MALWARE PROVENANCE:
DETECTING CODE REUSE IN MALICIOUS SOFTWARE

by

Jason R Upchurch

B.S. Regis University
2007

A dissertation submitted to the Graduate Faculty of the University of Colorado at Colorado Springs in partial fulfillment of the requirements for the degree of Doctor of Philosophy
Department of Computer Science
2016
This thesis for the Doctor of Philosophy degree by

Jason R Upchurch

has been approved for the

Department of Computer Science

by

______________________________
Xiaobo Zhou, Chair

______________________________
Edward Chow

______________________________
Jia Rao

______________________________
Rory Lewis

______________________________
Steven Fulton

______________________________
Date
Detecting code reuse in software has applications in malicious code analysis and malware code search and retrieval, but is complicated by the lack of available source code. Malware Provenance examines the difficulties in detecting code reuse in malware, particularly concerning the transformative processes during compilation. The thesis proposes methods of normalization to allow for code reuse detection within compiled x86 malicious software. It examines a method for filtering as well as visualizing results of similarity uncovered in the process. The thesis proposes a method to conduct similarity measurements that is $O(n \log n)$ for all-pairs comparisons. The approaches, contained within, also allow for the creation of signatures of supersets as well as malware variants, which allow Malware Provenance to function, as not only a code reuse detection approach, but also an accurate and fast method to detect malware family variants. Experimental results show that Malware Provenance is advantageous in detection accuracy and comparison time.
Dedicated to my wife Tiffiney and our fantastic children, Tom, Jordan, and Joseph, without which I could have never made it through the process.
Acknowledgements

I would like to thank the gang formally known as the Intrusions Section at DCFL. The experience of running the greatest Intrusion’s team on the planet was critical to this path of academic exploration of how to automate link analysis with data from malware. In addition, I would like to thank my advisor, Xiaobo Zhou, without whom I would not have navigated the mysteries of the process. Finally, I would like to thank my team at Intel and the Air Force Academy for sponsoring my research to bring Malware Provenance to life.
# Table of Contents

1 Introduction ................................................. 1
   1.1 What is Malware? ........................................ 1
   1.2 Types of Malware ........................................ 2
   1.3 Malware Based Intrusions ............................... 3
   1.4 Code Reuse in Malware Production ..................... 4
   1.5 General Approaches ..................................... 4
       1.5.1 Analysis-through-Execution (Dynamic Analysis) ... 4
       1.5.2 Analysis through Static Examination .............. 6
       1.5.3 Analysis through Comparison ...................... 8
       1.5.4 Research Issues and Approaches .................. 9
       1.5.5 Normalization ...................................... 9
       1.5.6 Feature Dataset Generation ....................... 10
       1.5.7 Dataset Reduction ................................ 12
       1.5.8 Dataset Comparison ................................ 12
       1.5.9 Comparison Reduction ............................. 13
       1.5.10 Comparison Visualization ......................... 14

2 Review of Literature ....................................... 15
   2.1 Traditional Signatures ................................. 15
   2.2 Heuristics .............................................. 16
# Table of Contents

2.3 System Call Analysis ............................................. 17  
2.4 System State Change Analysis ............................... 18  
2.5 Function Extraction .............................................. 18  
2.6 Algorithm Extraction ............................................ 20  
2.7 Code Based Similarity ........................................... 22  
2.7.1 Normalization .................................................. 24  
2.7.2 Vectors ......................................................... 24  
2.7.3 Feature Hashing ............................................... 25  
2.7.4 Control Flow Graph ......................................... 26  
2.7.5 Import Hashing ............................................... 27  
2.8 Visualization ..................................................... 27  

3 Detection of Code Reuse in Compiled Software .......... 28  
3.1 $n$-grams ......................................................... 28  
3.2 Hash Waterfalling ............................................... 30  
3.3 Normalization .................................................... 34  
3.3.1 Byte Code Transformation Examples .................... 36  
3.4 Instruction Normalization Through Substitution Study .... 46  
3.4.1 Approach: BBCP - the Basic Block Comparison Platform . 46  
3.4.2 Implementation ................................................. 47  
3.4.2.1 Normalization ............................................. 47  
3.4.2.2 Approach .................................................. 47  
3.4.2.3 $n$-grams ................................................ 48  
3.4.2.4 Comparison ................................................. 49  
3.5 Experimental Results ............................................ 50  
3.5.1 Same Source, Different Compilers ....................... 51  
3.5.2 Source Subset, Same Compilers ......................... 52  
3.5.3 Same Source Embedded into Dissimilar Programs ....... 53  

viii
Table of Contents

3.6  First Byte Normalization .............................................. 54
    3.6.1  Experimental Results ........................................... 56
            3.6.1.1  Equipment and Dataset ............................... 56
            3.6.1.2  Similarity ........................................... 57
    3.6.2  Normalization Selection ...................................... 59
3.7   Similarity Measures ................................................. 60
    3.7.1  Similarity Comparison Experiment ........................... 60
            3.7.1.1  Window Size for $n$-gram Based Measurements ...... 61
            3.7.1.2  Other Similarity Measurements ....................... 63
3.8   Code Diversity in Compiled Software .............................. 63
    3.8.1  Block Diversity and Feasibility Study ....................... 63
    3.8.2  Experimental Results ......................................... 65

4   Visualization of Results .............................................. 69
    4.1   Similarity via First Byte Feature Hash ....................... 69
            4.1.1  Notes on the Gold Standard ............................ 70
    4.2   Force Based Clustering ......................................... 70
    4.3   Other Similarity Techniques Visualized ....................... 73
    4.4   Removal of Trivial Relationships .............................. 74
            4.4.1  Filtering ............................................. 75
            4.4.2  Filtering Affect on Graph Complexity ................. 77
    4.5   Graph Complexity in Large Sets ................................ 78

5   Exponential Growth in All-Pairs Comparison ........................ 86
    5.1   MinHash ..................................................... 86
            5.1.1  MinHash as a Representation of a Feature Hash .......... 88
            5.1.2  Complexity Time Analysis .............................. 89
    5.2   Experimentation with Minhash ................................. 90
Table of Contents

5.2.1 Initial Feasibility of MinHash: Variance .................................. 90
5.2.2 Initial Feasibility of MinHash to Increase Speed of Comparison 91
  5.2.2.1 MinHash Error Rate ............................................. 92
5.3 Super Signatures ................................................................. 93
  5.3.1 Stream Similarity ........................................................... 95
  5.3.2 Derivative Datasets from MP Signatures Only ..................... 95
  5.3.3 Improving Accuracy and Performance through \( K \) Variance .. 98

6 Malware Provenance as a Variant Detection Tool 100
  6.1 Introduction ................................................................. 100
  6.2 Method: Measurements and Dataset .................................... 102
    6.2.1 Measurements ........................................................ 102
    6.2.2 Dataset .............................................................. 103
    6.2.3 Candidate Solutions for Comparison ................................ 109
      6.2.3.1 CTPH .......................................................... 109
      6.2.3.2 TLSH .......................................................... 109
      6.2.3.3 sdhash ...................................................... 110
      6.2.3.4 BitShred ..................................................... 111
      6.2.3.5 First Byte ................................................... 112
  6.3 Experimental Results ..................................................... 113
    6.3.1 Choosing Solution Options ......................................... 113
      6.3.1.1 BitShred Window Size ..................................... 113
      6.3.1.2 TLSH Bounding .............................................. 114
      6.3.1.3 First Byte Implementations ................................ 115
    6.3.2 Binary Classification Comparison ................................. 115
    6.3.3 Performance Comparisons .......................................... 118
      6.3.3.1 Signature Generation ...................................... 118
      6.3.3.2 All-Pairs Comparison ...................................... 119
Table of Contents

6.4 Observations ............................................. 120
   6.4.1 Inconsistencies ................................... 120
   6.4.2 Reproducibility ................................... 122
   6.4.3 Measurement Alignment ........................... 122
   6.4.4 Comparative Results ............................. 123

7 Malware Provenance Realized 125
   7.1 Experimental Results ............................... 126
      7.1.1 Equipment and Dataset ....................... 126
      7.1.2 Accuracy of First Byte Feature Hash Similarity .... 126
      7.1.3 Known Data Set Comparison .................. 127
         7.1.3.1 Block Based Clustering vs. Industry Comparative Results .... 129
      7.1.4 Filtering Using Available Function Identification ........ 130

8 Conclusion and Future Work 133
   8.1 Conclusions ........................................ 133
   8.2 Future Work ....................................... 137

9 Publication of Results 138

Bibliography 139
List of Figures

3.1 Full Byte Waterfall Fouling, Window=5, 31% Match. ............... 34
3.2 Substitution Process. .............................................. 49
3.3 Full Byte Waterfall Fouling, Window=5, 31% Match. ............... 55
3.4 First Byte Waterfall Fouling, Window=5, 73% Match. ............... 56
3.5 Similarity Threshold vs. Matches Detected. ........................ 58
3.6 Window Size vs. Matches Detected. ............................... 58
3.7 Feature Window 3 vs Edit Distance. ............................... 62
3.8 Feature Window 5 vs Edit Distance. ............................... 62
3.9 Feature Window 10 vs Edit Distance. ............................... 62
3.10 Fuzzy Hash vs Edit Distance. ................................. 62
3.11 Matching Vs. Exclusion based on hash window size. ................ 67
3.12 10 Instruction Window. ........................................... 68
3.13 20 Instruction Window. ........................................... 68
4.1 92 malware samples, clustered, no filter. ......................... 72
4.2 92 malware samples, Fuzzy Hash Cluster. ........................ 74
4.3 Affect of Filtering on Similarities. ................................. 77
4.4 25k Blocks, Unfiltered, 50172 similarities found. .................. 79
4.5 25k Blocks, Filtered, 9828 similarities found. ..................... 80
4.6 92 malware samples, Malware Provenance Clustering with Filtering. 81
List of Figures

4.7 2.3K malware samples, Malware Provenance Clustering with Filtering. 82
4.8 Focused Graph, 92 Samples. .................................................. 83
4.9 Focused Graph, 2.3K Samples. ............................................... 84
4.10 Focused Graph, 3 iterations, 500K Samples. ......................... 85

5.1 Feature Window 10 vs. MinHash 100. ................................. 91
5.2 Feature Window 10 vs. MinHash 1000. ............................. 91
5.3 Projected Feature hash Computation for 250M Filed. ............. 92
5.4 LSH vs. Feature Hash at Projected 250M blocks. ................. 93
5.5 Diagram Depicting MinHash Based Block Signature Creation. . 96
5.6 Diagram Depicting MinHash Based Section Signature Creation. . 96
5.7 Diagram Depicting MinHash Based Signature Compilation. .... 97

6.1 Precision/Recall Curve for BitShred Windows. .................... 113
6.2 Precision/Recall Curve for TLSH. ....................................... 114
6.3 FMeasure vs. Threshold for First Byte Types. ...................... 116
6.4 Precision/Recall Curve for First Byte Types. ....................... 117
6.5 FMeasure Comparison. ........................................................ 118
6.6 Precision/Recall Curve Comparison. ................................... 119
6.7 Signature generation performance per node. ....................... 120
6.8 Time to conduct all-pairs comparison. ............................... 121
6.9 Comparison Performance. .................................................. 122

7.1 92 malware samples, clustered, no filter. ............................ 128
7.2 Known grouping to clustered groups, lower is better. ............. 130
7.3 92 malware samples, clustered, Sys32/bin filter. ................... 131
7.4 92 malware samples, clustered, combined filter. ................... 132
List of Tables

3.1 GCC32 vs. GCC64 vs. VS2012 ........................................ 39
3.2 GCC32 vs. GCC64 vs. VS2012 Stack Prolog ..................... 40
3.3 GCC32 vs. GCC64 vs. VS2012 Align, Debug .................... 41
3.4 Same Source, Different Compilers ............................... 51
3.5 Source Subset, Same Compilers ................................... 53
3.6 Source Embedding .................................................... 53

6.1 Recall at Precision ............................................... 116
6.2 Precision at Recall ............................................... 117
6.3 Signature Generation Times ..................................... 118
6.4 All-Pairs Comparison Projections ............................... 119
6.5 Published Measurements vs. Tested Measurements .......... 121
Chapter 1

Introduction

Malware analysis means many things to many people. To network defenders, malware analysis is understanding malware so that it can be detected on the network or system. To risk mitigators, malware analysis is understanding the full functionality of a malware sample to determine or limit the possible impact a malware based intrusion has on the enterprise. To engagement leads, malware analysis can help determine the sophistication of the attackers. Finally, to law enforcement or intelligence units, malware analysis can help link intrusion sets and identify probable attackers or origins of attack. While the goals of malware analysis may differ, the basic process of examining malware remains the same.

1.1 What is Malware?

To understand malware analysis and the ongoing research in the field, we must first define malware and remove the many misconceptions surrounding it. Malware is short for malicious software; therefore, any definition must include software. While this definition should be implicit and immediately discernible to all, there remains confusion over the definition and components of malware. One misconception is that not all intrusions involve malware. The prolific use of malware is actually a
fairly recent addition to intrusions if you look at the history of the topic. However, while it is true that most of today’s intrusions are malware centric, it is a fallacy to believe that all intrusions, therefore, involve malware. Intrusions involving password guessing, social engineering and SQL/Command injection can all be accomplished quite successfully without creating or injecting any software. Another misconception is that malware includes one or more exploits. To examine this misconception, let us again look at the definition of malware: Malicious Software. The "mal" in malware is maliciousness. Software that is malicious is software that is using a computer for another’s gain without permission. Therefore, malware is software that:

1. arrived on a computer through exploit, misuse, or deceit and uses a computer for another’s gain.

Without examining this any further, we can see that malware does not always contain an exploit. If software intended for another’s gain arrives on a computer via deceit, for example a fishing email, then it is malware that could contain no exploit. Further, even if malware arrives via some initial exploit, that does not imply that such malware contains the exploit. Consider the ever popular PDF exploit. Many of these exploits give control to injected, simple, downloaders that retrieve more complex malware.

1.2 Types of Malware

Another issue with understanding malware analysis is the sheer number of terms associated with defining the specifics of malware. Worms, viruses, rootkits, et al. The confusion is compounded by our malware taxonomy being based on behavior, not implementation. Our classification system for malware is based, almost exclusively, upon behavior. Where in the natural world, alpha taxonomy is a fully developed taxonomy system that classifies organisms by tangible traits, our behavior system of classification for malicious software would classify an anteater, and many birds, in-
sects, reptiles, and amphibians as related because they all consume ants, while clearly they have only superficial biological relationships and little genetic relationship. While behaviors are very important in understanding how to locate and stop malware in any incident, we should not concentrate on grouping malware by behaviors. That is not to say identifying malware behaviors is not important, identifying malware behaviors is extremely important. It is the grouping that is not important, particularly since such groupings are not cleanly accomplished, as most malware samples can fit into more than one group. Unlike grouping malware into arbitrary buckets, describing malware behaviors, in a general sense, is important. Identifying malware behaviors can help determine the impact of malware on a particular machine. As such, categorizing malware behaviors can be helpful. Malware, as stated before, may or may not have an exploit. It can be self propagating. It could have the ability to scan other machines or self update. These are types of behaviors that we would like to discover and describe.

1.3 Malware Based Intrusions

Up until very recently, malware based intrusions have been rising for a decade [1]. While there has been a slight decrease this year, malware still represents the focus for cyber attackers to gain unauthorized access to systems and networks. These attacks can use malware to self-propagate and obtain access to millions of computers or use malware, such as a spear fishing attack, for a point of entry. Computer intrusions are conducted for profit, for espionage, to gain intellectual property, and other reasons; however, the cost to victims is multiplicative to the gain of the attackers. Consider the TJX intrusion in 2007, which I was personally involved in the investigation [2]. At last report, the loss of 94 million credit card numbers cost the cooperation $256M [3].
Chapter 1. Introduction

1.4 Code Reuse in Malware Production

Malware continues to grow at millions of samples a month [4][5]. While new technologies are focusing on exploit prevention, this proliferation of malware shows that the tide of malware on the world’s computers has not yet been stemmed. Malware analysts continue to be at a premium and demand cannot be met with the low supply of skilled professionals. There is a need for a technology to bridge the gap between malware analyst expertise and identifying malware that is not fully new. As a result, malware similarity is an active research area in industry and academia.

1.5 General Approaches

1.5.1 Analysis-through-Execution (Dynamic Analysis)

Analysis-through-Execution, commonly called dynamic analysis, is based on the simple premise that malware behavior during execution can give keen insight into the nature of the malware and its purpose. This naive method of this type of analysis is simple system state monitoring during execution. This process involves monitoring system state changes, to include file system changes, writes to databases that control OS level functionality (registry), network access, process creation, etc. This method is often very useful for creating signatures for detection software.

Malware is often packed or obfuscated in its functionality to prevent analysis [6]. The naive method overcomes many of the simple defenses malware authors write in, or otherwise add, to their software to prevent analysis. Simple packing and obfuscation can be overcome in many cases by monitoring a free execution of the malware [7], though work has been completed to statically disassemble obfuscated malware [8] and other techniques such as malware stack manipulation can present difficult challenges [9].
However, naïveté has its costs. First, free execution does not guarantee malicious algorithms are executed. For instance, many malware samples have detection algorithms for analysis environments or only display malicious behavior if certain conditions (such as a date, connectivity, location, switches) are met [1]. Other costs of naive execution analysis include no insight into how the malware was coded, who coded it, and the algorithms that generate malicious behavior. As a result, much of the research into malware analysis has been centered around overcoming these limitations in Analysis-through-Execution [10][11][12][13].

Generally, Analysis-through-Execution requires that the machine be reset after execution. It is the nature of malware analysis to determine how malware effects the machine, therefore it is not practical to clean the machine after the execution of the malware as changes to the machine are not known until examination. Even in a heavily monitored environment, there is always the possibility that the machine was affected in a novel way which was missed by the monitors. As such, the requirement exists to construct an environment that is capable of being reset to a controlled state.

Early approaches to constructing environments to conduct Analysis-through-Execution was through the use of virtual machines. Indeed, this approach exists as the most popular choice in industry to conduct malware analysis and testing. Virtual machines allow for snapshotting the machine state prior to execution and returning to that state to conduct further analysis. It is popular because it is quite inexpensive, easy to configure, and allows for freezing the execution in place when analysis is being conducted through a debugger.

However, problems with this technique were soon encountered when malware writers and writers of anti-analysis packers began to build into their packages VM detection engines. Reactive approaches in obscuring the VM from VM “aware malware” were the initial response to this anti-analysis problem. These approaches included custom patches to VMWare and other vitalization packages themselves and using de-
buggers to attach to the malware and sending it false information when the software was checking its environment.

While significant progress was made on the reactive front in preventing malware from detecting its environment, researchers looked to other methods to subvert detection of the environment. The most direct approach was to not use emulation of the environment at all, while still retaining the ability to reset the environment to a known state. This was accomplished through disk caching. By placing a write cache on the disk, the operating system would be free from permanent change. The analyst could simply use a prepared operating system to execute the malware and monitor the execution. Subsequent executions could be completed on a clean operating system by clearing the cache. This approach has been extended to full virtualization in environments such as Xen [14] as to evade host base virtualization detection that is now commonly found in some malware [15].

Other approaches involving analysis through execution involve full system virtualization and tracing, rather than traditional virtual machine time-sharing schemes. The approach is to control the environment fully by not executing on real hardware. By using this approach, hardware clock and other detection methods for traditional virtual machines do not work. In addition, full access to the hardware allows techniques like memory shadowing and full OS level tracing to be implemented. The use of TEMU [16] and QEMU [17] as a fully virtual environment has given rise to projects that allow fully virtualized malware analysis [12][18] and automated algorithm extraction [19].

1.5.2 Analysis through Static Examination

Static analysis refers to an analysis of code without execution. Typically, it involves the reversal of the assembly process from machine code, though some success has been found in much simpler structural comparison [20]. As there is a one-to-one correlation
Chapter 1. Introduction

between most assembly language instructions and their corresponding machine level
op codes, the idea is fairly straightforward at a high level.

In practice, disassembly is not so trivial. Consider function pointers. If a function
is called and addressed at runtime, there is no straightforward method to construct a
call graph to the function. Similar situations exist for data elements, embedded data,
indirect branching, and the variable length instruction set in popular processors that
can lead to alignment problems [21]. Also, two approaches to disassembly are possi-
bile: Linear Sweep and Recursive Traversal. In addition, types, data structures and
communication protocols in malware are often user defined and can often be key com-
ponents that must be reverse engineered to understand behavior. As such, automation
to produce understanding of data has also been explored [22][23][24][25][26].

Linear sweep disassembly starts at the first byte in executable section of a binary
and begins disassembly. Each following op code is considered to start at the next
byte following the previous instruction. Obviously, alignment becomes critical and
even well formed machine language generated from a typical compiler can quickly be-
come unmanageable. Optimizations to linear sweep disassembly such as realignment
and interactive sessions following a garbage instruction can help, but are not always
solutions.

Recursive traversal approaches disassembly from a different perspective. It does
not begin disassembly at the beginning of the executable section, but at the entry
point of the executable. For each call or conditional jump with a static reference, a
table is created for future entry points to examine. The disassembly continues until a
return or unconditional jump is found. At that point, the next entry point is processed
from the table. The disassembly continues until all entry points are exhausted.

In general, recursive traversal is considered more robust. While disassembling and
static analysis, is an active area of research, most effort is placed on modifying the
commercial dissembler IDA Pro [27]. Various projects that mirror much of the goals
covered here in dynamic analysis have been attempted or partially covered statically through plugins or otherwise augmenting IDA Pro.

While static analysis is of use in both academia and industry, it is easily defeated through obfuscation [28]. Simple packing of an executable makes analysis through static means difficult or impossible. An entire industry, with both legitimate and of more questionable application, has developed to prevent static analysis. This mostly focuses on obfuscation through post complication packing or through compilation add-on packages, though emulation environments and purposeful stack manipulation can be used to obscure malware analysis [29][9].

The widespread use of obfuscation in malware has lead to research in the area of semantic analysis. The Hyperion project from the Department of Energy’s Oak Ridge Laboratory is one such project [30]. Originally started in CMU’s CERT, Software Engineering Institute, the project seeks to derive program behavior from static code analysis without execution [31]. The project defines a method to transform procedural logic expressed in sequence into a deterministic, behaviorally equivalent, functional form. This method then allows for both analysis and similarity through behavior comparison. An important note is that this behavioral approach does not ignore implementation details entirely and may be a useful approach in determining code reuse in software.

1.5.3 Analysis through Comparison

The last general topic in malware analysis identified in this thesis is approaches to comparison methods used in malware analysis and detection. This area is critically important to developing the field as projects such as Apiary [32] and standards such as OASIS [33] rely on large, searchable, malware data repositories to achieve their goals. Where static and execution analysis provide insight and information for a capabilities-analysis and signature generation, in a pure concept view of the two areas, they do
Chapter 1. Introduction

not inherently compare malware samples against one another. It is comparisons that generate general knowledge about techniques and methods used in malware. This large dataset approach can be used for both detection and attribution.

The simplest form of comparison is the use of all, or some part of, the static representation of the code to compare against other samples. Examples of this technique range from file hashing and anti-virus signatures, to string comparison via regular expression of windowed hashing. The use of Bloom filters [34][35] to compare windowed hashing across an entire executable section of malware has been shown to be quite robust in identifying malware variants [36]. While these types of comparisons are easy to implement, current malware, packers, and polymorphic wrappers can easily defeat such methods and are a main driver for current research into other approaches to compare malware [37].

1.5.4 Research Issues and Approaches

The research issues for the thesis can be broken down into:

1. Normalization
2. Feature Dataset Generation
3. Dataset Reduction
4. Dataset Comparison
5. Comparison Reduction
6. Comparison Visualization

1.5.5 Normalization

Normalization with respect to comparing machine code to other machine code for the purpose of identifying code reuse is the process of removing compiler changes to
the code in order to deobfuscate differing transformations in output (machine code) given the same input (source code). This process itself has been an active research area. It is difficult, if not improbable, that transformations can be reversed entirely for each and every compiler and each and every optimization option. A compiler can be viewed as a black box transformation, even though some compilers are open source, their complexity is such that, even with access to the compiler as a white box, the transformation specifics are obscured. Therefore, normalization not only seeks to reverse differences introduced by compilers, but also remove changes that cannot be reliably reversed in a deterministic fashion. It is a balanced approach as normalization itself dilutes the value of identifying similarity.

In addition to the compilation process itself, the creators of malware have a vested interest in changing the syntax of malware to evade detection, while retaining the semantic execution of the code. In work [38], the authors point this out and test their similarity measurement to detect variants (TLSH)[39] against commonly encountered techniques that introduce these types of changes.

While all software, through the compilation process, may require normalization for meaningful similarity measures, malware is more prone to intentional changes specifically designed to thwart similarity measures [40]. As such, we anticipate that normalization will play a significant role in the accuracy and robustness of any project designed to detect code reuse in malware, so it is no surprise that work is also being conducted in advanced normalization techniques [41][42].

1.5.6 Feature Dataset Generation

In order to detect code reuse in malware, specifically code likely to have been written by malware writers, one must look at the individual blocks, functions, or algorithms that give malware its unique functionality. One approach would be to
analyze the malware file as a whole, thereby accessing code reuse by looking at all information in the file. Ssdeep [40] is widely used for this purpose. Competing works [38][43] have attempted to increase the accuracy and speed of this method. However, similarity measures with these approaches do not identify the location of the similarity or even focus only on code, nor do they have the ability to filter out unwanted information. The unit of input is the executable file, code and data, and the dataset represents only files.

Another approach would be to generate a dataset of particular sections of an executable file. This is the method used in works [44][45][36] in where the executable section is extracted from the file and used for comparison. Other sections could also be extracted and used in comparison, revealing more information on what lead to similarity. However, as the code section is still inputed as a whole, the feature dataset represents each entire executable section extracted from the malware file. Therefore, specific code that generated similarity is not easily identified.

To achieve the proposed goal of identifying code reuse in compiled malware through similarity measures, any dataset generation must be of fine enough detail to trace back to code reuse. Individual blocks (assembled basic blocks) are easily extracted from unpacked samples through industry tools used in reverse engineering [27]. They represent a more detailed view of the malware and therefore any representative dataset generated from these blocks is also more detailed.

The dataset itself can be a representation of the input data or the input data itself, with or without normalization. One case for using a representation of a dataset, rather than the data, is that the representation often reduces the storage space needed to conduct the comparison. Another case is that the representation of data also allows for dimensionality reduction of the dataset. Further, many dataset representations
Chapter 1. Introduction

obscure the input data such that the dataset representation is safely shared without compromising all information from the input data. A cryptographic hash is an example of this kind of obfuscation as the original input is not easily discovered by attacking the hash digest itself, particularly without access to the original input.

1.5.7 Dataset Reduction

Dataset reduction with respect to similarity analysis affects both the processing time of similarity calculations as well as the quality of the similarity itself. Processing time is straightforward. The less input, the less time to process the data. This is of interest in this project as there are 100’s of millions of malware samples in the wild today and growing. The quality of similarity measures are a function of the ability to filter known data that is of little interest.

Filtering, particularly of known data, is accomplished by identifying similarity to a known set. In the case of software, the most obvious candidate set for filtering of code is libraries. While library functionality could be essential in identifying malware variants as in work [46], identifying meaningful code reuse in new malware will rely on the ability to determine the code boundaries of code reuse. If boundaries, such as basic block boundaries, are identified and used in the generation of the dataset, then filtering of known code is possible. As a large portion of software is generated through library calls, this filtering can reuse input size as well as reduce noise from the identification of library code reuse.

1.5.8 Dataset Comparison

A straightforward technique for comparison would be to conduct a binary diff with a tool such as vbindiff. This method relies on the input data itself and is not normalized. In addition, the vbindiff method suffers from sequencing issues and is
Chapter 1. Introduction

unlikely to produce reliable results when comparing malware for variants. It is not useful in finding code reuse beyond assisting manual analysis.

To remove the sequencing issue, the input data can be compared using a “fuzzy hashing” tool. These tools are implementations of works [40][47][39] and use various methods of measuring similarity between files. All of these tools use data projection to condense and order the information in such a way to limit the impact of sequencing issues. Each successive work shows improvement over the previous in accuracy and/or speed of comparison; however, initial testing with known malware variants calls into question the usefulness of such tools in comparing malware, particularly when the input is raw executable files.

Creating a comparative dataset by projecting information has distinct advantages. In addition to allowing for the reduction of the dimensionality of the data, it is an opportunity to allow for structuring of the data to decrease comparison time. Approaches to reducing comparison time include parallelizing an $O(n^2)$ approach where all inputs are compared to all other inputs; conducting a candidate selection step to reduce $n$; or projecting the dimensionality to a fixed/ordered set to allow for a probabilistic search in $O(n \log n)$ time.

1.5.9 Comparison Reduction

A successful approach in calculating all-pairs similarity to identify code reuse in modest malware datasets (2.3K malware samples with 550K blocks) produces over a million similarities after the filtering of libraries. Increasing the effectiveness of filters can reduce the number of relationships; however, it is likely that a post similarity calculation reduction will be needed to allow for human understanding of relationship data. One approach is visualization of the data (discussed below); however, visualization alone may not compensate for a very large similarity graph size.
Chapter 1. Introduction

Other approaches to reduce the size and complexity of relationship data would be to limit the relationship information to only a focus. This approach would allow for a representation of similarity data with respect to a single malware sample of interest, removing noise from similarity data not specifically related to the sample at hand. Another approach would be to remove frequent similarity data beyond common libraries. If similarity for a block is found for other blocks beyond a certain threshold count, it could be declared uninteresting and removed from the output.

1.5.10 Comparison Visualization

The last research issue is creating a method to understand similarity data in large sets of malware intuitively. Visualization has long been used to understand complex information and is not new to the understanding of malware [48] and cyber analysis in general [49]. In this thesis, we seek to find code reuse in malware. As this is the case, identifying similarity between code blocks in malware samples would naturally produce an undirected or mixed network graph. Relationships from preprocessing would be defined as a malware sample to all of its constituent code blocks. Similarity calculations would link blocks from other malware samples, creating a network graph. There are a number of tools available to process network graphs to understand relationships. However, none of these tools are tailored towards understanding similarity. As such, it is likely that some research effort will be needed to identify how network graph visualization can be applied to understand software/malware code reuse.
Chapter 2

Review of Literature

2.1 Traditional Signatures

Virtually all current commercial anti-malware technologies rely on signature and/or anomaly detection methods to detect malware [50]. Signature detection involves a pattern matching scheme to detect malware based on previously identified malware. Often, the pattern is somewhat dynamic in that it is a regular expression. However, there are limitations to this scheme. First, patterns must be carefully selected to not match software that is not malicious, though automation of this process has been studied [51]. As anti-malware programs often, legitimately, require privileged system access, misidentifying non-malicious software as malicious can have significant impact [52]. Therefore, signatures must be chosen carefully.

The next problem with signature matching is the ease at which regular expressions can be defeated if they are strict enough to not lead to the false positive issue above. Simple techniques, such as hash waterfalloing, Listing 3.2 and Listing 3.4, in which relative offsets are changed inside the machine code can defeat regular expressions. Other techniques, such as changing compiler settings, changing compilers, even rearranging function order with no code changes [53] have been shown to defeat
Chapter 2. Review of Literature

signature-based detection schemes.

Another problem with signature-based detection methods is the volume of signatures themselves. With approximately 300 million wild malware samples, conducting a signature-based search for malware against all known malware is not possible on endpoint systems without significant performance issues. In addition, even continuous live memory monitoring is not yet viable due to both the volume of signatures and the computational overhead of regular expressions. These constrains have led to the practice of removing low-volume signatures to make room for signatures of trending malware.

Lastly, signature-based detection methods are reactive. They rely on back end systems and expertise to generate signatures based on known malware or malware analysis. In order to generate a signature for malware, a sample of that particular malware sample must be captured and identified. The signature generation then must comply with the constraints above, further slowing this reactionary process.

2.2 Heuristics

Heuristic malware detection is the process of identifying anomalous or known bad behavior in an attempt to identify malicious software. It is an algorithm based approach that attempts to identify malware without the constraints of signature-based systems. The concept is to partition software into groups based on behavior or content, winnowing to reduce either behavior or content to a small set of unknown software. The algorithms can be hard coded or use machine learning principals in an attempt to identify malicious software on a given system.

The draw back to the heuristic approach is the impact of a false positive. Just as with signatures, the misidentification of legitimate software as malicious can have serious consequences to machine and application availability. Even very small false
positive rates, which are within acceptable limits in many other fields, are not accept-
able in malware identification. As a result, heuristic malware identification has
been limited to user warnings or eliminated entirely in many instances. In the end,
heuristic based approaches can provide a user or analyst with a scoring system that
identifies the risk of malware, but in many instances provides too many false positives
to be a general solution [54][55][50].

2.3 System Call Analysis

The analysis of system calls can be used to identify malicious behaviors [46][56]. The
work [57] employs a system-call driven, behavior-mapping technique as an approach
to identifying malware. However, rather than use the behavioral mapping to derive
functionality from an analysis perspective, the work creates a map to compare against
other programs. The behavioral map is generated through dependency analysis be-
tween system calls. It is important to note that malicious system calls need not be
known prior to analysis. In this way, the work derives malicious behavior without
preexisting analysis or signatures.

To determine significant behaviors unique only to malware, two sets are used to
train the system. One is a known set of malware and the other is a known set of
non-malware. Behaviors are mined from both sets and a behavioral map is created.
The sets of maps are then filtered against each other via a process based on the
adjacent work in leap mining [58]. Each subset of malware maps, filtered through
negative maps, is then combined to produce a near optimal specification of malware
signatures.

Behaviors are mined from system calls. It is not just the calls themselves that
are traced, but the interrelation of calls with each other. As such, if a function
return of one call is used as an argument to another, those calls are interrelated. The
relationships of calls is used to form a map. The filtering process allows for a near complete positive graph that defines malicious actions. The process is near complete in that it uses leap mining to produce a near optimal solution to avoid exponential computational hurdles in the complete method. Leap Mining can be explored in more detail in [58].

2.4 System State Change Analysis

System state change as a method for identifying malware behavior and identification was one of the first developed malware analysis methods. The process is further explored as a way to automate malware classification in work [59]. The approach is to monitor the system as an output of black box and classify the software based on changes to the affected system. The approach is currently used in industry and research. However, as the approach is black box by nature, the implementation to accomplish behaviors is fully abstracted. Therefore, the approach is not viable as a method to identify code reuse in malware.

2.5 Function Extraction

One of the primary advantages of Analysis by Execution is the ability to monitor execution. One obvious point of interest during execution is to determine the use of any given function. Here dynamic analysis can provide much insight. For instance, while a function call is easy to identify statically in un-obfuscated code, its argument types are not. Dynamic analysis and system state monitoring through the use of an execution monitor can provide concise answers to what is required for a function to execute properly, such is the research in [60]. This research work involves the use of QEMU to construct an execution monitor below the operating system. The execution monitor tracks function entry and exit and produces a memory dump at the exit of
functions of interest.

The memory dump is used to evaluate the assembly code statically (see below) with dynamic information. Since the evaluation is done from a memory standpoint just after execution of the function, most obfuscation techniques are bypassed during this process. The process is still vulnerable to detection of the analysis environment, as are all dynamic processes [61]. The process is also vulnerable to obfuscation through added, but zero sum, instructions [60]; however, in practice it is not encountered often.

The [60] project uses the commercial off-the-shelf IDA Pro Disassembler [27] for static analysis, but uses dynamic information caught in the trace to resolve indirect and dynamic jumps and calls. All information is collected from the static binary, its memory images, and the trace to provide a path of execution and typing needed to generate a C type function.

The function generation process uses the disassembly and typing information to create a C interface for the function. The trace and disassembly is used to generate a function prototype for the extracted code. The code itself is arranged into basic blocks and static addresses are recoded to labels to function outside of the sample binary. Finally, a code generator assembles the C code necessary to create the function and corresponding header [60].

The process of function extraction has its limits. Many functions are interdependent on output of other code and functions; therefore, these functions are not self contained. The use of a function extracted from static code will not function correctly if some predetermined values are necessary to execute the function properly. Other difficulties encountered in function extraction are predicting or understanding function side effects and relevant auxiliary instructions [19].
2.6 Algorithm Extraction

In an effort to overcome these issues, researchers have moved to full algorithm extraction such as [19]. The approach is to identify an event within the malware functionality that an analyst would have interest in. The running example in [19] was the algorithm to download, decrypt, and execute an update to the malware. The most relevant event in the example would be to capture the file that was downloaded and decrypted, prior to execution of the update. As such, it would be possible to track updates by replaying the algorithm designed by the malware writers without actually executing the malicious content of the original program nor the update.

While such algorithms have been extracted prior to [19], the process was manual and extremely time consuming. The goal of [19] was to not extract simple functions, but to extract all information needed to recreate an entire algorithm that achieves some tangible result. The project goes further to create the code and process necessary to execute the algorithm without significant analyst intervention.

The researchers in [19] divide the work into two phases, the first is the extraction process and location of all of the necessary information needed to recreate the algorithm. The second is the technique to generate a stand alone application to execute the algorithm.

The first step in phase one is for an analyst to process the malware in the monitoring environment. The project uses a modified version of Anubis [12] to achieve this task. A detailed taint analysis is captured by the analysis environment by marking data elements returned by a system call. The system then tracks how the data interacts with the execution of the malicious code. The process allows for a tracking system of how functions and other code are interdependent, thus all of the code, data, and globals for the algorithm are identified. The relationships, values, and even the executed code and flow are tracked during the process. The execution trace defeats obfuscation as the actual executed code is tracked outside of what is stored at rest.
Following the initial execution, an analyst can use the data to locate specific functionality of the program for study. This can be an unpacking algorithm [62], file creation (as in the running example), or indeed any functionality of the program. The target point is identified by the analyst and the system can extract the interrelated information needed to recreate the event. This is accomplished by tracing all taint and code information that is used to achieve the event. The process uses both API call return/entry information as well as defining variables used during the process through mapping and examining memory locations used for variable storage during execution. The process is conducted recursively, backward through the program until no recursive triggers are left to follow and closure is met.

One additional avenue pursued by the researchers is to examine forward in the process prior to extraction. For instance, in the case of an encrypted update downloaded from the Internet. The update is not of much use if it has not yet been decrypted. Thus, the project allows for an analyst to examine how data is interrelated forward in execution prior to extraction.

The information needed to reconstruct an algorithm is only the first phase of the [19]. The automated reconstruction of the algorithm into a stand alone application is a non-trivial task. Here researchers, primarily dealing with MS Windows malware, used a combination of a Gadget Player and a dynamically loadable library to reanimate the extracted algorithm, reproduce the system result, all without running the malware itself.

As with all linear execution analysis methods, [19] cannot extract data or code not executed in the analysis environment. Elements that can detect the analysis environment, rely on external conditions not present in the analysis environment, or simply work on a schedule or time outside of the actual analysis, will be missed. This has lead to efforts to conduct Analysis by Execution in a multi-path manner [63].
2.7 Code Based Similarity

Other finer grained analysis research has been conducted to determine not if one software sample is similar to another, but to determine if code reuse is found in software samples [64]. The idea here is to extract functions from executables and compare those functions to others. This is the goal of [65]. In this work individual functions are identified through static analysis via call and return statements. The extraction and comparison information is generated in four stages. First, the code for each function is extracted and disassembled. Next, a normalization process removes references to specific registers or memory locations. Next a full function hash is created to identify exact duplicates and immediately return a match. Lastly, a feature vector is created from the function to do comparison. The feature vector index is based on a hash created from a sliding window with a set stride and all features encountered in it.

The comparison is based on an indexing scheme of encountered hashes within the sliding window. The hashes are store conventionally, thus the size of the stored values is large. The authors report that the database required at least 26 times the size of the input. This with a stride of 10. It can then be extrapolated that a full sliding window would be space prohibitive with this approach.

Another contribution of [65] is the removal of trivial clones and the consolidation of multiple input matches into larger clones. In the first case, trivial clones are those that are too small or so frequently encountered that they are not relevant. This is particularly true when the stride is larger than the window and the function code segment is small. The collapse of the structure of the function from the computation of the vector leaves a higher probability that there is vector overlap.

Another similarity scoring method that has come to light in malware analysis is [40]. This method is somewhat similar to the sliding window method in [65], except that there is no creation of vectors (structure is retained), stride is always equal to 1,
Chapter 2. Review of Literature

and the window size varies with input size.

The work in [40] was a modification of the spamsum algorithm. It can also be viewed as a modification to the piecewise hashing algorithm. Piecewise hashing is a sliding window hash without overlap and a user specified block length. Piecewise hashing is useful in determining homologous files (those file with matching contiguous bytes larger than the hashing block). Likewise, [40] is a sliding window hash; however, the block length varies with input on the fly. It maintains a machine state so that as input is received and the window is incremented, the influence of information sliding outside of the window can be removed. As such, it can change window sizes as needed without a restart.

Evaluation of similarity is done with a one-to-one comparison, measuring the distance from one result to the other. The result is then computed over the size of the inputs to determine the similarity score of the two files. While context triggered block size is valuable in evaluating input of like size, its one severe limitation is files that greatly differ in size as they will not match as block sizes vary from tolerances.

The use of ssdeep [40] has been applied to malware analysis similarity scoring as well [66]. This work, functions are extracted from the disassembly generated from a malware sample. The functions are then processed with ssdeep [40] to generate a fuzzy hash. Hashes are then compared within a relational database. While this is useful, there is no straight forward method to order a fuzzy hash signature. Thus comparisons are exponential to dataset size within [66].

The research in works [67][47][68][69][70][71][72][73] all center on evaluating, improving upon accuracy, and increasing the performance of the file similarity technique. While many of these projects have approaches of interest to increasing the performance/accuracy of similarity hashing, and some even address finding similarity in compiled code, few incorporate a normalization process to overcome transformative issues surrounding compiled software. However, these works are of particular interests
2.7.1 Normalization

To make conventional static comparison more robust, various techniques to normalize the code samples can be used to preprocess the sample before the comparison is made. Take for instance, simple obfuscation through packing. If the code can be unpacked, string comparisons and partial hash signatures can again be effective. Most commercial anti-virus products use proprietary unpacking or partial execution techniques to reveal obfuscated code.

Another example of normalization is to abstract registers and memory locations that may vary with different compilations and slight changes in source so that machine code or abstracted assembly can be directly compared with other samples [65]. This type of normalization, at least to some extent, occurs naturally as a part of the fixing of memory locations for function and algorithm extraction [74][19].

A more robust normalization would be to abstract the optimized machine language to a platform independent intermediate language. In work [75], this concept was implemented to allow for the modification and update of compiled binary software for which the source code was no longer available and to compile software in different languages. An intermediate language was used specifically for malware analysis in work [42]. In this project, an intermediate language generator (VINE) was used to abstract assembly to a platform independent language in which side effects (actions caused by CPU flags, etc) were preserved.

2.7.2 Vectors

While these types of normalization processes can combat similarity misses due to substitution and reordering of functions, they do nothing to compensate for the
reordering of instructions. In work [65], the authors normalize for substitution and then take a second step to remove reordering issues by creating another projection of the data in the form of a vector. The process orders all normalized instructions such that the structure, and therefore, similarity missed due to reordering, is lost. This is an interesting approach; however, there is a concern that normalization of both substitution and structure would result in an increase in false positives.

2.7.3 Feature Hashing

Feature Hash was defined in work [45] (BitShred), which was a result of a discovery that Bloom Filters used to store projected information in their first work [44], created through a sliding window hash, increased the probability of collision beyond that of storing a single projection of the window’s hash. The work was expanded in [36] to further define a framework for processing large malware datasets. The approach is to use a sliding window hash across the executable section of a malware sample. The sliding window hash, called a feature hash, is stored as a sparse matrix of features that represent the malware sample. The matrices are then compared and a Jaccard Index is calculated by taking the intersection and dividing by the union of the corresponding matrices. This creates a value between 0 and 1, which represents a close approximation of similarity between inputs.

The expanded work also addressed a normalization technique based on behavior rather than attempting to normalize the code for analysis. This approach uses techniques from [18] to extract features from malware samples and then calculates similarity using their feature similarity engine. This addresses limitations in static analysis pointed out in work [76]; however, while it is an interesting approach in identifying malware variants, it is black-box (in that behaviors can be implemented independently) and does not constitute identifying code reuse. Even while not rel-
evant to identifying code reuse specifically, behavior analysis is an active area of research in malware identification and analysis [77][78].

The BitShred project is still of particular interest due to its success in detecting malware variants statically. It improves upon the techniques found in the file similarity tools by using the code section of malware rather than the file as a whole. The feature hashes produced represent both the content and some structure of the executable code. While the hash window=5 reduces the structure representation in the data projection (discussed below), the success of the project led to the use of feature hashing in the initial studies in this project.

2.7.4 Control Flow Graph

Another technique to both normalize and analyze malware for similarity measurement is to ignore the syntax of the implementation and instead focus only on the structure of the program [79]. The approach in work [80] was to use control flow graphs (CFG) as the input unit in detecting similarity. The advantages of the approach is that compiler optimizations rarely introduce structural differences between code blocks. The distance between two samples of malware then becomes the edit distance between the CFG. The approach is unattractive for this work as it relies on large portions of code to have structural similarity to detect similarity. While it has been shown to be successful in detecting malware variants, the purpose of this work is to move beyond malware variant detection into code reuse detection in non-variants. In addition, graph similarity has been shown to be NP-hard when calculating similarity exactly and polynomial when estimating similarity between two graphs [81]. Other uses of control flow within malware have been to force execution of dormant blocks in an effort to measure behavior of malware that is attempting to hide functionality from analysts [82].


Chapter 2. Review of Literature

2.7.5 Import Hashing

Import hashing is another technique that evaluates similarity of malware variants by ignoring the specifics of the executable code, which is frequently manipulated to avoid detection algorithms, and focuses only on potential behaviors. The technique uses cryptograph hashing (or similarity hashing) to identify similarity in malware variants by comparing import tables. It is currently used in industry as part of a multi-step process to classify malware. While useful in identifying malware variants, as with many other works, the approach is not useful in identifying code reuse.

2.8 Visualization

Initial studies into the feasibility of the detecting code reuse quickly identified the need for a visualization framework to understand relationship data. As demonstrated in BitShred, relationship data produced when calculation similarity is many times the input size. BitShred used a nearest neighbor plot to show clustering data as well as a binary matrix to demonstrate similarity co-clustering of malware variants within a family. The need for a visualization component is driven by the large amount of data produced from a similarity comparison as multiple similarities are often found with each input.
Chapter 3

Detection of Code Reuse in Compiled Software

3.1 n-grams

Code reuse detection in source code can be related to natural language reuse detection. Such detection is commonly used in academic settings to detect plagiarism or other reuse of work, in web search engines to offer similar web pages or news feeds, and has been used to detect source code reuse in open source projects [83]. Work to move beyond single source languages has also been completed, which uses tokenization schemes to find code reuse [84].

Most of the processes in natural language reuse detection leverage n-grams. An n-gram is a window of tokens that is used to determine a hit or miss in similarity detection. The technique was first described in work [85] as a method for text categorization, refined in [86], and used for malicious variant detection in work [87]. While an n-gram can consist of any stream, in natural language detection, it is common to use words as individual tokens rather than individual bytes or letters. This approach gives no greater weight to longer words and allows for removal of very common words
Chapter 3. Detection of Code Reuse in Compiled Software

that may over represent similarity. This approach has proven very effective for the
detection of reuse of written work.

The number of hash windows within a given input is \( n = T - W \) where \( T \) is
the number of Tokens within a given input and \( W \) is the window size. Given any
set of tokens, the number of possible \( n \)-grams is exponential with respect to window
size. Therefore, \( n \)-grams in their natural form can represent both a very large search
space and a cumbersome computation comparing two strings to identify matches. To
eliminate the need for a CPU intensive string comparison, a cryptographic hash could
be taken of the \( n \)-gram string and stored in numerical form.

Using a hash to represent \( n \)-gram extraction from an input has two key advantages.
First, a conversion to an integer allows for more efficient comparison within the CPU
as well as the possibility of quickly sorting \( n \)-gram representations to allow for a
logarithmic lookup. Second, as the number of \( n \)-grams extracted from the input is
likely high, the need for a large distribution field from the cryptographic hash is
reduced [88]. Increasing the probability of collision from a single \( n \)-gram hash is
offset by the number of hashes taken. Therefore, hash output space can be reduced
significantly. In other works, this space has been reduced as far as 7 bits, without
significant error [36]. This property of using \( n \)-gram hashes with small output fields
allows for a dimensionality reduction in the search space.

Any match for individual \( n \)-grams taken from two or more inputs is the identifica-
tion of an isolated similarity between the inputs. For \( n \)-grams that represent the full
input (without normalization) and are represented in their natural form (not hashed),
every match represents a common feature that has been detected within the inputs.
A set of features compared with a second set of features in this manner results in a
distance, where each negative match increases the distance.

Similarity can also be calculated via a Jaccard index by calculating the union and
intersection of each individual feature. Given two feature sets \( F_a, F_b \) extracted from
inputs $I_a, I_b$, similarity of the inputs is determined via the Jaccard index: $J(F_a, F_b) = \frac{|F_a \cap F_b|}{|F_a \cup F_b|}$. However, storage of each n-gram hash from each basic block is not optimal. The dimensionality of the Feature Set in natural form is extremely large and the comparison is cumbersome. Using a hash representation allows for a reduction in storage space and comparison time. Further, it is possible to represent each hash value as an index into a large bit array, provided the hash is small. This creates a probabilistic data structure for which similarity can be calculated. Simple unions and intersections are easily, and more importantly, quickly calculated from such structures via simple bit operations. This circumstance can be used to calculate an approximate Jaccard Index as in work [36]. That is, $J(F_a, F_b) \approx \frac{S(B_a \land B_b)}{S(B_a \lor B_b)}$ where the Jaccard Index $J(F_a, F_b)$ is approximated by taking the bit AND and the bit OR of each bit array $B_i$, counting the set bits with function $S()$, and dividing the result. It is an approximation due to the probability of collision within each bit array.

### 3.2 Hash Waterfalling

In malware, source code is rarely obtainable. The usefulness of identifying code reuse as a tool for attribution leverages this fact. As such, malware code reuse detection must use code in its binary, machine language form. This presents a number of hurdles beyond natural language detection.

In high level languages, variable labels are used in place of memory locations to simplify programing, improve portability, and allow for flexibility in design. The transformation to assembly, then to machine language during the compilation process removes this abstraction [89]. The resulting executable byte-code contains no labels, only memory addresses for variables. Further, high level concepts for control structures, function segmenting, and direct access to variable information may also be lost during the compilation process.
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.1: Source Code snippet from CRC32a.

```c
unsigned int crc32a(unsigned char *message) {
    int i, j;
    unsigned int byte, crc;

    i = 0;
    crc = 0xFFFFFFFF;
    while (message[i] != 0) {
        byte = message[i];        // Get next byte.
        byte = reverse(byte);      // 32-bit reversal.
        for (j = 0; j <= 7; j++) {
            // Do eight times.
            if ((int)(crc ^ byte) < 0)
                crc = (crc << 1) ^ 0x04C11DB7;
            else crc = crc << 1;
            byte = byte << 1;        // Ready next msg bit.
        }
        i = i + 1;
    }
    return reverse(~crc);
}
```

Consider the Listing in 3.1 taken from the online compendium from work [90]. This is a very small function from an implementation of CRC32. The function contains 4 variables: \( i, j, \) byte, and crc. Now, consider the Listing in 3.2. This is a disassembly of a compiled portion of a function in CRC32. This code block contains two initial variables referenced in lines 4 and 5 with move instructions indexed from the base pointer of the stack which are \( i \) and crc. As the variables no longer exist in their directly accessible labeled name from source code, they can only be referenced by location. In fact, every variable, including functions, can only be accessed by location. As functions and data locations are specific to every compilation (or in some instances, every execution), the locations can pollute code reuse detection results.

Consider the code Listing 3.3, which is the same CRC function as before, but with an extra variable added. The code Listing in 3.4, which is again a disassembly from function, contains the dummy variable (a variable that is not used in the program
other than its declaration and initialization). The initial variables can be seen in lines 4, 5, and 6, which are \( i \), \( X \), and \( crc \) respectively. The resulting insertion causes multiple changes to the instructions below the insertion. The jump on line 7, the moves on lines 13 and 14 into the result variable, and the call in the last line are all changed with the variable insertion. This effect cascades throughout the code block. This effect is known as hash waterfalling as minor changes in offset are propagated down the code block.

The effect of hash waterfalling is more easily visualized when looking at the byte-code directly. In Listing 3.1, byte-code for the CRC32 block is shown as before, except that the disassembly information is removed. The effect of the waterfall from the variable insertion is easily seen to make changes throughout the code block. \( n \)-grams created from a sliding window through this byte-code would be changed often; thus, hash waterfalling has the effect of significantly fouling the feature hashing of each code segment. Indeed, the effect of inserting a single variable, which is not used except to be initialized, reduces the similarity via Jaccard index to 31\%, even with the
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.3: Source code snippet from CRC32a with extra variable.

```c
unsigned int crc32a(unsigned char *message) {
    int i, j, X;
    unsigned int byte, crc;

    i = 0;
    X=0;
    crc = 0xFFFFFFFF;
    while (message[i] != 0) {
        byte = message[i]; // Get next byte.
        byte = reverse(byte); // 32-bit reversal.
        for (j = 0; j <= 7; j++) { // Do eight times.
            if ((int)(crc ^ byte) < 0)
                crc = (crc << 1) ^ 0x04C11DB7;
            else
                crc = crc << 1;
        }
        byte = byte << 1; // Ready next msg bit.
    }
    i = i + 1;
}
return reverse(~crc);
```

Listing 3.4: Code snippet from CRC32a, waterfalled with an extra variable.

```
55     push    ebp
89E5    mov      ebp, esp
83EC24  sub      esp, 24h
C745FC00000000 mov [ebp+var_4], 0
C745F400000000 mov [ebp+var_C], 0
C745ECFFFFFFFH mov [ebp+var_14], 0xFFFFFFFF
EB5A    jmp short loc_4014C6

loc_40146C:
8B45FC mov eax, [ebp+var_4]
034508 add eax, [ebp+arg_0]
8A00 mov al, [eax]
0FB6C0 movzx eax, al
8945F0 mov [ebp+var_10], eax
8B45F0 mov eax, [ebp+var_10]
890424 mov [esp+24h+var_24], eax
E83FFFFFFF call __Z7reversej
```
Chapter 3. Detection of Code Reuse in Compiled Software

very small window of 5 bytes. With x86 machine language and other variable length

\[
\begin{array}{cccccccccccc}
55 & 89 & E5 & 83 & EC & 14 & C7 & 45 & FC & 00 & 00 & 00 & 00 \\
C7 & 45 & F0 & FF & FF & FF & EB & 52 & 8B & 45 & FC & 03 \\
45 & 08 & 8A & 00 & 0F & B6 & C0 & 89 & 45 & F4 & 8B & 45 & F4 \\
89 & 04 & 24 & E8 & 46 & FF & FF & FF \\
\end{array}
\]

Figure 3.1: Full Byte Waterfall Fouling, Window=5, 31% Match.

instruction languages, there also exists a weighting problem where some instructions affect the \( n \)-gram output more strongly than others. In the listings provided, the move immediate instructions and the call are all longer than 5 bytes. This circumstance is similar to the natural language similarity detection issue where longer words become more important than shorter words. In the natural language issue, the solution is to tokenize the words, not the letters or their byte representations. Thus, longer words are not arbitrarily more weighted in a similarity score than shorter words.

### 3.3 Normalization

One approach to alleviating the problems encountered with hash waterfalls and variable-length instruction weighting is to operate on and normalize the disassembly. Each instruction can be easily tokenized with a mnemonic and an operand(s). One such scheme is as follows:

- *Registers replaced with “REG”*
- *Locations replaced with “LOC”*
- *Constant memory references replaced with “MEM”*
- *Constant values replaced with “CONST”*
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.5: CRC32a, Normalized Disassembly.

```
push REG
mov REG, REG
sub REG, CONST
mov VAR, CONST
mov VAR, CONST  ← Dummy variable
mov VAR, CONST
jmp short LOC

mov REG, VAR
add REG, VAR
mov REG, [REG]
movzx REG, REG
mov VAR, REG
mov REG, VAR
mov VAR, REG
call LOC
```

- Variable References replaced with “VAR”

This normalization scheme applied to the CRC32 code snippet can be seen in 3.5. The differences in the two code blocks are reduced to the single instruction that was inserted into the code block. The resulting Jiccard similarity is 81% if tokens are defined as mnemonics or operands and 73% if defined as instructions. However, in order to complete this normalization process, each instruction must be fully disassembled, followed by a textual search and replace before being sent to a hash to create an \( n \)-gram. In addition, all substitution and replacement schemes require published tables of how mnemonics and operands are defined and replaced. This is particularly important given the use of a hash as a representation of an \( n \)-gram of tokens. Changes in formatting, naming, replacement tables, encoding, and capitalization (some of which would, quite easily, be overlooked in a specific implementation) would result in a false negative for a \( n \)-gram match.
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.6: Disassembly, First Byte of Instruction.

```
      push    ebp
     89 E5  mov    ebp,   esp
     EC24  sub    esp,  24h
    45FC00000000  mov    [ebp+var_4],  0
    45F00000000  mov    [ebp+var_C],  0
    45ECFFFFFFF  mov    [ebp+var_14],0FFFFFFFh
     5A  jmp    short  loc_4014C6
loc_4014C6:
     45FC  mov    eax,  [ebp+var_4]
     4508  add    eax,  [ebp+arg_0]
     00  mov    al,    [eax]
     B6C0  movzx   eax,   al
     45F0  mov    [ebp+var_10], eax
     45F0  mov    eax,  [ebp+var_10]
     0424  mov    [esp+24h+var_24], eax
     E8  call   _Z7reversej
```

However, not all normalization schemes of disassembly require replacement tables. Consider the same disassembly from before but with the first byte of each instruction highlighted as in Listing 3.6. A comparison of the unmodified CRC32 snippet with the modified, using only the first byte of each instruction, can be seen in Listing 3.4. Here, the fouling caused by the variable insertion is limited to only the insertion itself. Just as in the textual replacement scheme with instructions as tokens, the first byte represents the instruction and yields the same 73% match for Jaccard similarity.

### 3.3.1 Byte Code Transformation Examples

There are a number of different test cases in similarity research for compiled software. Small changes to source code can have a large effect on the final byte-code. However, while these changes are difficult, the compiler design itself is a much larger issue. Consider the code disassembly from 32bit Visual Studio 2012 (listing 3.8) and GCC32 4.9.3 (listing 3.7). To focus on the mnemonic variations, a listing of side by side
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.7: Code snippet from CRC32a GCC32.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>89E5</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>83EC28</td>
<td>sub esp, 28h</td>
</tr>
<tr>
<td>C745F4000000+</td>
<td>mov [ebp+var_C], 0</td>
</tr>
<tr>
<td>C745E8FFFFFF+</td>
<td>mov [ebp+var_18], 0FFFFFFFh</td>
</tr>
<tr>
<td>EB55</td>
<td>jmp short loc_401296</td>
</tr>
<tr>
<td>loc_401241:</td>
<td></td>
</tr>
<tr>
<td>8B55F4</td>
<td>mov edx, [ebp+var_C]</td>
</tr>
<tr>
<td>8B4508</td>
<td>mov eax, [ebp+arg_0]</td>
</tr>
<tr>
<td>0D0</td>
<td>add eax, edx</td>
</tr>
<tr>
<td>0FB600</td>
<td>movzx eax, byte ptr [eax]</td>
</tr>
<tr>
<td>0FB6C0</td>
<td>movzx eax, al</td>
</tr>
<tr>
<td>8945EC</td>
<td>mov [ebp+var_14], eax</td>
</tr>
<tr>
<td>8B45EC</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>890424</td>
<td>mov [esp+28h+Str], eax</td>
</tr>
<tr>
<td>E843FFFFFF</td>
<td>call _reverse</td>
</tr>
</tbody>
</table>

comparisons of mnemonics can be seen in Table 3.1. First observations are that the
code snippet from Visual Studio 2012 to the first jump (this is a partial disassembly of
a full block) is much larger than that of GCC32. Also, as GCC and Visual Studio use
differing stack management schemes, the Visual Studio code contains a larger function
prolog, as seen in Table 3.2. If the prologs are removed, the similarity improves, but
not by much as the Visual Studio byte-code includes a short addition of space to
debug buffer overflows as seen in Table 3.3. Additionally, GCC conducts the source
code while loop conditional test below the code and Visual Studio conducts the test in
line. Normalization of such changes is intensive as can be seen in this short example;
however, with our short example, there is a path forward to normalize at least this
example.

The comparison between a 64bit and 32bit compiler of the same make is also
interesting. In a comparison with 32bit and 64bit GCC of the same version (4.8.3),
the actual byte-code is quite different; however, when a comparison of mnemonics is
conducted, the code is very similar (one extra instruction converting a 32bit integer
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.8: Code snippet from CRC32a Visual Studio 2012.

```Assembly
55                  push    ebp
89E5                mov      ebp, esp
81ECF0000000        sub      esp, 0F0h
53                  push    ebx
56                  push    esi
57                  push    edi
8DBD10FFFFFF        lea      edi, [ebp+var_F0]
B93C000000          mov      ecx, 3Ch
B8CCCCCCCC          mov      eax, 0CCCCCCCCCh
F3AB                rep      stosd
C745F8000000000     mov      [ebp+var_8], 0
C745D4FFFFFFFF      mov      [ebp+var_2C], 0FFFFFFFFh

loc_41145C:
8B4508              mov      eax, [ebp+arg_0]
0345F8              add      eax, [ebp+var_8]
0FB608              movzx    ecx, byte ptr [eax]
85C9                test     ecx, ecx
7467                jz       short loc_4114D0
8B4508              mov      eax, [ebp+arg_0]
0345F8              add      eax, [ebp+var_8]
0FB608              movzx    ecx, byte ptr [eax]
894DE0              mov      [ebp+var_20], ecx
8B45E0              mov      eax, [ebp+var_20]
50                  push     eax
E86FCFFFFFF        call     .reverse
```

38
Table 3.1: GCC32 vs. GCC64 vs. VS2012.

<table>
<thead>
<tr>
<th></th>
<th>GCC32</th>
<th>GCC64</th>
<th>VS2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>push</td>
<td>push</td>
<td>push</td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td>push</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>push</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>push</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>jmp</td>
<td>lea</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>movsx</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>mov</td>
<td>rep</td>
<td>stosd</td>
</tr>
<tr>
<td>movzx</td>
<td>add</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>movzx</td>
<td>movzx</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>movzx</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>movzx</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>mov</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>jz</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movzx</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mov</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td></td>
<td>push</td>
<td>call</td>
<td></td>
</tr>
</tbody>
</table>
Table 3.2: GCC32 vs. GCC64 vs. VS2012 Stack Prolog.

<table>
<thead>
<tr>
<th></th>
<th>GCC32</th>
<th>GCC64</th>
<th>VS2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>push (Prolog)</td>
<td>push (Prolog)</td>
<td>push (Prolog)</td>
<td></td>
</tr>
<tr>
<td>mov (Prolog)</td>
<td>mov (Prolog)</td>
<td>mov (Prolog)</td>
<td></td>
</tr>
<tr>
<td>sub (Prolog)</td>
<td>sub (Prolog)</td>
<td>push (Prolog)</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>push (Prolog)</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>jmp</td>
<td>lea</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>movsxd</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>mov</td>
<td>rep stosd</td>
<td></td>
</tr>
<tr>
<td>movzx</td>
<td>add</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>movzx</td>
<td>movzx</td>
<td>mov</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>movzx</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>mov</td>
<td>test</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>jz</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Chapter 3. Detection of Code Reuse in Compiled Software

Table 3.3: GCC32 vs. GCC64 vs. VS2012 Align, Debug.

<table>
<thead>
<tr>
<th>GCC32</th>
<th>GCC64</th>
<th>VS2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov</td>
<td>mov</td>
<td>lea (debug code)</td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>mov (debug code)</td>
</tr>
<tr>
<td>jmp (while test)</td>
<td>jmp (while test)</td>
<td>mov (debug code)</td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>rep stosd (debug code)</td>
</tr>
<tr>
<td>mov</td>
<td>movsx</td>
<td>mov</td>
</tr>
<tr>
<td>add</td>
<td>mov</td>
<td>mov</td>
</tr>
<tr>
<td>movzx</td>
<td>add</td>
<td>mov (while test)</td>
</tr>
<tr>
<td>movzx</td>
<td>movzx</td>
<td>add (while test)</td>
</tr>
<tr>
<td>mov</td>
<td>movzx</td>
<td>movzx (while test)</td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>test (while test)</td>
</tr>
<tr>
<td>mov</td>
<td>mov</td>
<td>jz (while test)</td>
</tr>
<tr>
<td>call</td>
<td>mov</td>
<td>mov</td>
</tr>
<tr>
<td>call</td>
<td></td>
<td>add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>movzx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push</td>
</tr>
<tr>
<td></td>
<td></td>
<td>call</td>
</tr>
</tbody>
</table>

to a 64bit for processing. These small differences can be seen in Listing 3.9 and Table 3.1. In fact, all changes to the compiler can affect byte-code output, from differing compiler versions to optimizations. In the running example, a move to GCC 5.3 from 4.9 has no impact on the code snippet save for location changes. However, optimizing to level 3 with GCC 5.3 has a dramatic effect. A quick view of Listing 3.10 reveals that the optimized byte-code has little resemblance to the unoptimized code save for the initialization of the variables. While some straightforward changes, substitutions, or removals to prologs and out of order conditionals could be used to compensate for a number of differences, it is clear that a more complex process would be necessary to discover code similarity in at least some instances.

Even the use of the state of the art de-compilation, available in premium versions of IDA Pro, is not able normalize byte-code to the point of similarity being viable. While code between Visual Studio and Unoptimized GCC is quite similar (as can be
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.9: Code snippet from CRC32a GCC64.

```
55
4889E5
4883EC30
48894D10
C745FC00000000
C745F0FFFFFFFF
EB59

push rbp
mov rbp, rsp
sub rsp, 30h
mov [rbp+arg_0], rcx
mov [rbp+var_4], 0
mov [rbp+var_10], 0FFFFFFFFh
jmp short loc_1004011E4

loc_10040118B:

mov eax, [rbp+var_4]
movsx rdx, eax
mov rax, [rbp+arg_0]
add rax, rdx
movzx eax, byte ptr [rax]
movzx eax, al
mov [rbp+var_C], eax
mov eax, [rbp+var_C]
mov ecx, eax
call _reverse
```

seen in Listings 3.12 and 3.11, respectively), the decompilation of the optimized code
as seen in Listing 3.13 would be extremely difficult to transform into an unoptimized
version for comparison. However, it should be noted that GCC is open source, so GCC
specific optimizations are well documented; thus, it is likely that a de-optimization
algorithm could be written for such compilers.
### Listing 3.10: Code snippet from CRC32a GCC Optimize=3.

<table>
<thead>
<tr>
<th>Line</th>
<th>Machine Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
<td>push edi</td>
<td>push esi</td>
</tr>
<tr>
<td>56</td>
<td>push esi</td>
<td>push ebx</td>
</tr>
<tr>
<td>53</td>
<td></td>
<td>83EC10 sub esp, 10h</td>
</tr>
<tr>
<td>83EC10</td>
<td></td>
<td>8B742420 mov esi, [esp+1Ch+arg_0]</td>
</tr>
<tr>
<td>0FB606</td>
<td></td>
<td>movzx eax, byte ptr [esi]</td>
</tr>
<tr>
<td>84C0</td>
<td></td>
<td>test al, al</td>
</tr>
<tr>
<td>0F8451010000</td>
<td></td>
<td>jz loc_401386</td>
</tr>
<tr>
<td>83C601</td>
<td></td>
<td>add esi, 1</td>
</tr>
<tr>
<td>BBFFFFFFFFFF</td>
<td>mov ebx, 0FFFFFFFFh</td>
<td></td>
</tr>
<tr>
<td>8D7600</td>
<td>lea esi, [esi+0]</td>
<td></td>
</tr>
<tr>
<td>loc_401240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>89C1</td>
<td>mov ecx, eax</td>
<td></td>
</tr>
<tr>
<td>D1E8</td>
<td>shr eax, 1</td>
<td></td>
</tr>
<tr>
<td>81E1555555555</td>
<td>and ecx, 55555555h</td>
<td></td>
</tr>
<tr>
<td>255555555555</td>
<td>and eax, 55555555h</td>
<td></td>
</tr>
<tr>
<td>01C9</td>
<td>add ecx, ecx</td>
<td></td>
</tr>
<tr>
<td>09C1</td>
<td>or ecx, eax</td>
<td></td>
</tr>
<tr>
<td>89CA</td>
<td>mov edx, ecx</td>
<td></td>
</tr>
<tr>
<td>C1E902</td>
<td>shr ecx, 2</td>
<td></td>
</tr>
<tr>
<td>81E2333333333</td>
<td>and edx, 33333333h</td>
<td></td>
</tr>
<tr>
<td>81E1333333333</td>
<td>and ecx, 33333333h</td>
<td></td>
</tr>
<tr>
<td>C1E20</td>
<td>shl edx, 2</td>
<td></td>
</tr>
<tr>
<td>09CA</td>
<td>or edx, ecx</td>
<td></td>
</tr>
<tr>
<td>89D0</td>
<td>mov eax, edx</td>
<td></td>
</tr>
<tr>
<td>C1E604</td>
<td>shr edx, 4</td>
<td></td>
</tr>
</tbody>
</table>

... continues for many more lines / _reverse is inlined_
Chapter 3. Detection of Code Reuse in Compiled Software

Listing 3.11: Decompilation of Snippet: CRC32a GCC Optimize=0.

```c
int __cdecl sub_40122B(int a1)
{
    int v2; // [sp+10h] [bp-18h]@1
    int v3; // [sp+14h] [bp-14h]@2
    signed int i; // [sp+18h] [bp-10h]@2
    int v5; // [sp+1Ch] [bp-8h]@1
    v5 = 0;
    v2 = -1;
    while (*(_BYTE *)(v5 + a1))
    {
        v3 = sub_4011A0(*(BYTE *)(v5 + a1));
        for (i = 0; i <= 7; ++i)
        {
            if ((v3 ^ v2) >= 0)
                v2 <<= 2;
            else
                v2 = (((v2 ^ 0x4C11DB7) << 1) & 0x3F) | (v2 >> 31);
        }
        v3 <<= 2;
        ++v5;
    }
    puts("Jason");
    return sub_4011A0(~v2);
}
```


```c
int __cdecl crc32a(int a1)
{
    char v2; // [sp+Ch] [bp-F0h]@1
    int v3; // [sp+20h] [bp-2Ch]@1
    int v4; // [sp+8Ch] [bp-9Ch]@3
    int i; // [sp+48h] [bp-14h]@3
    int v6; // [sp+F4h] [bp-8h]@1
    memset(&v2, 0xCCu, 0xF0u);
    v6 = 0;
    v3 = -1;
    while (*(_BYTE *)(v6 + a1))
    {
        v4 = *(BYTE *)(v6 + a1);
        v4 = j_reverse(v4);
        for (i = 0; i <= 7; ++i)
        {
            if ((v4 ^ v3) >= 0)
                v3 <<= 2;
            else
                v3 = (((v3 ^ 0x4C11DB7) << 1) & 0x3F) | (v3 >> 31);
        }
        v4 <<= 2;
        ++v6;
    }
    puts("Jason");
    return j_reverse(~v3);
}
```
Listing 3.13: Decompilation of Snippet: CRC32a GCC Optimize=3.

```c
unsigned __int32 __cdecl sub_401220(unsigned int *a1)
{
    unsigned int v1; // eax@1
    int v2; // eax@2
    int v3; // ebx@2
    int v4; // ecx@3
    int v5; // edx@3
    int v6; // eax@3
    int v7; // edx@3
    bool v8; // sf@3
    int v9; // ebx@3
    int v10; // ecx@5
    int v11; // ebx@7
    int v12; // ecx@9
    int v13; // ebx@11
    int v14; // ecx@13
    int v15; // edx@15
    unsigned int v16; // eax@20
    int v17; // edx@20
    unsigned __int32 v18; // ebx@20

    v1 = *(_BYTE *)a1;
    if ( (_BYTE)v1 )
    {
        v2 = (int)((char *)a1 + 1);
        v3 = -1;
        do
        {
            v4 = (v1 >> 1) & 0x55555555;
            v5 = ((unsigned int)v4 >> 2) & 0x33333333 | 4 * (v4 & 0x33333333);
            v6 = ((unsigned int)v5 >> 4) | 16 * (v5 & 0xF0F0F0F);
            v7 = 2 * v3;
            v8 = (v3 ^ (v6 << 24)) < 0;
            v9 = 2 * v3 ^ 0x4C11DB7;
            if ( !v8 )
                v9 = v7;
            v10 = 2 * v9 ^ 0x4C11DB7;
            if ( ((v6 << 25) ^ v9) == 0 )
                v10 = 2 * v9;
            v11 = 2 * v10 ^ 0x4C11DB7;
            if ( ((v6 << 26) ^ v10) == 0 )
                v11 = 2 * v10;
            v12 = 2 * v11 ^ 0x4C11DB7;
            if ( ((v6 << 27) ^ v11) == 0 )
                v12 = 2 * v11;
            v13 = 2 * v12 ^ 0x4C11DB7;
            if ( ((v6 << 28) ^ v12) == 0 )
                v13 = 2 * v12;
            v14 = 2 * v13 ^ 0x4C11DB7;
            if ( ((v6 << 29) ^ v13) == 0 )
                v14 = 2 * v13;
            v15 = 2 * v14;
            if ( ((v6 << 30) ^ v14) < 0 )
                v15 = 2 * v14 ^ 0x4C11DB7;
            v3 = 2 * v15 ^ 0x4C11DB7;
            if ( (v15 ^ (v6 << 41)) == 0 )
                v3 = 2 * v15;
            ++v2;
            v1 = *(BYTE *)((v2 - 1);
        } while ( (_BYTE)v1 );

    v16 = ((unsigned int)v3 >> 1) & 0x55555555 | 2 * (~v3 & 0x55555555);
    v17 = (v16 >> 2) & 0x33333333 | 4 * (v16 & 0x33333333);
    v18 = byte_swap ulong((unsigned int)v17 >> 4) & 0xF0F0F0F | 16 * (v17 & 0xF0F0F0F));
    }
    else
    {
        v18 = 0;
    }
    puts("Jason");
    return v18;
}
```
3.4 Instruction Normalization Through Substitution Study

3.4.1 Approach: BBCP - the Basic Block Comparison Platform

This experiment tested the proposed algorithm to normalize via the mnemonic/operand tokenization and substitution scheme above. The proposed algorithm was labeled the BBCP - the Basic Block Comparison Platform [91]. The concept for this algorithm was derived from the normalization scheme and BitShred’s sliding window hash.

BBCP operated in the follow stages:

1. *Dissect either the full executable section or basic clocks of a binary file into n-length n-grams.*
2. *Generate a finderprint for the subject utilizing these n-grams through a process of feature hashing.*
3. *Compare two n-gram feature hashes to determine shared feature inclusion and generate a Jaccard similarity index.*

The goal of this study was to determine if feature hashing as described in works [44][45] could be improved through normalization. Could BBCP identify simple malware variants, without relying on direct signature comparison? Could BBCP detect malicious functionality from within an executable? Results show that BBCP could be used to detect “bad” code within a “good” executable, as would be the case in trojan horse malware. Finally, we demonstrate that individual basic-block comparison allows for the flagging of suspicious executables simply for the inclusion of a basic block, or set of blocks, known to exist in other malware.
3.4.2 Implementation

3.4.2.1 Normalization

BBCP was founded on the reverse engineering tool IDA Pro. IDA Pro with IDAPython was used to disassemble the executable programs into the constituent blocks and/or identifying the location of executable sections from within the executable.

An experiment was conducted with the mnemonic/operand tokenization and substitution scheme above. The normalization process of replacing compilation specific substitutions with generic ones minimized the effects produced by some code variations, which are expected given work [65]. In this experiment, these transformations minimize the influence of both variable and function reordering. As a result, the approach is more effective at detecting malware variants, which vary these indirect references to evaluate signature detection. The process for substitution can be visualized in Figure 3.2.

The structure of this algorithm allows us to identify shared executable code in a variety of situations. First, full-executable-section comparison allows us to identify simple malware variants. This process will detect if two programs share mostly similar code without relying on direct signature comparison. Second, BBCP can also detect if malicious functionality is contained within an executable. This process can be used to detect “bad” code within a predominately “good” executable, as could be the case in trojan horse malware. Finally, individual basic-block comparison allows us to flag software based simply on its inclusion of a known, malicious block.

3.4.2.2 Approach

As BBCP uses IDA to access the disassembled instructions from a submitted sample, the BBCP approach can access both the executable section and each basic block within the malware sample. Using the section as input is referred to as shallow-dive
Chapter 3. Detection of Code Reuse in Compiled Software

dissection (SDD) and is the input used in works [44][45]. This approach allows for the
detection of variants, or variations of software coding, often used to evade detection.
SDD mode is also able to identify containment of the code of one software instance
inside another, such as the case of a trojan horse.

When BBCP uses basic blocks as input, we refer to this as deep-dive dissection
(DDD). Basic blocks are defined by IDA. This process has a number of uses, including
detection of known-malicious blocks as well as filtration of known-good blocks to
identify unknown code. We will address these topics in greater depth in Chapter 4.4.

3.4.2.3  $n$-grams

The first step in the BBCP process is that of “shredding” or creating $n$-grams. The
shredding terminology is derived from the BitShred project, which the comparison
method and sliding window in this algorithm used in this study. In this step, the
input is broken into $n$-length $n$-grams via a sliding-window. However, in the Bit-
Shred project, these windows were extracted from individual bytes across the full
executable section of the malware or software input. In contrast, BBCP uses disas-
semply, extracted using IDA through IDAPython, as input. As we have access to the
instructions, we can normalize our results to mitigate the effects of the transformative
nature of the compilation process.

The normalization presented in BBCP operates through the following five trans-
formations, which were also described above:

- *Registers replaced with “REG”*

- *Locations replaced with “LOC”*

- *Constant memory references replaced with “MEM”*

- *Constant values replaced with “CONST”*
Variable references replaced with “VAR”

As can be seen from the figure, each stage of the process is still representative of the input byte-code and structure is maintained; however, location specifics are abstracted to improve matching. The input byte-code is ultimately used to produce $n$-grams with a window of 4 instructions each.

### 3.4.2.4 Comparison

Large scale comparison of malware based on individual comparisons of actual $n$-grams is time consuming, as discussed above. As such, it is not practical to directly use $n$-grams in comparison. As discussed above, we turn to a representation of the direct $n$-gram through hashing. As in BitShred [45], we construct a Feature Hash by initializing a bit array with the bounds of the hash range. In this case, we bound the hash to $x = 2^{22}$ or $\approx 4.2M$, resulting in each bit array consuming approximately $0.5MB$ of space. The process is represented in Algorithm 1.
Data: Normalized n-gram Instructions

Result: Feature Hash Bit Array

initialization of zeroed bit array of length $2^x$;

while n-gram list do
  read current;
  hash n-gram;
  index = retrieve final x bits of hash;
  array[index] = True;
end

Algorithm 1: BBCP Feature Hash Construction.

As only a subset of the full hashing algorithm is used in the index, the algorithm is subject to false positives due to collision. However, in the course of this experiment, it was concluded that a Feature Hash of size $2^{22}$ yields a reasonable tradeoff in disk/memory usage and collisions creating false positives.

To find similarity of two candidate feature hashes, the arrays are used to calculate the estimated Jiccard Index of $J(F_a, F_b) \approx \frac{S(B_a \land B_b)}{S(B_a \lor B_b)}$ discussed in detail in Chapter 6. This estimated index is simply the percent of similarity between two items. In this initial study, we sought to discover if normalization and probabilistic comparison were viable for code detection.

3.5 Experimental Results

Compiling the same source with differing compilers results in differing binaries as discussed and shown above. To show that normalization through substitution is a viable approach to detecting code reuse in malware and that it preforms better than
full byte stream methods, we constructed experiments that altered byte-code and then applied our algorithm to detect similarity via Jaccard Distance. We compared these results with a reimplemented version of BitShred.

### 3.5.1 Same Source, Different Compilers

For this test, we used a cryptographic library, written in C++, as the input. We extracted a single hashing algorithm from the library and used it as the source for the experiment. Microsoft Visual Studio and MinGW were used to compile the source to binary.

As a secondary metric for comparison, we also captured the $n$-grams from BitShred and BBCP prior to hashing to the 22-bit feature array. Duplicate $n$-grams were uniqued and sorted. The native Linux “comm” command was then used to determine how many were common between the comparative sets. The following data and all other BBCP data in this paper was collected using a Feature Hash of size $2^{22}$ bits.

As can be seen in Table 3.4, BitShred indicated a 0% similarity. As this is a true, not approximated similarity, the results indicate that no $n$-grams were found in common between the two software inputs. The similarity, as measured via the feature hash, was 0.2%. This discrepancy is the error rate caused by collision within the 22-bit hashes. To avoid collisions, the bit array produced by hashing windows must be very sparse. BitShred produced approximately $30K$ $n$-grams for this sample input. BBCP creates approximately $10K$ $n$-grams due to the tokenization of instructions rather than bytes within the byte-code. Feature hashes from BitShred are three times more likely to collide that those of BBCP in this example. As a result, BBCP’s
Chapter 3. Detection of Code Reuse in Compiled Software

Jaccard index and its terminal command analysis value show only a 0.25% difference in similarity, as collisions are much more rare with BBCP than BitShred.

From these results, BBCP outperforms BitShred in detecting code compiled with varied compilers. However, this observation is trivial as the similarity measurement is extremely small and thus negligible; however, it is significant that BitShred, via a sliding window of byte values, produced no \( n \)-grams which were a match between the samples, while BBCP, via a sliding window of mnemonics and opcodes on instruction boundaries, did find some \( n \)-grams that were the same. We consider this preferable to the alternative. However, the use of differing compilers and optimization levels have limits in the real world as compiler variety is small. Thus this test is a form of worst case. Given that code is believed to reused millions of times in malware, the cumbersome variation produced by utilizing differing compilers and optimizations is not seen often.

3.5.2 Source Subset, Same Compilers

This experiment used the cryptographic hash library from the previous test. For this test, a class was extracted from the source and then compiled with Microsoft Visual Studio. The full source of the cryptographic hash library was also compiled and a similarity test was conducted using the two processes, BBCP and BitShred. As the entirety of the extracted algorithm is within the cryptographic hash library, it should measure approximately 100% in containment. That is, approximately 100% of the algorithm’s \( n \)-grams should be inside the set of the cryptographic hash library’s \( n \)-grams. We use “approximate” intentionally as we found that certain functions, duplicated in the source, were compiled with slight variances between the subset and the whole.

The results of the experiment can be seen in Table 3.5. BBCP reported a similarity score near 98% while BitShed reported approximately 55%. As the expected result
Chapter 3. Detection of Code Reuse in Compiled Software

Table 3.5: Source Subset, Same Compilers.

<table>
<thead>
<tr>
<th></th>
<th>BitShred</th>
<th>BBCP,SDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard Index</td>
<td>0.37848196</td>
<td>0.94929424</td>
</tr>
<tr>
<td>Jaccard Containment</td>
<td>0.55411762</td>
<td>0.97705984</td>
</tr>
</tbody>
</table>

Table 3.6: Source Embedding.

<table>
<thead>
<tr>
<th></th>
<th>SDD-Jaccard Non-Containment</th>
<th>DDD - # of Functions greater than 65% Similar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison of Program A to Program B - Original</td>
<td>0.133178681</td>
<td>0</td>
</tr>
<tr>
<td>Comparison of Program A to Program B With Netcat Containment Case</td>
<td>0.195416442</td>
<td>165</td>
</tr>
</tbody>
</table>

should be near 100%, it can be seen that the BBCP method is nearly twice as accurate than BitShred in this test.

3.5.3 Same Source Embedded into Dissimilar Programs

The experiment was designed to demonstrate BBCP’s capability to detect specific shared code between two binaries. Conversely, we also wish to test whether BBCP can detect dissimilarity when appropriate. For the test setup, we retrieved two dissimilar programs from MSDN’s sample code repository. We used Netcat source to embed the program into the differing sources. We then test for similarity to detect the code reuse. The results from the experiment can be seen in Table 3.6.

It is worth mentioning again that two compilations of identical sources will not necessarily compile to the same assembly if the surrounding code is changed. Choices by the internal compiler algorithms that achieve the same result, with differing instructions, do occur often. For instance, (xor eax, eax) and (mov eax, 0) both set the
eax register to 0 and are used interchangeably, even within the same compiler. While it would be ideal if normalization could remedy these changes, we found that simply setting a 65% threshold was all that we needed to identify the contained Netcat functions.

### 3.6 First Byte Normalization

The compilation process is lossy as well as transformative. While each compiler could be studied and the transformation documented; the number of compilers, versions, optimizations, and other switches makes this approach infeasible. Thus, the transformation process of compilers must be considered black box in practice with the possibility of documenting compilers that are very common; however, this would be a transitional work.

When using the raw instruction method in work [36], we found that many of the potential matches between functions were obscured by operand values, particularly in the case of memory addresses. This should be of no surprise based on works [65][42]; however, initial results without code normalization were more sparse than desired. An examination of sample assembly extracted from test code shows that fouling due to relative memory location changes is pronounced with even small modifications to code, such as variable insertion. However, such fouling is reduced significantly if destination addresses can be ignored. In designing a table for substitution as with BBCP, the question arose that a table used for substitution, like that used in BBCP, could be continually modified. Each modification would depreciate any feature hashes that were created using older tables. In examining this issue, we looked for other ways to normalize where a substitution table was not necessary.

Consider the byte-code from CRC32a above (relisted in Figure 3.3 for ease of reference). Recall that the byte-code is a function from an implementation of CRC32.
Chapter 3. Detection of Code Reuse in Compiled Software

The byte-code on top is as sourced from the project. The bottom byte-code within the figure is fouled with the addition of an extra variable. As seen in the previous examples, this fouling can occur with debugging code as well as malware writer alteration.

As this byte-code is sourced from a basic assembly block, the instructions can be decoded as they were previously in the BBCP work. As Intel\textsuperscript{TM}x86 machine code is a mnemonic followed by operands, the mnemonic is likely contained within the first byte of the instruction. This is not the case with all mnemonics and some instructions contain more than just a mnemonic within the first byte (such as push). This being understood, the concept was explored for the possibility of a normalization process, which did not depend on any substitution tables. We refer to this normalization procedure as the First Byte method. Conversely, the n-gram consisting of the full byte-code instructions are referred to as the Full Byte method and the substitution method as BBCP.

In the familiar CRC32 code snippet, with all but the first byte of each instruction removed as seen in Figure 3.4, the effect of examining using First Byte on the snippet is interesting. Our First Byte method is much more resistant to hash waterfalling than the Full Byte method. Figure 3.3 shows the effect of simple hash waterfalling with a single variable addition in the byte-code of CRC.c. That modification adds a variable to the stack, changing the relative offsets. Hash waterfalling has the effect of fouling

![Figure 3.3: Full Byte Waterfall Fouling, Window=5, 31% Match.](image-url)
Chapter 3. Detection of Code Reuse in Compiled Software

the feature hashing of each code segment. Figure 3.4 shows the effect of feature hashing using our First Byte method. Figure 3.3 shows the effect of waterfalling when using full instruction machine code. The fouling produced by variable insertion is much more pronounced in the Full Byte method verses our First Byte method as shown in these examples. In the Full Byte method, it is reduced to a 31% match with the insertion of a single variable, while in the First Byte it is reduced to 73%.

\[
\begin{array}{c}
55 \ 89 \ 83 \ C7 \ C7 \ EB \ 8B \ 03 \ 8A \ 0F \ 89 \ 8B \ 89 \ E8 \\
55 \ 89 \ 83 \ C7 \ C7 \ 0F \ EB \ 8B \ 03 \ 8A \ 0F \ 89 \ 8B \ 89 \ E8
\end{array}
\]

Figure 3.4: First Byte Waterfall Fouling, Window=5, 73% Match.

As this method proved interesting, simple, and lacked any externally maintained or versioned substitution table, we chose to move forward with experimentation on using First Byte as a normalization method to resist relative location and variable insertion.

3.6.1 Experimental Results

3.6.1.1 Equipment and Dataset

All experiments were performed on an OS X machine (Intel 2.3 GHz i7 / 4 Core, 8 Thread / 16GB memory) using only a single core unless otherwise noted.

We performed our experiments on a malware data set supplied by the Open Mal-ware Project [92]. Our test subset was generated by processing our dataset in order of MD5 hash and identifying the first \( n \) samples that tested positive for C/C++ signatures via the PEiD [93] database. This selection process ensures that the samples are random within the domain of C/C++ malware. The Open Malware Project dataset consists of more than 5M malware samples by MD5 hash. We limit our analysis to C/C++; however, works such as Juxapp [64] show that n-gram analysis is effective across other languages.
Chapter 3. Detection of Code Reuse in Compiled Software

As we already used IDA [27] to extract basic blocks from each program, we instead extracted only the First Byte of each decoded instruction from the basic block. This not only removes some operands from specific instructions, particularly control flow instructions with relative offset destinations, but also preserve the structure of the basic block. In addition, the method greatly reduce the number of hashes calculated in the $n$-gram analysis and does not exclude basic blocks that don’t match simply due to operand specifics, just as in BBCP.

3.6.1.2 Similarity

Sample runs of 10k basic blocks were conducted to determine the affect of window size and similarity threshold on resulting similarity matches. Similarity was measured between 30-100% in 10% increments for both First Byte and Full Byte methods. Figure 3.5 shows the results of the effect of similarity thresholds on the number of matches recorded when the window size is held constant at 10 bytes in both the First Byte and Full Byte methods. In the First Byte method, each byte represents a single instruction, while the Full Byte method varies due to the x86 variable opcode sizing. The Full Byte method does not intelligently account for instruction size; however, x86 is confined to no more than 15 bytes per instruction [21]. Therefore, the $n$-grams of size 10 may not cover an entire instruction. Thus, First Byte analysis produces more matches while covering more instructions. This, of course, comes at the cost of more possible collisions due to more than one instruction being represented by a single byte.

Window size was measured between 5-500 bytes. Figure 3.6 shows the effect of window size on the number of matches when the similarity threshold is varied. As expected, as $n$-gram size increases, the number of matches decreases. However, the rate of decrease is higher with Full Byte $n$-grams verses First Byte. Despite the number of instructions included in First Byte $n$-grams being much higher and a
reduction in the number of basic blocks analyzed, there is an increase in matching with First Byte compared to Full Byte. We believe this is due to the result of n-gram fouling (above) being more pronounced in the Full Byte analysis when compared with the First Byte.

Without optimization, the $O(n^2)$ nature of a many-to-many comparison is taxing. Our first run on unoptimized code, conducting all comparisons was 82 minutes for 25K basic blocks. However, by observing that basic blocks that differ drastically in size are not likely to be similar, we were able to reduce execution for 25K blocks to 4.5 minutes by bypassing blocks that were beyond a size threshold. In addition, filtering for known code (Chapter 4.4), multi-threading, and superset comparisons (Chapter 5), we can compare 25K blocks (3.8M instructions) in approximately 75 seconds even
Chapter 3. Detection of Code Reuse in Compiled Software

with this $O(n^2)$ approach.

3.6.2 Normalization Selection

The effects of normalization can be seen in both the statistical analysis of numbers of matches found and the resulting graphs. Normalization in our method reduces each variable length instruction to a single byte. It reduces the overall basic block length and the number of sliding windows to compute the feature hash. Constrained at 25k basic blocks, our method discovered 2-5x more similarities (at reasonable thresholds) than the Full Byte method (Figure 3.5). Examination of both First and Full Byte verified the actual similarity of basic blocks and known variants (families) were still detected in our method.

The First Byte method of normalization provides similar results to the BBCP method, without the complication of a defined substitution scheme. BBCP adds processing time by substituting mnemonics and operands so that missed similarity is reduced. This text based substitution is less desirable for a few reasons. First, the text based scheme is slower at generating signatures. Second, the scheme is subject to implementation issues as the hash is taken from this text. Changes in capitalization, spacing, syntax, and OS defined new lines, just to name a few, would result in a hash that has no meaning when compared to another hash generated from a scheme with only a single difference. Lastly, while it is easy to find examples where a text based substitution scheme is more accurate, First Byte has performed well in tests against wild malware sample sets (see Chapter 6). As such, the rest of the thesis is based on First Byte Normalization as a component of Malware Provenance.
3.7 Similarity Measures

While a number of similarity measurements exist, arguably the most accurate measure of similarity in the case of binary signatures is the byte distance between two byte strings. To convert edit distance from Levenshtein Distance (the distance between two byte strings of different lengths), we subtract string distance from one such that

\[ Sim = 1 - \frac{(Size_1 - Distance)}{(Size_2)} \]

This measure is an accurate representation of similarity at near duplicate values, but deteriorates as the input size increases and there is a corresponding decrease in similarity. This fact is easily understood by comparing to random 257 byte strings for byte similarly. Such strings cannot be 0% similar as at least one byte must match another byte, resulting a distance of less than 257 bytes. Most probably, many such bytes match in such a string. However, the measure is the least complex direct measurement of similarity without estimation and is used for comparison.

While edit distance is arguably the most direct measurement of distance between two signatures, it is not possible to efficiently compute. Firstly, Levenshtein Distance can only be reduced to \( O(n + d^2) \), where \( n \) is the length of the longer of the two signatures being compared and \( d \) is the final distance being computed. Additionally, this type of comparison is pairwise, leading to complexity in each comparison and mass comparisons. While not optimal, the edit distance measurement between two First Byte normalized blocks is a standard to which we can compare.

3.7.1 Similarity Comparison Experiment

The goal of this measurement experiment is to determine the accuracy of Feature Hash based Jiccard Similarity estimates to a non-estimated distance with respect to First Byte malware comparison. While Jiccard similarity was useful in BBCP and First Byte, as compared to other techniques, the similarity measurement itself was
Chapter 3. Detection of Code Reuse in Compiled Software

not tested. To measure accuracy of the Jiccard estimation method, as presented in work [45], we tested this similarity measurement method with First Byte normalized blocks against other measurement techniques. As Jiccard similarity, as proposed, also depended on $n$-grams, we also tested the length of the hash window against the edit distance, which does not partition to $n$-grams.

### 3.7.1.1 Window Size for $n$-gram Based Measurements

Our first experiment was to measure the effect of window size in feature hashing First Byte input. An ideal mapping would be an $x = y$ mapping, except that our chosen distance measure is not linear in sampling and is bias to over estimate similarity in the lower ranges. Thus the ideal mappings are not straight lines, but very slight curves. We tested window sizes of $n = [3, 5, 10]$ to measure the accuracy with respect to the edit distance of each First Byte input. Each window represents the structure of exactly $n$ instructions due to the First Byte input. The data is generated from the same dataset in the above real world data.

The 3-instruction windows shown in Figure 3.7 suggest that the measurement is consistent with the edit distance with an acceptable variance. However, in similarity tests with the dataset above, 3-instruction windows produce a large amount of noise in the corresponding graph. Perhaps more importantly, the false positive rate that creates the noise also effects filtering, leading to steep false negative rate in similarity matching.

The 5-instruction window shown in Figure 3.8 also produces a measurement that is consistent with the edit distance with an acceptable variance.

A window size of 10 produces a Feature Hash that begins to introduce more variance from the true edit distance. This is due to small variances having large impacts as they are counted 10 times through the window. A window size of 10; however, produces the least noise in graphing similarity.
Chapter 3. Detection of Code Reuse in Compiled Software

Figure 3.7: Feature Window 3 vs Edit Distance.

Figure 3.8: Feature Window 5 vs Edit Distance.

Figure 3.9: Feature Window 10 vs Edit Distance.

Figure 3.10: Fuzzy Hash vs Edit Distance.
3.7.1.2 Other Similarity Measurements

Fuzzy hashing was tested for accuracy as well. The common implementation of CTPH [40] is ssdeep from the same author. While experimentation could prove to reveal better triggers for stop words, CTPH, as implemented in ssdeep, is very poor at determining similarity as seen in Figure 3.10. The CTPH tool ssdeep misses similarity candidates near 90% by edit distance. The measure has little correlation to the $x = y$ linear, optimal grouping.

This similarity comparison of actual edit distance with the selected First Byte normalization shows that a Jaccard similarity with Feature Hash storage is a viable comparative method for small datasets. However, as discussed in Chapter 5, the exponential nature of pairwise comparisons have real world limits with malware samples measured in 100’s of thousands.

3.8 Code Diversity in Compiled Software

The ability to identify code reuse by $n$-gram features is only useful if those features are fairly unique to software implementations. Before proceeding down the path of conducting full scale similarity of First Byte block signatures, a study was conducted to determine if Bloom Filters filled with hashes of First Byte windows was a valid approach to detect similarity. At the onset, questions like: ”How diverse are software blocks in a set of malware?” and ”What is the optimum window size for the sliding hash scheme?”, need to be answered before moving forward with more complex experiments.

3.8.1 Block Diversity and Feasibility Study

This experiment was not the first attempt to explore the use of statistical methods to identify code reuse in malware [44][45][36][94]. While works [44][45][36] use a sim-
ilar approach to identifying malware variants, the application of a similar method on normalized blocks left the above questions unanswered. In these works, the similarity was calculated in three phases. The first phases was to extract the executable text section of the malware sample. The second phase was to “shred” the program into \( n \) sized samples, again, referred to as \( n \)-grams. This method of identification has been shown to reliably compare malware samples, as a whole, to sum similarity through code reuse. However, this method does not identify explicit code reuse, only that programs likely share some byte-code. Such methods are useful for identifying potential variants within families; however, the methods would likely miss binaries that share only a few functions.

The intended purpose of this study was to test the feasibility of using windowed hashing of the first byte of each instruction to determine if similarity exists. The general method used in BitShred [44][45][36] was followed, except that each block from the executable text section was extracted and used as input rather than the executable text section as a whole. In addition, the simple normalization of only using the first byte of each instruction was employed to increase similarity scoring when minor changes in code are introduced. Each basic block signature was calculated in the same manner as BitShred identified entire programs, that is, a sliding window hash was calculated across the entire block. The hash used in the experiment was MD5 to reduce the possibility of error introduced by random collision to very near zero.

The malware used in the study was obtained from the OpenMalware Project [92]. We used a portion of the malware within the set to conduct the experiment. The first 100K samples were analyzed with PEiD [93] signatures to determine if the sample was packed, and if not, what compiler was used to create the sample. Using this method, we were able to select unpacked malware samples created using the C/C++ language. The result was the identification of 4414 Windows PE executables for the
3.8.2 Experimental Results

The test environment was a 2 x 2.66 Dual Core Xeon Mac Pro with 6 GB of ECC RAM running OS X 10.6.8. IDAPro Starter 6.2 for OS X was used as the recursive disassembler, IDAPython 1.5.3_final and IDA SDK 6.2.0 were used as the automation platform to extract the functions. The Bloom filter used in the experiment was the python-bloomfilter 1.1 package [95].

The C/C++ samples were processed to extract basic blocks from the executable section. The blocks were identified by first processing the files with the recursive disassembler and using the functionality of IDAPython to identify the head and all of the returns within the function. Each instruction was identified and the first byte of those instructions was extracted and put to an external file. The files were associated with the executable from which they originated.

Each extracted block’s information was compared for exact duplication via MD5. All exact duplicates were uniqued so that each function processed would represent some variation in code. The similarity check comprised of feeding a portion of the executable block byte-code to a Bloom filter as a binary key. Each block was “shredded” into $n$ number of keys of size $w$ via a sliding window. The Bloom filter was expanded as necessary to ensure error rates did not exceed 0.001%.

The Bloom filter was trained by extracting windows of size $w$ from the extracted blocks and adding them to the Bloom filter as keys. Any similarities from these training inputs were not recorded. Similarity matching was accomplished by extracting windows of the same size $w$ as the training input and checking for membership in the Bloom filter. These windows were not added to the filter. For each extracted function, the total number of window matches and mismatches was recorded. Percentage of membership for each block was calculated via post processing.
Chapter 3. Detection of Code Reuse in Compiled Software

The extracted samples used for the experiment totaled 560. Those 560 malware samples were processed and each First Byte representation of each samples component blocks were again put to disk. The processing took a total of approximately 120 minutes, or 12.8 seconds per program on average. The 560 malware programs produced 166,221 functions, including very small callers and tables. Of those $\approx 166K$ blocks, 85,459 were found to be unique via MD5. It is worth noting here that First Byte extraction itself resulted in a 50% reduction in input processing even in this early stage of experimentation. This fact and the use of a single large bloom filter (later feature filter) would later be used to reduce input in similarity matching.

The Bloom filter was created using the first 1M windows extracted from unique blocks and remained constant throughout the experiment. The first windows size used was the 5 byte value used in BitShred. Recall that BitShred uses the Full Byte, un-normalized byte-code, as input. Thus this window size represents 2-5 instructions in x86’s variable length instruction set. Using the First Byte method, this window represents a static 5-instruction window. This window size returned identification of similarity in 69% of submissions, while a window size of 10 resulted in a 33% hit rate. Window sizes of 5, 10, 20, 40, and 60 instructions were used in the experiment.

Observations from the experiment formed the basis for later experiments. As window sizes increased, smaller blocks began to be eliminated from processing as the window size becomes larger than the block itself. The exclusions were taken into account in choosing the window size for the followed experiment. The windows size of 10-instructions (bytes in the First Byte extraction method) was selected for the second half of this particular experiment for its balance of lower match probability and near 0% exclusion rate due to hashing windows exceeding instruction count. The results can be found in Figure 3.11.

Following the selection of the window size, a list of 26,329 unique functions extracted from the malware was again input into the Bloom based comparison engine.
Again, the first 1 million windows were used to train the filter. During this pass, the number of matches and total windows were tracked for each function and comparisons were stopped after 1 million windows. A total of 12,884 functions were used to train the filter and 12,846 were used for comparison. The results of the experiment are in Figure 3.12.

A second run with window size $w = 20$ was also conducted (as seen in Figure 3.13). The results were similarly distributed as $w = 10$ with the exception of the number of functions excluded due to size and the number of functions that did not match. This confirms that $w = 10$ is the better choice as it provides more granularity without increasing the probability of a false match as a majority of the matches appear to come from functions with size 12 - 22 ($w = 10$ and $w = 20$ respectively). As for scalability, a first pass run across the 26K function takes less than 6 minutes. This equates to approximately 72 functions a second. A second run in a short time span has much improved efficiency at 45 - 80 seconds (approximately 433 functions a second). This is likely due to OS X file caching. This suggests that disk I/O accounts for as much as 6 times the processing cost. There are several methods that are beyond the
Chapter 3. Detection of Code Reuse in Compiled Software

<table>
<thead>
<tr>
<th>Window = 10B</th>
<th>Window = 20B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matching Funct Count</td>
<td>Matching Funct Count</td>
</tr>
<tr>
<td>100%</td>
<td>251</td>
</tr>
<tr>
<td>90 - 100%</td>
<td>330</td>
</tr>
<tr>
<td>80 - 90%</td>
<td>315</td>
</tr>
<tr>
<td>70 - 80%</td>
<td>304</td>
</tr>
<tr>
<td>60 - 70%</td>
<td>298</td>
</tr>
<tr>
<td>50 - 60%</td>
<td>345</td>
</tr>
<tr>
<td>40 - 50%</td>
<td>328</td>
</tr>
<tr>
<td>30 - 40%</td>
<td>433</td>
</tr>
<tr>
<td>20 - 30%</td>
<td>627</td>
</tr>
<tr>
<td>10 - 20%</td>
<td>891</td>
</tr>
<tr>
<td>&gt; 10%</td>
<td>6448</td>
</tr>
<tr>
<td>Window &gt; size</td>
<td>2276</td>
</tr>
</tbody>
</table>

Figure 3.12: 10 Instruction Window.  Figure 3.13: 20 Instruction Window.

Scope of this thesis that could reduce this overhead to improve efficiencies to account for billions of functions found in the wild. The number of non-matching (> 10%) is clearly outside of the matching (<10%) distribution. This indicates that there is value in the testing method for identifying code similarity. This is not surprising given the success of BitShred when using a (less likely to match) Full Byte input.

The results of this code diversity study demonstrated that finding code reuse in diverse projects was possible. The simple pairwise matching was able to handle small input loads to find code blocks that were similar via the First Byte method of normalization.
Chapter 4

Visualization of Results

4.1 Similarity via First Byte Feature Hash

The results of the initial similarity study, that involved using the general method described in BitShred but substituting First Byte representations of Basic Blocks for Full Byte code extractions of the entire executable text section, were encouraging. As a result, code to conduct similarity calculations for these First Byte signatures to produce an all-pairs similarity output was completed. In addition, code to conduct similarity calculations for Full Byte representations of extracted blocks was also produced.

The first attempt to conduct all-pairs similarity between basic blocks was coded in Python. The general approach was to read in a listing of previously extracted blocks (First or Full Byte) until a block limit was reached. Calculate the similarity for all-pairs and insert malware as an empty node, blocks as content nodes, malware/block relationships as parent/child edges with weight 2, and block/block similarity as edges with a calculated similarity weight. The similarity, calculated by the estimated Jaccard Index of \( J(F_a, F_b) \approx \frac{S(B_a \land B_b)}{S(B_a \lor B_b)} \), can only be a floating point value between 0 and 1; therefore, the later clustering of the graph would tend to focus on malware as sets
of blocks, rather than a more evenly distributed field of random blocks.

### 4.1.1 Notes on the Gold Standard

During our initial studies, we obtained a Gold Standard dataset after more than two years of working with less desirable datasets derived from AV signatures and user submissions of malware. The set consists of 92 samples that have been fully analyzed by professional malware analysts. The sample set contains 9 groups of variants and 3 outliers.

As noted in BitShred [36] and FRASH[73], up until this point, there has been no Gold Standard available for algorithm comparisons to detect code similarity. As a result, most papers use improvised methods to estimate recall and precision or do not report on the robustness of the algorithm in terms of recall and precision. In many of our initial studies, we were also limited in this regard; however, the Gold Standard data was used to rerun the First Byte data below as it does not require recoding of the implemented algorithms.

### 4.2 Force Based Clustering

The statistical study of code diversity and code similarity along with success reported in other projects seeking to detect malware variants led to the calculation of similarity of hundreds of malware samples. The initial results were daunting as similarity detection in this small sample size was frequently many tens of thousands. It became clear that simply calculating similarity to detect code reuse and reporting results would produce data in sizes difficult to consume.
Chapter 4. Visualization of Results

**Data:** Input File Listing

**Result:** Gephi Network Graph

initialization;

while Input File Listing AND NOT reached block limit do
  read current;
  get hash windows;
  make Bloom from hash windows;
  insert Bloom Entry;
  insert block into graph as node;
  if block has new parent then
    insert parent into graph as node;
  end
  insert block/parent into graph as edge(weight 2);
end

while Bloom Entries do
  read currentOuter;
  while Bloom Entries + currentOuter as index do
    read currentInner;
    similarity = Intersection(CurrentOuter, CurrentInner) /
    Union(CurrentOuter, CurrentInner);
    insert similarity into graph as edge(weight similarity);
  end
end

**Algorithm 2:** Initial all-pairs similarity.

We desire to determine not just if a program belongs to a family, but identify what specific code in the malware was related to other specific code. To achieve
Chapter 4. Visualization of Results

Figure 4.1: 92 malware samples, clustered, no filter.

This goal, all relationships must be measured and relevant relationships recorded. The preprocessed and normalized basic blocks are hashed via sliding window as in works [45][36]. A graph is constructed with malware samples as parents and the sample corresponding basic blocks as children. Each basic block is compared to every other basic block and a similarity is calculated by dividing Feature Hash matches by possible matches. If the similarity score exceeds a prescribed threshold, an edge is created between the corresponding basic blocks. The edge weight is the similarity score.

Clustering is accomplished via Gephi [96] with the imported algorithm “openord” from Sandia [97]. The clustering is force-based rather than flat or approximate nearest neighbor [36][98][99]. This approach retains similarity connections between malware samples that are generally dissimilar as well as those that are variants from a malware
Chapter 4. Visualization of Results

family.

In Chapter 6 we will introduce a Gold Standard Comparative Set of malware. The set consists of 92 known malware samples in 8 groups with some outliers. This set is used in the graphing experiments.

The relationships plotted using the described technique and the Gold Standard Set produces a graph (Figure 4.1) that visually contributes to understanding the complex nature of the data. Not only are variants revealed, but each relationship is shown in the graph. The number of relationships can, however, be large in a relatively small number of malware samples. To reduce the number of relationships, we use filtering as described in Chapter 4.4.

### 4.3 Other Similarity Techniques Visualized

In contrast to using Malware Provenance as a method to identify malware relationships, the comparison of clustered Malware Provenance results and clustered fuzzy hashing visualizes the inaccuracy of fuzzy hashing for malware binaries. First, examine the comparison of a force base clustering on a fuzzy hash of the entire binary, as is common in industry. In the graph in Figure 4.2, the 92 input samples are clustered according to fuzzy hashing at 40% similarity as in other graphs. There are 29 samples that are not found to be similar in the set. In addition, groups 3, 4, and 6 are not only clustered into separate groupings, but are heavily cross clustered.

This visualization conveys what has already been discussed. The file based similarity used in CTPH (fuzzy hashing) is not optimal for comparing software. The lack of normalization is a likely culprit with the entire software file used as input. Additionally, CTPH does have restrictive minimums for input, making a block based comparison not viable. Chapter 6 discusses CTPH as well as other tools used to identify malware variants in comparison with a version of Malware Provenance used
Chapter 4. Visualization of Results

Figure 4.2: 92 malware samples, Fuzzy Hash Cluster.

to calculate similarity between entire software instances for the purposes of variant detection.

4.4 Removal of Trivial Relationships

The software development process often incorporates code reuse to save development time as well as promote the use of tested code. This process is widely adopted in the software industry. The use of libraries to accomplish common tasks is found in all software, including malware, though some malware attempts to hide this fact to prevent analysis. This fact has been exploited by security researchers to identify malware by the order of these function calls [46]. However, in trying to identify code
Chapter 4. Visualization of Results

reuse as a method to gain information on the origin of malware, such as Malware Provenance, the identification of libraries and other widely available code is not very useful.

The removal of trivial relationships is straightforward within a system such as Malware Provenance. The extraction of blocks as the basis for comparison with other malware blocks also allows for the comparison against other datasets. In this case, we can compare against a set of feature hashes gathered from sources we believe to be unimportant. The inclusion of static or otherwise inlined code from public sources such as libraries and compilers is not interesting, so we set out to remove it from comparison. As such, we began to explore filtering byte-code from malware to malware comparisons by first comparing it to known good code.

4.4.1 Filtering

Filtering was first accomplished by examining known goodware, in this case executable software found within Windows system32, in a similar method to the examination of malware. The goodware underwent the same processing as the sample malware. We first extracted the basic blocks from the executable text section; however, we did not list each basic block for comparison. We instead stored all feature hashes within a large bit field, creating a filter to test for set membership. Each basic block extracted from malware was then compared to the feature hashes of the filter. If the basic block is found to be a member of the filter, it is included in the graph for completeness. However, it is not compared to all basic blocks. Thus, this linear method reduces the number of similarity checks conducted given a malware set and reduces the complexity of the associated graph.

The development of an accurate filter has two goals. First is to reduce the number of trivial relationships and the second is to reduce the number of comparisons. This is last point is particularly important in the $O(n^2)$ complexity when using Feature
Chapter 4. Visualization of Results

Hashing.

Data: Input File Listing
Result: Gephi Network Graph

while Blocks to be filtered do
  get hash windows;
  build filter;
end

while Input File Listing AND NOT reached block limit do
  read current;
  get hash windows;
  if block index to filter IS LESS THAN threshold then
    make Feature Hash windows;
    insert feature Entry in feature matrix;
    insert block into graph as node;
    if block has new parent then
      insert parent into graph as node;
    end
  end
  insert block/parent into graph as edge(weight 2);
end

while Feature Entries do
  read currentOuter;
  while Feature Entries + currentOuter as index do
    read currentInner;
    similarity = Intersection(CurrentOuter, CurrentInner) / Union(CurrentOuter, CurrentInner);
    insert similarity into graph as edge(weight similarity);
  end
end

Algorithm 3: Filtered All-Pairs Similarity.

Filtering is accomplished in much the same way as work [36] as specific identity need not be retained, only set membership. In this way, a single large bit array was
Chapter 4. Visualization of Results

used to filter out trivial relationships. The filtering process was accomplished with linear computation time, reducing the $O(n^2)$ load of the similarity comparisons after filtering. Filters are Feature Hash based and constructed with the same algorithm used in similarity, except that the filter does not track individual relationships, only membership. In this way, the filter is built in linear time and the comparisons are done in linear time. If membership is determined to be true, that basic block is not processed within the similarity engine.

![Figure 4.3: Affect of Filtering on Similarities.](image)

### 4.4.2 Filtering Affect on Graph Complexity

Filter construction is based on known good-ware. To demonstrate the usefulness of filters, 100 random (non-obfuscated) binary executables were selected from our known clean examination system. A filter was generated from both the First and Full Byte basic block extraction methods over the same 10-byte window used in the base similarity experiment. The similarity model was run again with both the First Byte and the Full Byte methods, this time filtering out any basic blocks found in the filter set. A comparison of the reduction can be found in Figure 4.3. Any filtered basic block is added to the graph for completeness; however, such basic blocks are not compared to other blocks for similarity. As such, the 25k block constraint includes
filtered basic blocks. Examination of the unfiltered graph in Figure 4.4 and the filtered graph in Figure 4.5 shows a dramatic reduction in complexity, while retaining relevant similarity from malware basic block n-gram analysis.

The ability to filter known code from comparison of malware samples was a key discovery in the malware provenance project. The effect of the process to detect malware family variants is discussed in Chapter 6. The effect of filtering with block level code reuse detection is discussed in detail in Chapter 7.

4.5 Graph Complexity in Large Sets

Graphical clustering of this group after removal of trivial relationships produces the graph in Figure 4.6. Consider that the graph is constructed of relatively few malware samples (92) in a domain with an estimated 300M-1.5B samples in the wild. Also, the malware samples used in the graph were previously selected specifically because they belonged to a smaller subset of known families; that is, 8 families with 3 outliers. When producing graphs with thousands of samples, the clustering becomes much less clear.

The graph in Figure 4.7 contains the same data as the previous graph (colored green) with the addition of $\approx 550K$ additional blocks from $\approx 2300$ random malware samples (colored grey/red). The graph contains $1.38M$ detected similarities which are represented by edges between graph nodes. The graph is very complex, and while useful for finding common malware families (large clusters in the graph), it is unlikely to be useful for detailed examination of specific samples.

When seeking detailed examination of specific samples, it is not useful to include similarity information not relevant to the specific sample. We call this Focused Similarity Graphing, wherein a specific sample is the focus of all similarity comparisons, and only found similarity related to the focus is included in the graph. Consider
Figure 4.4: 25k Blocks, Unfiltered, 50172 similarities found.
Figure 4.5: 25k Blocks, Filtered, 9828 similarities found.
Figure 4.6: 92 malware samples, Malware Provenance Clustering with Filtering.

the grouped graph of Gold Standard data when only focusing on a single member of group 5. In the graph in Figure 4.8, the focus is magenta and the similarity is blue/green. While the search space included all the blocks from the first graph, only similarity linked to the focus was calculated and graphed. The process is iterative in that malware that is found to be similar is again submitted for similarity. This graph was calculated with 5 iterations, but all similarity in the original field of 92 samples was found within the second iteration.

While graph simplification was unnecessary because it was already informative, the same process dramatically simplifies the graph by focusing only on relevant information. The graph was calculated over a single iteration in the same field of $\approx 2300$ with $\approx 550K$ blocks found in the complex graph in Figure 4.7. This graph was read-
Figure 4.7: 2.3K malware samples, Malware Provenance Clustering with Filtering.

able and informative. It immediately identified a third malware sample related to the group.

Increasing the iterations used to construct the graph again presents complexity within this working example. The addition of the third malware sample (colored in red/grey in Figure 4.9) increases the graph complexity with two iterations to ≈ 84K nodes with ≈ 86K connections as seen in Figure 4.10. Nearly all of the connections are single similarities between a very few blocks. Examination of the disassembled blocks revealed that some of these similarities are non-trivial and represent functional code that was not identified as libraries by IDA Pro. The identification of wide-spread code reuse within the malware sample set, while increasing graph complexity, suggests that the project goal of finding optimal signature sets for malware is obtainable.
As shown by the similarity graphs constructed from malware samples in this study, when applied to the massive datasets that would be required for real-world applications, the graphs become too complex to be useful for the examination of specific samples. Therefore, various means of reducing the graphical complexity by excluding irrelevant similarity information were considered. In particular, this study found using focused similarity graphing to be an effective means of filtering the data so that only found similarity linked to the specific data of interest was calculated and graphed.
Chapter 4. Visualization of Results

Figure 4.9: Focused Graph, 2.3K Samples.
Figure 4.10: Focused Graph, 3 iterations, 500K Samples.
Chapter 5

Exponential Growth in All-Pairs Comparison

The comparisons needed to calculate each Jiccard index are pair-wise. It is the nature of pair-wise comparisons to be $O(n^2)$ when calculating all-pairs similarity, as discussed so far in this thesis.

5.1 MinHash

The inherent limitations of using a pure feature set as a comparative signature, such as what is found in works [69][44][45][36][100], are rooted in its varied length and its linear ordering. This can be easily demonstrated in the comparison of two sentences: “Jill walked the dog.” and “Jill and I walked the dog.” If each word is a feature, the similarity of the two sentences is measure as 66% as four words in each sentence out of a possible six ($4/6 = 66\%$) match. However, notice that measuring the similarity requires comparing every word in the first sentence to every word in the second. In addition, if we add the sentence “I walked the dog with Jill”, every word in this sentence must be compared to every other word, in every other sentence. This is the nature of the similarity comparison issue. Feature hashing allows for a linear
one-to-one comparison by constructing a bit array indexed with the hashes; however, the method is still pair-wise for each comparison.

However, consider that given strict ordering, comparison can be conducted in much shorter time. Such is the case with the last name/first name ordering in a phone book. By alphabetizing the last name followed by the first name, a search can be conducted much faster than comparing every name. If there were $n$ names within the phone book, each name could be compared to every other name by dividing the list of names in half and assessing if the subject of the comparison was less, more, or equal to the name found at the dividing point. Thus, comparison of each name can be conducted in $O(\log_2 n)$ time. In an all-pairs comparison, this would be $nO(\log_2 n)$ time. However, in the case of the sentences above, there is no such ordering as each word has no weight over another (in contrast with last name and first name).

Ordering could be accomplished by permuting the sentence to every possible order and storing each permutation in its own table. This method was first described in work [101]. The sentence “Jill walked the dog” would be stored as “Jill walked the dog”, “Jill walked dog the”, “Jill dog the walked”, “Jill dog walked the”, etc. each in its own table. The number of tables required is extreme if the sentence is large, as the number of tables required is $n!$, where $n$ is the number of words in a sentence. Notice that not only is permutation extremely costly in terms of space, it only conveniently works with a fixed length sentence.

Notice these issues could be solved if we could predictively permute each sentence $K$ times, where in $K$ searches, two sentences are likely to match at least once if they are similar and we could set the length of features in a sentence to $L$, without bias to features that begin with any arbitrary letter. If these conditions were met, each sentence could be “alphabetized” (ordered) in $K$ tables, thus reducing search to $NK O(\log_2 n)$ time to find a candidate match. If false positive matches were rare (as in the case of features being 7 words in a row, as opposed to a single word), overall
time reduces to \( n K O_{\text{search}}(\log_2 n) + O_{\text{sim}}(1) \), which reduces to \( O(n \log_2 n) \).

MinHash, or minimum hash, introduced in work [102][103], solves the problems described above. The approach is to use the random dispersal of a hash digest (the dispersal of a hash digest is pseudo random as compared to input) to solve both the variable length and the extreme number of tables needed for true permutation. First, each feature is hashed with \( K \) different hash functions, producing \( K \) signatures for each feature. As these signatures are dispersed randomly, but generated deterministically (each feature always produces the same hash for hash function \((H_K)\)), then selecting the minimum hash digest value for each hash function over all features represents both an unbiased feature input length of \( L \) and a probabilistic permutation of \( K \).

\[5.1.1 \text{ **MinHash as a Representation of a Feature Hash**}\]

As MinHash is directly translatable to an LSH sortable signature, the signature is appealing, though it is not the only LSH method available [104]. Both Feature Hashing and MinHashing take as input a sliding window hash of the First Byte block signature (as described in work [100]). Where Feature Hashing uses a single hash for each window, MinHashing uses \( K \) different hashes for each window (\( K = 100 \) is used in experimentation unless otherwise noted). Notice that only the random distribution, not the cryptographic nature, of the hash is needed to achieve the requirements of MinHash. Thus it is possible to hash every window with a single hash algorithm and use other means to distribute the minimum values randomly. For instance, a list of randomly distributed numbers could be used to \( XOR \) the hash output to redistribute the minimums. Another method could use a large hash output with a much smaller sliding mask to distribute the minimums randomly.

A Feature Hash contains a full accounting of a sliding window \( n \)-gram hash of a given input. As such, a MinHash can also be constructed from a Feature Hash, where the calculation of similarity between \( \text{block}_a \) and \( \text{block}_b \) for both methods is
duplicative. Thus, a MinHash can be viewed as a probabilistic representation of a Feature Hash, even if constructed directly from a sliding window. It can be shown that the probability of error in the probabilistic representation of a Feature Hash within a MinHash is a standard Chernoff bound [105], in that expected error rate is \( \frac{1}{\sqrt{K}} \), where \( K \) is the number of hashes per \( n \)-gram. In the case of \( K = 100 \), the expected error rate is \( \pm 10\% \).

5.1.2 Complexity Time Analysis

Each \( k \) MinHash taken from an input \( n \) need only be compared with other \( k \) MinHashes. As there is no meaning in cross comparing individual minimum hashes, each element need only be compared with elements with similar indexes. Also, notice that each element \( k' \) could be stored in a sorted table. A look up for the existence of an identical value would be \( O(\log_2 n) \). With a MinHash of \( K \) elements, lookups would be \( O(K \log_2 n) \); therefore, MinHash can be indexed within \( K \) tables and searched for all-pairs in \( n K O(\log_2 n) \) times.

The reduction in complexity makes MinHash much more attractive over Feature Hashing in \( O(n^2) \) time. Further reductions in time can be realized in MinHash as similarity calculations of two signatures is \( O(K) \) verses \( O(MN) \), where \( M \) and \( N \) are the number of features hashes in extracted blocks. Locality Sensitive Hashing (LSH) of the MinHash signatures can reduce the search time even further with an increase in error. Consider that every pair of minimum window hashes represented two Feature Hash windows in pseudo random locations. In choosing \( K' \) pairs, it can be shown that this is \( K' \) random permutations of hash signatures belonging to the full permuted set. Therefore, any match in \( K' \) represented a match in the full permuted pairing of signatures. This can be accomplished without an extremely costly full permutation and results in sub \( O(n \log_2 n) \) time as seen in Theorem 1.

**Theorem 1** Let \( m \in M \) where \( M \) is a Min-Wise determined hash array of length \( K \).
Chapter 5. Exponential Growth in All-Pairs Comparison

As $m \in \mathbb{Z}$ all $m$ can be stored as an ordered set $H$.

Searching $m$ in $H$ is a binary search as

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

where $n$ is the size of $H$

which is $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

where $a = 1, b = 2$ which is a form of the Master Theorem

$T(n) = \Theta(n^{\log_a b}(\log n))$

$T(n) = \Theta(\log_2 n)$

$T(n) = O(\log_2 n)$

Then for $M$ of length $K$

$T(n) = O(K(\log_2 n))$

$T(n) = O(\log_2 n)$

Then for all-pairs similarity, the search must be conducted $n$ times

$T(n) = nO(\log_2 n)$

$T(n) = O(n(\log_2 n))$

$T(n) = O(n \log_2 n)$

5.2 Expermentation with Minhash

All experiments were performed on a Linux64 machine (Intel 3.5 GHz 4770K / 32GB memory) unless otherwise noted.

5.2.1 Initial Feasibilty of MinHash: Variance

In our tests, we are interested to retaining First Byte Feature Hash accuracy in a sort compatible signature. As such, the first tests examine the accuracy of MinHash vs. Feature Hash and the effect of MinHash signature size on estimating similarity in our
test case. As our MinHash signature is also generated with a sliding window hash, the ideal mapping is in this case an unmodified $x = y$.

The method to the experiment was to conduct similarity on 1140 pairs of First Byte extracted, binary-code blocks. Then using those same pairs, measure the similarity via MinHash. Variance is the distance from this projected line. The first MinHash experiment was based on $K = 100$ and the second was $K = 1000$. In Figure 5.1 the Feature Hash vs. a MinHash of 100 signatures shows slight variance. The variance in a 1K MinHash is reduced in Figure 5.2, but only slightly.

\subsection{5.2.2 Initial Feasibility of MinHash to Increase Speed of Comparison}

The process of conducting a MinHash indexed search is to compute the MinHash of each block (filtering is also accomplished at the step); construct the MinHash and any LSH indices; query the system for candidates; conduct a similarity comparison for each candidate; and harvest the results. While the process is more complicated than the (compute, compare, harvest) of the Feature Hash method, in practice, it is much faster. In Feature Hash experiments, it was projected to consume approximately 800 node hours for the 1M Feature Hash signatures, with result time exponentially
increasing as signatures increase. Projecting the Feature Hash test results, it is estimated to consume $80M$ node hours to conduct an all-pair similarity test on a field of $250M$, even if optimizations in the search are able to dramatically decrease candidate comparisons as seen in Figure 5.3.

By contrast, MinHash is slightly sub-linear in practice (due to duplicate data being eliminated in $O(n\log_2 n)$ time during index construction. In this initial experiment, the single machine used during the experiment could compute (not projected compute) $1M$ comparisons in 316 seconds ($9K \times$ performance increase at $n = 1M$). To compare projected times from previous results, $250M$ basic blocks compared for all-pairs similarity is estimated at 22 node hours ($4M \times$ performance increase at $n = 250M$). This contrast can be seen in Figure 5.4.

5.2.2.1 MinHash Error Rate

The use of MinHash and more so, LSH, as a signature indexing scheme introduces a measurable error rate. Error rate can be in the form of a false positive (a candidate that is not within the similarity threshold) or a false negative (a real similarity that is missed in the candidate results). The false positive can be effectively eliminated
through additional compute time by calculating the real similarity. However, false negatives cannot be caught in a similar fashion. False negatives and false positives are inversely proportional; therefore, the choice to limit false negatives at the cost of increasing false positives is included in the testing. Even so, a false error rate of 4% was measured in the testing with $K = 100$, though most was near the threshold level. As such, to further reduce the false negative rate, a lower threshold could be used for candidate searches and the target threshold could be calculated on actual data if desired.

### 5.3 Super Signatures

Let us recall traditional hashing in forensics once more. The use of cryptographic hashing in computer forensics and malware identification is widespread and flawed. The traditional use of this type of hashing was to ensure that a file has NOT been modified. However, the availability of software tools and methods was a convenient, even if not optimal, solution of the need of law enforcement to quickly identify files of interest (or not of interest). As computer forensics was born out of law enforcement
and forensic accounting partitioners, they spent resources to learn how these existing technologies worked with the goal of presenting evidence in court. The need to understand how technologies worked was related to the requirements related to presenting testimony in a legal setting as an expert (Daubert Hearing in the US). Though computer evidence was first presented in 1970’s, it was decades before academic research into the area was fully applied to these real world problems.

The long term lack of computer scientists taking an interested in the computer forensic field has left relics such as the cryptographic hash to identify files of interest as the primary method for this task. However, as hashing was intended to find files that do not match exactly when they should, while forensics needs to find files that are similar or have certain content. While it is true that hashing can find exact matches of files that are known to be of interest, this does not solve the problem of files that are very close to those that are known but do not match exactly.

First notice that this computer forensic need is to find files that are very similar to those of interest. This would include the set of files that match exactly, which is a subset of the set of similar files. Next notice that in many cases, a file that is very similar to a known file of interest would contain many duplicative parts. Such cases can be easily seen in:

- Versioning of Documents: where a version is known to be of interest, but other unknown versions are sought
- Altered Pictures: where a version is known to be of interest, but has been edited or resized.
- Malware Variants: where a version has been detected but has been altered to evade detection
Chapter 5. Exponential Growth in All-Pairs Comparison

5.3.1 Stream Similarity

One of the first novel solutions to this problem was Context Triggered Piecewise Hashing [40]. It solved the issue of similar file matching through a sliding window hash, which was conducted via algorithmically identified sentinel values. While greatly more useful at finding similar files than traditional hash value comparison, it was flawed in that files of radically different sizes would not match. In addition, CTPH is a pairwise comparison in that stored hashes are ordered by location derived from the input file. As hashes are calculated with varying input ranges, and CTPH is input size sensitive, the usefulness of a CTPH signature of more than one input is questionable.

Other similarity driven signatures are not size sensitive. Feature hashing, as in work [36], can be used to store as many features as the upper bounds of the probabilistic hash storage will allow. The design of feature hashing allows for the augmentation of one Feature Hash with another. However, other than collisions, the addition of hashes grows the Feature Hash structure if stored as a sparse matrix. As the sparse matrix grows, comparisons are both slower due to the $O(n^2)$ nature of the design and less accurate due to collisions.

The MinHash [102] based design of Malware Provenance solves the problems with other approaches. First, as the length of the hash is not dependent on the length of the input but the number of hash algorithms (or field distributions) used to generate the hash, the addition of more input data to an individual signature does not increase comparison time as with feature hashing. Additionally, the proposed method also is not affected by input size as with CTPH as the window length is static in all hashing.

5.3.2 Derivative Datasets from MP Signatures Only

Consider a malware sample that has been processed by block hashing through the Malware Provenance method. Each block is normalized, truncating all but the first
byte of each instruction. Hashes are calculated via sliding window for each block. A signature is generated for each block by calculating $K$ minimum hashes via $\oplus$ with $K$ predetermined sequence of values. Such a process is depicted in Figure 5.5.

Now consider that a signature could be created via the Malware Provenance method for the Malware Sample as a whole. Instead of each block being extracted and normalized via the First Byte method, the entire executable byte-code is normalized. This executable section is processed in the same way as each block was processed before: i.e. a signature is generated by calculating $K$ minimum hashes via $\oplus$ with $K$ predetermined sequence of values. This process can be visualized in Figure 5.6.

Now consider that the hash windows for the first method are a subset of the second method. Those hashes excluded from the subset are hash windows calculated across block boundaries. As a floor operation of a set can be calculated in staged subsets, the MinHash signature for $n$ blocks is equivalent to a section signature with block barriers excluded. Thus, as depicted in Figure 5.7, a Malware Provenance signature calculated with a section as input is similar to that of block signatures combined to make a single signature. Notice that the compilation of signatures does not increase the signature size and that the degree of variance between a section signature and a block compilation is dependent on the number of blocks within a section and the number of total windows.
Figure 5.7: Diagram Depicting MinHash Based Signature Compilation.
Further consider that the compilation of signatures into a super signature need not be limited to blocks in a malware program. The method can be used to create malware family signatures where variations are accounted for as they are encountered. The process can be used in adjacent fields such as file forensics and network traffic monitoring where there is a need to find similar, but not exact, matching streams, and where accounting for variations could help the location of future variations. Using this property of Malware Provenance, it is possible for organizations to create new derivative datasets without requiring the original data source, which is often difficult to obtain.

5.3.3 Improving Accuracy and Performance through $K$

Variance

Recall that Malware Provenance defines $K$ to the number of different hash algorithms needed to diversify the field of possible minimum hash values and that $K$ number of different hashes can be closely approximated with a single hash XOR with a known series of pseudo random numbers. Let us propose, that in the case of Malware Provenance, we define a series of pseudo random numbers through a known algorithm where the list of pseudo random numbers can be generated to an arbitrary value $K$. In this method, $K$ at any value generates a set of numbers related to any other value of $K$. If $K > K'$, then the set $S_K \supset S_{K'}$; if $K < K'$, then the set $S_K \subset S_{K'}$; and in the trivial case of $K = K'$, then the set $S_K \equiv S_{K'}$. Thus, any Malware Provenance signature of length $K$ can be used for similarity testing with any other signature of length $K'$, so long as only the subset of the two generated number sets are used for comparison.

For instance, consider a Malware Provenance signature collection that was generated with $K = 100$, a second collection that was generated with $K = 500$, and a third collection generated with $K = 1000$. Each of these collections, when compared within themselves, produces an expected error rate of 10%, 4.5%, and 3.2%, respectively as
Chapter 5. Exponential Growth in All-Pairs Comparison

\[ P_{\text{error}} = \frac{1}{\sqrt{K}}. \] However, as any Malware Provenance signature of \( K = 100 \) is a subset of a signature generated with \( K > 100 \), it is possible to compare signatures of differing lengths by simply comparing \( K \) values of the smallest of the signature lengths. In this example, either set, where \( K = 500 \) or \( K = 1000 \), can be compared with set \( K = 100 \), while noting the increase in expected error to 10% from 4.5% and 3.2%, respectively. As memory requirements for comparison are linear to the size of \( K \), it is possible to capture signatures of a larger \( K \) for increased accuracy requirements at the expense of memory and retain the ability to reduce the signature size to allow for larger search indexes and thus larger datasets.
Chapter 6

Malware Provenance as a Variant Detection Tool

6.1 Introduction

In statistical analysis, a Gold Standard Test refers to a test or measurement that is the best available under reasonable conditions. Until this experiment and the publication of the results in work [106], these datasets have been derived through algorithm or sampled through malware repositories without manual analysis. For instance, in [36], samples were identified through algorithm with malware names reported in Virus Total [107] as input. The problem with this approach is that Virus Total’s database is ever changing, making it impossible to reproduce the exact test for peer review and/or comparison. In addition, the nature of choosing a dataset by antivirus naming is that antivirus identification is well known to be inexact and flawed. Further, even if these problems were solved, the evaluation is such that it tests the ability of a newly proposed algorithm to match that of another older algorithm, namely the signature driven algorithm of common antivirus products.

The failure of the signature system of antivirus products is one of the driving forces
in malware variant detection research. As such, a comparison of proposed solutions cannot be measured on the ability to find malware identified by these signatures. Other works such as [73] have sought to algorithmically change source code in a way to create variants of software packages. While this may be a superior approach to signature identification, the test was targeted toward general similarity comparisons rather than malware, specifically.

To further complicate the issue, in the wild, malware is produced by numerous attackers, malware writers, and packaging algorithms. This leads to a wide variety of techniques used to avoid signature detection and similarity analysis. This crowd sourced approach to malware generation produces a variety of malware variants that are very difficult to reproduce in the lab through algorithm. Thus, the only available solution is to use actual malware that has been sampled from this large crowd source as the basis for comparison. However, as mentioned above, the sampling should not be derived from signature detection algorithms as this would lead only to an evaluation of a new method to produce old results.

To address these issues, we proposed a Gold Standard Test; however, it should be discussed what should be included within this test. First, is the dataset itself. The dataset should represent data that is found in the wild. Knowledge of the dataset should be derived from the best available source under reasonable conditions. The tests against the dataset should enable comparison of performance in terms of both speed and accuracy on like systems. While similarity tests exist in works such as [73], we know of no tests with both a fixed dataset and standard, comparative results for future works that have been published specifically for malware variant detection.

As a result of this observations, we proposed a test and dataset in work [106] that provided:

1. A New Gold Standard Dataset for malware variant detection

2. A Standardized Nomenclature to bring malware variant detection inline with
other statistical classification projects

3. An initial comparative analysis of projects within the field of malware variant
detection based on static analysis

6.2 Method: Measurements and Dataset

6.2.1 Measurements

Our testing framework, Variant, was presented as a solution toward evaluating mal-
ware variant detection works. As a statistical analysis Gold Standard Test, Variant
reported on accuracy in terms of Precision, Recall, and F₁ Measure. Recall that
Precision is the positive predictive value, or the faction of retrieved samples that
are identified correctly. In this type of binary classification, Recall is the sensitivity
of an algorithm to retrieve relevant instances. In simpler terms, Precision is those
samples picked from a field that are correct, divided by the total picked. Recall is
those samples picked from a field that are correct, divided by the total number of
actual positive samples.

Precision and Recall are specific terms that describe general accuracy and, most
often, as one increases, the other decreases. To find a balance between these values it
is common to take the harmonic mean of Precision and Recall to generate F Measure
or balanced F₁ score. For the purposes of the Variant Framework we formally calculate
the following values:

1. \( \text{Recall} = \frac{\text{RelevantSamples} \cap \text{RetrievedSamples}}{\text{RelevantSamples}} \)

2. \( \text{Precision} = \frac{\text{RelevantSamples} \cap \text{RetrievedSamples}}{\text{RetrievedSamples}} \)

3. \( \text{F Measure} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \)
Chapter 6. Malware Provenance as a Variant Detection Tool

Performance is also of concern with respect to variant detection with suspicious files being submitted at between 400K and 1M per day to the antivirus providers [108]. However, performance is also divided into two measurements that can impact end points. First is the intake processing speed. This performance measure is the time it takes for an intake system to process the malware or suspicious software sample so that it can be compared to other samples. For example, for traditional hashing, where the sample is compared to other samples by taking a cryptographic hash of the sample and comparing it to other hashes, this processing time is the time it takes to generate the hash itself (without comparison). The generation of these signatures is often not dependent on previous signatures and is, therefore, linear in nature. Thus scaling projections of signature generation is straight-forward in these cases.

The next performance measure is the time it takes to conduct the comparison itself. While signature generation time is linear in the test cases provided in this paper, the comparison time in similarity analysis is not. Some of the methods included in this paper compare via sorted lists (LSH or other techniques) which is $O(n \log n)$ while others conduct a pair-wise comparison which is $O(n^2)$. Each of these present unique problems in transition. The $O(n^2)$ pair-wise comparison is CPU bound, requiring ever increasing computations to meet demand. The $O(n \log n)$ sorted indices comparison is memory bound, requiring ever increasing memory to account for the indexed tables used to calculate similarity. As we describe each of the tested projects for these papers, we will briefly discuss the implications of scalability for each of them.

6.2.2 Dataset

Dataset generation by algorithm is problematic as explained above. As such, we have obtained a dataset that was sampled at random via inputs in various incidents and analysis by professional malware analysts, whom grouped the dataset into eight distinct groups. Some of the malware samples within the dataset were originally
packed. We include the actual MD5s of the samples within this paper as the dataset is relatively small and this ensures that the dataset remains static and available in the future for the comparative accuracy tests.

The Malware set used consists of 85 samples. Within the set the malware has been put into 8 separate groups by function similarity. Analysis information was derived manually by malware professionals. Additional metadata has been gathered for each malware sample from Virus Total, McAfee, and/or Dell SecureWorks Counter Threat Unit.

Group 1: Consists of 12 malware samples. 10 of the 12 are identified as Ziyang RAT and were originally packed with Armadillo v1.71.
25721aa47fb29fcba9de1f3406d9f8d6
31da84e9dd9b865a7d0e4c3baa7b05a2
35f65bd2c9ff5c46186f84f19a3a7d18
3ce19fc2a1a6a42b8450d477a9919de2
47cc260cf70fc81995f651dc1c5b172a
718c6e47512bec8c585320d087041ace
7b30b4d95ed988881ec9fc3908df409e
8d64f279400d8e1f8bf2170d148203a7
90a219684b3b815d6b6c1add5e28c5b
b6e1a2048ea6bd6a941a72300b2d41ce
cbef7e4cd8b51b0b38fdebab6486f89a
ea66e664bd530124ff7993a4ad510d4

Group 2: Consists of 19 malware samples. The 19 samples are identified LinsensingSvr and were originally packed with Armadillo or Aspack.
0f171ff1a80822934439edaa7be1023b
10d7989355b5fc2915a18004df4f9074
Chapter 6. Malware Provenance as a Variant Detection Tool

156085a7cd31d272486193df10d7e26e
1a56c6eb1cd54ce642bdfd59168da127
35185b8c5e3cb928c97919aa5ad01315
356c9314ae95a18f3f6e630e04f4d8b6
3f7601f0aeb5e391638a597c15f80c9f
4734d158048c398f2ae44c035487e249
47803deb563d9ff917369b8c97c22a7e
49361de55268ff2ee67add42d359248d
5a5d26fe70521efdb875fecc961ff75a
5fa46b686c3a5e27fd4dfe0e1fbb1145
89e9bed692611692e244ed294c9904cc
9951f026f491ef90037a59f305269273
a90194c071ae6fb21331385ad7115fbc
a9a53cd80a12519429a9a40f9d34e563
b14ad1298928bb33613eb8e549c93e9e
d414c721c60df0282481df77c0c1cdae
e4cdfa15a380346ae7f80334e7d6a14

Group 3: Consists of 20 malware samples. The 20 samples are identified as BeepService and were originally packed with Armadillo v1.71.

0625b5b010a1acb92f02338b8e61bb34
15cb44831bddd295bb3c0decf7cea0dc0
1f3c731aed7d8085eb2d15132819eb8b
2393b93a762d4990ec88d25c9e809510
3a282da31bf93cfaaa8b5a11d441483b
3aa3846284b6e7112da90e1d5e4e7711
3c6ff8b69513bf338a2d5b3440b9a8cd
Chapter 6. Malware Provenance as a Variant Detection Tool

463a12f92652fc82b3c6e53bb917ecf2
4e95cb057f351af0f7c972800a07f350
52b8063f663563d549ec414a7caf38f9
54dc517c9f62dc5d435fb8bac0fd59f9
59534c90c3234fbd82492d1c1b38659
660b856f485fb8fa0ecb3533d88d405e
6b8ea95a729551fde76a28244cb95ac1
726d77fe00b4c00df1bb2c5af05ad21
73b8facac3e946354a89e58d308d8ebd
99f67381b3b389f0e6120603019e0ef9
a0f71497ca4c462c094c1843693381e
d5caf69c7a2ae416131133e0b1623066
e8ee22223b6475d7b3ef8f51383df1ef

Group 4: Consists of 13 malware samples. The 13 samples are identified as SimpleFileMover and were originally packed with Armadillo v1.71.

00d0382fe1b02b529701a48a1ee4a543
139ddf5aeff4602bf8168fe13a63da30
36093314059a9e7b95025437d523d259
59ee8762316018862d7405b595267d8d
5d7c34b6854d48d3da4f96b71550a221
5ff93637082c96de1650facdce95a970
721c56a617dfd2ceca790d9e9fa9ce
731089e10c20b13095df2624b6eb399f
8f73b7653ebf20f66a961cc39249b2e3
9cf67106cd1644125b773133f83b3d64
9f546188e0955737deff5cee8696d9a
Chapter 6. Malware Provenance as a Variant Detection Tool

dc1a284e82f4f38a628b84b0e43e65d5
e9d0a062b9b72d4c46ad9a70c80ade13

Group 5: Consists of 5 malware samples. The 5 samples are identified as DD Keylogger.
0e058126f26b54b3a4a950313ec5dbce
12b0e0525c4dc2510a26d4f1f2863c75
78f2acc3309e1e743f98109a16c2b481
96c28bdddba400ddc9a4b12d6cc806aa3
b13ab523e89d9bb055ae4d4566ab34f

Group 6: Consists of 10 malware samples. The 10 samples are identified as a PUP by McAfee and were originally packed with Armadillo v1.71 or InstallShield 2000.
02106ca9b92b8c268951b08fe3f80341
3685dd53195dae257528731ddd21d5c6
649d5fd5cbe4f5e1e16be346f8ae937
70306b773629f7fa71b301b5ee5bf087
8cdd5484985c62e38703314ff4d72642
99fde4bc652f2bd0e2085c2ea085aca
da33550dbcd821ead57f04349b04a7d4
dcc267ca10304fd38be926dc778108ca
edb7d0e894e929a5892bfc0b52c4752d
f46b1a119a53c16736a6b593df5f7eaf

Group 7: Consists of 3 malware samples. The 3 samples are identified as backdoors by McAfee and were originally packed with Armadillo v1.xx - v2.xx.
27e4610a97265110c7b79b2ba93f77f8
Chapter 6. Malware Provenance as a Variant Detection Tool

Group 8: Consists of 3 malware samples. The 3 Samples are identified as SvcInstaller and were originally packed with Armadillo v1.71.

For performance measurements with regard to signature generation and comparison times, the 85 samples included above do not represent a large enough dataset to project real world results. As one might expect, including 10K or 1M MD5s in a paper would not be useful. However, access to a randomized dataset of malware in large numbers from large repositories is likely to produce similar results for speed tests. Thus we acknowledge this is not an ideal solution; however, the solution to a randomized selection of malware for speed tests is practical.

The tests are targeted toward understanding malware variant detection in terms of binary classification as well as performance characteristics in terms of signature generation speed and comparison for all-pairs clustering. To understand these measurements, we plot a ROC curve of precision and recall, varying the threshold for each comparative result from a similarity score of 0.01 to 1 using our manually analyzed dataset detailed above. For performance measurements, we plot the signature generation time of our 85 sample dataset, plus 10K randomly selected samples in 1K increments. We conduct and visualize the comparison times for each candidate solution with the same inputs.
6.2.3 Candidate Solutions for Comparison

6.2.3.1 CTPH

Context Triggered Piecewise Hashing [40] was proposed in 2006 and implemented in the open source project ssdeep. The project is a general file similarity tool in that the entire content of the file is used to determine similarity. The tool is not completely input agnostic as hash windows are started and stopped on triggers based on hashes of file content. A fixed size window of 7 bytes slides through the content and the least significant 6 bits of a FNV hash are concatenated to the signature. Trigger points are dispersed across the file by modulating over a value that is determined by file size. Thus the signature is file size dependent and only files of similar size can be compared. A second signature with a modulating value of half the block size is also stored to compensate for this weakness.

As CTPH signatures are concatenated, the straight-forward comparison of these signatures is $O(n^2)$, though their very small size reduces this issue in small datasets ($O$ is very small). However, in large datasets, this would become an issue in practice. Work has been conducted to address the performance issues of CTPH [109][68]; however, as ssdeep is widely used in industry, the ssdeep implementation of CTPH is used in this paper.

6.2.3.2 TLSH

The tool ”TLSH” is a locality sensitive hashing scheme developed at Trend Micro[39] and, like CTPH and sdhash, it operates on the entire content of a file. The approach uses a sliding window hash that is temporarily stored in a byte array. Quartile points are then calculated from this array to allow for the selection of an abstracted view of the array, which is both easy to compare to other signatures and a small fixed length signature.
Chapter 6. Malware Provenance as a Variant Detection Tool

The comparison of TLSH signatures calculates a distance measure such that a measurement of 0 is exactly (or nearly exact) the same as the comparative file. The distance measure can exceed $1K$ with radically different files. This type of measurement was identified as superior in [39]; however, it presents an issue when comparing test results with other tests. In all the other tools tested for this paper, a weighted similarity measure from 0 to 1 is calculated to evaluate sample likeness. To deal with this disparity, we turn to the paper that describes TLSH. In the paper, the authors point out the value 300 as a point of diminishing returns, noting that the false positive rate ($1 - \text{Precision}$) and Detect rate ($\text{Recall}$) flatten at this point. To compare this tool with others, the measurement of distance was converted to a value of 0 to 1 by subtracting the distance from 300 then dividing by 3. Any value below 0 was bounded to 0. For completeness, we plot the recall vs. precision ROC curve of both the bounded TLSH and the natural TLSH output.

### 6.2.3.3 sdhash

The similarity digest sdhash was introduced by Roussev in 2010 in work [43]. The approach attempts to limit false positives by picking out features that are likely to be unique. The uniqueness of the features relies on entropy calculations and a large empirical study of file features. In addition to the feature selection, sdhash stores these features in Bloom filters, which address both comparison of arbitrary input sizes and efficient comparison. The sdhash implementation is a general file similarity tool designed for forensic file analysis that takes a file as input and the signature output represents the file in its entirety. In works [47][69], the authors point out several implementation flaws and a design flaw that would allow some manipulation of the similarity score.
6.2.3.4 BitShred

BitShred was introduced in 2009 by Jang and Brumley at CMU-Cylab [44]. It was further refined in works [45][36] and is a malware/software specific variant detection scheme. The approach was further developed to incorporate behaviors into signatures for comparative analysis and variant detection [36]. While this approach is reported to be promising, only the static analysis features of BitShred have been reimplemented for the purposes of this test.

The static analysis portion of BitShred incorporates a sliding window hashing scheme across the executable section of the malware. In the latest publishing, window sizes of both 4 and 12 are used to generate the hashes, which are stored in fixed-sized, bit-indexed arrays. The arrays are similar to Bloom filters in that the hash is effectively truncated and stored as a bit-indexed by the truncated hash. The two arrays can then be compared linearly through a sliding OR/AND operation to produce a Jaccard Similarity estimate such that \( J(F_a, F_b) = \frac{(F_a \cap F_b)}{(F_a \cup F_b)} \) is approximated by \( J(F_a, F_b) \approx \frac{S(B_a \land B_b)}{S(B_a \lor B_b)} \), however, while the comparison of two arrays is a linear calculation, comparison of multiple arrays is \( n^2 \) exponential.

We have reimplemented the static portion of BitShred for comparative analysis for this paper. As our implementation was built to reproduce the similarity comparison functionality of the static analysis portion of BitShred only, we make no claims that our implementation produces similar speeds; therefore, results for performance with respect to execution times have been extrapolated from the latest work [36].

In the latest work, BitShred has been implemented on a map/reduce framework with multiple map nodes. They report that signature generation times are linear, as expected, and can be produced at a rate of 4m 40s for 655,360 samples and 2m 25s for 327,680 samples. The generation times are on a 320 map node distributed system, which is approximately 2300 samples/per second on the distributed system or 7.1 samples per second per node. While our in-house implementation is slightly faster
Chapter 6. Malware Provenance as a Variant Detection Tool

in signature generation times (7.7 samples/sec on a single node), our comparison times are significantly slower. The work reports a comparison time of approximately 21,850,000 comparisons/sec on a 320 node system. This works out to approximately 68.3K comparisons/sec per node; therefore our reporting will project this comparison time.

6.2.3.5 First Byte

The First Byte malware similarity method was introduced in 2013 by Upchurch and Zhou in work [100]. It was realized after researchers in works [65][91] proved that simple disassembly can dramatically improve location of variant segments of code. The First Byte method was based on a full disassembly of malware, linking similar basic blocks to other basic blocks to form a graph from which relationships could be visualized. The basic blocks were normalized by extracting only the first byte of each instruction and using those as inputs to the sliding hash window. The project used sliding window hashes of size 5 or 10 to create a bit indexed array such as used in BitShred

Comparisons are completed in the same manner as in BitShred, that is an $O(n^2)$ based many-to-many (pairwise) comparison. This limitation lead to the development of a Locality Sensitive Hashing (LSH) scheme to represent the feature set that was fixed length and ordered. Thus one-to-one comparisons remain ordered as the original bit array, but look ups in a many to many comparison are $n \log n$ rather than $n^2$. To conduct the variant test for First Byte, which was developed to detect code blocks rather than entire variants, the blocks signatures for the malware samples were combined into one signature to represent an entire malware sample.

Additionally, First Byte originally used IDA Pro [27] to extract basic blocks from the malware samples. While IDA Pro is widely used in industry, the extraction of basic blocks in this manner proved too slow for en masse comparisons. While IDA
Chapter 6. Malware Provenance as a Variant Detection Tool

Pro’s recursive disassembly allows for more accuracy in identifying variants, a linear sweep disassembly proved more than an order of magnitude faster. A brief comparison of a recursive vs. linear sweep disassembler results from First Byte are included in the results; however, linear sweep is used throughout the rest of the paper via Distorm [110] (abbreviated FB-L in graphs and tables).

### 6.3 Experimental Results

All experiments were performed on a Linux64 machine (Intel 3.5 GHz 4770K / 32GB memory) unless otherwise noted.

#### 6.3.1 Choosing Solution Options

##### 6.3.1.1 BitShred Window Size

The work [36] describes BitShred with window sizes of 4 and 12, which represent a hash for every 4 and 12 consecutive bytes, respectively, within the executable section of a software package. To simplify comparative results with other tools, we first choose a single window size within BitShred in which to conduct the rest of the comparative tests.

In Figure 6.1, the two window sizes for BitShred are compared to each other. We do not address the signature size to avoid collisions, we instead use the maximum signature size, 32K, as reported in the paper to eliminate any noise in comparison. Signature size is an important component of speed of comparison; however, our performance comparisons are extracted from the paper and are, therefore, not affected by this choice.

As the graph in Figure 6.1 reveals that a window size of 4 generates a superior ROC curve (best is to the upper right), this window size was chosen for subsequent tests.
Chapter 6. Malware Provenance as a Variant Detection Tool

![Precision/Recall Curve for BitShred Windows.](image1)

**Figure 6.1:** Precision/Recall Curve for BitShred Windows.

6.3.1.2 TLSH Bounding

![Precision/Recall Curve for TLSH.](image2)

**Figure 6.2:** Precision/Recall Curve for TLSH.

Figure 6.2 shows the effect of bounding TLSH to a value between 0 and 1. As expected, the ROC curves are unaffected by the bounding except that recall increases to near $\text{recall} = 0.5$ at $\text{Precision} \approx 0.23$ where the bounding removed those results. While this concerns only those where $\text{Precision}$ requirements are very low, it should be noted for completeness.
6.3.1.3 First Byte Implementations

Two variations are available for comparisons using the First Byte technique. Recall in our early published work [100], the extraction of basic blocks allowed for the identification of libraries through a bit-array based filter. As this experiment uses the same process, it is possible to remove library features from signatures of malware samples. Therefore, we will examine the effect of this option with respect to $F$ Measure and a Recall - Precision ROC curve. In addition, as the linear sweep disassembler is much faster at generating signatures (4.1sigs/sec vs. 0.13sigs/sec), it is worth examining the cost in accuracy.

As can be seen in Figure 6.3, the $F$ Measure peaks steeply for linear with libraries, linear without libraries, and recursive with libraries. The removal of libraries from the recursive is peaked at $\text{threshold} = 0.01$. The ROC curve for the four techniques in Figure 6.4 shows that the precision for recursive never falls below $\text{Precision} = 0.925$. The peak $F$ Measure $= 0.755$ for all of the techniques is "recursive with libraries", though both linear sweep methods peek at $F$ Measure $\approx 0.71$. The $F$ Measure performance measurements with both linear sweep options are nearly identical throughout the curve of both graphs. While the FMeasure in the linear sweep disassembly is much more sensitive to changes in threshold, we will use the First Byte implementation with linear sweep disassembly and exclude the libraries in further tests.

6.3.2 Binary Classification Comparison

The goal of a variant detection scheme for malware, as with any binary classification system, is to correctly identify malware variants in a field of unknowns. To predict the field performance of these solutions, we test and measure if an implementation can correctly identify malware variants, how many instances of the identification are incorrect, and how many in the field have been missed. As explained above, we will measure these attributes with a manually sorted grouping of malware variants that
Chapter 6. Malware Provenance as a Variant Detection Tool

Figure 6.3: FMeasure vs. Threshold for First Byte Types.

Table 6.1: Recall at Precision.

<table>
<thead>
<tr>
<th></th>
<th>Precision</th>
<th>1</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitShred</td>
<td>0.166</td>
<td>0.348</td>
<td></td>
</tr>
<tr>
<td>ssdeep</td>
<td>0.255</td>
<td>0.343</td>
<td></td>
</tr>
<tr>
<td>First Byte</td>
<td>0.357</td>
<td>0.558</td>
<td></td>
</tr>
<tr>
<td>TLSH</td>
<td>0.156</td>
<td>0.207</td>
<td></td>
</tr>
<tr>
<td>sdhash</td>
<td>0.102</td>
<td>0.296</td>
<td></td>
</tr>
</tbody>
</table>

was not algorithmically generated. This will measure the performance of a potential solution to that of a human analyst rather than another algorithm.

For generalized binary classification performance, the peak $FMeasure$ of each of the techniques is most useful. The graph in Figure 6.5 displays these values for the tested solutions. BitShred maximizes $FMeasure$ at $FMeasure = 0.504$; ssdeep at $FMeasure = 0.531$; First Byte at $FMeasure = 0.710$; TLSH at $FMeasure = 0.360$; and sdhash at $FMeasure = 0.446$. While BitShred, First Byte, and TLSH all peak within their respective $FMeasure$ plot, both ssdeep and sdhash are maximized at the minimum possible threshold.

For a more tuned classification performance, we have plotted the ROC curve to contrast the tradeoffs in precision and recall. The graph in Figure 6.6 plots the points of recall and precision while varying threshold levels from 0.01 to 1 (the trivial case
Chapter 6. Malware Provenance as a Variant Detection Tool

![Precision/Recall Curve for First Byte Types.](image)

Table 6.2: Precision at Recall.

<table>
<thead>
<tr>
<th>Recall</th>
<th>BitShred</th>
<th>ssdeep</th>
<th>First Byte</th>
<th>TLSH</th>
<th>sdhash</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>na</td>
<td>na</td>
<td>0.919</td>
<td>na</td>
<td>0.89</td>
</tr>
<tr>
<td>0.3</td>
<td></td>
<td></td>
<td>0.971</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.74</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.35</td>
<td></td>
<td></td>
<td>0.35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of threshold = 0 was omitted for graph focus). For cases where Precision = 1, the tools display a large range of recall values. Table 6.1 shows the recall levels for high degrees of precision. First Byte has the highest recall measurement in the test with Precision = 1 where Recall = 0.357. The only other tool to push out of the teens at that precision level is ssdeep. At Precision = 90, First Byte again has the best performance with BitShred performing second best. In Table 6.2, Precision is measured at a required Recall level. The only project to reach a recall level beyond Recall = 0.50 in the test was First Byte with a maximum Recall = 0.74 at Precision = 0.318. All tools were able to retrieve at the 30% level, with First Byte having no false positives at this level.
6.3.3 Performance Comparisons

6.3.3.1 Signature Generation

Performance of the test tools varies widely. The signature generation times of First Byte, having to disassemble each malware sample, are the slowest of the group, with an average of 4.1 signatures/sec. The fastest signature generation was sdhash, which measured 348.4 signatures/sec. For in-processing malware, First Byte is capable of generating \( \approx 350K \) signatures/day/node in its current research state. The fastest, sdhash, is capable of more than 30M signatures/day/node. In Figure 6.7, the plot shows the dramatic difference in signature generation times over our Gold Standard dataset and the 10K additional samples.
6.3.3.2 All-Pairs Comparison

Signature generation times are linear in nature for all of the tools tested. In an all-pairs similarity clustering, each submission must be evaluated against all others, which leads to exponential comparison times as sample sets grow. A one-to-many comparison is linear with respect to field size with this method. There are known methods to reduce this comparison time, such as LSH, which is known to be $O(n \log n)$ and is used in two tools in this test, TLSH and First Byte.

In all-pairs comparison, the effects of algorithm choice become apparent. As seen in Figure 6.8, two of the three $n^2$ comparisons, BitShred and sdhash, consume hundreds of more node seconds than the rest. The CTPH tool ssdeep, though $n^2$ can perform inline with other tools with datasets in the 10K range. However, to
project computation times of much larger datasets, the operation time was computed at 10K samples for each of the algorithms, $n^2$ for ssdeep, BitShred, and sdhash; and $O(n \log n)$ for TLSH and First Byte. As seen in Table 6.4, ssdeep performs in league with TLSH and First Byte due to its very small operation time. However, as seen in Figure 6.9, the ssdeep algorithm is clearly exponential; thus, ssdeep’s maximum samples per day is 339K per node, where the $O(n \log n)$ algorithms can process 345M and 147M for First Byte and TLSH, respectively. As LSH is memory bound for lookup tables, these numbers may not be achievable. For instance, First Byte is limited to $\approx 2M \text{ signatures/GB}$ of available RAM.

### 6.4 Observations

#### 6.4.1 Inconsistencies

Variant locates several inconsistencies in reporting of recall and precision of other projects. In the various published works on tools, datasets have been generated by algorithms geared toward selection of sets by signature detection, limited manipulation of source code, or limited manipulation through byte-code changes. These
Chapter 6. Malware Provenance as a Variant Detection Tool

![Time to conduct all-pairs comparison](image)

Figure 6.8: Time to conduct all-pairs comparison.

<table>
<thead>
<tr>
<th></th>
<th>Published Recall</th>
<th>Published Precision</th>
<th>Tested (Peak FMeasure) Recall</th>
<th>Tested (Peak FMeasure) Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitShred[36]</td>
<td>.928</td>
<td>.932</td>
<td>.348</td>
<td>.914</td>
</tr>
<tr>
<td>TLSH[39]</td>
<td>.945</td>
<td>.935</td>
<td>.260</td>
<td>.583</td>
</tr>
<tr>
<td>sdhash[39]</td>
<td>.371</td>
<td>.995</td>
<td>.295</td>
<td>.910</td>
</tr>
<tr>
<td>ssdeep[39]</td>
<td>.312</td>
<td>.999</td>
<td>.337</td>
<td>.898</td>
</tr>
<tr>
<td>First Byte</td>
<td>na</td>
<td>na</td>
<td>.602</td>
<td>.872</td>
</tr>
</tbody>
</table>

derived datasets produced recall and precision measurements for the various tools in our tests that were much higher than our manually analyzed set that was selected from wild malware.

While the origin of these inconsistencies is unknown, we submit the first place to look is in the test sets used in other projects. As these sets have been derived by algorithm, the malware does not likely represent the breath of variation in wild malware samples. Variant allows for the direct comparison of works using static analysis techniques to detect malware variants.

In Table 6.5, we show that our manual selection of wild malware produces results inconsistent with other selection processes used in previous works. As our selection process represents a small portion of wild malware and is not derived or identified by
another algorithm, we submit these results are representative of real world tests and deployments.

6.4.2 Reproducibility

Our dataset is static. While we encourage others to expand upon the dataset by manually selecting malware that has been grouped by professionals, the original dataset can be reproduced simply by obtaining the malware identified by MD5 in this paper. Since our dataset can be reproduced exactly, future testing of malware comparative tools, with regards to binary classification measurements, can be compared directly to the included results outlined in this chapter and with other tools that publish based on the described dataset. This is an essential component of peer review and validation that has been absent in this field.

6.4.3 Measurement Alignment

Malware similarity works are relatively new to the greater binary classification field. The development of malware similarity works and their subsequent efficacy measurements have been reported in terms of false positive rates, detection rates, or other
accuracy measurements that are related to the standard binary classification terms (such as $FalsePositiveRate = 1 - Precision$), but are not aligned with reporting standards in the greater binary classification works. We submit that reporting of measurements be aligned with other binary classification works as reported as such in this paper.

### 6.4.4 Comparative Results

As outlined in [73], performance metrics are divided into three categories for these types of binary classification works. First is the preprocessing stage, where signatures are generated and times are measured. Second is the comparative stage, where a linear or all-pairs comparison is conducted and time is measured. Lastly, is the measurement in terms of binary classification. We identify these areas in our testing approach and produce results for 5 implementations from previous malware classification works to demonstrate the need for such a test. In conducting our tests, we uncovered some insights with regard to our test candidates.

In terms of speed, the approaches used in TLSH with respect to both signature generation and comparative speed are clearly preferred. While signature generation is 2.5x that of sdhash, this is more than compensated for through the use of an $O(n \log n)$ all-pairs comparison through the use of an LSH scheme. However, in terms of classification of malware, TLSH could not produce an $FMeasure > 0.4$.

In terms of binary classification, the First Byte method of Malware Provenance is a superior approach with our test set. First Byte produces the highest $FMeasure = 0.71$, highest $Recall = 0.740$, and $Precision = 1$ at $Recall = 0.357$. Performance with respect to comparative times is also a sound approach for the project as the comparative measurements are the fastest within the 5 projects tested. However, classification performance comes at the cost of time needed to generate signatures. First Byte is 85x slower at generating signatures than the fastest time, which was
produced by sdhash; however, comparison times will quickly overcome the signature
generation times in large datasets when using sdhash. First Byte is 30x slower at
generating signatures than TLSH, which also has a desirable $O(n \log n)$ comparison
algorithm.

Overall results are a matter of the specific needs of the transition project. If 365K
inputs per node, per day are enough to satisfy project requirements, First Byte in its
current state is the best solution tested. However, if memory bounding is an issue
with expected dataset sizes, the LSH solutions offered in TLSH and First Byte may
not meet requirements. Here, ssdeep and BitShred offer CPU bounded solutions that
could produce $n$ to $m$ results in reasonable times on distributed systems so long as $n$
was bound to a small fraction of $m$. 
Chapter 7

Malware Provenance Realized

Cyber security analysts are tasked with maintaining security of a network while detecting instructions in real time or performing a forensic investigation to see how an attack occurred [49]. A key component of this cyber analytic forensic process would include the ability to associate not just source code with intelligence data, but code with other code. As access to source code for malware is scarce, the code-to-code component of any cyber intelligence system must be based on some form of the compiled malicious binary. However, typically, compilation introduces significant variance in the binary. These variances are in the form of arbitrary register use, relative offsets, compiler specific optimizations, stack use, and other variances introduced through compiler implementation. The need for a reliable and accurate malware similarity engine combined with the difficulty in calculating such similarity has made malware similarity an active area of research. However, the limited availability of a baseline comparison dataset has, in turn, limited real accuracy measurements. As we have obtained a fully analyzed malware dataset from a third party, we have produced this accuracy study of byte-code similarity methods.
7.1 Experimental Results

7.1.1 Equipment and Dataset

All experiments were performed on a Linux64 machine (Intel 3.5 GHz 4770K / 32GB memory) unless otherwise noted.

We performed our experiments on a malware data set of 85 samples that has been professionally analyzed. The dataset is comprised of eight groups of malware variant with 3 outliers. The test was conducted in the blind without any prior knowledge of the dataset, except that it was to be limited to unpacked, x86 executables, for which our approach is designed. While the original work used IDA Pro [27] to disassemble and extract the First Byte input for processing, we instead use DiStorm. In all graphing, the manual analysis is color coded to group and the spacial location of each cluster represents algorithmic effectiveness of the approach.

7.1.2 Accuracy of First Byte Feature Hash Similarity

We performed a subjective test based on the expertise of a seasoned malware analyst outside of our group. A script was developed that presented two blocks from our similarity engine such that $40 < similarity < 100$. In this way, every block was unique, as determined through hash, in at least one mnemonic. After the evaluation of the two blocks, the analyst could choose if the block was truly similar and mark $y$ for a found similarity. If, however, the block was too simple and could be created at random, without access to the source code or was dissimilar, the analyst was instructed to mark $n$ for found similarity. If the analyst was undecided on the similarity of the code, the analyst was instructed to mark $u$ for undecided. The comparison was very labor intensive and was limited to 50 randomly sampled blocks due to analyst time constraints within an operational laboratory.

The evaluation shows that similarity is 82% accurate with 6% remaining undecided.
and 8% either false positives or too simple to distinguish from code that is likely to be coded in similar ways. More importantly, no similarly match greater than 60% was found to be in error or undetermined. While a more extensive evaluation is needed for analyst confidence to trust similarity calculations implicitly, the study shows promise in automation of malware analysis where similar code has already been evaluated and prototyped.

7.1.3 Known Data Set Comparison

In this test, we continue to use OpenOrd [97] algorithm, through Gephi [96], as our clustering method. The test set was disassembled to 4158 blocks that met our 20 instruction per block requirement. From within these blocks, the first byte of every byte-code instruction is extracted, removing relative locations from the input signature. From this normalized block, a MinHash was constructed using a sliding window size of 10 bytes, which represents 10-instructions in their compiled order for each window. The MinHash is then compared to other MinHashes within the set. A graph is constructed and clustered via push/pull physics clustering to visualize the relationships and bring forth detected variants.

One key advantage of using basic blocks as an input, rather than an entire executable block, is that blocks for known code can be excluded. This provides a great reduction in $n$ for computation as they can be excluded during preprocessing or linearly in similarity checking. Our early filtering experiments reduced graph complexity by 3x to 20x in large sets, but these early graphs were still very complex with malware sets as small as 100 samples. An unfiltered graph shown in Figure 7.1 of our malware test set produces 13832 intra-block relationships. The clustered graph in this case can visibly distinguish 4 or 5 sets of malware variants, though heavy cross cluster relationships are mapped. The graph is extremely noisy in that many libraries are found in substantial portions of all software, malware and goodware alike.
A filter of known code found outside of malware, comprised from the blocks of 100 programs from Windows system32 and 100 from Linux bin/sbin, was used to construct a filter for code found in goodware. This was the original proposal in our previous work and produces 5806 relationships. The clustered graph shown in Figure 7.3, which uses this type of filter, displays a significant reduction in noise created by uninteresting relationships. Seventeen groups are clustered together with many intra-group relationships. An examination of these relationships found that most continued to be libraries or compiler entry code. This was despite the use of early filtering of code found outside of malware. Relationships of this kind should be filtered from the graph as it only conveys relationships uninteresting for this work.

A new library filter was created using IDA Pro’s native detection of libraries via its F.L.I.R.T. database. This filter excluded 2869 of the 4158 blocks from the
similarity comparison and resulting graph. As a result, 4392 relationships were found producing a graph shown in Figure 7.4 that is nearly noise free. The table 7.2 shows the resulting clusters. Two samples each of group 4 and group 6 were found to be cross clustered via compiler entry code not excluded with. We performed our experiments on our malware data set of 85 samples that has been professionally analyzed presented in work [106]. The dataset is comprised of eight groups of malware variants with 3 outliers. All of the malware samples are unpacked. The dataset was obtained in the blind and the initial First Byte algorithm [100] was not developed with this set. In all graphing, the manual analysis is color coded to group and the spacial location of each cluster represents algorithmic effectiveness of the approach the filter in two separate clusters. All of group 5 and 4 samples of group 6 were cross clustered via a single block processing an exception handler call. One outlier was weakly mapped to a single sample of group 4 (again compiler entry code) and two outliers were strongly clustered to each other.

7.1.3.1 Block Based Clustering vs. Industry Comparitive Results

A common practice for comparison in forensics is to use “fuzzy” hashing as described in work [40] and some industry applications are using full instruction similarity engines like BitShred [36]. The “fuzzy” technique is a form of context triggered piecewise hashing (CTPH). Recall that CTPH creates shingles (variable length n-grams) by a trigger, in this case a specified byte or bytes that are used as stop words, much like stop words in document shingling. Also recall that BitShred is a Feature Hash comparison method that uses the full executable text section of the malware for comparison. A comparison of similarity methods was conducted within the same clustering scheme, but using these two methods as input for the graph.

The BitShred method revealed no relationships above 45% in any of the grouped malware datasets. Reducing the threshold increases reported relationships, but at a
cost of cross clustering the groups. The fuzzy hashing method fared better, finding 208 relationships. However, the method did suffer in accuracy, heavily cross clustering groups 2, 4, and 6 with some cross clustering in groups 3 and 8 as well. Neither method grouped as consistently as First Byte as can be seen in Figure 7.4.

7.1.4 Filtering Using Available Function Identification

The success of the addition of library blocks to the negative Feature Hash or MinHash filter demonstrated the need for comprehensive filters. However, Malware Provenance no longer relies on IDA Pro for disassembly, instead uses components of Distorm to directly disassemble the unpacked malware. Unfortunately, while Distorm allows for direct incorporation into the Malware Provenance engine, it is still inferior in terms of community support, particularly in any form of auto identification of known libraries.
The 15x speed increase in generating signatures with Distorm as opposed to IDA Pro was tainted by the need to use IDA to identify libraries. As a result, some time was spent examining the construction of IDA Pro F.L.I.R.T. (Fast Library Identification and Recognition Technology).

The appeal to directly incorporating F.L.I.R.T. was based on the malware analysis community support for the technology as well as the speed increases gained by using only the necessary components of the disassembler to obtain Malware Provenance objectives. While it should be possible to use compressed F.L.I.R.T. signatures directly, we instead choose to use the tools included with IDA to convert the F.L.I.R.T. signatures to their flat form. This has the advantage avoiding the questionable and time intensive activity of reverse engineering the compressed signature as implemented. Using this form allows for the creation of new library signatures that can be used in
Figure 7.4: 92 malware samples, clustered, combined filter.

IDA and Malware Provenance.
Chapter 8

Conclusion and Future Work

8.1 Conclusions

The early work in similarity detection of code reuse in malicious software explored the approaches to overcoming the transformative process of compilation from source code to machine language byte-code. Upon examination of other project’s detection and comparison techniques, we studied manufactured test samples to understand the limitation of these approaches. We studied why code, that was duplicate in source form, was under reported when similarity measures were calculated post compilation. This early work gave us the base understanding of the problems in detection of code reuse in malicious software.

During this period, we explored $n$-gram comparisons as an approach to calculating similarity via Jiccard Index. As opposed to other works that used this approach, we used normalized assembly that was extracted via IDA Pro as an input for the $n$-gram extraction. In addition, we compared actual Jiccard Indexes calculated with real $n$-grams with the estimated Jiccard Indexes that used hashing to reduce the dimensionality of the comparison space and bit arrays to store these hashes, which are referred to as Feature Hashes. Our BBCP project, published in work [91], showed
Chapter 8. Conclusion and Future Work

that a fairly simple substitution scheme could overcome some of these issues and led to the work in First Byte.

Our study continued along the path of normalization and introduced the First Byte normalization method for extracted byte-code. The normalization method was simple, easily implemented, and produced results similar to that of a substitution scheme. The First Byte normalization approach produced similar results to BBCP, without a substitution table. We concluded that the complexities of the substitution approach, as implemented in BBCP, did not produce significant improvements in similarity over the First Byte method; therefore, we proceeded to use First Byte throughout the rest of the thesis. We noted that Feature Hashes, visualization, and other aspects of the project were separate processes and, should we decide to explore more complex normalization methods, it would not likely interfere with these other methods.

During this period, we began to expand our work beyond manufactured samples used in BBCP. We obtained a malware dataset from the Open Malware Project and used this as input, rather than laboratory created samples. The dataset consisted of more than a million malware samples and the larger dataset allowed us to examine similarity en masse. The extraction and normalization of hundreds of malware samples produced thousands of blocks. The similarity comparison of thousands of blocks produced tens of thousands of relationships. In textual form, we concluded that the results of similarity would need to be visualized to be understood. We explored many visualization solutions used in adjacent fields such as biology and networking and found that Gephi provided performance, simplicity, and an open graph format that we could produce directly from our Feature Hash based comparison engine. The first graphs allowed us to study relationships at the block level and discover the extent that libraries play in malware variant similarity.

The complexity of the graphs and discovery of the influence of libraries in graph
complexity led to the development of a filtering method for library code. The approach used known “good” code, extracted from binaries located in the system directory within our clean testing system as a source of blocks to be used as a filter. As the project still used pair-wise comparisons to calculate similarity of blocks, but set membership to tests for the filter, the filtering of known code both decreased graph complexity as well as reduced the comparison time.

At the time of the publishing of work [100], we understood that we would be limited by the pair-wise nature of Feature Hashing. We spent some time exploring parallelization and intelligent block selection to understand the limitations of the method. This was the approach used in other works that used Feature Hash or other pair-wise comparisons; however, we found that it would be unlikely that pair-wise comparisons would be able to process the 300M to 1.5 billion wild malware samples. This led to an extended period of researching how to transform our pair-wise comparisons, which are $O(n \log n)$ in all-pairs searches, to an algorithm that could accomplish all-pairs similarity closer to linear time.

In the course of our work, we obtained a manually analyzed dataset of malicious software. The set was small with 92 samples (85 of which met our requirements that the sample be C/C++ and unpacked), however, the set provided us with real world results of the accuracy of our work. The first measured results of block accuracy seemed to be encouraging. However, in the course of our study, we manually examined the summation of block accuracy to determine the accuracy of variant detection. The results of variant accuracy were disappointing as our accuracy measurements appeared less than other published works. However, in the effort to discover why our method was apparently less accurate than other methods that did not employ normalization, we obtained, or reimplemented, these works so that we could measure the effectiveness of each work against the same test set.

To conduct this examination, we constructed a version of Malware Provenance
Chapter 8. Conclusion and Future Work

that would use block-level $n$-grams to construct a variant signature as well as reimplement the static comparison method in BitShred. The results of this test led to the discovery that other works, published to date, perform significantly more poorly against our test set than the sets used in the works. In examining why accuracy was over reported in other works, we theorized that the test set construction was, likely, the influencing factor. In each of these works, the test set was constructed by an algorithm that selected samples gathered mainly from antivirus signature hits or through laboratory manipulation of programs. We concluded that our wild, manually analyzed set was more representative of wild malware. The variance between the results produced with our set verses other published works was significant and led to the publishing of the set and testing method in work [106].

The solution to the $O(n \log n)$ nature of a pair-wise comparison was realized with an examination of varying approaches to Locality Sensitive Hashing. While not all approaches could be easily incorporated into the structure of Malware Provenance, it was discovered that MinHash from “Min-wise Independent Permutations” [103] was a good fit. The approach was developed with $n$-grams as input, thus the structure of our normalization/windowing method would be preserved. Upon further examination, it was discovered and shown that a MinHash could be constructed from a Feature Hash directly and that MinHash was a probabilistic representation of a Feature Hash. As MinHash is both fixed length and ordered, it could be searched in $O(\log n)$ time and all-pairs searches would be conducted with a theoretical maximum of $O(n \log n)$ time. Experimentation shows that the searches were slightly sub-linear in practice.

In addition to the improved complexity of MinHash vs. Feature Hash, other observations allowed for the construction of malware variant MinHash signatures from malware block MinHash signatures. This allowed us to construct a version of Malware Provence that could test for variants. This capability was important for comparisons to other works as there is a lack of works that accomplish the block-based code reuse
8.2 Future Work

The Malware Provenance approach for code reuse detection is the most accurate method of code reuse detection we have observed in our work. While signature generation is slower than other projects, due to Malware Provenance intelligently extracting and normalizing blocks from submitted code, its comparison time is also the fastest approach that we have observed. As MinHash can take as input any stream of data that is linear in nature, there are other applications for the technologies developed in Malware Provenance. We intend to explore how the Malware Provenance process could be abstracted to provide similarity comparisons and visualizations with data not sourced from malicious software basic blocks.

In addition to the abstraction of the process of providing similarity, we have already begun to explore how graphs could be explored in 3D environments. We believe that interactive graphing of Malware Provenance output could provide malware analysts with new ways to study malware and provide useful technologies for intrusion link analysis.

Further, we believe that a fully implemented Malware Provenance system that could truly provide a fully developed similarity graph, combined with other intrusion data and chronological information, could finally provide a dataset useful for malware genetics and origins.
Chapter 9

Publication of Results

Works already published: The following papers related to this work have been published.


F. Adkins, L. Jones, M. Carlisle, and J. Upchurch, “Heuristic malware detection via basic block comparison,” in *Malicious and Unwanted Software:” The Americas” (MALWARE), 2013 8th International Conference on*, pp. 11–18, IEEE


Planned Publications: At present, there is one additional paper submitted for peer review.
Chapter 9. Publication of Results


Submitted Patents: At present, there are two submitted patents.


Bibliography


Bibliography


Bibliography


143
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


