_smooth_render_generic: glyph too large:"
+                  " yMin = %d, yMax = %d\n",
+                  cbox.yMin >> 6, cbox.yMax >> 6 ));
+       return Smooth_Err_Raster_Overflow;
+     }
+     else
+       height = (FT_UInt)( ( cbox.yMax - cbox.yMin ) >> 6 );
+
      bitmap = &slot->bitmap;
      memory = render->root.memory;
```

Listing 6.2: Patch in `FreeType` for CVE-2010-2807

As a third example, consider the patch presented in Listing 6.3, which was issued in August 2008 to address uninitialized memory access issues in the LZW decoder used in LibTIFF v3.8.2 and earlier. An attacker could potentially execute arbitrary code via a maliciously-crafted TIFF image by exploiting the vulnerability in the CODE_CLEAR code handling routine [118]. Unfortunately, we identified 95 unpatched code clones of this bug in our November 2011 data set. For example, `argyll` and `vxl` in Debian, `ivtools` and `insighttoolkit` in Ubuntu, and `CamStudio` and `CinePaint` in SourceForge were all affected. In this case, a total of 95 packages were exposed to this known vulnerability for over 1,100 days past the initial patch date.

```
--- tiff-3.8.2.orig/libtiff/tif_lzw.c
+++ tiff-3.8.2/libtiff/tif_lzw.c
@@ -604,12 +619,20 @@
     if (code == CODE_CLEAR) {
         free_entp = sp->dec_codetab + CODE_FIRST;
+        _TIFFmemset(free_entp, 0, (CSIZE-CODE_FIRST)*sizeof (code_t));
         nbits = BITS_MIN;
         nbitsmask = MAXCODE(BITS_MIN);
         maxcodep = sp->dec_codetab + nbitsmask;
         NextCode(tif, sp, bp, code, GetNextCodeCompat);
         if (code == CODE_EOI)
             break;
+        if (code == CODE_CLEAR) {
+            TIFFErrorExt(tif->tif_clientdata, tif->tif_name,
+            "LZWDecode: Corrupted LZW table at scanline %d",
+            tif->tif_row);
+            return (0);
+        }
+
         *op++ = code, occ--;
         oldcodep = sp->dec_codetab + code;
```

Listing 6.3: Patch in `LibTIFF` for CVE-2008-2327

These are just a few examples of unpatched code clones for serious vulnerabilities. Such a large vulnerability window is a critical security problem because an adversary can spot (potentially) vulnerable packages using known vulnerabilities. For example, an attacker may be able to automatically create an exploit given a patch [32].

In this section, we present ReDeBug [74, 76], a lightweight syntax-based code clone detection system that identifies unpatched code clones at scale. We have used ReDeBug to determine how widespread the problem of unpatched code clone truly is, specifically: 1) how much (potentially) vulnerable code an attacker can identify when a patch is released, 2) how responsive the new version of an OS is to known security vulnerabilities, and 3) how many persisting unpatched code clones are copied from the previous version of an OS over to the latest version.

Existing research has focused on methods for improving the number of code clones detected, e.g., [57, 78, 80, 101]. While it is important to make advancements in finding more code clones, this line of research potentially requires comparison among all code pairs and uses various matching heuristics which can suffer from a higher false detection rate. ReDeBug tackles a new point in the design space where we trade more expensive, yet thorough, pattern matching algorithms for scalability, support for many different languages, and zero false detections.

- **Scalability.** To give a sense of the scale necessary to find all unpatched code clones, observe that Debian Squeeze alone contains 16 GB of non-empty and non-comment code, spanning over 348 million lines. ReDeBug uses a syntax-based pattern matching approach that can be implemented using extremely efficient data structures which allow fast querying for code clones when given a patch. Using ReDeBug on a machine with a 3.40 GHz i7 CPU and SSD, we were able to scan the 2.1 billion lines of code in our entire data set against 1,634 buggy code patterns in under 3 hours. With the ability to rapidly search for unpatched code clones, ReDeBug can be used to improve the security of code bases in day-to-day development by promptly and automatically checking for copies of known vulnerabilities. It is desirable to have a scalable solution that can be applied to hardware used by the average developer or user, such as a basic desktop.

  It is desirable to have a scalable solution that is applicable on hardware available to an average developer or user, e.g., an average desktop.

- **Support for many different languages.** OS distributions include programs written in a variety of languages. For example, Debian Squeeze consists of 288 million

lines of C/C++, 24 million lines of JAVA, 14 million lines of Python, 12 million lines of Perl, 5 million lines of PHP, and so on. To handle such a large variety of languages, ReDeBug uses a simple, fast, and language-agnostic syntax pattern matching approach to find unpatched code clones. We realize that there are more advanced matching algorithms that are applicable when the code is correctly parsed, and that such algorithms will likely find even more unpatched code clones. For example, Deckard [78], CP-Miner [101], CCFinder [80], and DejaVu [57] first parse a program and use a variety of matching heuristics based on high-level code representations, such as CFGs and parse trees. The challenge, however, is in the building of robust parsers for each language, which has proven difficult even for professional software assurance companies [28]. While we encourage future developers to add parsing support to ReDeBug, for now ReDeBug opts for a simpler robust algorithm that works across a wide variety of languages.

- **Zero false detection rate.** There are two types of false reports that any clone detection algorithm can make. The first type is a syntactic *false detection*. This happens when an algorithm says an unpatched code clone is present when it is not. ReDeBug eliminates false detections by performing a slower but exact match after all potential matches have been rapidly identified. Our approach means that each reported unpatched code clone is likely to be actionable. In contrast, the advanced heuristic matching algorithms used to find more code clones can suffer from a higher false detection rate. For example, CP-Miner had a false detection rate of 73% [101], and DejaVu had a false detection rate of 37% [57]. It is important to report only true matches to developers; otherwise, they end up wasting resources in examining the false reports. The second type is a semantic *false positive*. This happens when an algorithm detects an unpatched code clone, but the clone is used in a non-vulnerable way such as when checks have been inserted in earlier locations. Though ReDeBug inevitably can have false positives just like any other syntax-based method, we argue that false positives still present latent security problems because the code can be used in a vulnerable way due to a change in the future.

We have used ReDeBug to check for unpatched code clones in Debian 6.0 Squeeze (348,754,939 LoC[1]), Debian 5.0 Lenny (257,796,235 LoC), Ubuntu 11.10 Oneiric (397,399,865 LoC), Ubuntu 10.10 Maverick (245,237,215 LoC), Linux Kernel (8,968,871 LoC), and all C/C++ projects at SourceForge (922,424,743 LoC). So far, ReDeBug has found 15,546

[1]We always count non-empty, non-comment lines.

unpatched code clones in the total 2,180,581,868 LoC by checking 376 Debian/Ubuntu security-related patches. The patches address a variety of issues ranging from buffer overflows to information disclosure vulnerabilities to denial of service vulnerabilities. Our measurements indicate that even though ReDeBug is simpler, it actually finds a comparable number of code clones to existing approaches (§6.4.7).

Previous work has shown that once a patch is released, an attacker can use the patch to reverse engineer the bug and automatically create an exploit in only a few minutes [32]. Our experiments indicate that one security implication of ReDeBug is the ability of an attacker to find potentially thousands of vulnerable applications among billions of lines of code once a patch is released. This could be done on a simple laptop in only a few minutes, assuming the attacker has already preprocessed the code.

Contributions.

- We analyze entire OS distributions to comprehend the current trends of unpatched code cloning. To the best of our knowledge, ReDeBug is the first tool to explore over 2.1 billion lines of entire OS distributions to understand unpatched code clone problems. We show that unpatched code clones are a recurring problem in modern distributions, and find 15,546 unpatched code clones from Debian Lenny/Squeeze, Ubuntu Maverick/Oneiric distributions, the Linux kernel, and SourceForge. So far, ReDeBug has confirmed 145 real bugs.

- We describe ReDeBug, which suggests a new design space for code clone detection in terms of scalability, speed, support for different languages, and false detection rate. ReDeBug uses syntax-based pattern matching, which allows it to (i) scale to entire OS distributions, (ii) support many different languages, and (iii) guarantee zero false detections. In particular, the design point makes ReDeBug realistic for use by typical developers in everyday environments in order to improve the security of their code by quickly querying known vulnerabilities.

- We build a website and provide ReDeBug as an open-source tool to help developers fight against unpatched code clones and improve the security of code bases in day-to-day development.

6.1 Fingerprinting for Containment Detection

Finding unpatched code clones is a problem of detecting containment of known vulnerable code. Searching for all unpatched code clones is tricky and involves numerous considerations. For example, how many lines of code need to be similar for a case to be reported? Is one copied line enough, or are we only interested in multiple line matches? Should whitespace matter? Should the order of statements matter, and if so, should we only consider some syntactic classes? Do we consider the syntactic text, tokens, or the parse of files? For example, in C the order of declarations likely does not matter, but the order of computation may. What if two segments are equivalent up to variable naming? What about semantic equivalence, e.g., one code sequence multiples by 2 while the other performs a logical left shift. Are these similar or different?

These questions all involve trade-offs between accuracy, efficiency, and how ease of implementation of a robust algorithm. For example, consider code that is the same up to variable names and variable declaration order. A straight string match of the files may find virtually no commonality. We could certainly address these problems by normalizing declaration order, and parse code to determine variable name equivalence (so-called α-equivalence) [33]. However, running such algorithms requires us to implement parsing engines (which can be fragile) and run additional algorithms that take time, thereby reducing overall throughput. If we are not careful, we may end up subtly analyzing a model of the original program that is not right, e.g., declaration order matters when looking for buggy code clones of incorrect shadow variable declarations.

Our choices with ReDeBug were motivated by the design space goals of: (1) focusing on unpatched code clones, (2) scaling to large and diverse code bases such as OS distributions, (3) minimizing false detection, (4) being modular when possible and offering users a choice of parameters, and (5) being language-agnostic as much as possible so that we can work with the wealth of languages found within an OS distribution code base.

6.1.1 Bloom Filters

ReDeBug utilizes Bloom filters to identify unpatched code clones containing any specific bugs. A Bloom filter [29] is a space efficient randomized data structure used for set membership tests.

Suppose there is a data set S. A Bloom filter represents set S as a vector of m bits initially all set to 0. To add an element $x \in S$ to the Bloom filter we first apply k independent hash functions of range $\{1..m\}$. For each hash $h(x) = i$, we set the i'th bit of the bit vector

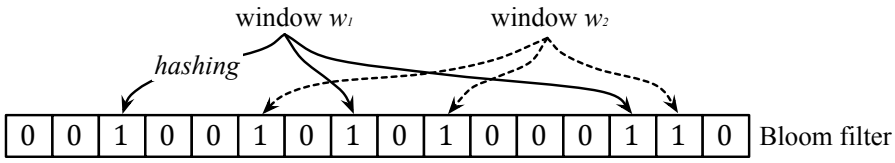


Figure 6.1: Bloom filter with $k = 3$

to 1. In a Bloom filter, to test if an element of $y \in S$, we again apply the k hash functions and check if all the bits are 1. If at least one of the hashed bits is 0, then we return $y \notin S$. If all bits are set to 1, then we return $y \in S$. Figure 6.1 shows a Bloom filter with $k = 3$.

Bloom filters have one-sided error for set membership tests. A false positive occurs when the set membership test returns $x \in S$ based on the bloom filter when x is not really in S. False positives occur because of collisions in hash functions. The false positive rate of the Bloom filter depends on the size of the bit array (m), the number of hash functions (k), and the number of elements in S (N). The probability of getting a false positive can be made negligible by an appropriate choice of parameters [31]. Bloom filters have no false negatives.

6.1.2 Containment Detection

The core of the ReDeBug system consists of the following steps:

1. ReDeBug normalizes each file. By default, ReDeBug removes typical language comments, all non-ASCII characters, and redundant whitespace (except new lines), and converts all characters to lowercase. We also ignore curly braces if the file is C, C++, Java, or Perl (as identified by extension or the UNIX `file` command).

   Normalization is modularized so that the exact normalization steps can easily be changed.

2. The normalized file is tokenized based on new lines and regex substrings. A token is a string ending in a new line.

3. ReDeBug slides a window of length n over the token stream. Each set of n tokens is considered a unit of code to compare.

4. Given two sets f_a and f_b of n-tokens, we compute the amount of code in common. When finding unpatched code clones, if f_a is the original buggy code snippet we

calculate

$$\text{CONTAINMENT}(f_a, f_b) = \frac{|f_a \cap f_b|}{|f_a|} \tag{6.1}$$

It is common to only consider cases where the containment is greater than or equal to some pre-determined threshold θ. In our implementation, we also perform obvious optimizations, such as only verifying $f_a \subseteq f_b$ instead of calculating an actual ratio for CONTAINMENT when $\theta = 1$.

5. ReDeBug performs an exact match test on the identified unpatched code clones to remove Bloom filter errors. When possible, ReDeBug also uses the compiler to identify when a code clone is dead code.

For example, suppose we have three files $A = t_1t_2t_3t_4$, $B = t_1t_3t_4t_2$, and $C = t_3t_4$ where each t_i is a token (note that tokens are written in the order in which they appear in the file). The tokenization is then $A = \{t_1, t_2, t_3, t_4\}$, $B = \{t_1, t_3, t_4, t_2\}$, and $C = \{t_3, t_4\}$. When $n = 2$, there are three 2-token strings in A and B: $f_A = \{(t_1, t_2), (t_2, t_3), (t_3, t_4)\}$ and $f_B = \{(t_1, t_3), (t_3, t_4), (t_4, t_2)\}$, and one 2-token string in C: $f_C = \{(t_3, t_4)\}$. The containment of (f_B, f_A) is $1/3$ since one out of three 2-token sets are shared, (t_3, t_4), even though the shared token sequence appears at different places in the file. As a result, ReDeBug works with reorderings, insertions, and deletions of up to n-tokens. In addition, the containment of (f_C, f_A) is 1 because f_A is a superset of f_C.

6.2 ReDeBug Architecture

At a high level, there are two approaches for finding unpatched code clones in OS distributions: (1) first find all code clones among the source code and then check to see if a patch applies to the copies, or (2) check for clones of the patched code only. Previous work has focused on techniques for the first approach. This makes sense when finding bugs on whole code bases is cheaper than on unique code snippets. ReDeBug takes the second approach because we only want to find clones of the original unpatched buggy code.

ReDeBug looks for unpatched code clones where patches are in UNIX unified `diff` format. Unified `diff`s are popular among open source kernel developers and OS distribution maintainers, and are well-integrated into popular revision control systems like Subversion [41].

```
// Original buggy code
char buf[8];
strcpy(buf, input);
```

```
// Possible patch 1
char buf[8];
- strcpy(buf, input);
+ strncpy(buf, input, 8);
```

```
// Possible patch 2
char buf[8];
+ if(strlen(input) < 8)
  strcpy(buf, input);
```

Figure 6.2: Buggy code example and two possible patches

A unified `diff` patch consists of a sequence of diff hunks. Each hunk contains the changed filename and a sequence of additions and deletions. Added source code lines are prefixed by a "+" symbol and deletions are prefixed by a "−" symbol. Line changes are represented by deleting the original line and adding back the changed lines.

The original buggy code includes all code deleted by the patch. However, simply looking for the lines that were changed (by being deleted) is insufficient: we must also consider the surrounding context of the patch.

Consider the buggy code and two possible patch scenarios, as shown in Figure 6.2. Patch 1 signifies that `strcpy` is buggy by deleting the line of code. The code is then replaced with the safe `strncpy` version. We can go looking for the deleted line of code and flag it as buggy everywhere we see it. However, patch 2 simply adds a check. Looking for the missing check is not straightforward since we cannot directly look for missing lines of code. Our approach is to look for copies of the surrounding context tokens, c, for each changed line and to report clones of the context.

6.2.1 ReDeBug Overview

The overall steps used by ReDeBug to detect buggy code clones, as shown in Figure 6.3, are:

- **Step 1: Pre-process the source.** A user obtains all source files used in their distribution. For Debian, this is done using the `apt` tool. ReDeBug then automatically:

  1. Performs normalization and tokenization as described in the fingerprinting method (§6.1.2).
  2. Moves a window of n-length over the token stream. For each window, the resulting n-tokens are hashed into a Bloom filter.
  3. Stores the Bloom filter for each source file in a raw data format. ReDeBug compresses Bloom filters before storing to disk to save space and to reduce the

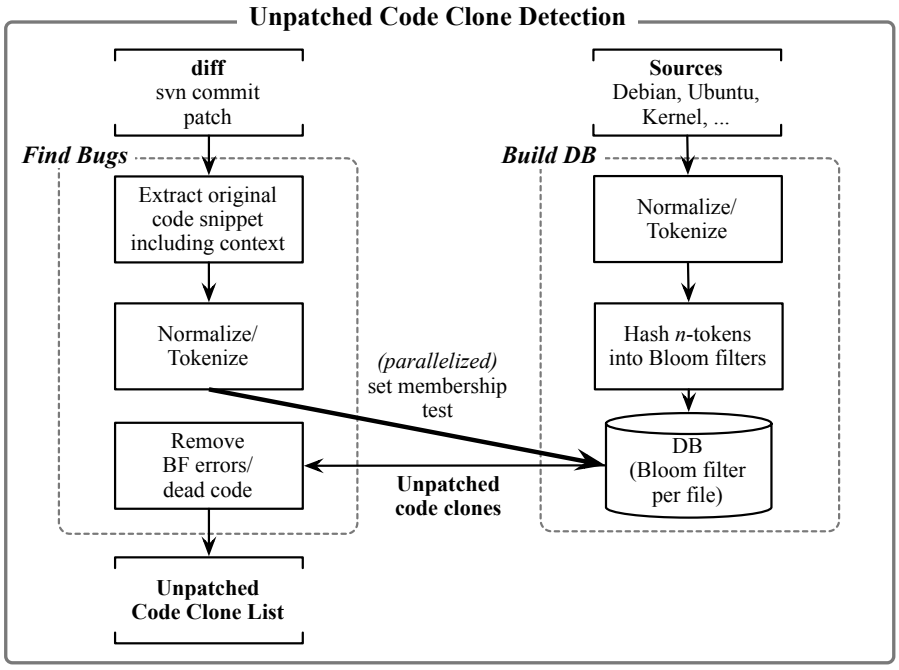


Figure 6.3: The ReDeBug workflow.

amount of disk access at query time. Since Bloom filters are typically sparse bit vectors, ReDeBug can significantly save disk space through compression.

While initially the above steps would be performed over the entire distribution, day-to-day use would only run the steps on modified files, e.g., as part of a revision control check-in. In our experiments and implementation, we also concatenate per-file Bloom filters for a project into a single bitmap before saving to disk. This is purely an optimization; loading the single large Bloom filter is much quicker than loading a bunch of small Bloom filters onto our machines.

- **Step 2: Check for unpatched code copies.** A user obtains a unified `diff` software patch. ReDeBug then automatically:

  1. Extracts the original code snippet and the c tokens of context information from the pre-patch source. The mechanics for the code snippet are simple: we extract lines prefixed by a "-" symbol in the patch (lines prefixed with a "+" symbol are added and thus not present in the original buggy code). If context information is given in the patch, we use that, or else we obtain it from the original source

files.

2. Normalizes and tokenizes the extracted original buggy code snippets. The normalization process is the same as described in Step 1. For C, C++, Java, and Python, we remove any partial comments in the c context lines since those languages support multi-line comments and c context lines may have only the head or tail part of multi-line comments.
3. Hashes the n-token window into a set of hashes f_p.
4. Performs a Bloom filter set membership test on each hashed n-token window. We report an unpatched code clone with file f if $\text{CONTAINMENT}(f_p, f) \geq \theta$.

- **Step 3: Post-process the reported clones.** Given reported unpatched code clones, ReDeBug automatically:

1. Performs an exact-matching test to remove Bloom filter errors.
2. Excludes dead code which is not included at build time. For C, specifically, we add `assert` statements to the buggy code region, and compile with `-g` option which allows us to check the presence of `assert` statements using `objdump -S`. For non-compiled languages this step is omitted.
3. Reports only *non*-dead code to the user.

Bloom filters have one-sided error for set membership tests. In our setting, the one-sided error means we may mistakenly say that an n-token is present in the set when it is not. The probability of this happening can be made arbitrarily low with appropriate parameter selection, e.g., it is 0.3% in our implementation.

In our evaluation, we only report an unpatched code clone if a file contains all context lines and all original n-tokens as described above, i.e., $\theta = 1$. This is a conservative configuration.

6.2.2 ReDeBug Parameters

ReDeBug is parameterized in two ways: the number of consecutive tokens to consider together, n, and the threshold, θ. n determines the sensitivity for statement reordering, e.g., if $n = 1$ then statement order does not matter at all, $n = 2$ looks at statement pairs, and so on. θ acts as a knob to indicate a significant amount of copying. When $\theta = 1$, two files must have exactly the same n-tokens (after normalization). When $\theta = 0$, any match is considered

significant. Values in between represent thresholds for the amount of similarity of interest. There is no "right" value for these parameters. In our experiments we show typical values that produce meaningful results. For example, $n = 4$ works well with existing patches.

6.2.3 Design Point Comparison

Our approach is in stark contrast to current research trends in code clone detection, such as Deckard [78], CP-Miner [101], DejaVu [57], and others [80, 130], which focus on minimizing missed code clones at the expense of other factors. These approaches also normalize the code, but then perform additional steps such as parsing the code into high-level representations like parse trees and control flow graphs. They then employ advanced fuzzy matching algorithms on the abstractions to find additional code clones that we may miss. On the other hand, these approaches may report more false code clones, which then require significant human effort to inspect. Furthermore, it is known to be very hard to implement good parsers [28].

Overall, the main difference is that by employing simpler techniques that are language agnostic, we can focus on efficient data structures and algorithms and ultimately scale to much larger code bases written in many different languages. Our techniques may miss some clones, but they minimize false clone detection rates. This is important for at least two reasons. First, by quickly checking all code in a distribution, we can make basic guarantees that at least syntactically similar unpatched code clones do not exist. Second, we can conservatively estimate the amount of code cloning in existing large code bases. The more advanced algorithms in the above work have not demonstrated they can make either claim.

6.3 Implementation

ReDeBug is implemented in about 1,000 lines of C code and 250 lines of Python. Normalization is modularized within the Python code. We use the QuickLZ library[2] to perform compression/decompression while setting `QLZ_COMPRESSION_LEVEL` to 3 for faster decompression speed.

[2] http://www.quicklz.com/

Distributions		Lines of Code	Date Collected
Early 2011 (Σ_1)	Debian Lenny	257,796,235	Jan 2011
	Ubuntu Maverick	245,237,215	Mar 2011
	Linux Kernel 2.6.37.4	8,968,871	Mar 2011
	SourceForge (C/C++)	922,424,743	Mar 2011
Late 2011 (Σ_2)	Debian Squeeze	348,754,939	Nov 2011
	Ubuntu Oneiric	397,399,865	Nov 2011
Total		2,180,581,868	-

Table 6.1: Source data set

6.4 Evaluation

6.4.1 Experimental Setup

System Environment. We performed all experiments to find unpatched code clones by both building and querying the database on a desktop machine running Linux 2.6.38 (3.4 GHz Intel Core i7 CPU, 8GB memory, 256 GB SSD drive). We utilized 8 threads to build a DB and to query bugs.

Data Set. We collected our source code data set twice: early in 2011 and late in 2011. We first collected our Early 2011 data set (Σ_1) in January/March 2011, which included all source packages for Debian 5.0 Lenny and Ubuntu 10.10 Maverick, as well as all public SourceForge C/C++ projects using version control systems such as Subversion, CVS and Git, and the Linux kernel v2.6.37.4. For the SourceForge data set we excluded identifiable non-active code branches, such as `branches` and `tags` directories. In November 2011, we prepared our Late 2011 data set (Σ_2), which included all source packages for Debian 6.0 Squeeze and Ubuntu 11.10 Oneiric. Table 6.1 shows the detailed breakup of our collected source code data set. The data set consists of a large number of projects written in a wealth of languages including C, C++, Java, Shell, Perl, Python, Ruby, and PHP.

In order to find security-critical bugs, we collected security-related patches from the Debian/Ubuntu security advisories that included the information about the corresponding packages and patches/`diffs`. We downloaded 1,634 `diffs` whose related CVE numbers were recognizable by the patch file names. As described in Table 6.2, pre-2011 patches (δ_1) were available at the time of collecting Σ_1, and 2011 patches (δ_2) were released between the download dates of Σ_1 and Σ_2.

In the original source packages for Debian and Ubuntu there are a number of existing

Data set	# files	# diffs	Date Released
Patches before 2011 (δ_1)	274	1,079	2001∼2010
Patches in 2011 (δ_2)	102	555	2011
Total	376	1,634	-

Table 6.2: Security-related patch data set

patches (e.g., `debian/patches/`) that can be applied during a build; we applied these patches as well. As a result, the patched packages were current up to security advisories on the download date. Since we downloaded the SourceForge packages via revision control systems, we assume that all patches were already applied.

We performed experiments to identify the number of duplicate buggy code segments that were still likely to be vulnerable. Then we verified the presence of all reported unpatched code clones, i.e., clones of the exact same buggy code, to confirm that the ReDeBug implementation was correct. We discuss this measurement in §6.4.5.

Default Parameters. The default context in a `diff` file is 3 lines of code. Unless otherwise noted, we set $n = 4$. $n = 4$ when the amount of context $c = 3$ guarantees that every reported duplicate had at least one changed line along with surrounding context. In all experiments for unpatched code clones we set $\theta = 1$, i.e., with the default parameters all n-tokens from the original buggy code segment needed to be found in an unpatched copy to report a bug. m is the size of a Bloom filter and N is the number of n-tokens to be hashed into a Bloom filter. ReDeBug used 256KB-sized Bloom filters where the m/N ratio was greater than 32. ReDeBug took advantage of 3 fast hash functions: djb2, sdbm, and jenkins[3]. A theoretical Bloom filter false positive rate for these parameter selections is 0.0717% [34].

6.4.2 Performance

We ran ReDeBug to create a database for each source code data set. Figure 6.4 shows the database build time. It took about 19 minutes each to build the databases for Ubuntu Maverick and Debian Lenny. Building the database for SourceForge took about 69 minutes. This is the end-to-end time including the time to read files, to normalize and tokenize source code, to create and store Bloom filters for a variety of languages, e.g., C/C++, Java, Shell, Perl, Python, Ruby, and PHP. The experimental results suggest that the time to build a

[3] http://www.cse.yorku.ca/~oz/hash.html

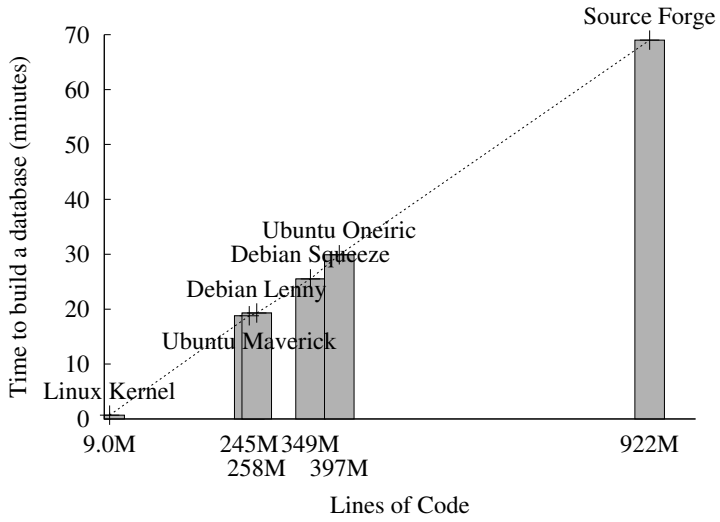


Figure 6.4: Time to build a database with various sizes of data sets

Distributions	DB Size	Projects #	Files #
Debian Lenny	6.0GB	10,699	1,155,594
Ubuntu Maverick	5.6GB	11,237	1,067,579
Linux Kernel 2.6.37.4	344MB	-	57,653
SourceForge (C/C++)	29GB	30,437	5,574,905
Debian Squeeze	8.2GB	14,977	1,586,325
Ubuntu Oneiric	9.8GB	18,240	1,892,911

Table 6.3: Size of created databases

database increases linearly as the size of the source code increases. Once ReDeBug has built the initial database, incremental update is quickly done by adding/changing only the relevant parts of the database.

The resulting database sizes and the number of projects and files in the databases are described in Table 6.3. As a reference point, Debian Lenny required 282 GB to store 1,155,594 files without compression, but only 6.0 GB with compression in ReDeBug. The large compression factor is due to the sparseness of the Bloom filters.

Figure 6.5 depicts the time to query 1,634 security-related patches (δ_1 and δ_2) to each database. As the size of a database (the number of files in a database) grew, the time to query bugs increased linearly. Though there was an overhead to recover compressed Bloom filters to perform the set membership test, the querying time was fast, e.g., 0.04 second per bug against about 1 million source files in the case of Debian Lenny.

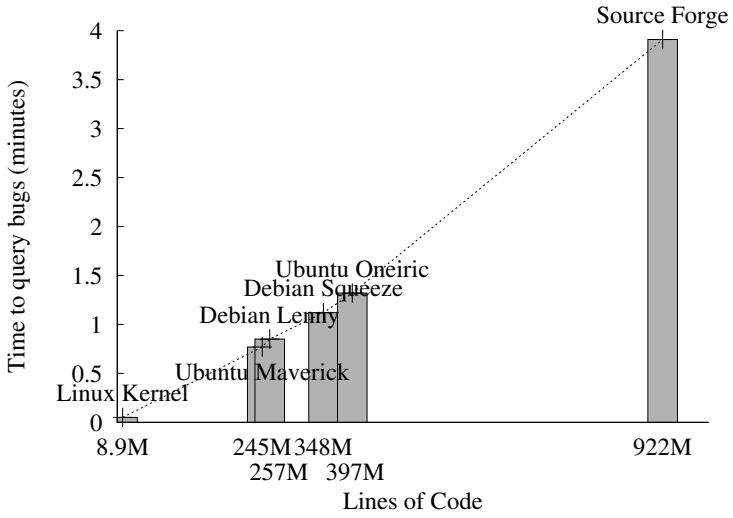


Figure 6.5: Time to query 1,634 bugs to various sizes of DBs

Figure 6.6 shows the time to search for different number of bugs against the whole database including both Σ_1 and Σ_2. The query time has a very gentle upward slope. The results suggest that querying even a large number of patches should only take a few minutes. For example, it took about 6 minutes 21 seconds to query 15 `diffs`, and this time increased only slightly to 7 minutes 46 seconds for 1,634 `diffs`.

Together these 3 graphs show that ReDeBug is highly scalable and can be applied to find unpatched code copies in day-to-day development. The time it takes us to perform all operations increases linearly with the size of the database, and grows very slowly with the number of `diffs`.

We compared ReDeBug with GNU `grep` which has a reputation of efficiently finding patterns. As an example, consider the task of finding all unpatched code clones that match all known buggy code in Debian Squeeze. Debian released 236 security advisories in 2011, which means there were at least 236 security-related patches in that year. Debian Squeeze contains 16 GB of non-empty and non-comment code, spanning over 348 million lines. By using GNU `grep` it took 54 minutes using 455 MB of memory to search for 236 buggy code patterns[4]. In order to match 400 buggy code patterns in Debian Squeeze, `grep` consumed over 1 GB of memory for a large deterministic finite automaton (DFA) and took 7 hours. `Grep`'s requirement of a large memory for a DFA causes poor cache use and performance. Given the limitations of `grep`, it is desirable to have more scalable methods for examining

[4]In order to find multi-line buggy code patterns, we ran `grep -Grz -f "patterns" "squeeze"`.

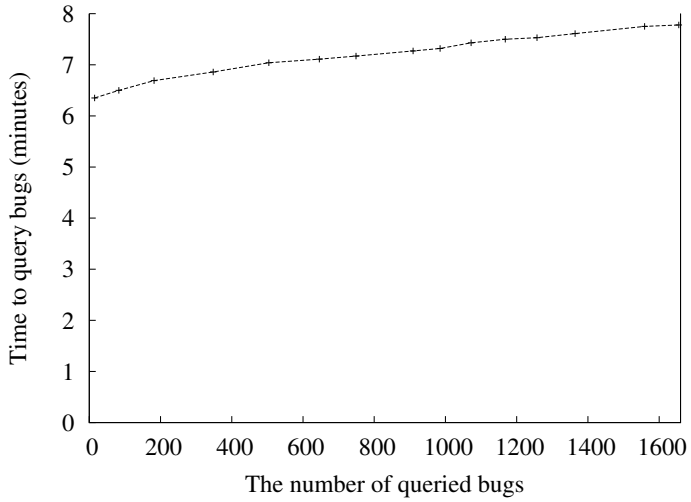


Figure 6.6: Time to check for various numbers of bugs against the entire DB

several years of known vulnerabilities. Using ReDeBug, it took 9 minutes to match Debian Squeeze against 236 buggy code patterns with 13 MB of memory.

6.4.3 Security-Related Bugs

In total, ReDeBug found 15,546 unpatched code clones in the two data sets Σ_1 and Σ_2. Figure 6.7 shows the detailed breakup of unpatched code clones identified in Σ_1 and Σ_2 when querying for δ_1 and δ_2. We include some examples of the identified unpatched code clones on our website `http://security.ece.cmu.edu/redebug/` (§6.5). We considered three scenarios to understand the current situation of unpatched code clones.

- $\{\delta_1 \ \& \ \delta_2\} \rightarrow \Sigma_1$: The unpatched code clones found in Σ_1 using δ_1 and δ_2 approximate how many (potentially) vulnerable packages an adversary may be able to spot when a patch becomes available. 10,248 unpatched code clones were detected in the SourceForge data set. Debian Lenny and Ubuntu Maverick, which were still supported on the download date, also had 1,482 and 1,058 unpatched code clones respectively. When security-related bugs are fixed in the original packages, the resulting vulnerabilities must be detected before an adversary identifies them.

- $\{\delta_1 \ \& \ \delta_2\} \rightarrow \Sigma_2$: The unpatched code clones identified in Σ_2 using δ_1 and δ_2 roughly indicate how new versions of an OS respond to previously known security vulnerabilities. Debian Squeeze and Ubuntu Oneiric included 1,532 and 1,223 such unpatched

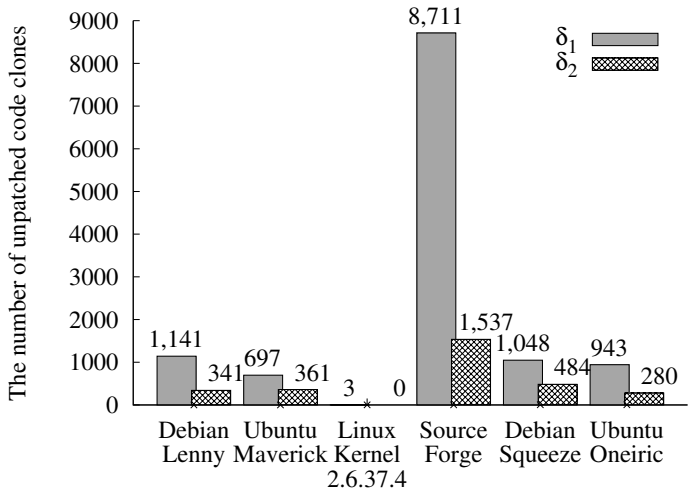


Figure 6.7: The number of unpatched code clones in Σ_1 and Σ_2

code clones respectively, which indicates that unpatched code clones are recurring in modern OS distributions. We reported the 1,532 unpatched code clones identified in Debian Squeeze packages to the Debian security team and package developers. So far, 145 *real bugs* have been confirmed by developers either by private emails or by issuing a patch. This showcases the real-world impact of ReDeBug.

- $\delta_1 \rightarrow \Sigma_1$ vs. $\delta_1 \rightarrow \Sigma_2$: The unpatched code clones found in both Σ_1 and Σ_2 using δ_1 demonstrate how many unpatched code clones persisted from the previous version of an OS to the latest version of an OS. In our evaluation, we compared the 1,838 unpatched code clones from δ_1 in Σ_1 and the 1,991 unpatched code clones from δ_1 in Σ_2. Among these 3,829 clones, 1,379 persisted. Figure 6.8 shows the number of unpatched code clones identified from patches released in different years. Note that 21 of the unpatched code clones are security vulnerabilities that were patched over a decade ago (in 2001). This indicates that unpatched code clones are long-lived in modern OS distributions.

6.4.4 The Identified Unpatched Code Clones

Figure 6.9 depicts the distribution of how often we found clones for patches. The maximum was 386 unpatched code clones of the patch shown in Listing 6.4, with most patches having less than 50 respective unpatched code clones. This result demonstrates that there

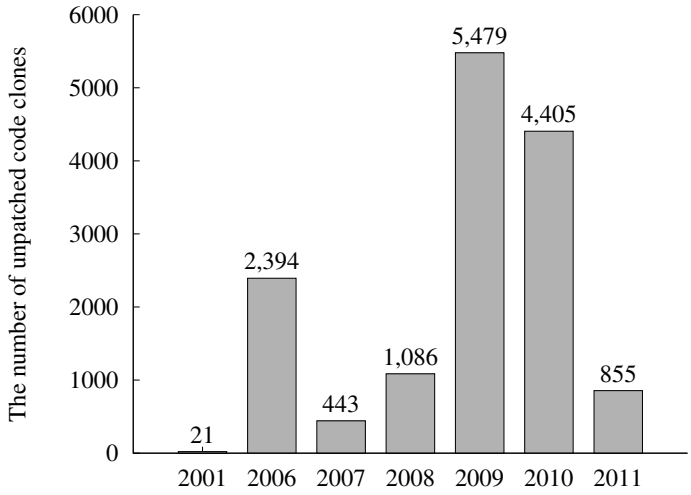


Figure 6.8: Unpatched code clones from patches in different years

	2001	2006	2007	2008	2009	2010	2011
Lenny	2	109	76	88	565	301	341
Maverick	0	161	35	62	248	191	361
Kernel	0	0	0	0	1	2	0
SrcForge	19	1162	227	746	3845	2712	1537
Squeeze	0	264	46	77	379	282	484
Oneiric	0	232	45	73	341	252	280
Total	21	1928	429	1046	5379	3740	3003

Table 6.4: Unpatched code clones in each distribution from different years' patches

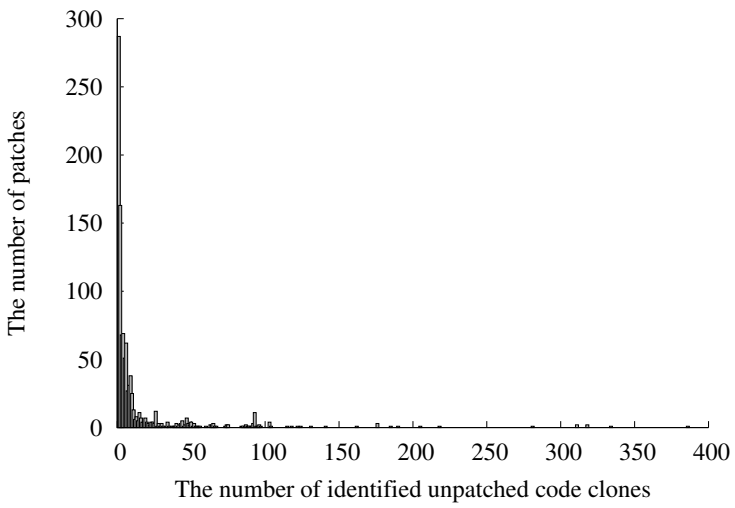


Figure 6.9: The identified unpatched code clones per patch

are potentially many vulnerable code clones for each new patch, highlighting the need to implement unpatched code clone detection as part of the developer lifecycle.

```
                              const char *end,
                              POSITION *pos)
    {
-       while (ptr != end) {
+       while (ptr < end) {
            switch (BYTE_TYPE(enc, ptr)) {
    #define LEAD_CASE(n) \
        case BT_LEAD ## n: \
```

Listing 6.4: Patch in Expat for CVE-2009-3720

Table 6.5 shows the number of identified unpatched code clones with various sizes of n. When n increases from 4 to 7, ReDeBug hashes every 7 consecutive tokens and each match represents an exact matching of 7 sequential tokens. Overall, this represents a larger number of exactly matched tokens, which yields a more conservative metric for "real" bugs (see §6.4.8 for discussion).

As we increased n, the total number of unpatched code clones that ReDeBug found decreased. Note that as n increases, the total number of `diffs` we queried decreased. The reason is that some `diffs` had fewer than n tokens in total. Overall, in the most conservative experiment, ReDeBug identified 3,374 unique unpatched code copies that likely constitute real bugs.

The size of n		$n = 4$	$n = 5$	$n = 7$
# of queried `diffs`		1,634	1,248	503
Unpatched code clones	Debian Lenny	1,482	1,013	309
	Ubuntu Maverick	1,058	736	251
	Linux Kernel 2.6.37.4	3	2	0
	SourceForge (C/C++)	10,248	6,211	2,130
	Debian Squeeze	1,532	1,061	391
	Ubuntu Oneiric	1,223	828	293
Total		15,546	9,851	3,374

Table 6.5: Unpatched code clones with various n for δ_1 and δ_2

6.4.5 Code Clone Detection Errors

A key question is, "What is the false detection rate of ReDeBug?" There are several ways to answer this. One popular metric is the accuracy of the matching process. In ReDeBug, this is the Bloom filter test. Bloom filter tests have no false negatives, but may have false positives. We performed an exact match test on the 15,599 unpatched code clones initially reported, 15,546 of which were confirmed. Thus, overall we had a 0.3% false positive rate in the Bloom filters. Our post-processing system removes this source of errors from the final output.

In some cases, unpatched code clones may be found in dead code, e.g., vulnerable code that is present but not included at build time, or vulnerable code that is included but never gets executed due to logical conditions. The first situation usually happens when external library code is embedded in a source package, but the package is written to prefer the available system library to the embedded library. For example, Listing 6.5 shows the vulnerability found in dead code, which is not included at build time. This vulnerability can lead to an integer overflow that allows denial of service [120].

```
--- bzip2-1.0.5.orig/decompress.c
+++ bzip2-1.0.5/decompress.c
@@ -381,6 +381,13 @@
  es = -1;
  N = 1;
  do {
+   /* Check that N doesn't get too big, so that es doesn't
+      go negative.  The maximum value that can be
+      RUNA/RUNB encoded is equal to the block size (post
+      the initial RLE), viz, 900k, so bounding N at 2
+      million should guard against overflow without
+      rejecting any legitimate inputs. */
+   if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
    if (nextSym == BZ_RUNA) es = es + (0+1) * N; else
```

```
    if (nextSym == BZ_RUNB) es = es + (1+1) * N;
    N = N * 2;
```

Listing 6.5: CVE-2010-0405

We matched the above code to `libcompress-bzip2-perl`. However, the package maintainers stated that the matched code was not an actual vulnerability since it was dead code. Dead code, however, may still be a latent vulnerability in that the accompanied vulnerable library code can be used depending on the availability of the system library during compilation on the user's machine. We discuss the second situation in §6.4.8.

For C, specifically, we compile code with an assert statement inserted into the identified buggy code region and look for its corresponding assembly in the binary file to weed out such cases. We measured the number of code clones in *non*-dead code for the 1,354 reported unpatched code clones from 149 Debian Squeeze packages. We confirmed that 831 out of 1,354 (61%) unpatched code clones were *non*-dead code and were likely to represent real vulnerabilities. Dead code may still present a problem that should be fixed because the code may be used in a vulnerable way due to a change in the future.

Note that in the overall system, these errors are ultimately removed, and would never be shown or affect the end user.

6.4.6 Examples of Security-Related Bugs

In order to evaluate the practical impact of ReDeBug, we reported 1,532 unpatched code clones identified in Debian Squeeze packages to the Debian security team and developers. So far, developers have confirmed 145 *real bugs* either via email or by issuing a patch. In this section, we show several examples of the real bugs we found.

Some of the bugs were found in `Qemu`, a processor emulator that can be used as a hosted virtual machine monitor. Various bugs, such as the one in Listing 6.6, which allows root access on the host machine [117], have been fixed over the past few years. Two such bugs include CVE-2008-0928 and CVE-2010-2784.

```
    int len, i, shift, ret;
    QCowHeader header;

-   ret = bdrv_file_open(&s->hd, filename, flags);
+   ret = bdrv_file_open(&s->hd, filename, flags | BDRV_O_AUTOGROW);
    if (ret < 0)
        return ret;
    if (bdrv_pread(s->hd, 0, &header, sizeof(header)) != sizeof(header))
```

Listing 6.6: CVE-2008-0928

The patches for these bugs were not applied to the derivative package `xen-qemu`, the Xen version of `Qemu`. When contacted, Debian and upstream developers confirmed the presence of real bugs and indicated that fixing these bugs was necessary.

The patch in Listing 6.7 was issued to fix a vulnerability in `rsyslog`, a Linux and Unix system logger. This vulnerability involved sending a specially crafted log message that led to denial of service [124].

```
--- rsyslog-4.6.4.orig/tools/syslogd.c
+++ rsyslog-4.6.4/tools/syslogd.c
@@ -1291,7 +1291,7 @@
   * outputs so that only 32 characters max are used by default.
   */
  i = 0;
- while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' && i < ↩
    CONF_TAG_MAXSIZE) {
+ while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' && i < ↩
    CONF_TAG_MAXSIZE - 2) {
    bufParseTAG[i++] = *p2parse++;
    --lenMsg;
  }
```

Listing 6.7: CVE-2011-3200

The patch above was not applied to the Debian package `rsyslog-gssapi`, a version of `rsyslog` with plugins that allowed `rsyslog` to write and receive GSSAPI encrypted logging messages. When contacted, the package maintainers decided to fix the vulnerability by issuing an update.

The patch below was issued to fix a heap-based buffer overflow vulnerability in the `Paint Shop Pro` plugin in `GIMP` 2.6.11 [123].

```
-       if (code >= max_code)
+       if (code == max_code)
          {
-           *sp++ = firstcode;
+           if (sp < &(stack[STACK_SIZE]))
+             *sp++ = firstcode;
            code = oldcode;
          }

-       while (code >= clear_code)
+       while (code >= clear_code && sp < &(stack[STACK_SIZE]))
          {
            *sp++ = table[1][code];
            if (code == table[0][code])
```

Listing 6.8: CVE-2011-1782

When contacted, the developers of Deutex, a Debian package used to manipulate files for various games, indicated that this was likely a real vulnerability.

The following patch was issued to fix an integer overflow in PHP before 5.3.6 which could lead to a denial of service and possibly an information leak [122]. This patch was not employed to the Debian PHP package. When contacted, the package maintainer issued a patch to fix the bug.

```
- if (start + count > shmop->size || count < 0) {
+ if (count < 0 || start > (INT_MAX - count) || start + count > shmop->↩
    size) {
    php_error_docref(NULL TSRMLS_CC, E_WARNING, "count is out of range"↩
        );
    RETURN_FALSE;
```

Listing 6.9: CVE-2011-1092

Listing 6.10 shows a recent patch for CVE-2011-3145, which was successfully patched in an Ubuntu Oneiric package, but not in a Debian Squeeze package. This patch fixed an incorrect /etc/mtab ownership in the ecryptfs-utils package, which might cause arbitrary location unmount [157].

```
      if (setreuid(uid, uid) < 0) {
        perror("setreuid");
      }
+     if (setregid(gid, gid) < 0) {
+       perror("setregid");
+     }
      goto fail;
    }
  } else {
```

Listing 6.10: CVE-2011-3145

After we contacted the developers, the same patch that was applied to the Ubuntu Oneiric package was issued for the Debian Squeeze package to fix the vulnerability.

Listing 6.11 shows a security patch applied to the Ubuntu Oneiric apache2 package to fix a vulnerability in which remote attackers could send requests to intranet servers with a well-crafted URI [125]. The same patch was applied to the Debian Squeeze package to fix the bug after we reported it.

```
 ap_parse_uri(r, uri);

+/* RFC 2616:
+ *   Request-URI    = "*" | absoluteURI | abs_path | authority
+ *
+ * authority is a special case for CONNECT.  If the request is not
+ * using CONNECT, and the parsed URI does not have scheme, and
+ * it does not begin with '/', and it is not '*', then, fail
```

```
+ * and give a 400 response. */
+if (r->method_number != M_CONNECT
+ && !r->parsed_uri.scheme
+ && uri[0] != '/'
+ && !(uri[0] == '*' && uri[1] == '\0')) {
+ ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
+     "invalid request-URI %s", uri);
+ r->args = NULL;
+ r->hostname = NULL;
+ r->status = HTTP_BAD_REQUEST;
+ r->uri = apr_pstrdup(r->pool, uri);
+}
+
 if (ll[0]) {
  r->assbackwards = 0;
  pro = ll;
```

Listing 6.11: CVE-2011-3368

We highlighted the need to handle many languages in OS distributions. Here, we show a non-C example in Ruby. Listing 6.12 shows a security patch for the `puppet` package to fix a vulnerability in which an attacker could impersonate a master by exploiting a non-default `certdnsnames` option when generating certificates [126]. After we reported this bug, the package maintainer fixed the vulnerability by issuing a security patch.

```
 # Sign a given certificate request.
-def sign(hostname, cert_type = :server, self_signing_csr = nil)
+def sign(hostname, allow_dns_alt_names = false, self_signing_csr = nil↩
    )
   # This is a self-signed certificate
   if self_signing_csr
+    # # This is a self-signed certificate, which is for the CA.  Since↩
     this
+    # # forces the certificate to be self-signed, anyone who manages ↩
    to trick
+    # # the system into going through this path gets a certificate ↩
    they could
+    # # generate anyway.  There should be no security risk from that.
     csr = self_signing_csr
+    cert_type = :ca
     issuer = csr.content
   else
+    allow_dns_alt_names = true if hostname == Puppet[:certname].↩
    downcase
     unless csr = Puppet::SSL::CertificateRequest.find(hostname)
       raise ArgumentError, "Could not find certificate request for #{↩
           hostname}"
     end
+
+    cert_type = :server
     issuer = host.certificate.content
```

```
+
+     # Make sure that the CSR conforms to our internal signing policies↩
    .
+     # This will raise if the CSR doesn't conform, but just in case...
+     check_internal_signing_policies(hostname, csr, allow_dns_alt_names↩
    ) or
+       raise CertificateSigningError.new(hostname), "CSR had an unknown↩
     failure checking internal signing policies, will not sign!"
    end

    cert = Puppet::SSL::Certificate.new(hostname)
-   cert.content = Puppet::SSL::CertificateFactory.new(cert_type, csr.↩
     content, issuer, next_serial).result
+   cert.content = Puppet::SSL::CertificateFactory.
+     build(cert_type, csr, issuer, next_serial)
    cert.content.sign(host.key.content, OpenSSL::Digest::SHA1.new)
```

Listing 6.12: CVE-2011-3872

6.4.7 Comparison to Prior Work

Existing tools such as Deckard [78], CP-Miner [101], CCFinder [80], and DejaVu [57] parse programs for high-level code representations and perform "fuzzy" matching to find clones. However, building robust parsers is not a straightforward task [28]. We need language-independent techniques that can be easily adopted to detect unpatched code clones in many different languages. ReDeBug's use of syntax-based pattern matching allows it to be easily adopted to detect unpatched code clones in many different languages. Roughly speaking, ReDeBug performs simple normalization where whitespaces are removed and all characters are converted into lowercase equivalent. Such simple normalization can increase the possibility of finding unpatched code clones in many different languages. Moreover, it is less error-prone than sophisticated language parsers.

ReDeBug improves scalability with a decreased false detection rate, but may find fewer code clones than previous code clone detection work. In order to measure the number of unpatched code clones that ReDeBug missed, we compared the number of code clones detected by ReDeBug to the number of code clones reported by Deckard [78]. We chose Deckard because it claims better code clone detection performance than CP-Miner [101] and CloneDR [20].

Theoretically, the code clones reported by Deckard should be the superset of the code clones found by ReDeBug. In practice, however, Deckard missed more code clones than ReDeBug. We used Deckard v1.2[5] for our experiments, and set parameters as follows:

[5] https://github.com/skyhover/Deckard

Clone Detection	Real Clones	False Detection	Missed
ReDeBug	180	0	15
Deckard	96	183	99

Table 6.6: Code clone detection performance

`minT` (minimum number of tokens required for clones) = 30, `stride` (size of the sliding window) = 2 for their conservative results, and `Similarity` = 1 to minimize their false detection. This was similar to the setup in the Deckard paper.

Deckard did not scale to the entire Debian Lenny distribution (257,796,235 LoC) in our test setup due to its pairwise similarity calculation. During pairwise comparisons, Deckard consumed more than 20 GB of memory in less than 2 minutes, after which we killed the process. We ran Deckard on each package at a time instead of on the entire OS with 28 randomly selected C code files which contained security bugs. We only reported code clones that matched the buggy code regions. Deckard took more than 12 hours to complete the code clone detection in Debian Lenny, utilizing 8 threads to process 8 packages at the same time. While Deckard processed only the source code written in C (Deckard can only process one language at a time), ReDeBug processed a wealth of languages (e.g., C/C++, Java, Shell, Python, Perl, Ruby, and PHP) in 6 hours.

Table 6.6 shows the code clone detection results of Deckard and ReDeBug. As expected, ReDeBug had no false detections, and surprisingly, missed one sixth as many code clones compared to Deckard. The code clones that ReDeBug missed were due to the use of different variable names or types.

Deckard faired worse than ReDeBug despite using a more sophisticated strategy. When we investigated the causes, we found that 38 of the 99 cases were due to parse failures in Deckard, and the remaining 61 cases were due to the algorithm for detecting code clones. This result lends support to the idea that parsing code is difficult and can be a limiting factor in practice, and that ReDeBug's relatively simple approach can be valuable in such circumstances.

6.4.8 Discussion

Unpatched code clones that are not vulnerable. Since ReDeBug gets rid of Bloom filter errors and dead code, a metric for false positives is the number of unpatched code clones that were not vulnerable for some other reason. We have identified two other causes for this type of false positive. First, normalization may be too aggressive in some circumstances

```
// case 1
for(i=0;i<maxlen;i++){
  a[i]=0;
```

```
// case 2
for(i=0;i<maxlen;i++)
{
  a[i]=0;
```

Figure 6.10: Two syntactically equivalent cases

and thus the identified code clone is not really a code clone. Second, we may find real unpatched code clones, but other code modifications may prevent the unpatched code from being used in an exploitable context.

Normalization reduces the false negative rate. Figure 6.10 shows an example. Without normalization, these two code samples would not match; In the second case, the curly brace would be considered a separate token, while in the first case it would be the part of the token with the `for` loop statement. However, normalization may also increase the false positive rate. For example, imagine two equivalent code sequences, with one performed on an unsigned integer "A" and the other on a signed integer "a". If the bug relates to signedness, only the latter code is vulnerable. However, normalization converts all variables to lowercase, thus we would mistakenly report both code sequences as being buggy.

Listing 6.13 shows an example of an unpatched code clone that is present but not vulnerable. The patch fixes an integer signedness bug in various BSD kernels. NetBSD contained the same vulnerable code, but fixed the problem by changing the type of `crom_buf->len` from signed integer to unsigned integer instead of using the shown patch.

```
- if (crom_buf->len < len)
+ if (crom_buf->len < len && crom_buf->len > 0)
```

Listing 6.13: CVE-2006-6013

ReDeBug may have false positives when unpatched code clone is present but not vulnerable. For example, an unpatched code clone was detected in `ircd-ratbox` package from the patch for CVE-2009-4016 shown in Figure 6.11a. The package maintainer informed us that the integer underflow vulnerability was fixed in a different location, as shown in Figure 6.11b, which shows two new checks to guard against the vulnerable code, i.e., inserting separate checks `if(len <= 1) break;` ahead of the vulnerable code. As a result, this unpatched code clone is used in a way that makes it unexploitable. ReDeBug and all other syntax-based approaches share the same problem.

```
        else
            *d++ = *src;
-       ++src;
-       --len;
+       if (len > 0) {
+           ++src, --len;
+       }
    }
    *d = '\0';
    return dest;
```

(a) Patch for CVE-2009-4016

```
-   while (*src && (len > 0)) {
+   while (*src && (len > 1)) {
        if(*src & 0x80) {
            *d++ = '.';
            --len;
+           if(len <= 1)
+               break;
        ...
        else
            *d++ = *src;
        ++src;
        --len;
    }
    *d = '\0';
    return dest;
```

(b) Another patch for CVE-2009-4016

Figure 6.11: Different fix for CVE-2009-4016

6.5 ReDeBug to Enhance Code Security

ReDeBug is available for download as an open-source tool on our website http://security.ece.cmu.edu/redebug/ that can help developers fight against unpatched code clones.

ReDeBug is rewritten in Python to make the tool (i) easy to use without the need to compile first, (ii) useful on multiple platforms, and (iii) simple to extend with language-specific optimizations. The website also offers an online unpatched code clone detection service where developers can submit their code to test whether it contains any known vulnerabilities stored in our database. If a match is found, a report is presented showing both the original buggy code and unpatched code clones found in the submitted code.

In this section, we concentrate on how to use ReDeBug to find unpatched code clones in practice. As shown in Listing 6.14, ReDeBug takes two positional arguments: ⟨`patch path`⟩ and ⟨`source path`⟩. The first refers to the top-level patch directory from which we extract original buggy code snippets; the second points to the top-level directory of the source tree to be checked. As optional arguments, `-n` defines how many lines of code are to be considered as a unit of code to compare, `-c` sets how many surrounding lines of code are to be reported as context, and `-v` enables verbose output.

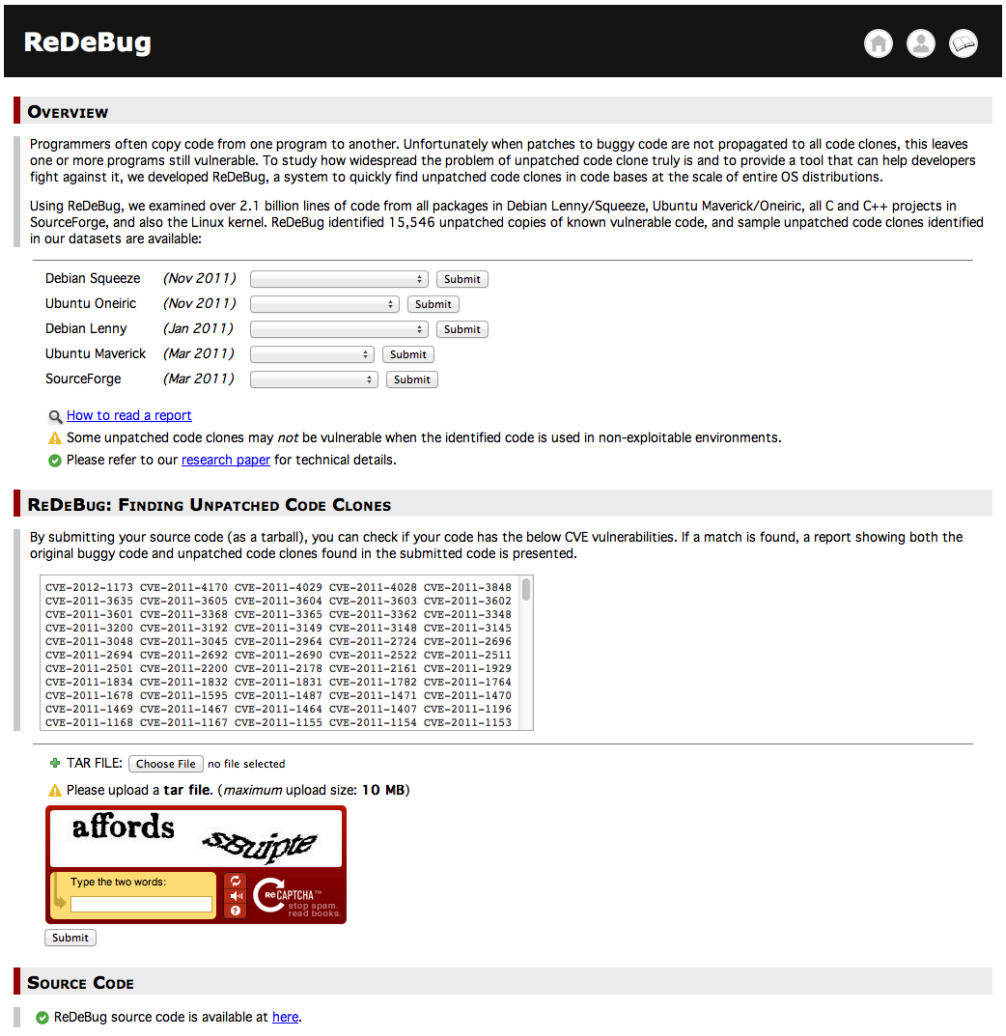


Figure 6.12: The ReDeBug website

```
$ redebug.py -h
usage: redebug.py [-h] [-n NUM] [-c NUM] [-v] patch_path ↩
    source_path

positional arguments:
  patch_path            path to patch files (in unified diff ↩
      format)
  source_path           path to source files

optional arguments:
  -h, --help            show this help message and exit
  -n NUM, --ngram NUM   use n-gram of NUM lines (default: 4)
  -c NUM, --context NUM print NUM lines of context (default: 10)
  -v, --verbose         enable verbose mode (default: False)
```

Listing 6.14: Help message of ReDeBug

ReDeBug consists of three major components: (i) `PatchLoader`, which extracts original buggy code snippets from patch files, (ii) `SourceLoader`, which matches source files against known buggy code, and (iii) `Reporter`, which generates a report after performing an exact-matching test. We explain each component of ReDeBug below with an example of identifying the unpatched code clone for the CVE-2009-3379 vulnerability in the Debian mplayer package.

PatchLoader: ReDeBug takes patch files in the UNIX unified `diff` format, which is popular among open source developers. Listing 6.1 shows a patch for the CVE-2009-3379 vulnerability in libvorbis in the unified `diff` format. A unified `diff` patch consists of a sequence of `diff` hunks where each hunk includes the filename of a modified file, deleted source code lines that are prefixed by a "-", and inserted source code lines that are prefixed by a "+". Modifications are represented as deletions of old source code lines followed by insertions of new source code lines.

1. Consider a set of patches P_i. ReDeBug extracts original code snippets P_i' from P_i by excluding the lines prefixed by a "+" symbol. This is because the inserted lines are not present in the original buggy code. The surrounding context lines are included to conservatively identify unpatched code clones. ReDeBug requires only the patches, not the pre-patch source code. Since we do not have to keep the original source code, ReDeBug is able to save significant space.

2. ReDeBug normalizes the extracted original buggy code P_i' to $\bar{P}_i$ by removing all whitespaces except new lines and converting all characters into lower-case. We keep

new lines since patches in the unified `diff` format operate at the line level. ReDeBug also identifies file types using the `libmagic` library, and performs language-specific normalization to increase the probability of identifying unpatched code clones. For example, for C, C++, and JAVA, we remove single-line comments (`//`), multi-line comments (`/* */`), and curly braces ({}). The code in Listing 6.15 shows the normalized buggy code extracted from the code in Listing 6.1. Regular expressions for such language-specific optimization are defined in `common.py`, which can be easily extended to add more optimizations and support other languages.

```
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
intcascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 6.15: Normalized buggy code

3. ReDeBug slides a window of n-lines over the normalized code $\bar{P}_i$. For example, we have 5 windows from the code in Listing 6.15 when $n = 4$: lines 1–4, 2–5, 3–6, 4–7, and 5–8. For each window w, we apply a list of hash functions H to build a list of hash values $h_i = \{h(w) | w \in \bar{P}_i, h \in H\}$. At present, ReDeBug utilizes 3 hash functions: FNV-1a hash[6], djb2 hash, and sdbm hash[7] (refer to `common.py`). The default context in a `diff` file is 3 lines of code. Therefore, we can guarantee that each window has at least 1 changed line by setting $n \geq 4$ (the default n is 4).

SourceLoader: ReDeBug builds a Bloom filter [29] for each source file to check for the presence of known vulnerabilities. For example, ReDeBug checks for the CVE-2009-3379 vulnerability in the code in Listing 6.16 as follows:

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);

for(j=0;j<info->partitions;j++){
    int cascade=oggpack_read(opb,3);
```

[6] http://isthe.com/chongo/tech/comp/fnv/
[7] http://www.cse.yorku.ca/$\sim$oz/hash.html

```
    if(oggpack_read(opb,1))
        cascade|=(oggpack_read(opb,5)<<3);
    info->secondstages[j]=cascade;

    acc+=icount(cascade);
}
```

Listing 6.16: Source code snippet from mplayer package

```
info->begin=oggpack_read(opb,24);
info->end=oggpack_read(opb,24);
info->grouping=oggpack_read(opb,24)+1;
info->partitions=oggpack_read(opb,6)+1;
info->groupbook=oggpack_read(opb,8);
for(j=0;j<info->partitions;j++)
intcascade=oggpack_read(opb,3);
if(oggpack_read(opb,1))
cascade|=(oggpack_read(opb,5)<<3);
info->secondstages[j]=cascade;
acc+=icount(cascade);
```

Listing 6.17: Normalized source code snippet

1. ReDeBug normalizes source file F_j to $\bar{F}_j$ in a similar way by removing all whitespaces except new lines and converting all characters into lowercase. Then, language-specific optimizations, such as comment removal, are applied according to the identified file type. For example, the code in Listing 6.16 is normalized into the code in Listing 6.17.

2. ReDeBug slides a window of n-lines over the normalized source code $\bar{F}_j$. We hash each window w using the same list of hash functions H. Specifically, for each $h \in H$, we set the $h(w)$-th bit of the Bloom filter BF_j to 1. Each source file is now represented by its corresponding Bloom filter.

3. ReDeBug tests whether a normalized source file $\bar{F}_j$ includes normalized buggy code $\bar{P}_i$ by checking if every bit in the locations specified by h_i is set to 1 in BF_j. For example, for all the hash values h_i generated from the code in Listing 6.15, we check if the corresponding bits are set to 1. If at least one of the bits is 0, then the corresponding window of $\bar{P}_i$ is not present in $\bar{F}_j$. ReDeBug only records the pair $(\bar{P}_i, \bar{F}_j)$ as a potential match if $\bar{F}_j$ contains the entire $\bar{P}_i$.

Reporter: For every pair $(\bar{P}_i, \bar{F}_j)$ recorded, ReDeBug verifies if $\bar{P}_i$ really occurs in $\bar{F}_j$. A Bloom filter may cause false detection due to hash collisions. This is why ReDeBug performs exact matching to eliminate any possible false detection due to the use of Bloom

filters. For example, the code in Listing 6.17 does indeed contain the buggy code in Listing 6.15. Finally, ReDeBug reports that the Debian `mplayer` package contains an unpatched code clone of CVE-2009-3379. The report also presents a pair of the patch in Listing 6.1 and the matched source code in Listing 6.16, which helps developers to easily inspect the identified unpatched code clone.

While we encourage future developers to add parsing support to ReDeBug, for now ReDeBug opts for a simpler robust algorithm that works across a wide variety of languages.

6.6 Summary

In this chapter, we presented ReDeBug, an architecture designed for unpatched code clone detection. ReDeBug was designed for scalability to entire OS distributions, the ability to handle real code, and the minimalization of false detection rates. ReDeBug found 15,546 unpatched code clones, which likely represent real vulnerabilities, by analyzing 2.1 billion lines of code on a commodity desktop. We demonstrated the practical impact of ReDeBug by confirming 145 real bugs in the latest versions of Debian Squeeze packages. We believe ReDeBug can be a realistic solution for regular developers seeking to enhance the security of their code in day-to-day development, and make ReDeBug available as an open-source tool.

Chapter 7

Code Resemblance vs. Code Containment

7.1 Introduction

We propose two fingerprinting methods: code similarity detection using feature hashing and code containment detection using Bloom filters. We originally wanted a system that used a single algorithm. While conceptually more elegant, such a design wasn't optimal in either scenario. For example, if we had based our similarity metric on Bloom filters with multiple hash functions, we would have had a larger error rate than with feature hashing due to extra collisions from the extra hash functions. If we had used feature hashing instead of Bloom filters, we would again have had lower accuracy when performing set membership tests. Although the internals of implementing both feature hashing and Bloom filters are almost identical, specific parameters are selected in different ways because the goals are different. Code similarity detection uses a single hash function and a distance metric interface, whereas code containment detection utilizes multiple hash functions and a set membership interface.

Both approaches are also algorithmically different. The time for code similarity detection goes up quadratically with more files because we have to perform $\binom{|N|}{2}$ pairwise similarity calculation. On the contrary, the time for code containment detection increases linearly as we have more files and patches because we query patches against files, i.e., a single sweep over files.

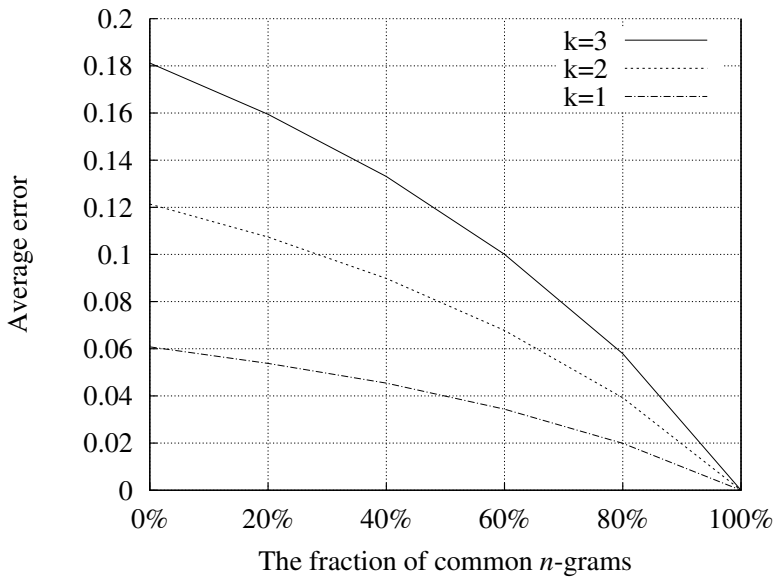


Figure 7.1: Error with various k where m=8,192

7.2 Feature Hashing vs. Bloom filters

A Bloom filter is a probabilistic data structure used to efficiently encode sets and perform set membership tests. Let $h_1, h_2, h_3, ..., h_k$ be a set of hash functions of type $D \rightarrow R$ and $|D| \gg |R|$, i.e., each hash is a compression function. A Bloom filter calculates $h_i(x) = d$ and sets the d'th bit in the m-length bit vector for all hash functions h_i and each feature value x. To test if an element x' is in the feature set, a check is performed to see that the $h_i(x')$ bit is set for all i. If any bits are not set, then x' is not in the set. Bloom filters have false positives due to hash collisions, but never false negatives. The false positive rate is reduced, all things being equal, by adding more hash functions.

Bloom filters did not work well for code resemblance detection because we wanted to approximate the Jaccard similarity, not to perform set membership tests. Feature hashing is similar to Bloom filters where we compute $h(x) = d$ and set the d'th bit, except that only a single hash function is used. Theorem 1 (§3.2.2) shows that the software fingerprints provide a near-optimal approximation of the true Jaccard index. To the best of our knowledge, no previous work (e.g., [31]) has performed a similar analysis. Indeed, the proof shows that increasing the number of hash functions increases error, which is why Bloom filters don't work well. This corresponds well to feature hashing, where only one hash function is used. Further, requiring only one hash has obvious performance improvement implications.

Through simulations on random sets of n-grams, we also showed that the use of single hash function minimized the difference between the Jaccard in Equation 3.1 and the bit vector Jaccard in Equation 3.2. In particular, we created two sets that contain 1000 n-grams each, with varying numbers of overlapping n-grams, and measured how much the bit vector Jaccard differed from the true Jaccard. Figure 7.1 shows the average error as the fraction of common n-grams; the number of hash functions k varies (the standard deviation is very small and is therefore not shown). We note that the error increases as k increases, with minimum error achieved at $k = 1$, which is different from the usual Bloom filter set membership tests.

7.3 Summary

We present two software similarity measures and their underlying fingerprinting algorithms. First, code resemblance between two programs can be measured to tell how similar the programs are. For this purpose, we use *feature hashing* [150, 159] to dramatically reduce the high-dimensional feature spaces that are common in program analysis. Our evaluation shows that our fingerprinting algorithm is up to an order of magnitude faster than previous approaches and uses less memory with comparable accuracy. Second, code containment is detected to find unpatched code clones. For this purpose, we utilize *Bloom filters* [29] to quickly check if code contains known vulnerable code in it. By checking over 2.1 billion lines of code for 376 security-related patches, we found 15,546 unpatched copies of known vulnerable code.

Part III

Conclusion

Chapter 8

Conclusion

To keep modern systems secure, software analysis must be performed to determine whether software contains malicious and/or buggy code. Unfortunately, the sheer volume of new software fueled by extensive code reuse far outpaces the current capacity of software analysis. If we cannot cope with the ever increasing volume of software, we may miss critical security problems. Therefore, it is critical to develop scalable analysis to bridge the gap.

In this dissertation, we show that automatic code reuse detection enables an efficient data reduction of a high volume of incoming malware for downstream analysis and enhances software security by efficiently finding known vulnerabilities across large code bases.

We propose a new software fingerprinting algorithm using feature hashing with bit vectors and approximate software similarity calculation for scalability purposes. We also prove a theoretical bound of our approximation and demonstrate the practical uses of our techniques in many security scenarios, such as malware clustering, software lineage inference, and unpatched code clone detection with large-scale real-world data sets.

Bibliography

[1] Apache hadoop. http://hadoop.apache.org/. (Referenced on pages 6, 40, and 42.)

[2] CMU cloud computer cluster. http://www2.pdl.cmu.edu/~twiki/cgi-bin/view/OpenCloud/ClusterOverview. (Referenced on page 49.)

[3] DARPA-BAA-10-36, Cyber Genome Program. https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-10-36/listing.html. Page checked 5/16/2013. (Referenced on pages 8 and 25.)

[4] Debian security advisories. http://www.debian.org/security/. (Referenced on page 1.)

[5] Malware analysis system. http://mwanalysis.org/. (Referenced on pages 14 and 49.)

[6] Measure of software similarity. http://theory.stanford.edu/~aiken/moss/. (Referenced on page 19.)

[7] National vulnerabilities database. http://nvd.nist.gov/. (Referenced on page 1.)

[8] Offensive computing. http://www.offensivecomputing.net/. (Referenced on page 49.)

[9] Ubuntu security notices. http://www.ubuntu.com/usn. (Referenced on page 1.)

[10] VirusTotal. http://www.virustotal.com/. (Referenced on page 55.)

[11] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts*, 2004. (Referenced on pages 26, 34, 37, 49, and 52.)

[12] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the Association for Computing Machinery*, 51(1), 2008. (Referenced on pages 6, 18, and 21.)

[13] F. Angiulli, E. Cesario, and C. Pizzuti. A greedy search approach to co-clustering sparse binary matrices. In *IEEE International Conference on Tools with Artificial Intelligence*, 2006. (Referenced on page 24.)

[14] A. Asiaee T., M. Tepper, A. Banerjee, and G. Sapiro. If you are happy and you know it... tweet. In *ACM International Conference on Information and Knowledge Management*, 2012. (Referenced on page 21.)

[15] J. Attenberg, K. Weinberger, A. Dasgupta, A. Smola, and M. Zinkevich. Collaborative email-spam filtering with the hashing-trick. In *Conference on Email and Anti-Spam*, 2009. (Referenced on pages 21 and 36.)

[16] 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*, 2007. (Referenced on pages 14, 27, and 58.)

[17] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conference on Reverse Engineering*, 1995. (Referenced on page 14.)

[18] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5), 1997. (Referenced on page 14.)

[19] A. Banerjee, I. Dhillion, J. Ghosh, S. Nerugu, and D. Modha. A generalized maximum entropy approach to bregman co-clustering and matrix approximation. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2004. (Referenced on page 68.)

[20] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: program transformations for practical scalable software evolution. In *ACM/IEEE International Conference on Software Engineering*, 2004. (Referenced on page 134.)

[21] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *IEEE International Conference on Software Maintenance*, 1998. (Referenced on page 15.)

[22] 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. (Referenced on pages 2, 3, 5, 11, 14, 17, 18, 24, 25, 26, 27, 34, 35, 37, 38, 39, 55, 58, 60, 61, and 70.)

[23] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: a tool for analyzing malware. In *European Institute for Computer Antivirus Research*, 2006. (Referenced on page 14.)

[24] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3), 1976. (Referenced on page 30.)

[25] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9), 2007. (Referenced on page 30.)

[26] 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), 2005. (Referenced on page 25.)

[27] D. Bernstein. http://www.cse.yorku.ca/~oz/hash.html. Page checked 3/4/2012. (Referenced on page 38.)

[28] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the Association for Computing Machinery*, 53(2), 2010. (Referenced on pages 29, 112, 120, and 134.)

[29] B. H. Bloom. Space/Time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7), 1970. (Referenced on pages 8, 114, 140, and 145.)

[30] A. Broder. On the resemblance and containment of documents. In *IEEE Conference on Compression and Complexity of Sequences*, 1997. (Referenced on pages 6 and 18.)

[31] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2005. (Referenced on pages 46, 115, and 144.)

[32] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, 2008. (Referenced on pages 111 and 113.)

[33] C. Calvès and M. Fernández. A polynomial nominal unification algorithm. *Theoretical Computer Science*, 403, 2008. (Referenced on page 114.)

[34] P. Cao. http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html. (Referenced on page 122.)

[35] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsous. Fully automatic cross associations. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2004. (Referenced on pages 7, 22, 23, 24, 66, and 68.)

[36] S. Chapman. SimMetrics. http://sourceforge.net/projects/simmetrics/. Page checked 5/16/2013. (Referenced on pages 4, 35, and 50.)

[37] P. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. In *SIAM International Conference on Data Mining*, 2011. (Referenced on page 27.)

[38] H. Cho, I. S. Dhillon, Y. Guan, and S. Sra. Minimum sum-squared residue co-clustering of gene expression data. In *SIAM International Conference on Data Mining*, 2004. (Referenced on pages 23 and 24.)

[39] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, 2005. (Referenced on page 27.)

[40] ClamAV. Creating signatures for ClamAV. `http://www.clamav.net/doc/latest/signatures.pdf`. (Referenced on page 3.)

[41] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. Version control with subversion. `http://svnbook.red-bean.com/en/1.0/svn-book.html#svn-ch-3-sect-4.3.2`. Page checked 3/4/2012. (Referenced on page 116.)

[42] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *ACM Conference on Computer and Communications Security*, 2011. (Referenced on page 28.)

[43] 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*, 2011. (Referenced on page 30.)

[44] F. de la Cuadra. The geneology of malware. *Network Security*, 2007. (Referenced on pages 80, 83, and 84.)

[45] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating System Design and Implementation*, 2004. (Referenced on pages 40 and 42.)

[46] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2003. (Referenced on pages 23, 24, and 68.)

[47] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *ACM Conference on Computer and Communications Security*, 2008. (Referenced on page 28.)

[48] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *IEEE International Conference on Software Maintenance*, 1999. (Referenced on page 14.)

[49] T. Dumitras and I. Neamtiu. Experimental challenges in cyber security: a story of provenance and lineage for malware. In *Conference on Cyber Security Experimentation and Test*, 2011. (Referenced on pages 25 and 31.)

[50] T. Dumitras and D. Shou. Toward a standard benchmark for computer security research: the worldwide intelligence network environment (WINE). In *Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2011. (Referenced on pages 84, 100, and 104.)

[51] N. Edwards and L. Chen. An historical examination of open source releases and their vulnerabilities. In *ACM Conference on Computer and Communications Security*, 2012. (Referenced on page 31.)

[52] D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. In *ACM Symposium on Discrete Algorithms*, 1998. (Referenced on page 38.)

[53] H. Flake. Structural comparison of executable objects. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2004. (Referenced on pages 12 and 50.)

[54] W. B. Frakes and K. Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7), 2005. (Referenced on page 2.)

[55] 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. (Referenced on pages 14 and 27.)

[56] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ACM/IEEE International Conference on Software Engineering*, 2008. (Referenced on page 15.)

[57] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su. Scalable and systematic detection of buggy inconsistencies in source code. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010. (Referenced on pages 15, 29, 111, 112, 120, and 134.)

[58] T. George and S. Merugu. A scalable collaborative filtering framework based on co-clustering. In *IEEE International Conference on Data Mining*, 2005. (Referenced on pages 24 and 68.)

[59] M. Gheorghescu. An automated virus classification system. In *Virus Bulletin Conference*, 2005. (Referenced on pages 12 and 13.)

[60] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *IEEE International Conference on Software Maintenance*, 2000. (Referenced on pages 24, 30, 77, and 80.)

[61] M. Graziano, C. Leita, and D. Balzarotti. Towards network containment in malware analysis systems. In *Annual Computer Security Applications Conference*, 2012. (Referenced on page 14.)

[62] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. Cuckoo sandbox. http://cuckoosandbox.org/. Page checked 4/16/2013. (Referenced on page 14.)

[63] F. Guo, P. Ferrie, and T. Chiueh. A study of the packer problem and its solutions. In *International Symposium on Recent Advances in Intrusion Detection*, 2008. (Referenced on pages 14, 28, and 49.)

[64] A. Gupta, P. Kuppili, A. Akella, and P. Barford. An empirical study of malware evolution. In *International Communication Systems and Networks and Workshops*, 2009. (Referenced on page 31.)

[65] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2012. (Referenced on page 21.)

[66] J. Hartigan. Direct clustering of a data matrix. *Journal of the American Statistical Association*, 67(337), 1972. (Referenced on page 68.)

[67] M. Hayes, A. Walenstein, and A. Lakhotia. Evaluation of malware phylogeny modelling systems using automated variant generation. *Journal in Computer Virology*, 5(4), 2008. (Referenced on page 31.)

[68] A. Hemel and S. Coughlan. Binary analysis tool. http://www.binaryanalysis.org/. (Referenced on page 61.)

[69] X. Hu, T. cker Chiueh, and K. G. Shin. Large-scale malware indexing using function call graphs. In *ACM Conference on Computer and Communications Security*, 2009. (Referenced on pages 2, 3, 5, 11, 13, 24, 25, 26, 36, 38, 55, 58, and 86.)

[70] Q. Huang, X. Chen, J. Z. Huang, S. Feng, and J. Fan. Scalable ensemble information-theoretic co-clustering for massive data. In *International MultiConference of Engineers and Computer Scientists*, 2012. (Referenced on pages 23 and 24.)

[71] International Secure Systems Lab. Anubis: Analyzing unknown binaries. http://anubis.iseclab.org/. Page checked 12/12/2009. (Referenced on pages 14 and 27.)

[72] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2012. (Referenced on pages 26 and 28.)

[73] N. Jain, M. Dahlin, and R. Tewari. Using bloom filters to refine web search results. In *International Workshop on the Web and Databases*, 2005. (Referenced on page 45.)

[74] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy*, 2012. (Referenced on pages 9, 25, and 111.)

[75] 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. (Referenced on pages 6, 7, 11, 17, 24, 25, 33, 86, and 101.)

[76] J. Jang, M. Woo, and D. Brumley. ReDeBug: finding unpatched code clones in entire os distributions. *USENIX ;login:*, 37(6), 2012. (Referenced on page 111.)

[77] J. Jang, M. Woo, and D. Brumley. Towards automatic software lineage inference. In *USENIX Security Symposium*, 2013. (Referenced on page 8.)

[78] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *ACM/IEEE International Conference on Software Engineering*, 2007. (Referenced on pages 29, 111, 112, 120, and 134.)

[79] Jibz, Qwerton, snaker, and xineohP. PEiD. `http://www.peid.info/`. (Referenced on pages 28 and 49.)

[80] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7), 2002. (Referenced on pages 2, 14, 29, 111, 112, 120, and 134.)

[81] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *ACM Workshop on Recurring Malcode*, 2007. (Referenced on pages 14, 28, and 49.)

[82] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful. In *Working Conference on Reverse Engineering*, 2006. (Referenced on page 30.)

[83] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *International Workshop on Evolution of Large-scale Industrial Software Applications*, 2003. (Referenced on page 2.)

[84] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1, 2005. (Referenced on pages 3, 11, 12, 25, 26, 31, 55, 58, 77, and 78.)

[85] A. Karnik, S. Goswami, and R. Guha. Detecting Obfuscated Viruses Using Cosine Similarity Analysis. In *Asia International Conference on Modelling & Simulation*, 2007. (Referenced on page 13.)

[86] G. Karypis. CLUTO: a clustering toolkit, release 2.1.1. Technical report, University of Minnesota, 2003. (Referenced on page 48.)

[87] W. M. Khoo and P. Lio. Unity in diversity: Phylogenetic-inspired techniques for reverse engineering and detection of malware families. In *SysSec Workshop*, 2011. (Referenced on pages 13, 25, 31, 77, and 78.)

[88] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Working Conference on Mining Software Repositories*, 2013. (Referenced on page 105.)

[89] 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. (Referenced on page 30.)

[90] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *ACM Conference on Computer and Communications Security*, 2011. (Referenced on page 14.)

[91] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7, 2006. (Referenced on pages 11, 26, 34, 37, 49, and 52.)

[92] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Symposium on Static Analysis*, 2001. (Referenced on page 15.)

[93] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2), 1996. (Referenced on page 15.)

[94] J. Krinke. Identifying similar code with program dependence graphs. In *Working Conference on Reverse Engineering*, 2001. (Referenced on page 15.)

[95] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymophic worm detection using structural information of executables. In *International Symposium on Recent Advances in Intrusion Detection*, 2005. (Referenced on page 13.)

[96] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004. (Referenced on pages 13 and 81.)

[97] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *IEEE International Conference on Software Maintenance*, 1997. (Referenced on page 29.)

[98] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. AccessMiner: using system-centric models for malware protection. In *ACM Conference on Computer and Communications Security*, 2010. (Referenced on page 27.)

[99] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1), 2001. (Referenced on pages 3, 7, 24, 30, 80, and 84.)

[100] P. Li, L. Lu, D. Gao, and M. Reiter. On challenges in evaluating malware clustering. In *International Symposium on Recent Advances in Intrusion Detection*, 2010. (Referenced on page 28.)

[101] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32, 2006. (Referenced on pages 2, 14, 29, 111, 112, 120, and 134.)

[102] 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. (Referenced on page 31.)

[103] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *ACM Conference on Computer and Communications Security*, 2003. (Referenced on pages 13 and 81.)

[104] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ACM/IEEE International Conference on Software Engineering*, 2007. (Referenced on page 29.)

[105] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *ACM SIGCOMM Conference on Internet Measurement*, 2006. (Referenced on pages 25, 31, and 77.)

[106] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1(1), 2004. (Referenced on page 24.)

[107] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference*, 2007. (Referenced on pages 14, 28, and 49.)

[108] 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. (Referenced on pages 24, 30, and 77.)

[109] J. Mayrand, C. Leblanc, and E. M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *IEEE International Conference on Software Maintenance*, 1996. (Referenced on page 15.)

[110] McAfee. McAfee threats report: Fourth quarter 2012. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2012.pdf, 2012. (Referenced on pages 1 and 3.)

[111] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 1976. (Referenced on page 80.)

[112] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011. (Referenced on pages 15 and 30.)

[113] A. K. Menon, K. Chitrapura, S. Garg, D. Agarwal, and N. Kota. Response prediction using collaborative filtering with hierarchies and side-information. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2011. (Referenced on page 21.)

[114] A. Mockus. Large-scale code reuse in open source software. In *International Workshop on Emerging Trends in FLOSS Research and Development*, 2007. (Referenced on page 61.)

[115] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, 2007. (Referenced on page 58.)

[116] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference*, 2007. (Referenced on page 58.)

[117] National Vulnerability Database. CVE-2008-0928. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0928. Page checked 3/4/2012. (Referenced on page 130.)

[118] National Vulnerability Database. CVE-2008-2327. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-2327. Page checked 9/11/2012. (Referenced on page 110.)

[119] National Vulnerability Database. CVE-2009-3379. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3379. Page checked 9/11/2012. (Referenced on page 108.)

[120] National Vulnerability Database. CVE-2010-0405. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0405. Page checked 9/11/2012. (Referenced on page 129.)

[121] National Vulnerability Database. CVE-2010-2807. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-2807. Page checked 9/11/2012. (Referenced on page 109.)

[122] National Vulnerability Database. CVE-2011-1092. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1092. Page checked 3/4/2012. (Referenced on page 132.)

[123] National Vulnerability Database. CVE-2011-1782. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-1782. Page checked 3/4/2012. (Referenced on page 131.)

[124] National Vulnerability Database. CVE-2011-3200. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3200. Page checked 3/4/2012. (Referenced on page 131.)

[125] National Vulnerability Database. CVE-2011-3368. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3368. Page checked 3/4/2012. (Referenced on page 132.)

[126] National Vulnerability Database. CVE-2011-3872. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3872. Page checked 3/4/2012. (Referenced on page 133.)

[127] M. Neugschwandtner, P. M. Comparetti, G. Jacob, and C. Kruegel. FORECAST: skimming off the malware cream. In *Annual Computer Security Applications Conference*, 2011. (Referenced on page 26.)

[128] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ACM/IEEE International Conference on Software Engineering*, 2010. (Referenced on pages 4 and 29.)

[129] S. Papadimitrou and J. Sun. Disco: Distributed co-clustering with map-reduce. In *IEEE International Conference on Data Mining*, 2008. (Referenced on pages 7, 22, 23, 66, and 68.)

[130] Pattern Insight. Code assurance. http://patterninsight.com/products/code-assurance/. Page checked 3/4/2012. (Referenced on pages 29 and 120.)

[131] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14), 2008. (Referenced on pages 3, 26, and 28.)

[132] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *USENIX Symposium on Networked Systems Design and Implementation*, 2010. (Referenced on pages 17, 27, and 35.)

[133] R. Perdisci and M. U. VAMO: towards a fully automated malware clustering validity analysis. In *Annual Computer Security Applications Conference*, 2012. (Referenced on page 28.)

[134] N. H. Pham, T. T. Nguyen, H. A. Nguyen, X. Wang, A. T. Nguyen, and T. N. Nguyen. Detecting recurring and similar software vulnerabilities. In *International Conference on Software Engineering*, 2010. (Referenced on page 29.)

[135] M. S. Rahman, T. Huang, H. V. Madhyastha, and M. Faloutsos. Efficient and scalable socware detection in online social networks socware on facebook. In *USENIX Security Symposium*, 2012. (Referenced on page 27.)

[136] M. A. Rajab, L. Ballard, P. Mavrommatis, and N. Provos. CAMP: content-agnostic malware protection. In *Network and Distributed System Security Symposium*, 2013. (Referenced on page 27.)

[137] V. Ramanathan, W. Ma, V. T. Ravi, T. Liu, and G. Agrawal. Parallelizing an information theoretic co-clustering algorithm using a cloud middleware. In *IEEE International Conference on Data Mining Workshops*, 2010. (Referenced on page 23.)

[138] K. Rieck. Malheur: Automatic analysis of malware behavior. `http://www.mlsec.org/malheur/`. (Referenced on page 27.)

[139] K. Rieck, T. Holz, C. Willems, D. Patrick, and P. Laskov. Learning and classification of malware behavior. In *IEEE Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, 2008. (Referenced on pages .)

[140] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4), 2011. (Referenced on pages 3, 14, 26, and 27.)

[141] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, 2011. (Referenced on pages 84 and 104.)

[142] 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. (Referenced on pages 14 and 28.)

[143] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of Computer Security Applications Conference*, 2006. (Referenced on pages 14, 28, 49, and 58.)

[144] RSA FraudAction Research Labs. Anatomy of an attack. `http://blogs.rsa.com/anatomy-of-an-attack/`. (Referenced on page 1.)

[145] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis*, 2009. (Referenced on page 13.)

[146] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD Conference on Management of Data*, 2003. (Referenced on pages 6, 11, 18, 19, 30, 39, 52, and 61.)

[147] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *IEEE Symposium on Security and Privacy*, 2001. (Referenced on page 26.)

[148] 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. (Referenced on page 105.)

[149] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *IEEE Symposium on Security and Privacy*, 2009. (Referenced on pages 14, 28, and 58.)

[150] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *Journal of Machine Learning Research*, 10, 2009. (Referenced on pages 6, 20, 21, 34, 35, and 145.)

[151] 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. (Referenced on page 30.)

[152] N. Srndi and P. Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Network and Distributed System Security Symposium*, 2013. (Referenced on page 27.)

[153] Symantec. W32.duqu: The precursor to the next stuxnet. http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet. (Referenced on page 2.)

[154] Symantec. Symantec internet security threat report, volume 17. http://www.symantec.com/threatreport/, 2012. Page checked 5/23/2013. (Referenced on pages 1, 3, 22, 24, and 33.)

[155] S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, 2009. (Referenced on page 11.)

[156] Trustwave. 2013 trustwave global security report. https://www2.trustwave.com/2013GSR.html. Page checked 5/16/2013. (Referenced on page 4.)

[157] Ubuntu Security Notice. CVE-2011-3145. http://www.ubuntu.com/usn/usn-1196-1/. Page checked 3/4/2012. (Referenced on page 132.)

[158] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In *Duplication, Redundancy, and Similarity in Software*, 2007. (Referenced on pages 34, 37, 49, and 52.)

[159] 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. (Referenced on pages 6, 20, 34, 35, and 145.)

[160] G. Wicherski. peHash: a novel approach to fast malware clustering. In *USENIX Workshop on Large-Scale Exploits and Emerging Threats*, 2009. (Referenced on page 26.)

[161] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007. (Referenced on pages 14 and 27.)

[162] 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. (Referenced on pages 24, 30, 77, 80, and 83.)

[163] R. Yang, P. Kalnis, and A. K. H. Tung. Similarity evaluation on tree-structured data. In *ACM SIGMOD Conference on Management of Data*, 2005. (Referenced on page 15.)

[164] S. Yang, B. Long, A. J. Smola, H. Zha, and Z. Zheng. Collaborative competitive filtering: learning recommender using context of user choice. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2011. (Referenced on page 21.)

[165] Y. Ye, T. Li, Y. Chen, and Q. Jiang. Automatic malware categorization using cluster ensemble. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2010. (Referenced on pages 13, 26, and 86.)

[166] J. Zhou and A. Khokhar. ParRescue: scalable parallel algorithm and implementation for biclustering over large distributed datasets. In *IEEE International Conference on Distributed Computing Systems*, 2006. (Referenced on page 23.)

[167] zynamics. BinDiff. `http://www.zynamics.com/bindiff.html`. Page checked 8/5/2012. (Referenced on pages 13, 26, 36, and 86.)