Software Similarity and Classification

By

Silvio Cesare, BIT, M.Info

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

Deakin University

June, 2013

DEAKIN UNIVERSITY

ACCESS TO THESIS - A

I am the author of the thesis entitled

Software Similarity and Classification

submitted for the degree of

Doctor of Philosophy

This thesis may be made available for consultation, loan and limited copying in accordance with the Copyright Act 1968.

'I certify that I am the student named below and that the information provided in the form is correct'

Full Name SILVIO CESARE

Signed Signature Redacted by Library

Date 11 November 2013

DEAKIN UNIVERSITY

CANDIDATE DECLARATION

I certify that the thesis entitled

Software Similarity and Classification

submitted for the degree of

Doctor of Philosophy

is the result of my own work and that where reference is made to the work of others, due acknowledgment is given.

I also certify that any material in the thesis which has been accepted for a degree or diploma by any other university or institution is identified in the text.

Full Name SILVIO CESARE

Signed Signature Redacted by Library

Date 17 June 2013

Acknowledgments

I would like to thank the people who made writing this thesis possible. A special thanks to my partner, Kylie, who has supported me greatly. Thanks to Dani, Joe, and Eva who always add a degree of the unexpected to life. Thanks to my mother, Maxine, who enabled me early on to pursue academia. Thanks to my sister, Paloma, who knows too well the life of academic pursuits. Finally, thanks to my supervisor, Prof. Yang Xiang, who has supported me for many years during my Masters degree and PhD.

Publications

Books

1. Silvio Cesare, Yang Xiang, “Software Similarity and Classification”, Springer, 2012.

Refereed Journal Papers

1. Silvio Cesare, Yang Xiang, Wanlei Zhou, "Control Flow-based Malware Variant Detection", IEEE Transactions on Dependable and Secure Computing, IEEE, 2013, (in press). (ERA A)
2. Silvio Cesare, Yang Xiang, Wanlei Zhou, “Malwise - An Effective and Efficient Classification System for Packed and Polymorphic Malware”, IEEE Transactions on Computers, IEEE, vol. 62, no. 6, pp. 1193-1206, 2013. (ERA A*)
3. Yini Wang, Sheng Wen, Silvio Cesare, Wanlie Zhou, Yang Xiang, "Eliminating Errors in Worm Propagation Models", Communication Letters, IEEE, vol. 15, no. 9, pp. 1022-1024, 2011. (ERA A)
4. Yini Wang, Sheng Wen, Silvio Cesare, Wanlie Zhou, Yang Xiang, "The Microcosmic Model of Worm Propagation", The Computer Journal, vol. 54, no. 10, pp. 1700-1720, 2011. (ERA A*)
5. 5. Yongrui Cui, Mingchu Li, Yang Xiang, Yizhi Ren, Silvio Cesare, "A Quality-of-Service based Fine-grained Reputation System in the Grid Economy", Concurrency and Computation: Practice and Experience, 2011. (ERA A)

Refereed Conference Papers

1. Silvio Cesare, Yang Xiang, Jun Zhang, "Clonewise - Detecting Package-level Clones Using Machine Learning", 9th International Conference on Security and Privacy in Communication Networks (SecureComm 2013), 2013. (ERA A)
2. Silvio Cesare, Yang Xiang, "Simseer and Bugwise - Web Services for Binary-level Software Similarity and Defect Detection", 10th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), 2012.
3. Silvio Cesare, Yang Xiang, “Wire – A Formal Intermediate Language for Binary Analysis”, IEEE Trustcom, IEEE, 2012. (ERA A)

4. Silvio Cesare, Yang Xiang, "Malware Variant Detection Using Similarity Search over Sets of Control Flow Graphs", IEEE Trustcom, IEEE, 2011. (ERA A)

Industry Conferences

1. Black Hat, 2013 - Bugalyze.com - Detecting Bugs Using Decompilation and Data Flow Analysis
2. AusCERT, 2013 - Simseer.com - Malware Detection in a Cloud
3. Ruxcon, 2012 - FooCodeChu - Web Services for Software Analysis, Malware Detection, and Vulnerability Research
4. Ruxcon Breakpoint, 2012 - Detecting Bugs in Binaries Using Decompilation and Data Flow Analysis
5. Black Hat, 2012 - Clonewise – Automated Package Clone Detection
6. AusCERT, 2012 - Effective Flowgraph-based Malware Variant Detection
7. Ruxcon, Professional Delegates Event, 2011 - Faster, More Effective Flowgraph-based Malware Classification
8. Ruxcon, 2011 - Automated Detection of Software Bugs and Vulnerabilities in Linux

Media Articles and Interviews

1. "AusCERT 2013: Cloud-based scanner identifies new malware by its ancestry" CSO
2. "Scanner Identifies Malware Strains, Could Be Future of AV" Slashdot
3. ""“Tool detects software plagiarism, theft and malware outbreaks” SC Magazine
4. “Research offers software salvation from AV friendly-fire” SC Magazine
5. “Tool kills hidden Linux bugs, vulnerabilities” Slashdot
6. "'Clonewise' Security Service Helps Identify Vulnerable Code" Dark Reading
7. Risky Business #177 -- Silvio Cesare discusses his AV PhD
8. Risky Business #203 -- LulzSec: They're baaaaaaaack

Software Similarity and Classification

This thesis identifies the key topics in software similarity and classification. It examines the task of detecting software variants, clones, derivatives, and classes of software. From this theory, we propose a novel system to detect package-level clones of software using pattern classification techniques enabling us to discover software vulnerabilities in Linux. We also propose a formal language to aid binary analysis and using this framework, propose a novel system to detect malware variants through unique malware signatures, database indexing, and searching algorithms. These systems have been evaluated on real data sets including over 10,000 Linux packages making up the Debian Linux distribution where 34 previously unknown clones and over 30 previously unknown vulnerabilities were identified. Our malware system was evaluated on over 15,000 real malware and is demonstrated to be more effective and efficient than previous systems maintaining a near real-time scan performance.

Contents

Table of Figures

Tables

Chapter 1: Introduction

This thesis introduces the major applications related to software similarity and classification and proposes novel contributions to the theory and practice of malware detection and clone detection. The topic of software similarity and classification covers the areas of detecting software variants, clones, derivatives, and classes of software. The literature of those individual areas can be combined into a cohesive topic that we examine in a unified manner. We demonstrate that considering these applied problems as a software similarity and classification problem enables techniques to be shared between areas.

1.1 Background

The software similarity problem is to determine the similarity between two pieces of software. Software that is similar has a common origin. This allows for relationships between software to be inferred such as when used in evolutionary trees to identify a software's ancestry and derivatives. The software classification problem is to assign classes to software. For example, software may be labelled as belonging to the class of malicious programs, or the class of non malicious programs. Software similarity and software classification are closely related and based on the problem of feature extraction. Feature extraction concerns itself with identifying invariant properties of a program.

A number of applications make use of identifying program features including malware classification, software theft detection, plagiarism detection, and code clone detection.

Malware classification is the process of determining if a program is malicious. One approach to perform classification is to obtain a fingerprint of the malware based on program feature extraction. This fingerprint creates an invariant signature that can be used to identify evolutionary malware variants. For detection of completely novel malware, program features can be extracted to create feature vectors which can be subsequently used in machine learning algorithms and statistical classification.

Software theft detection identifies unauthorized copying of a program in binary form. An example of this is if a software library is illegally being used with regards to its license. One

approach to detect software theft is to identify birthmarks in the software. A birthmark is a program feature or feature set that is invariant when the software is illegally copied.

Plagiarism detection identifies similar or identical copying of source code. An example of its use would be to detect student cheating in programming assignments. Plagiarism detection works by extracting program features that are invariant when plagiarised. The program features are then detected in plagiarised copies.

Code clone detection [1] seeks to identify duplicate fragments of code in a source tree. The value in detecting code clones is that it is often bad software development practice to have redundant or duplicate code fragments. By refactoring the code to eliminate clones, the software becomes easier to maintain and is less likely to have bugs. Code clone detection works by identifying program features for code fragments and identifying those features in other locations.

1.2 Motivation

Malware classification helps fight the threat of malicious software. Such malicious software presents a significant challenge to modern desktop computing. According to the Symantec Internet Threat Report [2], 499,811 new malware samples were received in the second half of 2007. In 2010, over 1.5 billion malicious code detections were identified [3] by the same vendor. F-Secure published, “As much malware [was] produced in 2007 as in the previous 20 years altogether“ [4]. This trend is continuing and makes the detection of malware before it adversely affects computer systems highly desirable. To achieve this, static detection of malware is still the dominant technique to secure computer networks and systems against untrusted executable content.

Detecting malware variants improves signature based detection methods. The size of signature databases is growing exponentially, and detecting entire families of related malicious software can prevent the blowout in the number of stored malware signatures.

Detecting malware variants improves signature based detection methods. The size of signature databases is growing exponentially, and detecting entire families of related malicious software can prevent the blowout in the number of stored malware signatures.

Detecting entire families of malware by using similarity measures instead of exact matching makes malware detection less fragile and more robust in the face of malware evolution and change.

Software theft detection is an important problem with serious consequences. In 2005, a federal court determined that the independent software vendor Compuserve be paid $140 million by IBM to license its software or $260 million to purchase its services because it was discovered that IBM products had illegitimately used code from Compuware without authorization [5]. The software theft problem is growing as the internet and software companies become more ubiquitous. For example, in SourceForge.net there were over 230,000 registered open source projects as of February 2009 [5]. Clearly, an automated approach to detecting software theft is the only way to scale with the problem.

Plagiarism detection is an important task to ensure that students do not cheat when submitting assignments. Without plagiarism detection systems, teachers rely on their own memory when marking. If the number of assignments is high, or the cheating occurs from previous years, or the assignments are divided between markers, plagiarism may go undetected. An automated approach to detecting plagiarism is therefore an important component in a teacher's arsenal against student cheating.

Code clone detection helps improve the maintainability of large software systems. Several studies have shown this that duplicated copy and paste fragments of code make code harder to maintain [6, 7]. This increases the cost of developing and maintaining software. Therefore, an effort to detect clones and refactor solutions leads to less cost in the software life cycle.

1.3 Problem Formulization

The static feature extraction problem is related to identifying invariant properties or approximations of the program.

Definition 1. *Let r be a property for program p if for all possible executions r is true.*

The software similarity problem is to determine if program p is a copy or derivative of program q. We use an extended definition based on software theft detection [8].

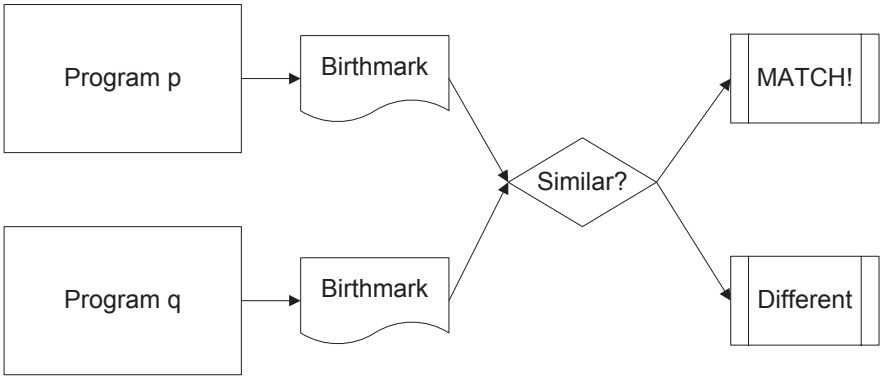

Fig. 1. The software similarity problem.

Definition 2. *A program q is a copy of program p if it is exactly the same as p or it is the result of a semantic preserving transformation (e.g., obfuscation, recompilation, or optimisation) over p.*

Definition 3. *Programs p and q are similar if they are derived from the same works.*

Definition 4. *Let p, q be programs. Let f be a method for extracting a set of characteristics extracted from p. We say f(p) is a birthmark of p, only if both of the following conditions hold.*

- *f(p) is obtained only from p itself*
- *Program q is a copy of p → f(p) = f(q)*

Definition 5. *Let p, q be programs or program components. Let f(p) → a and f(q) → b be the birthmarks extracted from p and q. Let s(a,b) → [0,1] be a similarity function and a value e < 1. The birthmarking system is resilient if p and q are similar and 1 - s(a,b) < e.*

Definition 6. *Let p and q be independently written programs. The software birthmarking system is credible if the system can discriminate between the two programs; that is s(f(p),f(q)) < 1-e*

The software classification problem uses the birthmark feature to identify class membership of software.

Definition 7. *Given a set of programs and their classes $\{(p_1,c_1),...(p_n,c_n)\}$, the software classification function c'=h(f(p)) will yield a similar classification as close as possible to the true data set.*

1.4 Problem Overview

The problem of software similarity and classification is approached by constructing a software birthmark for a program and then using a similarity function on that birthmark for comparisons. Program features are used to construct a birthmark. Different program features enable different birthmarks, so taxonomy of program features is useful. Different features have different properties which are better or worse at different qualities. A simple breakdown is to divide the features into syntactic and semantic properties. Syntax describes the structure or form of a program whereas the semantics describe the meaning

of a program's instructions. Semantics are sometimes more useful than syntax when constructing birthmarks due to the fact that obfuscations and transformations applied to programs can modify that syntax while maintaining equivalent semantics. There are different approaches in extracting features such as extracting properties from execution of the program or extracting properties statically. For static analysis, program analysis techniques offer benefit. Decompilation is a specific program analysis technique that recovers high level source-like information from a binary. Decompilation offers some benefits to birthmark construction that we examine in this thesis. If program features are used to construct birthmarks, they must be represented in mathematical form. Different features are naturally represented using different structures. Once a birthmark is constructed, they can be compared using mathematical measures and metrics. The final result is a measure of similarity, or classification of birthmarks into classes using statistical machine learning.

1.5 Aims and Scope

The aim of this thesis is to review state-of-the-art literature and propose advances in the field of software similarity and classification. The thesis makes cohesive much of the disparate literature and surveys software feature extraction, similarity, classification, and their applications by investigating the principal concepts that constitute the construction of algorithms that tackle these problems. The intended purpose is to provide an opportunity for researchers and software engineers to understand the state-of-the-art, lay foundation for the creation of extended works, and then use that foundation to propose new ideas, concepts, and algorithms to extract software features, determine software similarity, and perform software and classification.

The scope of this thesis is limited to the theory of software feature extraction, similarity, and classification. The applied areas surveyed in software similarity and classification are limited to:

- Software Theft Detection
- Plagiarism Detection

- Software Clone Detection
- Malware Variant Detection and Classification

For applications that fall outside of this scope, readers are advised to find other relevant sources and references.

The novel research proposed and implemented in this thesis is limited to 3 research works which improve specific state-of-the-art techniques to detect clones, analyse binary executables, and detect malware variants. While, not improving all the state-of-the-art in software similarity and classification, the proposed work contributes significantly to knowledge and the cohesive literature review lays foundation for future advances.

1.6 Contributions

This thesis makes the following contributions to advance the state-of-the-art in the field of software similarity and classification:

1. The literature of software similarity and classification is combined into a unified field.
2. We propose the concept of package-level clones which has immediate practical benefit to Linux vendors, package repositories operating systems.
3. We propose considering package-level clone detection as a pattern classification problem.
4. We propose over 30 features for the purposes of package-level clone detection.
5. We formulate a solution for Debian Linux in identifying security vulnerabilities based on package-level clones.
6. We propose a formal intermediate language to analyse binary-level executables.
7. We propose combining high level information obtained through decompilation into our low-level language.

8. We apply our language to a number of tasks related to code equivalence, software similarity, and classification in a formal context.

9. We propose new ways of representing graph-based signatures of programs that enable more efficient processing.

10. We propose string, set of strings, and vector based signatures to approximate a set of control flow graphs.

11. We propose new ways of comparing, indexing, and searching those signatures very efficiently.

1.7 Thesis Organization

The structure of this thesis is as follows:

- Chapter 2 gives a survey of state-of-the-art literature.
- Chapter 3 proposes, implements, and evaluates a novel system to identify package-level clones and infer security problems in Linux distributions.
- Chapter 4 proposes and implements a novel system to analyse binary-level executables.
- Chapter 5 proposes, implements, and evaluates a novel system to detect malware variants.
- Chapter 6 proposes and implements a novel cloud-based system for exposing the research systems presented in this thesis.
- Chapter 7 examines future work and concludes the thesis.

The thesis may be read in different manners. A recommended order to read the chapters is shown in the following workflow.

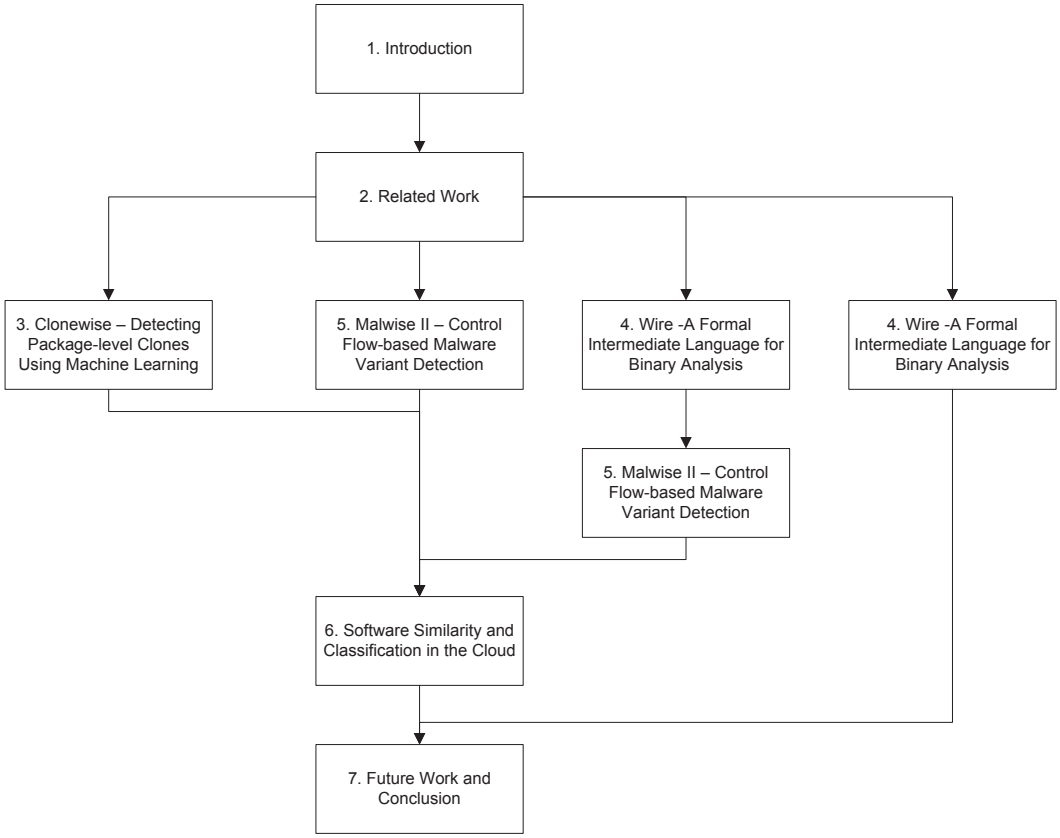

Fig. 2. Recommended order of reading chapters.

Chapter 2: Related Work

This chapter examines related work in the area of software similarity and classification. Extracting program features, processing those features into suitable representations, and constructing distance metrics to define similarity and dissimilarity are the key methods to identify software variants, clones, derivatives, and classes of software. This chapter reviews the literature of those core concepts, in addition to relevant literature in each application and demonstrates that considering these applied problems as a similarity and classification problem enables techniques to be shared between areas. Additionally, in-depth case studies are presented using the software similarity and classification techniques developed throughout the chapter.

2.1 Taxonomy of Program Features

All programs have common features and abstractions which are used to create birthmarks. Features can be divided into syntactic and semantic groups. Syntactic features concern themselves with program structure and program form. Semantic features examine the meaning of the program. In this chapter we examine those syntactic and semantic features of programs.

Syntactic Features include:

- Raw Code
- Abstract Syntax Trees
- Variables
- Pointers
- Instructions
- Basic Blocks
- Procedures

- Control Flow Graphs
- Call Graphs
- Object Inheritances and Dependencies

Semantic features include:

- API Calls
- Data Flow
- Procedure Dependency Graphs
- System Dependency Graphs

2.1.1 Syntactic Features

2.1.1.1 Raw Code

The raw code of the program can be analysed directly. For source code this is the textual stream, possibly normalized by removing comments and whitespace. For binaries, the raw code is the byte sequences.

Definition 8. *Let $\sum$ be an alphabet of symbols .The raw code of program p is defined by the function r that evaluates to a string over the alphabet.*

$$r : P \to S$$
$$p \to s, s \in \sum *$$

2.1.1.2 Abstract Syntax Trees

Abstract syntax trees (AST) examine the syntax of source code and construct a tree representing the syntactical structure. For binaries, decompilation is required to reconstruct

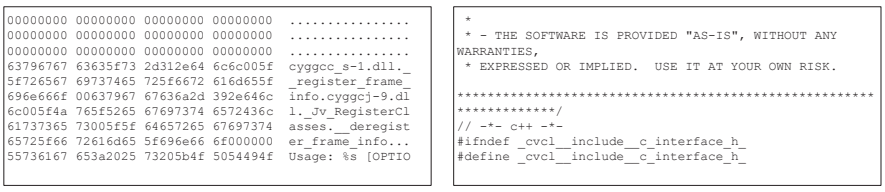

Fig. 3. Raw code for a binary (left) and source code (right).

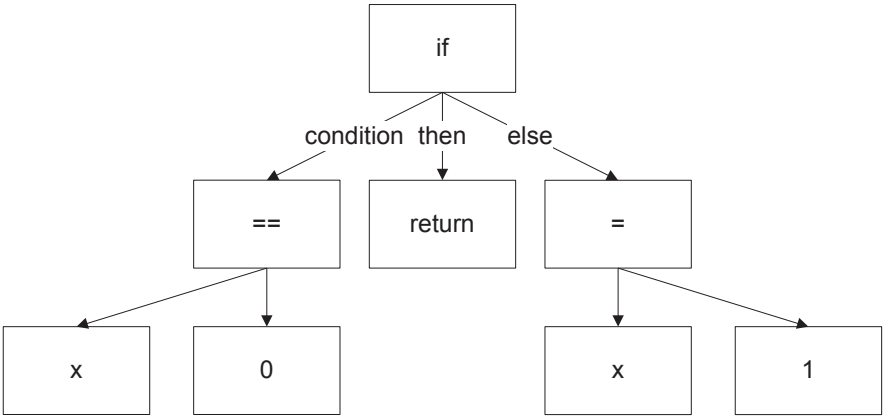

Fig. 4. An abstract syntax tree (AST).

an abstract syntax tree.

2.1.1.3 Variables

Variables represent the state of data. Programs typically maintain separate regions of memory for different classes of data handled by the run time environment. Run times may separate the stack from the heap to store data. The stack is used for local variables in a procedure and survives for the scope of that procedure or activation record. The run time creates a stack segment to achieve this outcome. In contrast, the heap is used for dynamically generated memory. Global variables conceptually belong to a different region than the heap, but for practical purposes are normally grouped together at run time in a data segment.

2.1.1.4 Pointers

Pointers are a type of variable that contain links or pointers to other variables. Pointers can be dereferenced, which allows for referencing the data the pointer is pointing to. Pointers may allow pointer arithmetic to be performed which allows for such operations as incrementing the value of a pointer. Some languages allow seemingly arbitrary pointer arithmetic, while other languages heavily restrict their use. Restricting pointer arithmetic allows for easier automated analysis.

2.1.1.5 Instructions

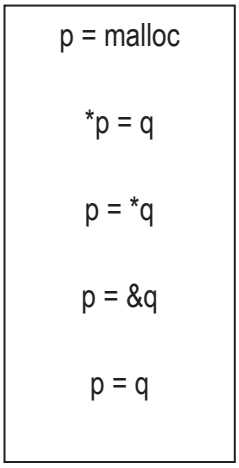

Fig. 5. Typical pointer operations.

Instructions capture the basic unit of computation. Computations can include such things as unary and binary operations, procedure or library calls. An instruction is defined by its operand and opcodes.

Definition 9. *Let I be set of all instructions such that I={(opcode,operand$_1$,...,operand$_n$)}*

Definition 10. *Let InstrSequence be a string of instructions such that*

$$InstrSequence \in \sum *, \sum = I$$

Assembly

Assembly is a low level instruction format that can be executed on the native processing unit. It consists of opcodes which describe the type of operation to perform, and operands which are the arguments or parameters. Assembly language can be roughly divided into Complex Instruction Set Computing (CISC) architectures, or Reduced Instruction Set Architectures (RISC). RISC architectures favour simplified and small instruction sets while CISC architectures favour a rich and large instruction set. x86 is the dominant architecture for personal computing and is a CISC based architecture.

Intermediate Representations

Instructions can be abstracted into intermediate representations. A common representation is Three-Address-Code which consists of three operands and one opcode. Typically, two fixed operands are inputs and the remaining operand is the output. For unary operations, the extra operands are ignored. Using intermediate representation has the advantage of normalizing a complex instruction set into a series of simpler standardized operations.

Definition 11. *Let TAC=(opcode, operand$_1$, operand$_2$, operand$_3$)*

2.1.1.6 Basic Blocks

A basic block is a sequence of instructions that satisfy the following conditions:

- Execution flow can only enter the basic block through the first instruction.
- Execution flow can only exit the block at the last instruction.

A basic block can also be represented as s directed cyclic graph showing the data dependencies between instructions.

Definition 12. *Let InstrSequence(b) be a string of instructions such that* $InstrSequence \in \sum^*, \sum = I$ *for basic block b.*

2.1.1.7 Procedures

Procedures and functions are found in structured programming which allows for making modular maintainable code. A program uses a set of procedures *F=procedures(P)={f$_1$,...,f$_n$}*

2.1.1.8 Control Flow Graphs

The control flow graph is a directed graph representing the possible flow of execution within a procedure. The nodes in the graph represent basic blocks.

Definition 13. *The control flow graph of procedure f is the directed graph C=(B,E) such that B is the set of basic blocks and E is the set of edges between them.*

Alternative representations of control flow are possible using graphs such as dominator trees or control dependency graphs.

Definition 14. *d dom n or node d dominates a node n if every path from the start node to n must go through d.*

Definition 15. *A node d strictly dominates a node n if d dominates n and d does not equal n.*

Definition 16. *The immediate dominator or idom of a node n is the node that strictly dominates n but does not strictly dominate any other node that strictly dominate n.*

Definition 17. *A dominator tree is a tree where each node's children are those nodes it immediately dominates.*

2.1.1.9 Call Graphs

The call graph represents the control flow between procedures and is again represented by a directed graph. If the program does not have recursive procedures, then the graph is

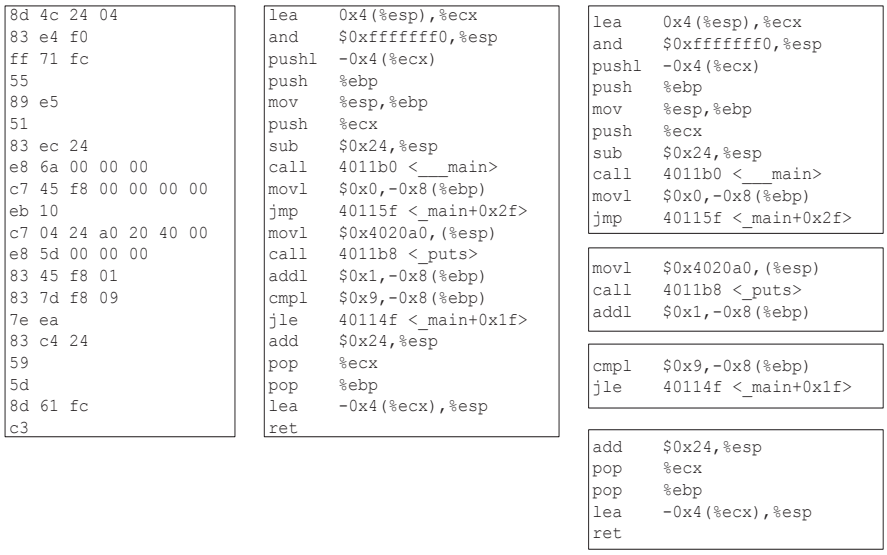

Fig. 6. Assembly instructions and basic blocks.

acyclic. Like the control flow graph, dominator trees can be equally representative of the call graph.

Definition 18. *The call graph of a program is the directed graph CallGraph=(F,E) such that F is the set of procedures and E is the set of edges between them.*

The interprocedural control flow graph combines the control flow graphs with the call graph. It is defined as *ICFG=(B',E)*:

- The set of control flow graphs.
- Each control flow graph is given an additional exit node, which is successor to the set of return nodes in the cfg.
- For all basic blocks, a call instruction divides the block into two parts. The first part is connected to a call_return node, and that in turn is connected to the remaining basic block part.
- For each basic block that now ends with a call instruction, the block's successor is additionally the control flow graph of the call target. The successor of the exit node of the target control flow graph is additionally the call_return node.

2.1.1.10 Object Inheritances and Dependencies

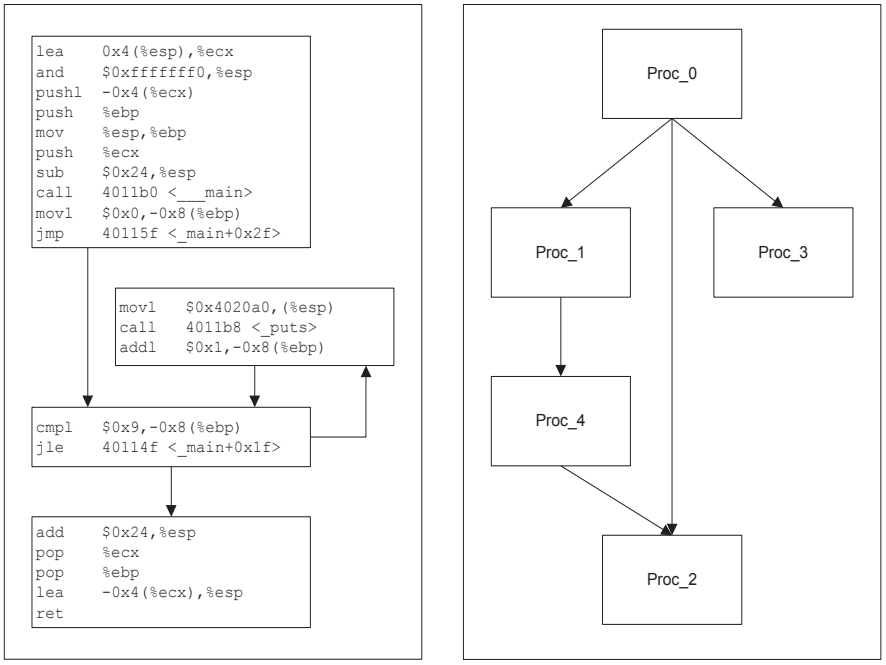

Fig. 7. A control flow graph (left) and a call graph (right).

Objects come from object oriented languages which group procedures (known as methods) and data into modular units. Objects are related to other objects via inheritance of their functionality.

2.1.2 Semantic Features

2.1.2.1 API Calls

API calls represent calls to libraries and other imports.

2.1.2.2 Data Flow

Data flow statically represents the data at run time entering and leaving each basic block. Many types of data flow analyses [9] are possible including reaching definitions, liveness, available expressions, and very busy expressions.

2.1.2.3 Procedure Dependence Graphs

The control dependencies and data dependencies of a procedure can be represented in a single graph using a procedure dependence graph [10].

2.1.2.4 System Dependence Graph

The system dependence graph combines the set of procedure dependency graphs of each procedure into a unified representation.

2.1.3 Taxonomy of Features in Program Binaries

Programs may begin as source code, but are typically compiled into a target binary for execution on the native platform or in another run time environment. The target binary is a container for all the information necessary for its execution in the target environment. This container is known as the object file format [11].

2.1.3.1 Object File Formats

Object File Formats contain five types of data:

- Headers
- Object Code
- Symbols
- Debugging Information
- Relocations

Most modern object files also contain:

- Dynamic Linking Information

2.1.3.2 Headers

The object file format is often described by a variety of headers. Headers may be used to define where the object code, symbols, debugging information, etc, is present in the binary.

2.1.3.3 Object Code

Object code contains the code and data of the program. For native executables the object code can consist of assembly or machine code. For object file formats such as Java class files, the object code contains byte code which is the instruction set architecture of the Java Virtual Machine.

2.1.3.4 Symbols

Parts of the code, data and binary may be associated with symbolic names. These associations are organized and stored in a Symbol Table.

2.1.3.5 Debugging Information
The binary may contain debugging information such as line numbers of source code associated with object code, or naming of information for different codes or data.

2.1.3.6 Relocations
If the binary has not been associated with a specific load address at compile time, the binary may need to be link edited at runtime. Relocations or fixups contain the necessary information to bind the object code to a specific load address.

2.1.3.7 Dynamic Linking Information
If the binary requires the use of external libraries, then the names of the required library functions must be present. Likewise, if the binary's functions are being exported as a library, then this information must also be present.

2.1.4 Case Studies

2.1.4.1 Portable Executable
The Portable Executable (PE) format [12] is the native object file format for the Windows family of operating systems. It is a modern file format which can contain all the information we have described in this section. It is identified by a series of magic bytes in its headers. Object code is defined in PE sections and an Import Address Table allows for dynamic linking.

2.1.4.2 Executable and Linking Format
The Executable and Linking Format [13] is the object file format in use on Linux and other operating systems. It replaced the previous a.out object file format in Linux. The a.out

```
/bin/ls:     file format pei-i386

architecture: i386, flags 0x00000102:

EXEC_P, D_PAGED

start address 0x00401000

Sections:

Idx Name          Size      VMA       LMA       File off  Algn

  0 .text         00019528  00401000  00401000  00000400  2**4
```

Fig. 8. The output of objdump on a PE executable.

object file format did not natively support dynamic linking and ELF brought a much more modern format to Linux and enabled the transition to shared libraries using dynamic linking. An ELF binary is identified by a magic sequence in its header. There are three types of ELF object files.

- Executable Objects
- Relocatable Objects
- Dynamic Objects

Executable objects have been linked and bound to an address. Relocatable objects have not been bound to a load address and require linking. Dynamic objects have both a relocatable view and an executable view - shared libraries use this format.

Dynamic linking is slightly different to the PE format and uses a Global Offset Table (GOT) and a stub call to the runtime linker to resolve imports.

2.1.4.3 Java Class File

Java class files [14] contain object code in sections defined in the file's headers. The object code is in the instruction format for execution on the Java Virtual Machine. Like the previous object file format, a sequence of marker bytes (the magic bytes) in the header identifies the file format.

2.2 Program Transformations and Obfuscations

Software feature extraction must cope with transformations that are intended to obscure, evolve, or rewrite the program. For example, malware polymorphism and metamorphism are transformations applied to the malicious code to evade signature detection. Robust signatures must identify the invariant birthmarks under these transformations. This chapter focuses on analysing these types of program transformations and obfuscations including compiler optimsations, recompilation, plagiarism, software theft, derivative works, malware packing, malware polymorphism and malware metamorphism.

2.2.1 Compiler Optimisation and Recompilation

Compiler optimisations and recompilation are semantic preserving transformations. These transformations rewrite the program but do not alter the behavioural properties of the

software. Compiler optimisations make feature extraction more difficult. Even very minor changes to a program's source code can result in significant changes to the program's instruction stream once recompiled.

Many compiler optimisations are possible. We examine some in this section. Typical classes of code optimisation that may affect the birthmarks and feature extraction are:

- Instruction Reordering
- Loop Invariant Code Motion
- Code Fusion
- Function Inlining
- Loop Unrolling
- Branch/Loop Inversion
- Strength Reduction
- Algebraic Identities
- Register Assignment

2.2.1.1 Instruction Reordering

Instructions can be reordered or scheduled in such a way that they are semantically equivalent but perform faster due to caching. To determine if instructions inside a basic block can be reordered, a directed acyclic graph can be drawn of the data dependencies. Only instructions that have data dependencies between each other require strict ordering between those instructions.

2.2.1.2 Loop Invariant Code Motion

Code that is inside a loop may be moved to outside the loop if no semantic change occurs. This improves the efficiency of the code.

2.2.1.3 Code Fusion

Code inside loops in sequence can be fused into a single loop.

2.2.1.4 Function Inlining

Functions can be inlined to improve performance. Inlining a function means that a clone or copy of that function replaces the function call. This means that a function call is avoided and therefore improves performance.

2.2.1.5 Loop Unrolling

It can improve efficiency to unroll the loop by duplicating the loop body and termination condition.

2.2.1.6 Branch/Loop Inversion

Branching on equality or non equality can be inverted and may improve efficiency in some cases.

2.2.1.7 Strength Reduction

Strength reduction replaces expensive operations with equivalent but less expensive operations.

2.2.1.8 Algebraic Identities

Algebraic identities take note that some expressions are algebraically equivalent to other less expensive operations. For example, x+0 is equivalent to the less expensive expression x.

2.2.1.9 Register Reassignment

Register allocation is the process of assigning specific registers to instructions. The assignment of these registers can change while maintaining semantically equivalent code.

2.2.2 Program Obfuscation

Program obfuscation obscures the workings of a program [15].

Definition 19. *Let* $P \xrightarrow{T} P'$ *be a transformation of a source program P into a target program P'.* $P \xrightarrow{T} P'$ *is an obfuscating transformation, if P and P' have the same observable behaviour. More precisely, in order for* $P \xrightarrow{T} P'$ *to be a legal obfuscating transformation the following conditions must hold:*

- *If P fails to terminate or terminates with an error condition, then P' may or may not terminate.*
- *Otherwise, P' must terminate and produce the same output as P.*

2.2.3 Plagiarism, Software Theft, and Derivative Works

An incomplete list of source code plagiarism techniques is described in [16]. The authors state that such a list is never ending, so a comprehensive list is impossible. Nevertheless, they identified the following forms of plagiarism:

- Lexical Changes
    - Comments can be reworded, added and omitted
    - Formatting can be changed.
    - Identifier names can be modified.
    - Line numbers can be changed (e.g., in Fortran programs).
- Structural Changes
    - Loops can be replaced (e.g, replacing a while loop with a for loop)
    - Nested if statements can be replaced by case statements and vice versa.
    - Statement order can be changed.
    - Procedures can be replaced by functions (e.g., in Pascal)
    - Procedures may be inlined
    - Ordering of operands may be changed (e.g., x < y becomes x >= y)

2.2.3.1 Semantic Changes

An extension to syntactic changes is that of semantic changes where the new variant is a derived work of the original malware. Semantic changes occur due to the software authors modifying the original source code or functionality. This can occur to a natural evolution of the software during its development life cycle. Additionally, it can occur when a software author reuses existing code in a new program instance.

2.2.3.2 Code Insertion

Code insertion occurs when new functionality is added to the malware.

2.2.3.3 Code Deletion
Code deletion occurs when functionality is removed from the malware.

2.2.3.4 Code Substitution
Code substitution occurs when functionality in the malware is replaced by an alternative algorithm or code.

2.2.3.4 Code Transposition
Code transposition occurs when specific code and functionality of the malware is removed from its initial location and inserted into a semantically different location in the malware.

2.2.4 Malware Packing, Polymorphism, and Metamorphism

The two categories of malware obfuscation are syntactic and semantic changes. Semantic changes include those described for plagiarism and software theft. A syntactic polymorphic malware technique is a method that changes the syntactic structure of the malware [17]. Though the syntactic structure changes in polymorphic malware, the malware semantically remains identical. The technique is predominantly used to evade byte level signature based detection and classification that is routinely employed by traditional Antivirus. Polymorphism borrows many of the techniques from the field of program obfuscation.

Polymorphism is sometimes described by the similar term of metamorphism. In that usage it is used to describe the automated syntactic mutation of the malware's code and instructions. Under such terminology, polymorphism is used to describe syntactic mutation of limited parts of the malware's instruction content. The remaining parts of the malware are encoded at the byte level without regard to the instruction syntax or semantics. In this book we treat polymorphism and metamorphism as identical to each other.

Syntactic malware obfuscations and transformations include:

- Dead Code Insertion
- Instruction Substitution
- Variable Renaming
- Code Reordering
- Branch Inversion and Flipping

- Opaque Predicate Insertion
- Code Packing

2.2.4.1 Dead Code Insertion

Dead code is also known as junk code and a semantic nop [17]. Dead code is semantically equivalent to a nil operation. Insertion of this type of code has no semantic impact on the malware. The insertion increases the size of the malware and modifies the byte and instruction level content of the malware.

```
push %ebx
pop %ebx
```

Fig. 9. A semantic nop

2.2.4.2 Instruction Substitution

Instruction substitution replaces specific instructions or sequences of instructions with semantically equivalent, but differing instructions and instruction sequences. The size of the malware may grow or shrink in this procedure.

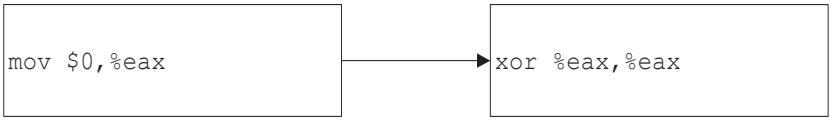

Fig. 10. Instruction substituion.

2.2.4.3 Variable Renaming

Variable renaming [18] and the associated technique of register reassignment alters the use of variables and registers in a sequence of code such that the instructions are semantically equivalent but use different variables and registers when compared to the

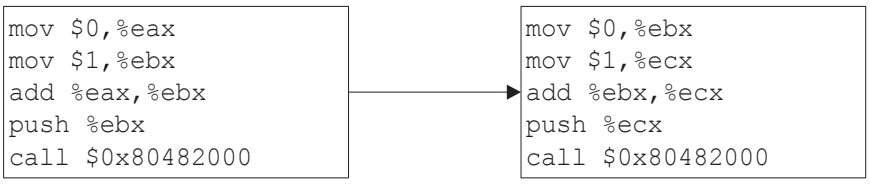

Fig. 11. Register reassignment.

original code.

2.2.4.4 Code Reordering

Code reordering [18] changes the syntactic order of the code in the malware [17]. The actual or semantic execution path of the program does not change. However, the syntactic order as present in the malware image is altered. Code reordering includes the techniques of branch obfuscation, branch inversion, branch flipping, and the use of opaque predicates.

2.2.4.5 Branch Obfuscation

Branch obfuscation attempts to hide the target of a branch instruction. Examples include the use of Structured Exception Handling (SEH) on the Microsoft Windows platform. The use of SEH to obscure control flow is common in modern malware. Similar techniques involve indirect branching. Indirect branching uses data content as the target of a branch. This translates control flow identification into a harder data flow analysis problem. The use of a branch function [19] extends this approach and dispatches multiple branches through a single routine. The main purpose of branch obfuscation is to make the static analysis of the malware by an analyst or automated system harder to perform.

```
mov $0x8048200,%eax
jmp *%eax
```

Fig. 12. An indirect branch.

2.2.4.6 Branch Inversion and Flipping

Branch inversion inverts the branch condition in conditional branches. Whereas the branch may originally transfer control when the condition is true, branch inversion alters the condition to branch when false. To maintain the original semantics of the program the branch instruction is also inverted. For example, a branch on condition true statement can be changed to a branch on condition false statement. Additionally, the condition being tested would also be inverted. Branch inversion is effectively a form of instruction substitution on control flow statements.

Branch flipping [19] is a similar technique to branch inversion and rewrites the branch instruction by substituting it with semantically equivalent code with different control flow properties. For example, if the original code is to branch on condition true then the new code branches on condition false to the original fall-through instruction. The new fall-through instruction then unconditionally branches to the original conditional branch target.

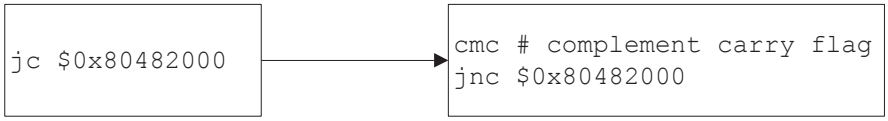

Fig. 13. Branch inversion.

Fig. 14. Branch flipping.

2.2.4.7 Opaque Predicate Insertion

An opaque predicate [19] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in the malware's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis.

2.2.4.8 Malware Obfuscation Using Code Packing

Code packing [20, 21] is the dominant technique used to obfuscate malware and hinder an analyst's understanding of the malware's intent. In one month during 2007, 79% of identified malware from a commercial Antivirus vendor was found to be packed [22]. Additionally, almost 50% of new malware in 2006 were repacked versions of existing malware [23].

Code packing, in addition to obfuscating the understanding of the malware by an analyst, is also used by malware to evade an Antivirus system's detection. Polypack [24] evaluated the effectiveness of code packing against Antivirus detection by providing a service to pack malware using a variety of code packing tools. Antivirus systems often have the capabilities of unpacking known code packing tools, and unpacking unknown tools has also had commercial interest [25]. However, Polypack demonstrated that packing can be an effective tool to defeat an Antivirus system with many commercial malware detection systems failing to identify the packed versions of existing malware.

Code packing is used in the majority of malware, but code packing also serves to provide compression and software protection for the intellectual property contained in a program. It is not necessarily advantageous to flag all occurrences of code packing as being indicative of malicious activity. Code packing tools are freely available [26] and commercially sold to

the public as legitimate software [27]. For this reason, unpacking of packed programs provides benefit. It is advisable to determine if the packed contents are malicious, rather than identifying only the fact that unknown contents are packed.

2.2.4.9 Traditional Code Packing

From [28]: The most common method of code packing is described in [20] and [28]. Malware employing this method of code packing transforms executable code into data as a post-processing stage in the malware development cycle. This transformation may perform compression or encryption, hindering an analyst's understanding of the malware when using static analysis. At runtime, the data, or hidden code, is restored to its original executable form through dynamic code generation using an associated restoration routine [29]. Execution then resumes as normal to the original entry point. The original entry point marks the entry point of the original malware, before the code packing transformation is applied. Execution of the malware, once the restoration routine is complete and control is transferred to the original entry point, is transparent to the fact that code packing and restoration had been performed. A malware may have the code packing transformation applied more than once. After the restoration routine of one packing transformation has been applied, control may transfer another packed layer. The original entry point is derived from the last such layer. The process of this form of malware packing is shown in Fig. 15.

2.2.4.10 Shifting Decode Frame

From [28]: An extension to traditional code packing is to maintain as much of the packed image in an encrypted form at run-time. During execution of the malware, blocks of

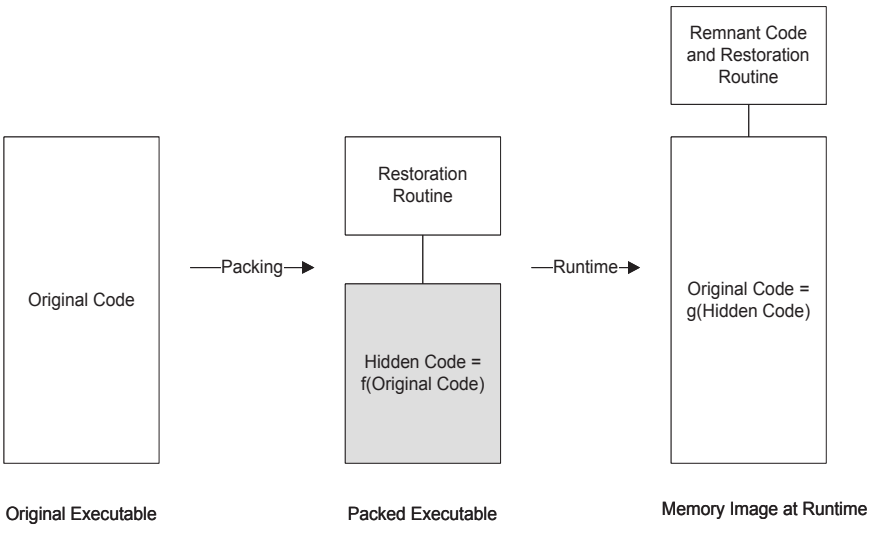

Fig. 15. The traditional code packing transformation.

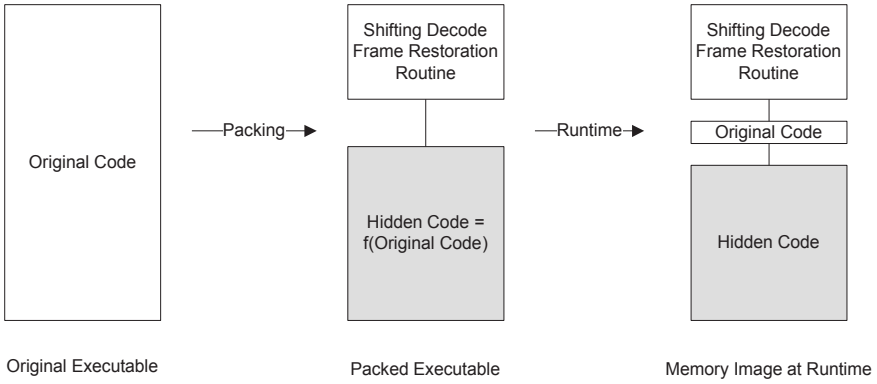

Fig. 16. Code packing using the shifting decode frame.

memory can be decrypted as needed and subsequently re-encrypted to prevent an analyst or automated system from having access to all the hidden code at any single moment in time. This technique is known as the shifting decode frame [30]. The granularity of encryption can occur at the page level, the basic block level, and the instruction level. This type of code packing is not often used in wild malware, and in practice, traditional code packing and instruction virtualization are the dominant techniques used in real malware. The process of this form of malware packing is shown in Fig. 15..

2.2.4.11 Instruction Virtualization and Malware Emulators

From [28]: Code packing may employ the use of instruction virtualization also known as a malware emulator [21]. An emulator used by a malware should not be confused with an emulator used for automated unpacking of the malware. This type of code packing transformation employing an emulator is used in a minority of malware. In this form of code packing, packing translates the original native code into a byte-code which is subsequently emulated by the malware at run-time. Using this form of code packing, the hidden code in

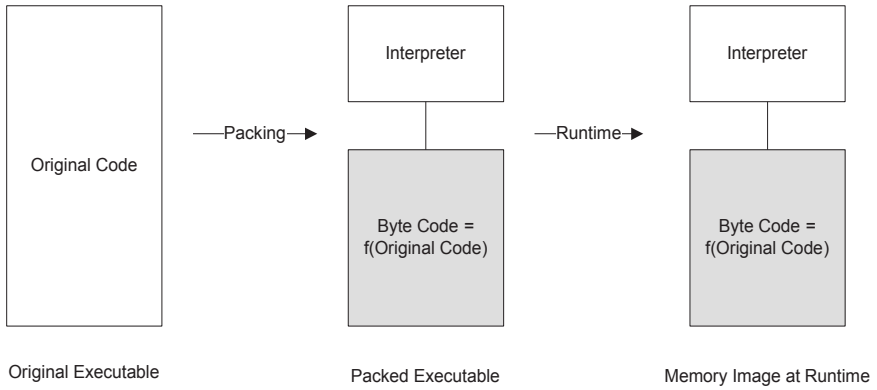

Fig. 17. Code packing using instruction virtualization.

its original form is never revealed. The process of this form of malware packing is shown in Fig. 16.

2.2.5 Features under Program Transformations

Program features may change under program transformations and obfuscation. The challenge then is in choosing features which remain invariant under these conditions. The raw or byte level content deals poorly with program transformations. Small changes in high level source code may result in large changes in the raw content. Instruction level content is also prone to large changes under transformations such as when registers are reassigned or the instruction stream is modified. Control flow is more invariant than most syntactic features and can be a good choice. At a source code level, program and system dependency graphs have been popular. The APIs used by a program represent a good choice and have been widely used in behavioural analysis of malware. For static analysis of malware, the malware must be unpacked to reveal its hidden code. Unpacking of malware is not addressed in this book.

2.3 Formal Methods of Program Analysis

Feature extraction is a necessary component to construct a birthmark, show similarity and classify a program as belonging to a particular class. Program analysis is an important component in feature extraction. The analysis reveals information on the syntax, semantics, and behaviour of the program being inspected. This section focuses on formal methods of program analysis which can be used for the purpose of property and feature extraction.

2.3.1 Static Feature Extraction

The majority of formal methods we will examine in this section are based on analysing a static view of a program without performing execution of it. A number of possible choices exist to perform feature extraction statically from a program. There is some equivalence between source code and binary feature extraction, however differences also exist.

The possible stages to extract static features from source code are:

- Raw Code Analysis

- Lexical Analysis
- Parsing
- Static Program Analysis

For binary only software, analyses can be divided into:

- Raw Code Analysis
- Object File Parsing
- Static Program Analysis of Binaries
- Decompilation

Static program analysis is an approximation of program behaviour. For an analysis to be sound, then no behaviour should be omitted. For an analysis to be precise, the over-approximation should be close to the actual behaviour. This over approximation leads to false positives in the case of bug detection, or conservation optimisations in the case of compiler techniques. A perfectly precise analysis is undecidable due to Rice's theorem [31], however even without perfect precision the results are still practical and useful.

2.3.2 Formal Syntax and Lexical Analysis

Lexical analysis is the process of producing a sequence of tokens given a sequence of characters. Lexical analysis is performed before parsing. The parser uses the tokens generated from the lexical analysis.

2.3.3 Parsing

Definition 20. *A context-free grammar G is defined by the 4-tuple:*

G=(V,Σ,R,S) where
V is a finite set of non terminal variables.
Σ is a finite set of terminals.
R is a finite set of rules or productions of the grammar.
S is the start variable.

Rules are of the form $V \rightarrow w$ where V is a non terminal symbol and w is a string of terminals and/or non terminals.

```
digit          [0-9]

letter         [a-zA-Z]

%%

"<="                  { return LEQ;         }

">="                  { return GEQ;         }

"begin"               { return BEGINSYM;    }

"call"                { return CALLSYM;     }

"const"               { return CONSTSYM;    }

"do"                  { return DOSYM;       }

"end"                 { return ENDSYM;      }
```

Fig. 18. Implementation of lexical analysis.

Context-free grammars are the basis for recognizing and representing programming languages in source code. However, in practice, a number of widely used languages such as C++ are not strictly context-free in all cases.

The process of parsing in static analysis is to transform source code into a concrete or abstract syntax tree.

2.3.4 Intermediate Representations

2.3.4.1 Intermediate Code Generation

The process of code generation is typically performed by traversing the abstract syntax tree and generating intermediate code for each unit in the tree.

2.3.4.2 Abstract Machines

The intermediate language used for the intermediate code runs on an abstract machine that has a correspondence to the actual machine. Typical models of computation for the

```
input:

        expr { ((SParserParam*)data)->expression = $1; }

        ;

 expr:

      expr TOKEN_PLUS expr { $$=createOperation( ePLUS, $1, $3 ); }

    | expr TOKEN_MULTIPLY expr { $$=createOperation( eMULTIPLY, $1, $3 ); }

    | TOKEN_LPAREN expr TOKEN_RPAREN { $$=$2; }

    | TOKEN_NUMBER { $$ = createNumber($1); }

;
```

Fig. 19. Implementation of parsing.

abstract machine are register machines or random access machines. A typical implementation useful for static analysis consists of:

- An unlimited number of uniquely labelled registers.
- A small number of instruction prototypes to make an instruction set.
- An instruction pointer.
- A sequence of labelled instructions.
- A random access memory.
- An entry point.
- The instruction set can further be divided into:
- Data (arithmetic etc)
- Control (conditional and unconditional branching etc)

- API Calls (operating system and library interface etc)

2.3.4.3 Basic Blocks
To partition the intermediate code into basic blocks [32] we determine instructions that are leaders. Leaders are the first instruction in each basic block. An instruction is a leader when it satisfies one of the following properties:

- The first instruction in the intermediate code.
- Any instruction that is the target of a branch.
- Any instruction that follows a branch.

2.3.4.4 Control Flow Graph
The successors of a basic block b, succ(b), are:

- The target of the basic block's branch instruction.
- The basic block immediately following the current basic block in the instruction stream.

Thus, a control flow graph [32] is defined as the directed graph C=(B,E) such that B is the set of basic blocks, and $E = \{(u, v) \mid u \in B, v \in succ(u)\}$

2.3.4.5 Call Graph
The successors of a procedure f, call_succ(f), are:

- The set of call targets in the procedure body.

Thus, a call graph is defined as the directed graph CallGraph=(F,E) such that F is the set of procedures, and $E = \{(u, v) \mid u \in F, v \in call\_succ(u)\}$

2.3.5 Formal Semantics of Programming Languages
The formal semantics of programming languages aims to rigourously reason about program meaning by having a strict mathematical representation of a program's semantics. Multiple methods are available to represent program semantics and the three main techniques are:

- Operational Semantics
- Denotational Semantics
- Axiomatic Semantics

Other approaches are also possible, including algebraic semantics [33] which has been used successfully to show equivalence between code fragments of metamorphic malware.

2.3.5.1 Operational Semantics

Operational semantics capture the state transition that occurs when a program instruction is executed. It can be thought of as defining an interpreter for a language [34]. Operational semantics can be expressed using the following notation:

$$\frac{\begin{array}{l} premise_1 \\ \ldots \\ premise_n \end{array}}{(i,P) \Rightarrow P'} NAME$$

Where i is the current instruction, P is the current state and P' is the next state following execution of the instruction i.

2.3.5.2 Denotational Semantics

Denotational semantics transform instructions to mathematical objects [34]. It can be thought of as defining a compiler for a language.

2.3.5.3 Axiomatic Semantics

Axiomatic semantics give an axiomatic basis for a program. Typically this is achieved by using preconditions and postconditions for instructions. These preconditions and postconditions can be analysed with logic, typically first order logic. The most common use of axiomatic semantics is to prove program correctness using Hoare logic [35] and its variants.

2.3.6 Theorem Proving

2.3.6.1 Hoare Logic

Hoare logic is a means for proving the correctness of structured programs [35]. It is based on axiomatic semantics. Hoare logic provides a deductive method for proving correctness,

however loop invariants must be synthesised and this represents a significant challenge in developing program proofs.

2.3.6.2 Predicate Transformer Semantics

Predicate transformer semantics [36] provide a method to generate verification conditions through the weakest precondition. This is a form of axiomatic semantics and reformulates Hoare logic to provide an automated construction of first order logic formula to prove program correctness.

2.3.6.3 Symbolic Execution

Symbolic execution [37] is the process of executing a program using symbolic represents for variables and data. The program executes by generating constraints of the symbols for each instruction. Mixed symbolic execution [38] allows a more efficient implementation by concretely executing part of the program using native computations, and symbolically execution those variables of interest. Symbolic execution is path based execution. At every control transfer point, a decision must be made of which path to follow. The feasibility of paths and the symbolic constraints are modelled using an SMT decision procedure. The decision procedure can report if a set of constraints is feasible, or provide a counter example to prove otherwise. Symbolic execution has been applied to binaries for applications such as malware analysis [39].

2.3.7 Model Checking

Model checking is used to verify that a model meets the properties of a specification [40]. It achieves this by enumerating the state space of the model to verify the specification.

2.3.8 Data Flow Analysis

Data flow analysis tries to statically determine the behaviour of data [9]. Perfectly precise data is undecidable so data flow analysis seeks to find an approximation of the data by discovering conservative program invariants. Data flow analyses are flow-sensitive which means the ordering of instructions is taken into account. The solution of data flow problems is based on lattice and order theory. The problems are represented as monotone functions which can be approximated and computed using fixed point solutions.

2.3.8.1 Dataflow Equations

Dataflow analysis is performed by reaching a fixpoint solution in a semilattice for a system of monotone equations that describe the dataflow. Typical data flow analyses require control flow information to perform the analysis. The basic approach is to set up data flow equations to track data entering and leaving each node in the control flow graph. In a forward flow analysis, a transfer function is applied on the data entering a basic block which results in the data leaving the basic block. Merging of control flow edges is applied using a join operator. The analysis can be forwards or backwards merging successor or predecessor nodes. In some literature a meet operator is used instead of a join. This is arbitrarily dependent on whether a meet-semilattice or join-semilatice is used for analysis.

In a forward analysis using a join-semilattice, for each block b:

$$out_b = transfer\_function(in_b)$$
$$in_b = join(\{p \mid p \in predecessor_b\}, out_b)$$

A backwards analysis replaces in with out, and out with in. It also uses the successor blocks instead of the predecessor blocks in the join.

Typical join operators include union or intersection. Data flow analyses are usually constructed to be conservative so that precision is sacrificed to capture all possible behaviours. The analysis proceeds by iteratively computing the functions for all blocks until a fixed point is reached.

2.3.8.2 Dataflow Analysis Examples

Common data flow analyses include reaching definitions and live variable analysis. These analyses are use-def analyses. They resolve the problem of identifying which instructions subsequently use a variable as in the case of liveness and upwards exposed uses, or which variable definitions reach an instruction as in the case of reaching definitions. There may be more than one reaching definition of the same variable at an instruction if multiple paths lead to that instruction and the same variable is defined along those separate paths.

If an accurate control flow graph is available, then data flow analysis performs equally accurate. Data flow analyses has been heavily used in the decompilation of binaries [41]. If

data flow analyses is performed interprocedually, then the call graph must be accurately generated.

2.3.8.3 Reaching Definitions

The lattice for reaching definitions is the power set of definitions ordered by set inclusion. The data flow equations for reaching definitions are:

$$REACH_{OUT}[S] = GEN[S] \cup (REACH_{IN} - KILL[S])$$
$$REACH_{IN}[S] = \bigcup_{p \in pred[S]} REACH_{OUT}[p]$$
$$GEN[d : y \leftarrow f(x_1,...,x_n)] = \{d\}$$
$$KILL[d : y \leftarrow f(x_1,...,x_n)] = DEFS[y] - \{d\}$$

where DEFS[y] is the set of all definitions that assign to variable y. d is a unique label attached to the assigning instruction.

2.3.8.4 Live Variables

The lattice for live variable analysis is the power set of used variables ordered by set inclusion. The data flow equations for live variable analysis are:

$$LIVE_{IN}[S] = GEN[S] \cup (LIVE_{OUT} - KILL[S])$$
$$LIVE_{OUT}[final] = 0$$
$$LIVE_{OUT}[S] = \bigcup_{p \in succ[S]} LIVE_{IN}[p]$$
$$GEN[d : y \leftarrow f(x_1,...,x_n)] = \{x_1,...,x_n\}$$
$$KILL[d : y \leftarrow f(x_1,...,x_n)] = DEFS[y] - \{y\}$$

2.3.9 Abstract Interpretation

Abstract interpretation [42] is closely related to data flow analysis. Abstract interpretation concerns to the sound approximation of programs. A classic example of abstract interpretation used for pedagogical purposes is the abstract domain of signs which represents numerical variables by the possible sign they have. A variable may be positive, negative, possibly both, or zero. Abstract interpretation has been applied to, in amongst other things, malware detection.

2.3.10 Intermediate Code Optimisation

Data flow analysis is used in intermediate code optimisation. A very small set of possible optimisations are:

- Dead Store Elimination
- Constant Folding
- Copy Propagation

For example, in dead store elimination, if a variable is defined, but is not live, then the definition can be safely removed from the code.

2.3.11 Research Opportunities

Algebraic semantics [43] have been used to show equivalence between metamorphic malware. However, the general approach of using formal semantics to show semantic equivalence between programs is under-utilised. We believe this presents an opportunity for researchers looking at the software similarity problem in future work. We tackle part of this problem and propose using operational semantics to analyse malware codes in Chapter 4. The notion of non exact matching of semantics is an area that needs investigation if we are to detect similar but not identical program copies.

2.4 Static Analysis of Binaries

Static binary analysis is more difficult than if source code is available. In many cases, the analyses are unsound and behaviours are omitted to make problems feasible. Heuristics may be required to separate code and data in a disassembly or pointer behaviour may be weakly modelled to make statically analysing programs feasible. Nevertheless, static analysis of binaries is an important area of research with a number of practical applications including the detection of software theft and the classification and detection of malware. This section examines static analysis of binaries with the intent that properties and features of binary programs can be extracted to create useful birthmarks for software similarity and classification.

```
disassemble_program(program)

{

  address = disassemble_linear_sweep(start(program), end(program))

}



disassemble_linear_sweep(start, end) {

  address = start

  while (address < end) {

    instruction = Disassemble(program, address)

    if (error) {

      address += 1;

    } else {

      disassembly[address] = instruction;

      address += length(instruction);

    }

  }

}
```

Fig. 20. Linear sweep disassembly.

2.4.1 Disassembly

Disassembly is the process of translating machine code to assembly language [44]. This is typically the first stage of a static analysis. Static disassembly parses the entire binary image statically without execution. In static disassembly, there are two main algorithms. In the Linear Sweep algorithm, the instructions are disassembled one instruction after another, starting from the beginning of code. The disadvantage of this method is that data introduced into instruction stream may be erroneously disassembled.

```
disassemble_program(program) {

  disassemble(entry_point(program))

}

disassemble_recursive_traversal(address) {

  while (has_address(program, address)) {

    if (disassembly[address] not null)

     return

    instruction = Disassemble(program, address)

    if (error)

      return

    disassembly[address] = instruction

    if (is_return_instruction(instruction))

      return

    if (is_transfer_instruction(instruction))

       disassemble(transfer_target(instruction);

    address += length(instruction);

  }

}
```

Fig. 21. Recursive traversal disassembly.

The other main algorithm to perform disassembly is the Recursive Traversal algorithm. This algorithm decodes each instruction following the order of the control flow. This resolves the issue of embedded data, but may miss decoding instructions that are the target of indirect jumps or other situations when it is hard to resolve control flow statically.

```
disassemble_speculative(program) {

  disassemble_recursive_traversal(entry_point(program))

  for all intervals in

    [start(program), end(program)] and not in disassembly

  {

    disassemble_linear_sweep(

      start(interval), end(interval))

  }

}
```

Fig. 22. Speculative disassembly.

Speculative Disassembly attempts to remedy the problems of the Recursive Traversal algorithm problem by first performing the Recursive Traversal, and then performing a Linear Sweep in regions that are not decoded.

Disassembly results in the following data.

$$disassembly = \{address, opcode, operand_1, ..., operand_n\}$$

2.4.2 Intermediate Code Generation

A simple approach to transforming assembly into an intermediate language is to translate each instruction without maintaining intermediate state. This approach has been used successfully in the Reverse Engineering Intermediate Language (REIL) [45]. Other popular intermediate languages are Vex as used in the Valgrind binary instrumentation framework [46] and Vine as used in the BitBlaze project [47]. An example to translate native assembly into three address code is shown below.

$$native\_assembly\_instruction \rightarrow (TAC_1, ...TAC_n)$$

```
disassemble_procedure(address) {

  while (has_address(program, address)) {

    if (disassembly[address] not null)

     return

    instruction = Disassemble(program, address)

    if (error)

      return

    disassembly[address] = instruction

    if (is_return_instruction(instruction))

      return

    if (is_transfer_instruction(instruction)

        and not is_call_instruction(instruction))

      disassemble_procedure(transfer_target(instruction);

    address += length(instruction);

  }

}
```

Fig. 23. Procedure disassembly.

2.4.3 Procedure Identification

An important stage in reconstruction the control flow of an executable is identifying procedures. There are roughly four approaches that can be employed.

- Using object file format information (e.g., symbols and exports)
- Using static targets of call site $F = \{f \mid (address, call\_direct, f) \in disassembly\}$
- Using idioms to identify procedure prologues

- Using static analysis and data flow analysis to reconstruct indirect call targets

The main hindrance to generating accurate representations is when a program uses indirect branches and procedure calls. The analysis of indirect targets requires data flow analysis. A number of approaches have been employed [48-50]. Using idioms to identify procedures requires string matching algorithms to identify common byte sequences.

2.4.4 Procedure Disassembly

Procedures consist of a body of instructions which must be recovered from the disassembly. The algorithm is a very slight variation of the recursive traversal disassembly algorithm. The difference is that inter procedural control flow is not traversed.

2.4.5 Control Flow Analysis, Deobfuscation and Reconstruction

Control flow analysis is more difficult on binaries because of the difficultly in separating code and data. Likewise, the presence of indirect branch and call targets in assembly language makes precisely determining the static control flow undecidable.

The simplest approach is to ignore indirect targets completely. The edges of the graphs representing the call graph control flow can be constructed by connecting the call site to the static call target. For control flow graphs the approach is similarly applied to branch targets.

Control flow may also be obfuscated. An opaque predicate [19] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in a binary's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis.

The presence of opaque predicates in a control flow graph reduces the accuracy of the graph because of misleading branch targets. In [51] it was proposed to use the program analysis technique of abstract interpretation to detect specific classes of opaque predicate algorithms.

2.4.6 Pointer Analysis

Pointer and alias analysis tries to determine the variables that a pointer may point to. In assembly this problem is difficult. A conservative approach to alias analysis of assembly using datalog constraints was proposed in [52], however this work was to introduce formal rigour and is not practical to deploy. Value-Set Analysis [53] has been proposed as an alias analysis, suitable for binary programs and assembly language. Value-Set Analysis has been used in malware detection [54] and the automated static unpacking of malware [55].

2.4.7 Decompilation of Binaries

Decompilation [41] is the process of recovering source code from executable binaries. In general, decompilation can be seen as a form of static analysis of a binary that recovers additional information from its intermediate representation. Research connecting the type of static analysis a compiler performs to the requirements of a decompiler was proposed in [41] and [56].

2.4.7.1 Condition Code Elimination

In Instruction Set Architectures such as x86, many arithmetical instructions modify a status flag or condition code. For example, determining if two variables are equal is divided into two computations. An arithmetic instruction over the two variables that sets a condition code, and then a branch based on the resulting condition code. Decompilation requires these two computations be reduced to one conditional test.

An approach to solve this is by maintaining a reaching definition of the various conditions code set by each arithmetic instruction. At the point of a conditional branch based on the condition code, the reaching definitions are combined into a single condition.

2.4.7.2 Stack Variable Reconstruction

Stack variable reconstruction transforms variables allocated on the stack into native variables in the intermediate representation. The stack can be accessed in two main ways. The first method is by referencing variables relative to the top of the stack, or stack pointer. The second method accesses the stack relative to the frame pointer. The frame pointer is unique for each procedure or activation record. It points to the top of the stack as set on function entry. During procedure execution the stack pointer may change, but the frame pointer remains constant. This simplifies access to variables on the stack and is often used

in debug builds of application. It is clear that for a decompiler to be effective, it must handle both methods of accessing the stack. Both frame and stack based addressing may be intermixed in real life applications.

Another complication to using the stack pointer is that callees may or may not change the stack pointer. It is the responsibility of the caller to push arguments onto the stack, but the callee may or may not unwind these arguments based on the calling convention being used.

One approach [57] to reconstruct stack based variables takes advantage of the fact that in compiled programs, the position of the stack pointer in each basic block remains constant. The stack pointer can be modified within a basic block when calls are made or values or pushed and popped on or from the stack. Using this information, a set of constraints over the control flow graph can describe the stack pointer. Solving the constraints identifies the relative position of the stack pointer at the entry and exit of each basic block. Frame pointer relative addressing uses fixed offsets from the top of the stack at the beginning of the procedure, and knowing the position of the stack pointer at each basic block enables knowing exactly which memory location on the stack is being referenced. This enables a unified approach to modelling stack and frame based addressing.

Pointers and arrays complicate the process of stack variable reconstruction. In these cases, the stack variable may only be referencing the beginning of an array or pointing to the beginning of the object. Heuristics must be used to estimate the size of the object. An approach to estimate this is by looking at the size of the stack frame or looking at the next adjacent stack reference to predict a bounds on the object in question.

2.4.7.3 Preserved Register Detection

A typical problem that arises is determining if the register is modified in the life time of a procedure. If the register is used in procedure, but maintains its original value once returning from the procedure's callsite then the register is preserved. The process of preserving a register is to copy the register into a temporary variable and then restore it before leaving the function. Detecting preserved registers is important in the process of identifying which registers are arguments or return values from a procedure.

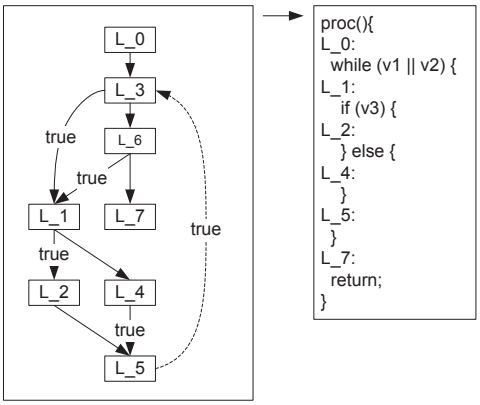

Fig. 24. A control flow graph and its linearized form.

Data flow analysis and a suitable intermediate representation can help solve the preserved register problem. If we ignore calls within a procedure, we can identify a preserved register by the fact that the reaching definitions for that register at each function exit, is the value of a copy of the register on function entry. To determine where the value is copied on entry to the function we can use a liveness analysis to identify where the register is used and check that instruction for a copy instruction.

This process of identifying preserved registers requires that local variable reconstruction be performed. The reason is that the temporary variable used to save a copy of the preserved register is typically represented by a local variable.

2.4.7.4 Procedure Parameter Reconstruction

The parameters to procedures may be passed on the stack, or passed via registers. The return values are typically passed by registers. The exact semantics are defined the calling convention on a particular procedure. The arguments used by a procedure can be determined by the procedure accessing variables outside the current stack frame. Once the arguments are known, at call sites, the stack is statically unwound to the required depth to retrieve them.

Registers may also be passed as arguments. Ignoring calls, arguments are registers that are live on procedure entry that aren't preserved. To take into account calls, the analysis is performed on inner calls first as defined by their depth first order in the call graph. Recursive calls require further analysis.

2.4.7.5 Reconstruction of Structured Control Flow

A standard technique in decompilation is transforming a control flow graph into higher level structured control flow [41, 58, 59]. This is the process of structuring. Identifying conditions, loops, and parts of the control flow graph that cannot be structured is required. Conditions may be compound conditional statements involving conjunction and disjunction. The higher the quality of structuring means the less the number of gotos in the generated code. Some graphs cannot be structured and the reducibility of the graph identifies these cases.

Structuring of control flow graphs was proposed in [60, 61] to generate string signatures that were later used to identify malware variants.

2.4.7.6 Type Reconstruction

Type information is lacking from binaries. Reconstruction of types enables higher quality code in the decompiled output. An approach to type reconstruction using the unification algorithm was proposed in [62]. A data flow analysis approach based on lattices and using single static analysis was proposed in [56].

2.4.8 Obfuscation and Limits to Static Analysis

It is known that perfectly precise disassembly is undecidable [63]. Branch targets can be indirect, and precise understanding of those run-time values can be problematic. In [64] an analysis of some limits to static analysis of malware were identified. The use of opaque predicates was shown to confound the problem of precise program representation. Determining whether two programs are semantically equivalent is also known to an undecidable problem which is why for example malware detection is often based on heuristic and unsound solutions. Likewise, perfect decompilation, for all possible binaries, is undecidable. If the binary does not originate from high level source then it is unlikely decompilation will give meaningful results.

2.4.9 Research Opportunities

Decompilation presents potential research opportunities when combined with other techniques such as static analysis or malware classification. Very little research has been performed on decompilation-based applications. The main application of decompilation thus far has been source code recovery. However, the high level information it recovers makes it a suitable abstraction for useful software features. In Chapter 5, we propose

extending decompilation-based approaches to malware variant detection. To achieve this task, we propose in Chapter 4 a novel formal intermediate language for binary analysis. Our intermediate language uses high level concepts from decompilation and bridges the gap between binary and higher level analyses.

2.5 Dynamic Analysis

In the previous sections we have examined static extraction of program features for the purpose of birthmark construction. Dynamic analysis is examined in this section. It is an alternative approach to static analysis that can be used for birthmark construction. Dynamic analysis concerns itself with analysing a running program. The program being run is typically isolated in an environment which allows its behaviour to be inspected. Typical behaviours that are extracted are the API call sequence. Instruction sequences, basic block sequences and control flow are amongst other behaviours that can also be identified.

2.5.1 Relationship to Static Analysis

There are roughly two approaches to extract program features from software. In the static approach, the software is never executed and the features are extracted from a static view of the program. In dynamic analysis the software is executed, possibly in a virtual machine, and its run-time behaviour examined. The run-time behaviours exhibit the properties or features being extracted.

Static analysis is effective because it is able to examine to represent the set of all possible execution paths by approximating program behaviour. This is important because behaviours of specific programs may be hard to trigger dynamically. It is often difficult to trigger corner cases in programs and as a result a number of dynamic analysis testing methodologies exist to address this such as the use of analysing code coverage during execution. In the case of malicious code, malware authors actively change the behaviour of the code when under analysis.

The main advantage of dynamic analysis is that the semantics of the program are exhibited, and obfuscations applied to the program have less effect on these exhibited semantics. Attempting to identify run-time behaviour properties for multiple paths of execution has been researched [39]. It is still a new area, but using symbolic execution to

trigger different behaviours has had some success. The results of exploring these multiple paths can be accumulated into a final report to infer the intent or potential behaviour of a piece of software.

2.5.2 Environments

Dynamic analysis requires an environment in which to run and isolate the program being analysed. The environment in which to run a program can be categorized in the following list:

- Hooking
- Dynamic Binary Instrumentation
- Virtualization
- Application Level Emulation
- Whole System Emulation

2.5.3 Debugging

An operating system typically provides an API to debug processes. Debugging can allow for operations including single stepping through execution an instruction at a time, or setting a breakpoint at a particular code address. Debugging can be useful to monitor non malicious programs, however most malware today implements anti-debugging functionality which can detect the presence of a debugger.

2.5.4 Hooking

Hooking is the process of intercepting API calls allowing for possible instrumentation. Hooks can be placed in user space or kernel space. Hooking is commonly used by commercial Antivirus software to monitor process behaviour and detect possible misuse. Detours [65] is an implementation of hooking for the Windows operating system. The basic mode of operation is to overwrite the function in memory with a trampoline to the intercept handling code. The intercept handling code performs any instrumentation or monitoring as necessary then restores control back to the original function. Another method of hooking is overwriting dispatch tables such as system call tables or import addresses. It is also

possible in Linux to natively intercept API calls to dynamic libraries by preloading another library. Malware today often can detect the presence of hooking by implementing checksums over their executable code.

2.5.5 Dynamic Binary Instrumentation

Dynamic binary instrumentation is an approach that instruments native code on the fly. The binary being executed is controlled from a dispatcher which analyses the code, instruments it, and then rewrites it for execution. Some examples of dynamic binary instrumentation include PIN [66], DynamoRIO [67], and Valgrind [46]. Dynamic binary instrumentation based on PIN has been used for malware unpacking and analysis in [68, 69].

2.5.6 Virtualization

Virtualization is a technique that supports native execution of a guest operating system by exploiting separation and isolation mechanisms implemented by the native hardware architecture or software. A number of methods are available to implement virtualization including paravirtualization which must be supported by both the host and the guest operating systems. The most important type of virtualization for providing an environment to perform feature extractions is hardware assisted virtualization. In the x86 architecture, hardware assisted virtualization was not always supported and detection of the virtualized environment was implemented by many strains of malware [68]. Hardware assisted virtualization has been used for malware analysis [70]. This type of analysis is harder to detect but attacks still exist to detect virtualization from a guest [71]. For example, it is known that memory caching between guests and hosts are different in the virtualized environment. However, as virtualization becomes a standard tool on the desktop, malware authors might no longer be able to associate virtualization with threat analysis.

2.5.7 Application Level Emulation

Application level emulation emulates the operating system and instruction set architecture for specific applications. This approach has been predominantly employed in Antivirus systems to perform real-time analysis of malware and automated unpacking [60]. Its main disadvantage is its inability to faithfully emulate the desired system which makes it susceptible to detection as has been the case with modern malware.

The typical features emulated in an application level emulator on the x86 Windows platform for the purposes of malware detection include:

- Instruction Set Architecture (ISA).
- Virtual Memory.
- Windows API emulation.
- Linking and Loading.
- Thread and Process Management.
- OS Specific Structures.

The instruction set architecture (ISA) must be faithfully emulated. In practice, most deployed emulators only simulate part of the complete x86 ISA. Malware authors have responded by using uncommon instructions such as those associated with MMX and FPU to detect and thwart the emulation process.

Virtual memory must be emulated. 32-bit x86 employs a segmented memory architecture. In Windows the segment registers are utilised to reference thread specific data. This data is additionally used by Windows Structured Exception Handling (SEH). SEH is used to gracefully handle abnormal conditions such as division by zero and is routinely used by packers and malware to obfuscate control flow.

The Windows API is the official system call interface provided by Windows. There are too many Windows API functions to full emulate in a typical environment so only the most common APIs are implemented. This also presents a method for malware to detect and thwart an emulator using uncommon API calls.

Linking and loading must be implemented by an emulator. Program loading entails allocating the appropriate virtual memory, loading the program text, data and dynamic libraries. Relocations must be performed and run-time linking performed.

Threads and process management must be performed. Malware can sometimes try to detect and thwart a debugger or emulator by being multi-process or multi-threaded.

OS specific structures must also be simulated. Windows has a number of these including the Process Environment Block, the Thread Environment Block and the Loader Module. These structures are visible to applications and can be used by malware.

2.5.8 Whole System Emulation

A whole system emulator emulates the hardware of a PC. This allows an operating system to be installed as a guest. There are roughly two approaches to implement a whole system emulator or any emulator in general:

- Interpretation
- Dynamic Binary Instrumentation

An example of whole system emulators includes QEMU [72] which is based on dynamic binary translation. Bochs is another whole system emulator that uses interpretation instead of dynamic binary translation. Bochs has been used for malware unpacking and analysis [30]. Interpretation is slower than dynamic binary translation which makes QEMU a popular choice.

Interpretation works by implementing a fetch, decode and execute loop inside the emulator. Dynamic binary translation translates sequences of code from the guest into native code on the host. It can perform optimisations on these blocks of code which improves efficiency. The blocks are also cached reducing the costs of translation. In general, dynamic binary translation offers significant performance improvements over an interpretation based emulator.

It is possible to modify a whole system emulator to monitor or instrument guest execution [73]. The BitBlaze project [47] is a project for binary analysis that makes heavy use of whole system emulation to perform tasks including malware analysis. Whole system emulation is effective for behavioural analysis of code but attacks exist to detect its presence from the guest [71].

2.6 Feature Extraction

To recap the survey so far, we have examined static and dynamic methods of program analysis. These features must be translated into mathematical representations and birthmarks to be useful. Furthermore, mathematical representations may be embedded in other mathematical types to make birthmarks more amenable to similarity comparisons and for use in classification algorithms. Another approach is to represent features using kernels. This allows for the use of classification algorithms including the support vector machine for complex data types. This section examines the mathematical representations that we use to describe program features.

2.6.1 Processing Program Features

Program features are the basis of software similarity and classification, but must be transformed or into a meaningful representation that allows for similarity comparisons and indexing. Different representations are possible ranging from highly efficient but least expressive, to highly expressive but least efficient. For example, representing birthmarks as vectors allows for very efficient comparisons, but tends to lose structural information that is present in graph based representations.

Combining features into a unified form may result in the establishment of software metrics. Attribute counting is one approach. Attributes that can be tallied might include the number of specific keywords, the number of conditionals, the number of loops and so forth. The final metric is the set of counted attributes. Processing might be done on these counted attributes to result in other measures. The Halstead complexity measures [74] are a set of software metrics that uses attribute counting at its core to give a measure on a programs complexity. Its initial use was for the purpose of software maintenance metrics but it has also been applied to software similarity.

Another approach to combine the expressiveness of complex objects, such as graphs, is to transform or embed one representation into another. For example, a graph can be transformed into a vector based representation. Information is lost, but in many cases this is still useful as a birthmark.

2.6.2 Strings

A string describes a sequence of tokens or characters. An example of a string could be a sequence of instruction opcodes making up a program path.

Definition 21. *Let* $\sum$ *be an alphabet of symbols .Let s be a string over the alphabet where* $s \in \sum^*$.

2.6.3 Vectors

Vectors are one of the simplest representations and are efficient to work with. A vector is an an ordered list or tuple of a fixed number of elements or dimensions. A feature vector describes the frequency of particular features occurring. If the number of features is very large then dimensionality reduction can be used to filter unimportant features, or combine features together such as when using Principle Component Analysis (PCA).

Examples of using vectors include describing features based on the occurrence of a specific n characters or n-grams.

2.6.4 Sets

A set is a collection of unique objects. A set of features is sometimes a useful representation. It ignores ordering of those features. An example use of sets is to describe the set of API calls a program makes.

2.6.5 Sets of Vectors

A set of vectors may sometimes be useful. If we consider that a procedure can be represented as a vector, then the set of procedures can be represented as a set of vectors.

2.6.6 Trees

Trees capture the structure of data, but are not as general as graphs. A tree is a connected undirected graph without cycles. Abstract syntax trees and parse trees are naturally represented by trees. Structured control flow can also be represented by trees. Trees can have a defined ordering of child nodes or be unordered.

2.6.7 Graphs

Graphs model structure in the data. Many program features are naturally represented as graphs include control flow graphs, call graphs, and dependency graphs.

Definition 22. *A graph is g=(V,E) where V is a set of vertices.* $E = \{(u,v) \mid u,v \in V\} \subseteq V \times V$

Definition 23. *A labelled graph* $g = (V, \alpha, \beta)$ *where V is a set of vertices* $\alpha : V \to L$ *is the node labelling function, and* $\beta : V \times V \to L$ *is the edge labelling function.*

2.6.8 Embeddings

Strings may be embedded in vectors. To reduce the string problem into an n-gram vector problem, a string may be divided into n-grams where the specific n-grams represent features.

Definition 24. *Given a set of strings L, and a set of vectors V there is a function f such that* $f : L \to V$

Strings may be embedded in sets. To reduce the string problem into a set problem, a string may be divided into n-grams or shingles where the unique n-grams represent set elements.

Definition 25. *Given a set of strings L, and a set of sets S there is a function f such that* $f : L \to S$

Trees may be embedded in vectors. A tree may be decomposed into fixed sized subtrees. These subtrees can represent features in a feature vector.

Definition 26. *Given a set of trees T, and a set of vectors V there is a function f such that* $f : T \to V$

2.6.9 Kernels

Kernels are most used in kernel based statistical machine learning classifiers. A kernel function operates in feature space which is typically of much higher dimensionality. A string kernel based on the subsequences in the string known as a subsequence kernels was proposed in [75]. A kernel for sets of features was proposed in [76]. A kernel for vector sets was proposed in [77]. A kernel for trees was proposed in [78]. A kernel based on random walks in a graph was proposed in [79]. Subtree kernels have been proposed. A kernel based the set of all paths in a graph has also been proposed. A kernel based on the shortest paths in a graph was proposed in [80].

2.6.10 Research Opportunities

Embeddings and kernels present a significant opportunity for researchers. Embeddings have been investigated somewhat, but a comprehensive treatment of different embeddings for different structures has not been performed in the context of software similarity. In Chapter 5, we propose new methods of embedding structured data into vectors. We

propose approximating graphs as vectors and this allows us to implement a very efficient malware variant detection system. Kernel methods are effectively unused in software similarity and this presents many opportunities for researchers to apply kernel methods to so the software similarity and classification problem. Graph kernels could be used to perform software classification in applications such as malware classification.

2.7 Software Birthmark Similarity

Comparing birthmarks is necessary to identify similarities between software. If two birthmarks are similar, then the software is similar. Birthmarks may be compared to show similarity, or an alternative to showing similarity is to show dissimilarity or distance. Similarity measures and metrics exist for the different types of data such as strings, vectors, trees, graphs, etc. This section examines the different similarity measures and metrics for the different classes of birthmarks.

Keywords: Software similarity, birthmark similarity, distance metrics, string similarity, vector similarity, set similarity, set of vectors similarity, tree similarity, graph similarity.

2.7.1 Distance Metrics

Definition 27. *A metric on a set X is a function (known as the distance function or distance):*

$$d : X \times D \to \mathbb{N}$$

For all x, y, z in X, this function is required to satisfy the following conditions:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0$ *iff x=y*
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$ *(triangle inequality)*

If the distance function has the properties of a distance metric then indexing and searching a database can be performed more efficiently. Therefore it is beneficial to compare software using distance functions that are metric. Examples of metric access methods are in [81-83].

2.7.2 String Similarity

Strings can be compared using string metrics. The Levenshtein distance between two strings defines the number of edit operations that must be performed to transform one string to the other. An edit operation includes character insertion, deletion, and substitution. Other string metrics include the Smith-Waterman algorithm which is used to perform local string alignment, or using the longest common subsequence. Optimal solutions to edit distance and alignments are normally O(n.m) where n and m are the lengths of each respective string. The solutions are typically implemented using dynamic programming. The Levenshtein distance, Smith Waterman distance and Normalized Compression Distance are all metric.

2.7.2.1 Levenshtein Distance

Definition 28. *For two strings s and t, the Levenshtein distance is measured as follows:*

D(i,0)=0 $0 \le i \le len(s)$

D(0,j)=0 $0 \le j \le len(t)$

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + d(si,tj), & substitution \\ D(i-1,j)+1, & insertion \\ D(i,j-1)+1 & deletion \end{cases}$$

d(i,j) is a function whereby d(c,d)=0 if c=d, 1 else.

The Levenshtein distance is metric.

Definition 29. *A method of normalizing the edit distance to give a similarity in [0,1] is:*

$$sim(s,t) = 1 - \frac{ed(s,t)}{\max(len(s), len(t))}$$

2.7.2.2 Smith-Waterman Algorithm

Definition 30. *For two strings s and t, the Smith-Waterman similarity score is measured as follows:*

D(i,0)=0 $0 \le i \le len(s)$

D(0,j)=0 $0 \le j \le len(t)$

If $a_i = b_j$ $w(a_i,b_j)=w(match)$ *or* $a_i \neq b_j$ $w(a_i,b_j)=w(mismatch)$

$$D(i,j) = \max \begin{cases} 0 & \\ H(i-1,j-1) + w(ai,bj) & match/mismatch \\ H(i-1,j) + w(ai,-) & deletion \\ H(i,j-1) + w(-,bj) & insertion \end{cases}$$

The Smith-Waterman algorithm when constructed as a distance instead of a similarity is known to be metric. The similarity algorithm is known as an optimal local string alignment.

2.7.2.3 Longest Common Subsequence (LCS)

Definition 31. *For two strings X and Y, the LCS is found as follows:*

$$LCS(Xi,Yi) = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ (LCS(Xi-1,Yj-1),xi) & if\ xi = yj \\ longest(LCS(Xi,Yj-1),LCS(Xi-1,Yj)) & if\ xi \neq yj \end{cases}$$

The similarity between two strings X and Y is defined as $\left|LCS(X,Y)\right|$

2.7.2.3 Normalized Compression Distance

Definition 32. *For two strings x and y where C(x) is the length of a compressed x, the normalized compression distance (NCD) [84] is:*

$$NCD(x,y) = \frac{C(x,y) - \min(C(x),C(y))}{\max(C(x),C(y))}$$

The NCD is metric.

2.7.3 Vector Similarity

Vector distance can be performed using metrics such as the Euclidean distance or Manhattan distance. Non metric similarity measures can include the cosine similarity which is often used in text mining.

2.7.3.1 Euclidean Distance

Definition 33. *The Euclidean distance between vectors p and q is:*

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

The Euclidean distance is metric.

2.7.3.2 Manhattan Distance

Definition 34. *The Manhattan distance between vectors p and q is:*

$$d(p,q) = \sum_{i=1}^{n} \left| q_i - p_i \right|$$

The Manhattan distance is metric.

2.7.3.3 Cosine Similarity

Definition 35. *The cosine similarity between vectors A and B is:*

$$similarity = \cos(\phi) = \frac{A \cdot B}{\|A\|\|B\|}$$

The cosine similarity is not metric.

2.7.4 Set Similarity

Two sets can be compared using a variety of measures. The Dice coefficient and Jaccard Index are two such measures. The Jaccard Index is not metric, but its parallel the Jaccard Distance is, which allows for efficient indexing and searching. Containment and the Tversky index are examples of asymmetric similarity measures. Because they are asymmetric, they do not qualify as metric distance functions.

2.7.4.1 Dice Coefficient

Definition 36. *The Dice coefficient betweens sets A and B is:*

$$s = \frac{2\left|A \cap B\right|}{\left|A\right| + \left|B\right|}$$

The Dice coefficient is not metric.

2.7.4.2 Jaccard Index

Definition 37. *The Jaccard Index between sets A and B is:*

$$J(A,B) = \frac{\left|A \cap B\right|}{\left|A \cup B\right|}$$

The Jaccard Index is not metric, however the Jaccard distance is.

2.7.4.3 Jaccard Distance

Definition 38. *The Jaccard distance between sets A and B is:*

$$J_d(A,B) = 1 - J(A,B)$$

The Jaccard distance is metric.

2.7.4.4 Containment

Definition 39. *The Containment of set B in A is:*

$$C(A,B) = \frac{|A \cap B|}{|A|}$$

Containment is an asymmetric measure and therefore not metric.

2.7.4.5 Overlap Coefficient

Definition 40. *The overlap coefficient between sets A and B.*

$$overlap(X,Y) = \frac{|A \cap B|}{\min(|X|,|Y|)}$$

The overlap coefficient is not metric.

2.7.4.6 Tversky Index

Definition 41. *The Tversky Index of sets X and Y is:*

$$S(X,Y) = \frac{|X \cap Y|}{|A \cap B| + \alpha|X - Y| + \beta|Y - X|}$$

The Tversky index is an asymmetric measure and therefore not metric.

2.7.5 Set of Vectors Similarity

A set of vectors can be compared using the minimum matching distance [85], which constructs a minimum weight matching between pairs of vectors in each set. This distance is metric and can be evaluated in polynomial time. We extend this problem in Chapter 5 to our novel set of strings problems.

2.7.6 Tree Similarity

Trees can be compared for equality using tree isomorphism. Ordered trees are trees such that the children of each node are in a specific sequence. Ordered trees are significantly

more efficient to process than unordered trees. Approximate matching and similarity between trees can also be found using the tree edit distance [86]. The tree edit distance is metric. Alternatives to the tree edit distance include using the largest common subtree as an indicator of similarity. These are similar to the graph based version of the problem and are shown in the next section.

Definition 42. *The tree edit distance between two graphs* $d : T_1 \times T_2 \rightarrow \mathbb{N}$ *is the minimum number of edge or vertex insertions, deletions, and substitutions to transform one tree to the other.*

2.7.7 Graph Similarity

2.7.7.1 Graph Isomorphism

Graphs can be tested for structural equality by graph isomorphism testing. Graph isomorphism has not been demonstrated to belong to the complexity class P but it has not been proven to be in NP either.

Definition 43. *Let* $g_1 = (V_1, \alpha_1, \beta_1)$ *and* $g_2 = (V_2, \alpha_2, \beta_2)$ *be two graphs. A graph isomorphism between* g_1 *and* g_2 *is a bijective mapping* $f : V_1 \rightarrow V_2$ *such that*

$$\alpha_1(x) = \alpha_2(f(x)) \forall x \in V_1$$
$$\beta_1((x, y)) = \beta_2((f(x), f(y))) \forall (x, y) \in V_1 \times V_1$$

If $V_1=V_2=0$ then f is called the empty graph isomorphism

2.7.2.2 Graph Edit Distance

A harder problem is calculating the approximate similarity or distance between two graphs. The two main approaches are the graph edit distance and using the maximum common subgraph. The graph edit distance is metric. These problems are proven not to belong to P. However, polynomial time approximate solutions exist to the graph edit distance.

Definition 44. *The graph edit distance* $d : G_1 \times G_2 \rightarrow \mathbb{N}$ *between two graphs is the minimum sum cost of basic edit operations to transform one graph to another.*

2.7.2.3 Maximum Common Subgraph

Definition 45. *Let* $g_1 = (V_1, \alpha_1, \beta_1)$ *and* $g_2 = (V_2, \alpha_2, \beta_2)$ *be two graphs and* $g_1' \subseteq g_1, g_2' \subseteq g_2$. *If there exists a graph isomorphism between* g_1' *and* g_2', *then both* g_1' *and* g_2' *are called a common subgraph of* g_1 *and* g_2.

Definition 46. *Let g_1 and g_2 be two graphs. A graph g is called the maximum common subgraph of g_1 and g_2 if g is a common subgraph of g_1 and g_2 and there exists no other common subgraph of g_1 and g_2 that has more nodes than g.*

Definition 47. *The distance between graphs g_1 and g_2 is:*

$$d(g_1, g_2) = \frac{|MCS(g_1, g_2)|}{|g1|} \text{ where } |g|=|V|+|E|$$

Definition 48. *The distance between graphs g_1 and g_2 is:*

$$d(g_1, g_2) = \frac{|MCS(g_1, g_2)|}{\max(|g_1|, |g_2|)} \text{ where } |g|=|V|+|E|$$

An approximate or inexact maximum common subgraph is also possible.

Definition 49. *The graph edit distance between two graphs $d : G_1 \times G_2 \to \mathbb{N}$ is the minimum number of edge or vertex insertions, deletions, and substitutions to transform one graph to the other.*

Distances based on the maximum common subgraph are not metric.

2.8 Software Similarity Searching and Classification

The ultimate problem of this chapter is to search for similar software to our query from a database and to classify a program as belonging to a particular class. This section examines how we transform the pair-wise similarity problem into a similarity search problem over a database. Moreover, we examine statistical classification of birthmarks to identify the class of software it belongs to.

2.8.1 Instance-based Learning and Nearest Neighbour

Instance-based learning is a form of machine learning used in classification. To classify an object, it is compared to known instances of that object. If the query is similar to a known instance, or alternatively closest to an instance, known as its nearest neighbour, then it is classified as belonging to the same class. Nearest neighbour and range searches are the fundamental basis for software similarity using software features. If a piece of software represented as an object is in very close range or distance to known software instances, then it is declared a variant.

2.8.1.1 k Nearest Neighbours query

Definition 50. *Given a set of objects P and a query Q, and an integer k > 0, the k nearest neighbours (kNN) query is to find a result set kNN that consists of k objects such that for any* $p \in (P - kNN)$ *and any* $p' \in kNN, dist(p', q) \leq dist(p, q)$

2.8.1.2 Range query

Definition 51. *Given a set of objects P and a query Q, and a range r > 0, the range query is to find a result set rNN that consists of objects such that for any* $p' \in rNN, dist(p', q) \leq r$

2.8.1.3 Metric Trees

Metric trees allow similarity searches (nearest neighbour and range searches) for objects that have a metric distance function. A number of algorithms have been proposed such as BK Trees [87], Vantage Point trees [81], M-Trees [83], Slim trees [88], or DBM Trees [82]. Metric access methods can be categorized by different qualities such as whether the data structures allow for efficient insertion and deletion of objects allowing for dynamic access, or whether the data structures are kept in main memory or on disk.

2.8.1.4 Locality Sensitive Hashing

Locality sensitive hashing [89] is a scheme whereby similar objects are hashed to the same buckets. This allows a similarity search to perform nearest neighbour searches by hashing.

Definition 52. *Let d be a metric distance function. Let* $B(v, r) = \{q \in X \mid (v, q) \leq r\}$. *A family* $H = \{h : S \to U\}$ *is called {r1,r2,p1,p2} sensitive for D if for any* $v, q \in S$

- *If* $v \in B(q, r_1)$ *then* $\Pr_H[h(q) = h(v)] \geq p_1$
- *If* $v \notin B(q, r_2)$ *then* $\Pr_H[h(q) = h(v)] \leq p_2$

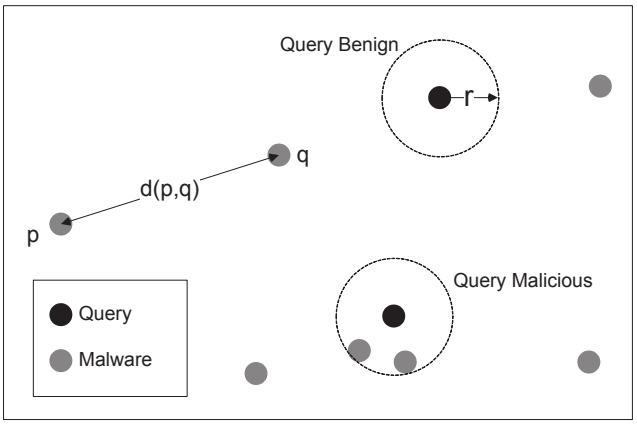

Fig. 25. The software similarity search to detect malware.

In order of a locality-sensitive hash (LSH) family to be useful, it has to satisfy inequalities p1 > p2 and r1 < r2.

2.8.1.5 Distributed Similarity Search

Scalability becomes a problem when database sizes increase. For example, malware databases have been growing exponentially [4] and efficient algorithms are required to handle the problem. Distributed algorithms are one solution to scale similarity searches. Distributed metric space similarity search algorithms include M-Chord [90] and GHT* [91, 92]. An approach based on Locality Sensitive Hashing is proposed in [93].

2.8.2 Statistical Machine Learning

Statistical classification is the process of assigning objects to classes. A typical example is the malware classification problem which is the process of assigning an unknown executable to the class of malicious or non malicious software.

Machine learning can be supervised or unsupervised. In the unsupervised model, none of the objects are labelled, and their class designation is unknown. The usual approach is to perform clustering to identify separate classes. In the supervised approach, a training set of data is labelled and used to build a model of classes in relation to their characteristics. After training, the system classifies unlabelled data and determines their classes.

Statistical classifiers include the popular and efficient Bayesian classifiers. Artificial Neural Networks (ANN) are another popular choice. The classifiers can also be grouped into linear and non linear systems. In a linear classifier, the input space can divide the classes using hyperplanes.

Vectors are used in many machine learning algorithms so often it is most useful to represent software as feature vectors. Features that are extracted from software can be used to construct feature vectors. Kernel machines provide an alternative approach to using feature effects and the most popular kernel method based classifier is the Support Vector Machine [94]. In this approach, a kernel for a particular object must be constructed. For classification of objects such as graphs, a variety of graph kernels can be used.

2.8.2.1 Vector Space Models

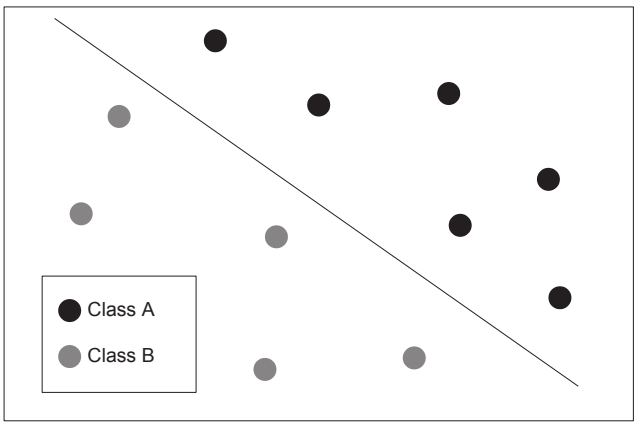

Fig. 26. A linear classifier separating two classes.

In the vector space model, a feature vector is constructed in $\mathbb{R}^n$ and classes are separated by partitioning over that space. The original feature vectors may have a high dimensionality, but in reality many of these features may be of low importance or redundant. Dimensionality reduction reduces the size of the feature vector.

2.8.2.2 Kernel Methods

The most well known kernel based classifier is the support vector machine (SVM) [94]. It is a linear classifier and works by constructing a hyperplane that maximally separates the margins between each class.

2.8.3 Research Opportunities

Nearest neighbour searches using metric distance functions to perform similarity searches has been employed in some malware detection literature. Much existing literature on software similarity has only focused on pairwise similarity and ignored the indexing and searching problem. Opportunities exist to transfer existing techniques into metric indexing methods. In Chapter 5, we make the novel application of metric indexing methods to our malware indexing and searching system.

Locality sensitive hashing also represents an opportunity as this indexing and searching technique has not been employed in all areas such as malware detection. Likewise, distributed similarity search algorithms are still to be exploited in the domain of software similarity.

The use of kernel methods for graph and tree based features is an area which is unexplored. The use of graph kernels to enable graph based classification presents much opportunity for researchers in future work.

2.9 Applications

This section surveys the application specific literature in software similarity and classification. It examines malware classification, software theft detection, plagiarism detection and code clone detection. We group the literature based on the class of program feature that is used to construct birthmarks. Finally, we critically analyse the approaches used.

2.9.1 Malware Classification

2.9.1.1 Raw Code

An approach employed by commercial Antivirus avoids static analysis by automatically extracting string signatures [95, 96].The main problem with this approach is that polymorphic malware makes string signatures prone to failure when the byte level content changes due to mutation, recompilation, and source code modification.

Kolmogorov complexity is a theoretical measure of the computational complexity, or minimum string length in a universal description language, required to represent an object or set of data. It is a theoretical measure that is not computable. To estimate the Kolmogorov complexity, an object may be compressed and concatenated with the associated decompression routine, to give the approximate minimum string length to describe the object. The observation, when this theory is related to malware, is that similar malware have similar measures of Kolmogorov complexity. This form of analysis occurs on the malwares raw file or section content.

Estimating Kolmogorov complexity was proposed in peHash [97] by identifying the compression ratio of a malicious sample that was subsequently used for clustering malware families. Another measure of similarity related to Kolmogorov complexity is the Normalized Compression Distance (NCD). The NCD was used in [98] to cluster worms into families. This approach, like peHash [97], was not used to classify samples as being benign or malicious, but to cluster malicious samples only.

It was the observation in [99] that malware and benign programs can be classified according to a likeness to a compression model for each of the malicious and benign classes. In this research, it was proposed that two compression models be constructed from a two training sets, one of malicious samples, and one of benign samples. To classify a query sample as being malicious or benign, the number of bits required to encode the query was calculated for each compression model. The query was classified by identifying the class that requires the least data to encode the query.

2.9.1.2 Instructions

An approach that employs static analysis is code normalization [17, 100]. Code normalization canonizes malware before Antivirus string scanning. In [17], static analysis eliminated superfluous control flow by merging redundant control flow nodes together. Instruction sequences within basic blocks that had no effect were also removed using an SMT decision procedure. The malware normalization approach improves on Antivirus detection but does not always effectively canonize a program to a unique form. This can affect the effectiveness and efficiency of malicious code detection.

A simple approach requiring only disassembly is fingerprinting malware based on opcode distributions [101]. An improved approach was proposed by using n-gram analysis of opcode and byte sequences. N-grams and n-perms can identify similarity between malicious programs and build evolutionary trees [102]. N-gram based feature vectors were used in instance-based learning and statistical classification. Statistical classification allowed for the detection of novel and unknown malware in [103, 104]. These systems improve the effectiveness of static string signatures, but instruction level classification has similar problems when the instruction stream changes to any significant degree.

2.9.1.3 Basic Blocks

Malware classification using the basic blocks of a program has been investigated in [105]. This approach requires disassembly and ideally a reasonable control flow analysis to identify targets of branchs and calls. The edit distance can be used between basic blocks to identify similarity. Existence of a basic block in a malicious sample can be determined using an inverted index or bloom filters. The main problem with this approach is polymorphic malware that changes the instructions within a basic block.

2.9.1.4 API Calls

The static ordering of system API calls can be extracted and used for malware classification. Association mining was proposed in [106] proposed to detect unknown malicious programs. Dynamic analysis of API calls or the combination of API calls and data flow can also be used as proposed in [107].

2.9.1.5 Control Flow and Data Flow

Control flow has been shown to be one of the more invariant features of a polymorphic malware and is resistant to byte and instruction level changes. Combining data flow analysis and control flow analysis was proposed in [108, 109]. Annotated flowgraphs combining data flow were compared to signatures, or automata, that describe the malware.

2.9.1.6 Data Flow

A data flow analysis was proposed in [54] where value set analysis was used to construct signatures.

2.9.1.7 Call Graph

Interprocedural control flow using the call graphs of a program have been compared to show similarity to existing malware [110-113]. An approach to transform the interprocedural control flow information into a context free grammar, allowing for homomorphism testing using string equality was also proposed in [24].

2.9.1.8 Control Flow Graphs

Control flow graphs have also been employed in [60, 61, 114, 115] using graph edit distances, maximum common subgraphs and decomposition of graphs into small fixed sized subgraphs.

2.9.2 Software Theft Detection (Static Approaches)

2.9.2.1 Instructions

Considering the static instruction sequences in control flow graphs was proposed for Java programs in [116]. This approach proposed using control flow graphs to build static instruction traces. The traces were constructed by imposing a tree structure on the control flow graphs and performing tree traversals to generate an ordering of the instructions. To compare traces a sequence alignment algorithm was used. The similarities between traces in control flow graphs were accumulated to generate a program level similarity score.

K-grams were proposed in [117] to compare two programs. In this work, a k-gram was defined as a unique sequence of k instructions as laid out in the executable. The resulting birthmark is a set of k-grams. To compare two programs, set similarity measures were used which parallel the Jaccard index and the detection of subsets.

The operands of instructions have also been proposed as a useful birthmark in Java programs. [118] proposed four birthmarks, one being the sequence of constant values in field variables. Operand stack patterns were proposed in [119] [120]. Operand stack patterns looked at sequences of bytecode that shared operands through the operand stack.

2.9.2.2 Control Flow

Control flow has been proposed as a static feature from which birthmarks can be constructed [121, 122]. In the proposed approaches, the edges in the control flow graph were used. The instructions in the basic blocks making up the edge were concatenated with each other to construct a possible execution sequence of code. To compare two of these features, the longest common subsequence (LCS) algorithm was used. To compare two sets of these features, as when all the control flow edges are considered, a maximum weight matching was performed on the set of all pairwise comparisons of those features. This matching sum allows for a calculation of similarity.

2.9.2.3 API Calls

Static API calls were proposed as birthmarks in [123, 124]. The API calls made in each procedure of a program were grouped together in sets. To compare two sets, the Dice coefficient which measures the similarity between two sets was used. To compare two programs, where each program consists of multiple sets, a maximum weight matching was used on the set of all pairwise comparisons between those sets. This matching allows for calculation of similarity.

2.9.2.4 Object Dependencies

Object inheritance graphs in Java programs and the objects other objects used was proposed in [118] as a birthmark. This paper proposed a total of four birthmarks that could be used for software theft detection.

2.9.3 Software Theft Detection (Dynamic Approaches)

2.9.3.1 Instructions

Dynamic extraction of instruction N-grams was proposed in [125]. This is analogous to k-grams and n-grams in the static approach.

2.9.3.2 Control Flow

An interesting approach to capture the dynamic nature of control flow was proposed in [126]. The control flow is dynamically traced, and the edges in the associated control flow graph labelled. The execution trace generates a sequence of those labels. The sequence is converted into a context free grammar using the SEQUITUR algorithm which is useful in capturing the repetitive nature of dynamic control flow. The grammar produces a graph and the terminal nodes are removed. This final graph is the birthmark. To compare two birthmarks, a maximum common subgraph is used to identify similarity.

2.9.3.3 API Calls

Dynamic tracing of API class has had a considerable amount of research [8, 127-130]. The dynamic API trace exhibits properties of the programs semantics and is less prone to the problems of obfuscation that static API call traces have. However, triggering all behaviours can be difficult.

2.9.3.4 Dependence Graphs

A dynamically generated system call dependence graph approach to building a birthmark was employed in [5]. Nodes in the graph represented system calls and control and data dependencies were represented by edges. The graphs, or birthmarks, were compared to show similarity using subgraph isomorphism testing.

2.9.4 Plagiarism Detection

Plagiarism detection systems often make the distinction between attribute counting and structure based techniques. Attribute counting is based on software metrics, or the frequencies of particular features occurring. Typical approaches include Halstead metrics and other metrics which take into account attributes including the number of tokens, the number of operators, the number of variables, or the number of source lines [131]. Structure based techniques rely on using program structure which typically include the use of dependency graphs or parse trees.

2.9.4.1 Raw Code and Tokens
JPlag [132] and YAP3 [133] consider tokens from source code as features and perform similarity comparisons using greedy string tiling. Another approach [134] considers tokenization and linearization of the source code and uses an adaptive sequence alignment to construct a similarity measure.

2.9.4.2 Parse Trees
Parse trees are related to abstract syntax trees and have been proposed for plagiarism detection [135] by using tree comparisons to identify similarity. Tree similarity can be based on algorithms including tree edit distances or largest common subtrees.

2.9.4.3 Program Dependency Graph
GPLAG used program dependency graphs of programs [136]. Similarity between program dependency graphs uses similarity metrics such as the graph edit distances.

2.9.5 Code Clone Detection
Large scale manual attempts at auditing specific Linux distributions for embedded packages-level clones have occasionally occurred in the past. In 2005, the Debian package repository was scanned for vulnerable zlib fingerprints based on version strings [137]. Antivirus signatures were generated and ClamAV performed the scanning. We extend this work significantly with a completely automated approach in Chapter 3. Work has been done on detecting higher-level clones, including file-level clones [138]. Additionally, clone detection has been used on industrial sources like the Linux kernel [139] or as used by Microsoft engineers [140]. An interesting semantic approach to clone detectioni is to use the memory states of a program [141]. . In [142], trees were used to represent sourcecode, and subtrees transformed to a vector representation. This allowed for the Euclidean distance and clustering to identify clones.

2.9.5.1 Raw Code and Tokens
Clone detection can be performed on the textual stream in a source file once whitespace and comments are removed [143]. The key concept is that a fingerprint of a code fragment is obtained and then the remainder of the source scanned for possible matching duplicates. More recently [144, 145] has used the token approach with good success in large scale evaluations. Large scale copy and paste clones using a data mining approach was investigated in [146, 147].

2.9.5.2 Abstract Syntax Tree

An alternative approach is to use the abstract syntax tree of the source to generate a fingerprint [148]. Tree matching can subsequently be used to discover software clones. Abstract syntax trees are more impervious to superficial changes to the textual stream and textual organization of the code.

2.9.5.3 Program Dependency Graph

Other program abstractions can be used to fingerprint code fragments such as the program dependency graph which is a graph combining control and data dependencies [149].

2.9.6 Critical Analysis

All applications of software similarity and classification share common themes of feature extraction, similarity functions and statistical classification. The literature reviewed in this section on applications should be in the context of the theory presented in this chapter. Initial work on malware detection was based primarily on the raw code contents. As noted in earlier sections, raw code is ineffective when trying to detect malware variants including polymorphic and metamorphic samples. Instruction opcodes and sequences also face similar problems. Control flow has been used successfully in most of the above applications when perform static analyses. The danger of including data flow as a feature is that the birthmarks created become too specific to the instance of code and therefore suffer the same fate as using byte-level content. Therefore, control flow might be the best choice for the time being. Control flow can be obfuscated however using packing and other techniques so a trend has been to perform dynamic analysis by running the sample program in a virtualized environment. The feature of choice has been the API calls the program makes. Dynamic analysis is not without fault though and that has also been discussed in earlier sections. Of note, there is a distinction in the literature between the software similarity problem and the software classification problem. Some applications such as software theft detection will

always be based upon software similarity. However, applications such as malware detection only care for a signature-less binary classification. Nevertheless, software similarity is still useful for identifying families of malware and attributing authorship of those malicious executables.

2.10 Future Trends

Software similarity and classification may see the unification of malware classification with other technologies such as software theft detection or software clone detection. These topics will see sharing of concepts and techniques and the use of program features will become comprehensive. It may indicate that a combination approach to software similarity and classification is appropriate. Many of the features are useful at representing a particular property of software, but obfuscations or transformations may alter these properties. Using a variety of properties in combination may be a suitable response for increasing accuracy.

Static binary analysis is an emerging field and continues to improve. The analyses are becoming stronger and able to model more complex behaviour without gross under-approximations or over-approximations. This will continue to improve as this area of static analysis becomes more recognized. In particular, malware classification and software theft detection are driving forces of the need for analyses.

Static binary analysis is used in academic malware classification. It has not seen widespread use in commercial Antivirus. We believe this situation will change due to the more effective signatures and the ability to use machine learning and statistical classification to detect novel samples of malware. The trend in malware classification is to use higher level of abstractions and more emphasis is placed on combining data flow analysis with control flow analysis. Appropriate database technologies are being used more as the problem is becoming how to effectively perform indexing and searching of program features for an instance-based signature approach of malware variant detection. Statistical classification continues to improve on the effectiveness of program features used. We are likely to see the combination of program features, and the combination of

different classifiers to improve system accuracy. Complex objects such as graphs will continue to be used with an emphasis on problems in graph mining.

Software theft detection is not widely used by all vendors, but as technology improves and matures, this may become more common. Software theft detection is a program variant detection problem and therefore uses instance-based learning. Database technology as in the case of malware variant detection will take important roles.

Network speeds are improving and cloud services are becoming more popular. Antivirus vendors have already taken advantage of this and have provided an initial set of offerings for cloud based malware detection. Services already exist that provide AV scanning on demand using a large number of commercial scanners. A hybrid scheme may also be used where some of the processing and feature extraction is done on the endpoint. We expect that as bandwidth becomes less of an issue, cloud Antivirus will become popular. Placing malware classification in the cloud allows the use of huge signature databases along with correlation not possible when end users are disconnected. Mobile platforms are less powerful than their desktop counterparts, so these devices would benefit from cloud services where the majority of processing is done away from the user's device. Finally, cloud services may provide an opportunity to detect attackers, through service misuse, from tuning their malware or plagiarised code to evade detection.

Concluding Remarks

In conclusion of this chapter, software similarity and classification is an important topic that unifies and tackles the problems of malware classification, plagiarism detection, software theft detection and code clone detection. Many techniques are pioneered or formalized in one topic but only later applied, if at all, to other domains. We have presented the core concepts of how to approach this problem and identify new areas of research. Much research is possible simply by applying existing research across domains.

Chapter 3: Clonewise – Detecting Package-level Clones Using Machine Learning

Developers sometimes maintain an internal copy of another software or fork development of an existing project. This practice can lead to software vulnerabilities when the embedded code is not kept up to date with upstream sources. As a result, manual techniques have been applied by Linux vendors to track embedded code copies and identify vulnerabilities. We propose an automated solution to identify clones of packages without any prior knowledge of these relationships. We then correlate clones with vulnerability information to identify outstanding security problems. This motivates package maintainers to avoid using cloned packages and link against system wide libraries. Our approach identifies similar source files based on file names and content to identify relationships between packages. We propose over 30 novel features that enable us to use to use pattern classification to accurately identify package-level clones. To our knowledge, we are the first to consider clone detection as a classification problem. Our results show Clonewise compares well to manually tracked databases. These results are now starting to be used by Linux vendors to track embedded packages. Red Hat started to track clones in a new wiki, and Debian are planning to integrate Clonewise into the operating procedures used by their security team. Based on our work, over 30 unknown package clones and vulnerabilities have been identified and patched.

3.1 Introduction

Developers of software sometimes embed code from other projects. They statically link against an external library, maintain an internal copy of an external library's source code, or fork the development of an external library. A canonical example is the zlib compression library which is embedded in much software due to its functionality and permissive software license. In general, embedding software is considered a bad development practice, but the reasons for doing so include reducing external dependencies for installation, or the need to modify functionality of an external library. The practice of embedding code is generally ill advised because it has implications on software maintenance and software security. It is a security problem because at least two versions

of the same software exist when it is embedded in another package. Therefore, bug fixes and security patches must be integrated for each specific instance instead of being applied once to a system wide library. Because of these issues, for most Linux vendors, package policies exist that oppose the embedding of code, unless specific exceptions are required.

In the example of zlib, each time a vulnerability was discovered in the original upstream source, all embedded copies required patching. However, in the past, uncertainty existed in Linux distributions of which packages were embedding zlib and which packages required patching. In 2005, after a zlib [150] vulnerability was reported, Debian Linux [151] made a specific project to perform binary signature scans against packages in the repository to find vulnerable versions of the embedded library. To create a signature the source code of zlib was manually inspected to find a version string that uniquely identified it. This manual and time consuming approach still finds vulnerable embedded versions of software today. We constructed signatures for vulnerable versions of compression and image processing libraries including bzip2, libtiff, and libpng. We performed a scan of the Debian and Fedora Linux [152] package repository and found 5 packages with previously unknown vulnerabilities. Even for actively developed projects such as the Mozilla Firefox web browser, we saw windows of exploitability between upstream security fixes and the correction of embedded copies of the image processing libraries. Even in mainstream applications such as Firefox, these windows of opportunity sometimes extended for periods of over 3 months.

The traditional approach for discovering duplicated fragments of insecure code has been through the use of code clone detection. However, clone detection is sometimes too fine grained to be of practical benefit for Linux vendors and package maintainers.

3.1.1 Motivation for Package-level Clone Detection

Clone detection theoretically solves the problem of insecure code fragments propagating to other locations. However, in practice the number of code clones is significantly high. For developers of individual projects, clone information may be useful. Yet, package maintainers and operating system distributions have no realistic actions to take with such clone information since they are not the primary developers of the software they release. What package maintainers and operating system vendors want is the ability to repackage

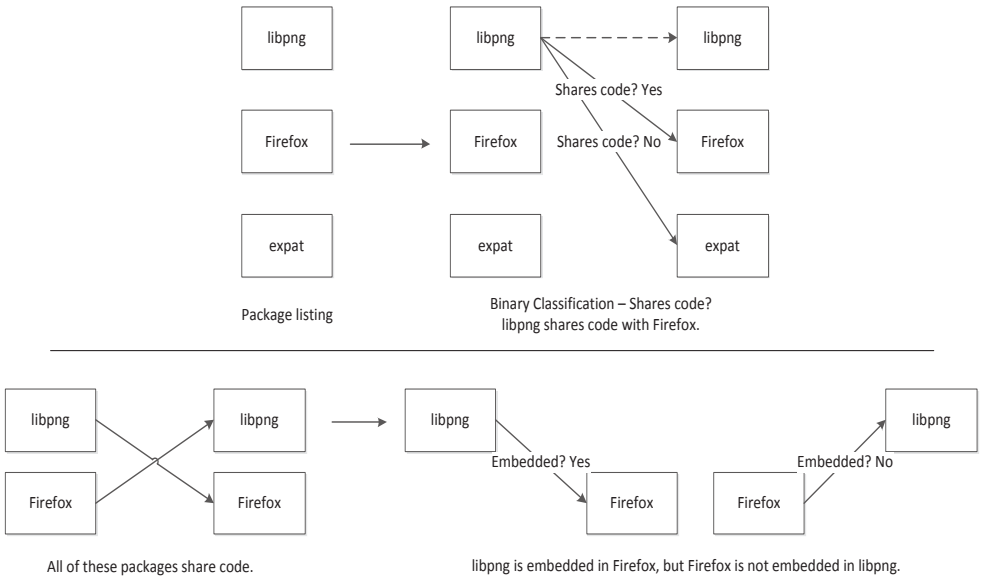

Fig. 27. Shared package clone detection (above) and embedded package clone detection (below).

or build the software in such a way that improves security and eliminates clones. If vendors know that an entire package is cloned in another, then they can modify the build process to use the operating system's system wide library package. This is an achievable goal and improves the security and stability of the system. This is our motivation and the reason we see package-level clone detection as an important addition to software engineering that traditional clone detection does not address.

3.1.2 Motivation for Automated Approaches

The approach of manually searching for embedded copies of specific libraries deals poorly with the scale of the problem. According to the list of tracked embedded packages in Debian Linux, there are over 420 packages which are embedded in other software in the repository. This list was created manually and our results show that it is incomplete. Other Linux vendors were not even tracking embedded copies before our research supplied them with relevant data. It is evident from this that an automated approach is needed for identifying embedded packages without prior knowledge of which packages to search for. This would aid security teams in performing audits on new vulnerabilities in upstream sources. This identifies the motivation for our system named Clonewise to identify package-level clones.

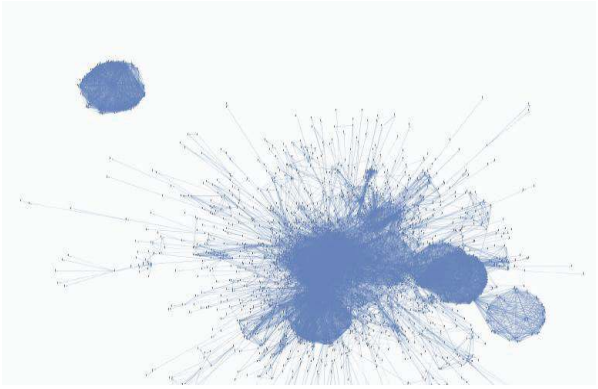

Fig. 28. Graph of Fedora 13 package relationships.

Previous systems that automate and address part of the problem are software provenance systems. Our system extends such suck by recognising more features in software that can be used to fingerprint it. Our system also addresses the problem of software being implemented in multiple languages, even within the same package. Our work is language agnostic. We also address the problem of requiring every version of a software to match it against a query. Our system can determine if a package is embedded, irrespective of which version number is used. This has advantages, but also makes identifying security problems in specific versions harder. We overcome this by using side-information that tracks the necessary information and is maintained by operating system vendors.

Our work is also similar to the concept of structural or higher-level clones as proposed in [138]. We are much more specific in the type of structure we are searching for. That is, package-level clones. Likewise, the structural clones in [138] use directory-level clones to simulate module-level clones which is not as accurate.

3.1.2 Generability

At first glance, package-level clone detection may appear to be a Linux distribution specific problem. However, this problem applies to any vendor who maintains a repository of software packages and shares common code amongst packages. It also applies to any vendor which for legal reasons needs to know the provenance of embedded packages such as open source libraries. Finally, this problem applies to any vendor who needs to know what open source libraries have been embedded so as to keep up-to-date with upstream releases. It is quite conceivable that any large software project may incorporate some permissively licensed open source software as an embedded library or package. For

all of these reasons, software engineering needs to incorporate automated means to provide assurance that the state of software and the existence of package-level clones is known.

3.1.3 Innovation

Our approach is to consider code reuse detection as a binary classification problem between two packages. The classification problem is 'do these two packages share code?' We achieve this by performing feature extraction from the two packages and then performing statistical classification using a vector space model. The features we use are based on the filenames, hashes, and fuzzy content of files within the source packages

To identify security vulnerabilities we associate vulnerability information from public vulnerability advisories to vulnerable packages and vulnerable source files. We then discover all clones of these packages in a Linux distribution. Finally, we check the manually tracked vulnerable packages that Debian Linux maintain for each vulnerability and report if any of our discovered clones are not identified as being vulnerable.

In this chapter we make the following contributions:

We define the problem of package clone detection, and the sub-categories of shared and embedded package clone detection.

We are the first ones to formulate code reuse detection as a pattern classification problem. Then, it is feasible to apply traditional pattern classification algorithms to achieve accurate clone detection. We employ a novel asymmetric bagging based classifier combination method to address the specific classification problem.

We propose over 30 new features for the purpose of clone detection, which are fundamental to solve the specific pattern classification problem. In particular, the proposed features are basis to the accuracy of clone detection.

We propose applications of package clone detection. We present algorithms to identify outstanding security vulnerabilities based on out-of-date clones.

We implement a complete system, Clonewise, which demonstrates our system effectively identifies package clones, finds vulnerabilities and is useful to vendors. For example, Debian Linux is planning infrastructure integration of Clonewise.

3.1.4 Structure of the Chapter

The structure of this chapter is as follows: Section 3.2 defines the problem of package clone detection and outlines our approach. Section 3.3 discusses some early attempts we made to perform package-level clone detection. Section 3.4 describes how Clonewise detects shared and embedded package clones using machine learning. Section 3.5 describes the algorithms we use to identify vulnerabilities based on clone information. Section 3.6 gives an outline of our implementation and Section 3.7 evaluates our system. Section 3.8 is discussion. Finally we present our concluding remarks.

3.2 Problem Definition and Our Approach

3.2.1 Problem Definition

A package clone is the duplication of one package's code in another package. It is the presence of code reuse between packages. How do we find these package clones?

A package can be embedded in another package. How do we determine this knowing that a package clone exists?

A package clone may contain vulnerabilities or other security problems because the clone is out of date. How do we find these?

3.2.2 Our Approach

Our approach for detecting clones is based on binary classification. This approach is shown in Fig. 27 and described below. A key point is that if two packages share code, one is not necessarily embedded in the other. We therefore detect code reuse and embedding as related but distinct problems.

Our approach is to consider code reuse detection as a binary classification problem between two packages. The classification problem is 'do these two packages share code?' We achieve this by performing feature extraction from the two packages and then perform

statistical classification using a vector space model. The features we use are based on the filenames, hashes, and fuzzy content of files within the source packages.

A package clone consisting of two packages can be analysed to determine if one package is embedded in the other. We use a binary classification problem to answer this. The features we use are based on the size of the cloned code relative to the size of each package, and other features such has how many packages are dependent on the packages we are analysing.

We determine vulnerable packages by correlating security tracking information with our package clone detection analysis.

3.3 Initial Attempts

Clonewise has been under development over a period of time and we have experimented with a number of approaches before deciding to use the machine learning-based system we currently employ.

3.3.1 Containment for Embedded Package Clone Detection

In our first attempts, we recognized that shared filenames can be used to identify commonality between two software packages. We experimented with using set theory proposed in related work to show that one package was embedded in another:

Definition 53. *Package containment is:*

$$\frac{|A \cap B|}{|A|} > t$$

where A and B are the sets of filenames in each package and t is a threshold of similarity. This equation is similar to the containment similarity measure [153] to show that one object is embedded in another. This is the same similarity measure as used to determine software provenance.

The first point we noticed was that some filenames are very common and skewed our results. We decided to exclude the most frequent filenames from our analysis to address this problem.

3.3.2 Intersection for Shared Package Clone Detection

A failure with the previous approach is that almost the entire package must be embedded for detection to occur. It is often the case that only the core code is embedded. We then tried the following:

Definition 54. *Package clone intersection is:*

$$|A \cap B| > t$$

This equation shows us the number of shared filenames between two packages which indicates sharing, not embedding, of code. We chose a low threshold and analysed the Fedora 13 Linux distribution. We made a directed graph where each node was a package, and an edge between nodes indicated the above equation was true. The graph is shown in Fig 28.

The graph gives us insight into package relationships. Cliques, or fully connected subgraphs, are packages that all share code with each other. If we relax clique detection to detect quasi-cliques or detect densely connected subgraphs, as in community structure [154], we can reveal even more relationships. It is likely, that one of the nodes in the clique, quasi-clique, or community is a library that is embedded in the other nodes.

3.3.3 Motivations for Other Approaches

We chose not to continue along this line of research for a number of reasons. 1) Choosing thresholds can be difficult and machine learning to select these values is a sound alternative. 2) All filenames should be considered, but it would be ideal if they were weighted based on their frequency. 3) Other features besides filenames should be considered. The set theory approach fails at this point without significant redesign.

3.4 Package Clone Detection

Clonewise is based on machine learning and we have found this approach to be most versatile and successful. We employ statistical classification to learn and then classify two packages as sharing or not sharing code.

Classification is a well-studied problem in machine learning and software is available to make analyses easy. Weka [155] is a popular data mining toolkit using machine learning that Clonewise uses to perform machine learning.

3.4.1 Shared Package Clone Detection

Feature extraction is necessary to perform shared package clone classification. We need to select features that reflect if two packages share or do not share code. The feature vector we extract is obtained from a pair of packages that we are testing for sharing of code. The 26 features we use are discussed in the following subsections.

3.4.1.1 Number of Filenames

Our first set of features is simply the number of filenames in the source trees of the two packages being compared.

3.4.1.2 Source Filenames and Data Filenames

In Clonewise, we distinguish between two types of filename features. Filenames that represent program source code and programs that represent non program source code. We distinguish these two types of filenames by their file extension. The list of extensions used to identify source code are c, cpp, cxx, cc, php, inc, java, py, rb, js, pl, m, mli, and lua. Almost all of the features in Clonewise are applied for both source and data filenames.

3.4.1.3 Number of Common Filenames

To identify that a relationship exists between two packages such that they share common code, we use common filenames in their source packages as a feature. Filenames tends to remain somewhat constant between minor version revisions, and many filenames remain present even from the initial release of that software. For our purposes we can ignore directory structure and consider the package as a set of files, or we can include directory structure and consider the package as a tree of files. We noted several things while experimenting with this feature:

Many files in a package do not contribute to the actual program code.

C code is sometimes repackaged as C++ code when cloned. For example, lib3ds.c might become lib3ds.cxx.

The filenames of small libraries can often be referred to as libfoo.xx or foo.xx in cloned form.

Some files that are cloned may include the version number. For example, libfoo.c might become libfoo43.c.

We therefore employ a normalization process on the filenames to make this feature counting the number of similar filenames more effective.

Normalization works by changing the case of each filename to be all lower case. If the filename is prefixed with lib, it is removed from the filename. The file extensions .cxx, .cpp, .cc are replaced with the extension .c. Any hyphens, underscores, numbers, or dots excluding the file extension component are removed.

3.4.1.4 Number of Similar Filenames

It is useful to identify similar filenames since they may refer to nearly identical source code. A fuzzy string similarity function is used that matches if the two filenames are 85% or more similar in relation to their edit distance.

Definition 55. *Our similarity measure is:*

$$similarity(s,t) = 1 - \frac{edit\_dist(s,t)}{\max(len(s), len(t))}$$

We chose the edit distance as our string metric after experimenting with other metrics including the smith-waterman local sequence alignment algorithm and the longest common subsequence string metric.

3.4.1.5 Number of Files with Identical Content

We perform hashing of file content using the ssdeep software and do a comparison of hashes between packages to identify identical content without respect to the filenames used. Like the previous class of feature, we have a feature for the number of files having identical content that are all program source code, and a feature for the number of files having identical content that are non-program source code.

3.4.1.6 Number of Files with Common Filenames and Similar Content

To increase the precision of file matching from the previous feature, we employ a fuzzy hash of the file contents and then perform an approximate comparison of those hashes for files with similar filenames. While the previous approach is based on file names alone, this approach is a combination of file names and content. Fuzzy hashing can be used to identify near identical data based on sequences within the data that remain constant using context triggered piecewise hashing [156]. The result of fuzzy hashing file content is a string signature known as its fuzzy hash. Approximate matching between hashes is performed using the string edit distance known as the Levenshtein distance. The distance is then transformed to a similarity measure. The similarity is a number between 0 and 100 indicates the hashes are not at all similar, and 100 indicates that the hashes are equal.

We have features for the number of files of similar content with a similarity greater than 0 of program source code and non-program source code. We also count the number of similar files having a similarity greater than 80.

3.4.1.7 Scoring Filenames

Not all filenames should be considered equal. Filenames, such as README or Makefile that frequently occur in different packages should have a lower importance than those filenames which are very specific to a package such as libpng.h. We account for this by assigning a weight for each filename based on its inverse document frequency [157]. The inverse document frequency lowers the weight of a term the more times it appears in a corpus and is often used in the field of information retrieval.

Definition 56. *The inverse document frequency is:*

$$idf(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

where D is the set of packages, d is a package, and t is a filename in a package.

We use features scoring the sum of matching filename weights to the number of similar files, the number of similar files and similar content with similarity greater than 0 and 80, for both program source code and non-program source code.

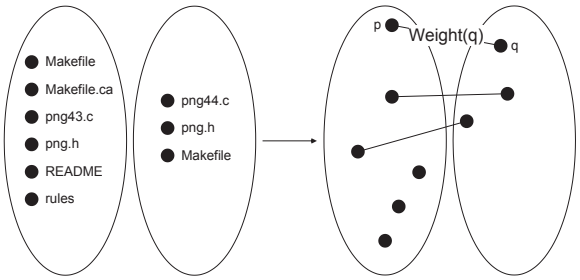

Fig. 29. The assignment problem.

3.4.1.8 Matching Filenames between Packages

If filename matching between two packages was performed as an exact match, then the number of filenames shared would be the cardinality of the intersection between the two sets of filenames. However, in Clonewise the filename matching is approximate based on the string edit distance. This means that some filenames such as Makefile.ca could potentially match the filenames Makefile.cba and Makefile.cb. Moreover, the scores for each filename as discussed in the previous section can be different depending on which filename is deemed to be a match. We solve this problem by employing an algorithm from combinatorial optimization known as the assignment problem as shown in Fig. 29.

The assignment problem is to construct a bijective mapping between two sets, where each possible mapping has a cost associated with it, such that the mappings are chosen so that the sum of costs is optimal. Formally, the assignment problem is defined as:

Definition 57. *Given two sets, A and T, of equal size, together with a weight function C: A × T → R. Find a bijection f: A →T such that the cost function (below) is optimal.*

$$\sum_{a \in A} C(a, f(a))$$

In our work the sets are the two packages and the elements of each set are the filenames in that package. The cost of the mapping between sets is the score of the matching filename in the second set according to its inverse document frequency. Our use of the assignment problem seeks to maximize the sum of costs.

The assignment problem can be solved in cubic time in relation to the cardinality of the sets using the Hungarian or Munkres [158] algorithm.

The Munkres algorithm is effective, however for large N, a cubic running time is not practical. We employ a greedy solution that is not optimal but is more efficient when N is large.

4.1.9 Features Selected.

We experimented with using a subset wrapper and a genetic search algorithm for feature selection. We did not arrive at a good feature set within a practical amount of time. Therefore, we chose not to perform feature selection in classification.

3.4.2 Shared Package Clone Classification

The output of Clonewise is the set of packages where the classification determines the package pairs share code. Clonewise also reports the filenames between the packages and the weights of those filenames.

Clonewise uses supervised learning to build a classification model. We use the manually created Debian embedded-code-copies database that tracks package clones to train and evaluate our system. We employ a number of classifiers to evaluate our system as described in Section 3.7.

3.4.3 Embedded Package Clone Detection

To detect embedded package clones we use the results of shared package clone detection and apply a filtering stage to exclude packages where the first package is not embedded in the second package. We solve this problem by considering the problem as a binary classification problem.

Similar to the shared package clone detection approach, we perform feature extraction before using statistical classification. The 18 features we use are summarized in the following:

3.4.3.1 Number of Filenames

As in shared package clone detection, the number of filenames that are source and data are used.

3.4.3.2 Percent of X embedded in Y

These features say how much of one package is embedded in the other package.

3.4.3.3 Package X has Lib in name

These features are useful in identifying if a package is a library, which increases its likelihood that it is an embedding. If the package name is prefixed with 'lib', then the feature is assigned a value of 1. If the prefix is not that, then the value is 0. The prefix is compared without regard to case.

3.4.3.4 A to B Ratio

These features inform us on how big the packages are relative to each other. It is typical that an embedded library is smaller than the software it is embedded in.

3.4.3.5 Package Dependents

These features inform us on how many other packages depend on the package in question. Libraries are typically used by many other packages and so the value for this feature will also be high. As explained earlier, that the package is library indicates that the package is more likely to be embedded.

3.4.4 Classification Using Asymmetric Bagging

For training our classifier, we have a finite set of labelled positive cases as obtained from vendor generated databases and we are able to arbitrarily generate labelled negative cases. We have many more negative cases than we have positive cases, wherein a positive case indicates an embedded package clone. This scenario represents the imbalanced class problem [159] where many classifiers favour the majority class. We decided to improve our detection rate of the positive class by addressing the imbalanced class problem by performing asymmetric bagging [160].

Asymmetric bagging uses all the labelled positive cases and use an equivalent number of negative cases obtained from a random sampling. This extends traditional bagging which uses a random and equal sampling from both classes. The asymmetric bagging approach described generates a single bag upon which a classification model is built from training. Many bags are created and classification models are built for each bag. When performing classification of an unlabelled instance, each bag makes a prediction and the results are aggregated using a majority vote. This has the effect or improving the accuracy when detecting positive cases. We implemented the asymmetric bagging algorithms by extending the bagging meta-classifier in the Weka machine learning toolkit.

3.5 Inferring Security Problems

In this section, we discuss standardization and tracking efforts by vendors. We then examine algorithms and approaches to detect software vulnerabilities. Package-level clone detection is not strictly the best method to discover security problems through code cloning. However, it is almost impossible in practice to apply code-level clone detection across tens of thousands of packages with potentially hundreds of thousands of clones and expect developers to integrate fixes. The reality is, a vendor's security team can fix high impact bugs and push package maintainers to build their software using system wide package-level libraries. In effect, the only practically used system of bug fixing on a large scale in regards to clones, is by fixing package-level clones. Yet the problem still exists of how to motivate package maintainers or security teams to apply these fixes. The current practice is to highlight that the cloned package contains known security problems and pointing out that there is less cost in rebuilding the software to eliminate the higher-level clone than it is to apply individual patches. Therefore, we see value in Clonewise as being a tool that can bring about good practices of eliminating package clones by highlighting vulnerabilities. To achieve the task of vulnerability detection, we propose use-cases for clone detection by Linux security teams. We also propose a completely automated solution to find out-of-date clones that have outstanding security vulnerabilities.

3.5.1 Use-case of Clone Detection to Detect Vulnerabilities

One method which we initially tried, for the purpose of vulnerability detection, was to look at packages that had reported vulnerabilities against them. We considered this to be a list of security sensitive packages. We used this list of packages as input to our clone detection analysis. Anytime a security sensitive package was cloned, we verified that the clone was not out of date. This is an effective method to detect vulnerabilities, but it requires manual analysis. Even though the technique we described is manual, it still has benefits today and can be used in an on-going basis to detect new vulnerabilities.

If a new vulnerability is found in a package, then clone detection should be performed on the complete Linux distribution because it is likely the same vulnerability is present in the cloned software. For example, if a vulnerability is reported for libpng, then clone detection should be performed and each libpng clone checked to see if the vulnerability is present as

is shown in Fig. 31. This method can be used by Linux security teams, but for old vulnerabilities it is not advisable since many clones would be patched but not reported by a Linux vendor. Therefore, we looked at other automated methods to detect out-of-date clones which we describe in the following sub-sections.

3.5.2 Standardization Efforts

Common Vulnerabilities and Exposure (CVE) is a standardization effort for public reporting of vulnerabilities. CVEs are maintained in the National Vulnerability Database (NVD). Each unique vulnerability is given a unique CVE identifier. In version 2 of the NVD content, the CVE information is stored as an XML database. CVE reports a vulnerability and gives a canonical name of the package or packages affected using the Common Platform Enumeration (CPE). Documented in the CVE entry is also a summary of the vulnerability in the package or program. This summary often includes a reference to the program's vulnerable function and vulnerable source file, if it exists. CVE makes it possible for different vendors to talk a common language of vulnerabilities and remediation when the same vulnerability affects multiple vendors. This is common because vulnerabilities often occur from upstream sources that are pushed downward and used by different vendors.

3.5.3 Debian Linux Security Tracking

Debian Linux make a significant effort to track security information and maintain a publicly accessible repository known as the security tracker for tracking security problems in their distribution.

A useful database that is unique to Debian is a manually generated list that is used to associate CPE names to Debian package names. This is done so Debian can check native packages against new vulnerabilities that appear as a CVE in the NVD.

Debian Linux also use CVE internally to track vulnerabilities. They maintain a database of every CVE. They then list every package in Debian affected by each particular CVE.

3.5.4 Automated Vulnerability Inference

In Clonewise, we can use clone detection in addition to the above information to identify untracked vulnerabilities.

> ***Summary:*** Off-by-one error in the __opiereadrec function in **readrec.c** in libopie in OPIE 2.4.1-test1 and earlier, as used on FreeBSD 6.4 through 8.1-PRERELEASE and other platforms, allows remote attackers to cause a denial of service (daemon crash) or possibly execute arbitrary code via a long username, as demonstrated by a long USER command to the FreeBSD 8.0 ftpd.

Fig. 30. An NVD CVE summary.

Clonewise takes a CVE number as input and extracts the vulnerable package from the report. The CPE package name is translated to a native Debian package name.

Clonewise then parses the summary to find the vulnerable source files. It is possible to extract theses vulnerable source files from the summary (Fig. 30) by tokenizing the summary into words and extracting words that have a file extension of known programming languages.

Clonewise then looks at all the clones of the vulnerable package and trims the list by ensuring one of the vulnerable source files is present in the clone and that the fuzzy hash between the vulnerable package's source is similar to the clone's.

We also trim the list by ignoring clones that we believe have been patched to use the system wide dynamic library. We did this by checking if in the binary version of the package the embedded package was a package dependency. If the embedded package is a dependency, then the main package almost certainly uses it for dynamic linking. Dynamic linking is the normal approach vendors use to address the security implications of package clones.

Finally, Clonewise checks to see if Debian Linux is tracking this package clone as being affected by that particular CVE. If it is not being tracked, then Clonewise will report the package as being potentially vulnerable as shown in Fig. 32.

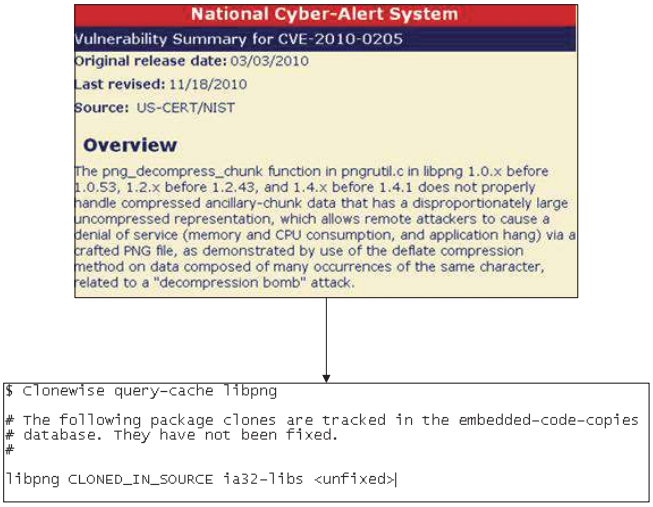

Fig. 31. Use-case of clone detection.

This process of finding outstanding vulnerabilities is applied to every CVE of interest in the database, and a final report is generated. The normal process is that a security analyst then verifies each reported vulnerability and eliminates any false positives.

One feature that we didn't implement was using the CVE summary's reference to vulnerable functions. We could potentially parse the sentence containing the vulnerable source filename to extract the vulnerable function and then check for the presence of this string in the source file. We did not do this because it requires the contents of each source tree to be maintained as signatures. This would increase the data storage requirements of Clonewise which we thought to be impractical. Potentially we could download the source as required, but this would cause issues doing analyses between distributions.

3.6 System Implementation

In this section we discuss the implementation.

3.6.1 Software

We implemented all of the above proposals in a complete system named Clonewise to identify package clones in Linux distributions. Clonewise automatically downloads a Linux distribution package repository and builds a database of signatures for each package. It then trains a model and uses statistical classification to perform clone detection for both the shared and embedded cases. We employ the Weka machine learning toolkit to perform the data mining aspects of our system. Our implementation uses C++ and shell scripting. It

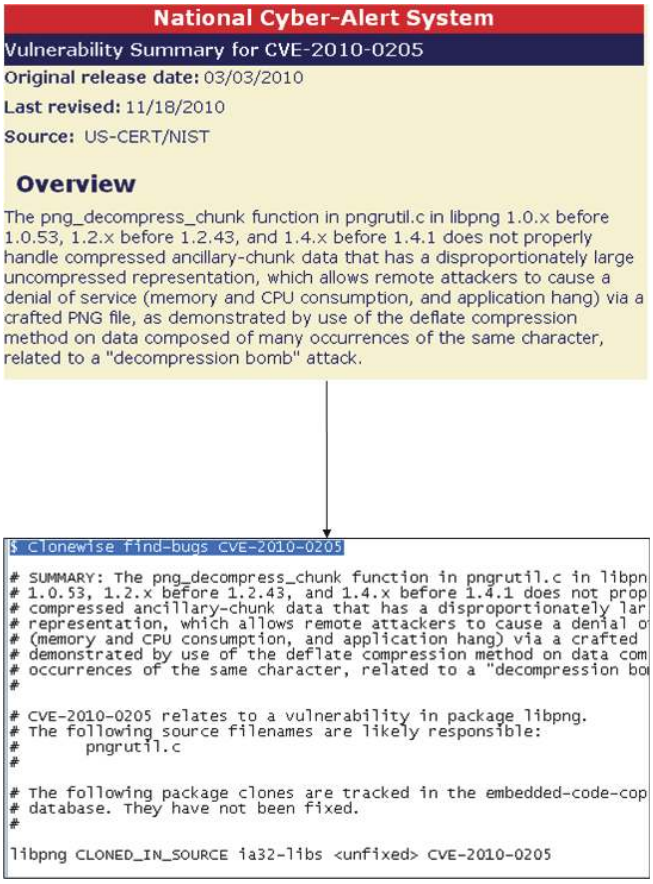

Fig. 21. Automated vulnerability inference.

consists of about 4,500 lines of code (LOC) to perform the package clone detection and security problem inference.

We performed an analysis of the Ubuntu Linux distribution and also performed some analysis of other distributions including Fedora 13 and Debian Linux. The package count in each distribution was in excess of 10,000.

Clonewise consists of multiple components. The components are divided into:

- Parsing Debian's package clone database
- Building the Clonewise database
- Training the classification models
- Clone detection

- Building a clone detection cache
- Querying the cache
- Finding cloned files
- Inferring vulnerabilities

We parse Debian's package clone database and convert it to XML or a text based format. We can optionally filter the results to ignore statically linked clones, or we can filter those clones which have been fixed, or those clones which remain unfixed. This component is necessary for generating the labelled training data to build a machine learnt model for classification. We can also find clones of files given as input. The output is the set of packages that have a similar file in their source trees. To build the Clonewise database, we download the entire source package repository for a Linux distribution, unpack the sources, and generate signatures. The signatures are the ssdeep signatures of the source trees for each package. We also build a package index relating binary packages to source packages. Finally we build a package dependency list for the purpose of identifying fixed clones. Clone detection is performed as explained earlier by using machine learning. XML output is optional. The clone detection cache is built using a cluster and the results of clone detection are stored to disk. The cache can be queried so that clone detection does not need to be performed again. XML output is optional. Finally, vulnerability inference relates clones in the cache to Debian's security tracker and the NVD CVE information.

3.6.2 Scaling The Analysis

Our system is effective and reasonably efficient at identifying clones in a single Linux package. However, in a typical Linux distribution there exist more than ten thousand individual packages. Our system would be impractically long if we performed clone detection on all packages without taking advantage of multicore and cluster computing.

3.6.2.1 Multicore

Given an input package to perform clone detection, Clonewise pairs that package with every other package in a Linux distribution. These package pairs are the input to a binary classification problem. Each binary classification problem can be evaluated independently

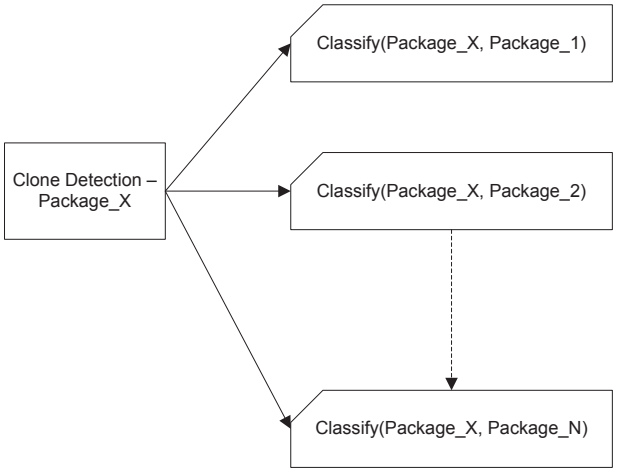

Fig. 33. Multicore.

of the other binary classification problems. This model of evaluation is embarrassingly parallel and leads to efficient parallel and distributed computing. The workflow is shown in Fig. 33.

We chose to solve this problem using multicore computing. We used the OpenMP multicore programming model [161] to implement our solution. OpenMP is a shared memory model based on the use of compiler directives. We parallelize the feature extraction and classification for each package pair. This process improves the speed it takes to perform clone detection on an individual package.

3.6.2.2 Clustering

Our multicore implementation improves the performance of clone detection on a single package. We use cluster computing to distribute clone detection of multiple packages. Each package can be scanned in parallel without regard to other packages and is also an embarrassingly parallel problem. The workflow is shown in Fig. 34.

We implemented our system using message passing with Open MPI [162]. In our implementation, a job is defined as performing clone detection on a single package. Since we have many packages to analyse, a master node distributes jobs to slave nodes. When the slaves complete a job they signal the master node requesting more work.

3.6.2.3 Running the Analysis

We analysed our Linux distribution using a high performance compute cluster. We purchased 4 hours of cluster computing time from the Amazon EC2 cloud computing

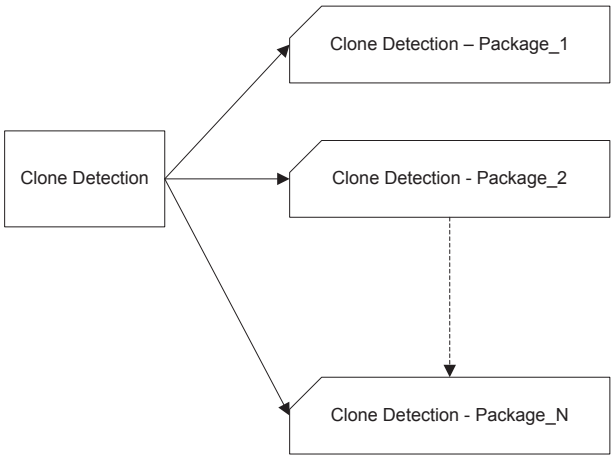

Fig. 34. Clustering.

service. We built a 4 node cluster with dual CPUs per node, Intel Xeon E5-2670, eight-core "Sandy Bridge" architecture), 60.5G of memory per node, and CPU performance identified as 88 EC2 compute units.

3.7 Evaluation

In this section we evaluate Clonewise using vendor labelled data and evaluate Clonewise's ability to discover security vulnerabilities.

3.7.1 Filenames as Features

In our first experiment we wanted to determine the distribution of unique filenames in a large Linux distribution. We tallied the frequency of filenames occurring in the Ubuntu Linux sources. We identified 3,077,363 unique filenames and ranked them according to their frequency of occurring. We sampled this distribution and performed a regression analysis. We observed that the frequency of filenames follows an inverse power law distribution with multiplicative constant 777892.740 and multiplicative exponent constant of 1.207. The R square value from the regression analysis was 0.928

3.7.2 Establishing the Ground Truth for Training and Evaluation

Debian Linux maintain a manually created database of packages that are cloned in their security tracker. We use this list of entries to establish the ground truth for our labelled data.

TABLE 1. ACCURACY OF SHARED PACKAGE CLONE DETECTION

CLASSIFIER	PRECISION	RECALL	ACCURACY	F-MEASURE
Naïve Bayes	0.47562	0.57687	0.98599	0.52137
Multi. Perceptron	0.80555	0.26806	0.98948	0.40225
C4.5	0.85878	0.68725	0.99436	0.76349
Random Forest	0.89881	0.70039	0.99499	0.78728
Rand. Forest (0.8)	0.96746	0.58607	0.99426	0.72994

TABLE 2. ACCURACY OF SHARED PACKAGE CLONE DETECTION

CLASSIFIER	TP/FN	FP/TN	TP RATE	FP RATE
Naïve Bayes	439/322	484/56296	57.69%	0.85%
Multilayer Perceptron	204/557	48/56732	26.81%	0.08%
C4.5	523/238	86/56694	68.73%	0.15%
Random Forest	533/228	60/56720	70.04%	0.11%
Random Forest (0.8)	446/315	15/56765	58.61%	0.03%

The Debian database was not originally created to be processed by a machine, so some of the data is not consistent in referencing packages with their correct machine readable names. Instead, shorthand or common names for packages and libraries are sometimes used. We cull all those entries which do not reference package sources and are therefore not suitable for our system.

We had two types of negative labeled entries. One case was for shared package clone detection, and the other was for embedded package clone detection. To establish true negatives for shared package clone detection, we randomly selected pairs of packages not in our true positive list. We label these package pairs as negatives. This data can be unclean since we observe the labeled true positives are incomplete, but even so, the true

TABLE 3. ACCURACY OF EMBEDDED PACKAGE CLONE DETECTION

CLASSIFIER	PRECISION	RECALL	ACCURACY	F-MEASURE
Naïve Bayes	0.10171	0.94349	0.35580	0.18362
Multi. Perceptron	0.75229	0.43101	0.94540	0.54802
C4.5	0.89235	0.75164	0.97396	0.81597
Random Forest	0.89067	0.72798	0.97225	0.80114
Asym. Bagging	0.53196	0.91852	0.93168	0.67372

TABLE 4. ACCURACY OF EMBEDDED PACKAGE CLONE DETECTION

CLASSIFIER	TP/FN	FP/TN	TP RATE	FP RATE
Naïve Bayes	718/43	6341/2808	94.35%	69.31%
Multilayer Perceptron	328/433	108/9041	43.10%	1.18%
C4.5	572/189	69/9080	75.16%	0.75%
Random Forest	554/207	68/9081	72.80%	0.74%
Asymmetric Bagging	699/62	615/8534	91.86%	6.72%

negatives we label are still useful for training our statistical model. In total, we obtained 761 labelled positives and 56780 negatives.

To generate true negatives for the embedded package clone detection, we paired up all packages that were reported as being embedded in X, ignoring those cases where X was the embedded code. This is what we expect our system to report – that X is embedded in Y and Z, but Y is not embedded in Z, and Z is not embedded in Y. In total, we were able to label 9149 negative cases.

3.7.3 Accuracy of Shared Package Clone Detection

We employed 10-fold validation from our labeled dataset to evaluate the accuracy of our system and experimented with a number of classifiers including Naïve Bayes [163], Multilayer Perceptron, C4.5 [164], and Random Forest [165]. Our results are shown in

Table 1 and Table 2. The data is very imbalanced and this skews the accuracy, which easily achieves better than 99%, because we can identify negative cases more easily than positive cases. We obtained the best result using the Random Forest classification algorithm. This classification algorithm performed significantly better than all other algorithms we evaluated. The true positive rate is 70.04%, the precision is 89.88%, the recall is 70.05%, and the f-measure is 78.73%, which we think is quite reasonable for the first implementation of an automated system for package clone detection. The false positive rate must be very low for our system to be used by Linux security teams. Our initial false positive rate is 0.11%. We then modified the decision threshold of the random forest algorithm to consider false positives as more significant than false negatives. Our false negative rate is 0.03% with a decision threshold of 0.8 which represents that 3 in every 10,000 package pairs is mislabeled as a positive. The true positive rate is lower with a higher decision threshold and is 58.61%. This is the trade-off we accept for a low false positive rate. There are about 18,000 source packages, so there are 18,000 package pairs that are classified when performing clone detection on an individual package. Therefore, if our training data were not noisy, we would predict 4 to 5 false positive per complete clone detection on an individual package. However, our labelled negatives are noisy, and some negatives are actually positives. Therefore, we think between 4 to 5 false positives is closer to an upper limit. This is reasonable for a manual analyst to verify and we think it will not cause significant burden on Linux security teams.

3.7.4 Accuracy of Embedded Package Clone Detection

We evaluated the embedded package clone detection using a number of classifiers including Naïve Bayes, Multilayer Perceptron, C4.5, and Random Forest. Our results are shown in Table 3 and Table 4. We obtained the best result using the C4.5 classification algorithm. The true positive rate was 75.16%, the false positive rate was 0.75%, the precision was 89.24%, the recall was 75.16%, and the f-measure was 81.60%. We then used this algorithm as a base classifier for our asymmetric bagging meta-classifier with 50 bags. This improved the true positive rate to 91.86% but also increased the false positive rate to 6.72%. We see this as an acceptable trade-off to improve the true positive rate.

3.7.5 Practical Package Clone Detection

As part of the practical results from our system we contributed 34 previously untracked package clones to Debian Linux's embedded code copies database. Thus, we feel that the package clone detection provides tangible benefit to the Linux community. We also verified if the embedded packages we detected were not in fact patched by the Linux vendors to link dynamically against a system wide library.

3.7.6 Vulnerability Detection

A consequence of package clone detection is determining if a clone is out of date and if it has any outstanding and unpatched vulnerabilities. As part of our work we detected over 30 vulnerabilities in Debian and Fedora Linux because of package clone issues by checking security sensitive packages manually, or using adhoc identification of out-of-date clones. The vulnerabilities in each package we found using clone detection are shown in Table 5 and 6.

3.7.7 Automated Vulnerability Detection

We performed a more recent evaluation of completely automated vulnerability inference over the years of 2010, 2011, and 2012. Clonewise reported 132 vulnerabilities across 19 packages. We submitted bug reports against each package to Debian Linux. Not all our submitted bug reports were actual vulnerabilities. Some reports were erroneous because Clonewise falsely identified a package clone when one did not exist. Another source of errors was that some bugs we reported as vulnerabilities could not be triggered, even though the clone was correctly identified and had unpatched CVEs. This was true of libpng image processing library being embedded in the syslinux boot loader package. Boot loading displays an image, but does not allow an attacker to control that image to trigger the vulnerability. A high number (64) of vulnerabilities were found in the ia32-libs package. This package contains a list of embedded libraries and is only updated by Debian on point releases. Debian informed us that this package would invariably contain vulnerabilities, but in the unstable release of Debian an alternative approach will be employed which resolves these issues by not embedding libraries.

TABLE 5. ADHOC DETECTION OF FEDORA LINUX VULNERABILITIES

PACKAGE	EMBEDDED PACKAGE
OpenSceneGraph	lib3ds
mrpt-opengl	lib3ds
mingw32-OpenSceneGraph	lib3ds
libtlen	expat
centerim	expat
mcabber	expat
udunits2	expat
libnodeupdown-backend-ganglia	expat
libwmf	gd
Kadu	mimetex
cgit	git
tkimg	libpng
tkimg	libtiff
ser	php-Smarty
pgpoolAdmin	php-Smarty
Sepostgresql	postgresql

Debian have not yet confirmed all our bug reports so we investigated each package manually to check that a package clone existed, and that the internal version number of the library was a version vulnerable to the CVE Clonewise reports. The results are shown in

TABLE 6. ADHOC DETECTION OF DEBIAN LINUX VULNERABILITIES

PACKAGE	EMBEDDED PACKAGE
boson	lib3ds
libopenscenegraph7	lib3ds
libfreeimage	libpng
libfreeimage	libtiff
libfreeimage	openexr
r-base-core	libbz2
r-base-core-ra	libbz2
lsb-rpm	libbz2
criticalmass	libcurl
albert	expat
mcabber	expat
centerim	expat
wengophone	gaim
libpam-opie	libopie
pysol-sound-server	libmikod
gnome-xcf-thumnailer	xcftool
plt-scheme	libgd

Table 7. It should be noted that the high number of true positives is largely accounted for by the 64 vulnerabilities we marked as such once Debian informed us that ia32-libs was by

TABLE 7. AUTOMATED VULNERABILITY INFERENCE

TP + FP (Packages)	19
TP (Packages)	10
FP (Packages)	9
TP + FP (CVEs)	132
TP (CVEs)	81
FP (CVEs)	51

nature collecting vulnerabilities until point releases. Nonetheless, we detected unverified vulnerabilities in more than 50% of the packages Clonewise reported. We performed this manual analysis stage of all vulnerabilities, except for those in ia32-libs, in less than 2 hours. Our results are shown in Table 8. In the case that these potential vulnerabilities are not confirmed by Debian, then Debian will still need to update their internal CVE database to report that those packages are unaffected. Therefore, our work still remains beneficial.

The results of our system demonstrate that we effectively identify vulnerabilities with a false positive rate that is practical for manual verification in a feasible amount of time.

3.8 Discussion

In this section we examine points of discussion, focusing on how our work has had practical consequence to Linux vendors. We also discuss how we think vulnerability reporting could be improved to take into account package cloning and embedding.

TABLE 8. AUTOMATED DETECTION OF POTENTIAL VULNERABILITIES

PACKAGE	EMBEDDED PACKAGE
freevo	feedparser
hedgewars	freetype
ia32-libs	* (see text)
libtk-img	tiff
likewise-open	curl
luatex	poppler
planet-venus	feedparser
syslinux	libpng
vnc4	freetype
vtk	tiff

3.8.1 Practical Consequences of Our Research

Key results of our research are the consequences and responses by Linux vendors in using our data. Linux vendors responded well and are using our results. Another consequence of our research was that we were given access to modify and update the Debian Linux embedded packages database and to enter vulnerabilities and other information in their security tracker. Debian Linux have also sought us to integrate our system into the security team's standard operating procedures and have offered access to a subdomain on the Debian website to offer a clone detection web service. We feel this validates our work and completing this integration is our next immediate focus. Red Hat Linux did not maintain an equivalent embedded packages list like Debian's, but have since created a database on their public wiki based on our research results. We believe similar

data would be useful for other Linux vendors, and non-Linux vendors such as the BSD family of operating systems and distributions.

3.8.2 Referencing CVEs in an advisory.

Ideally, CVE would include package relationships of vulnerabilities it reports. For example, if Firefox has a libpng vulnerability assigned a CVE, then libpng would be referenced as the canonical upstream package. The Common Platform Enumeration which canonically labels software and enables upstream tracking of packages may provide a useful system for tracking these related package clone vulnerabilities.

Concluding Remarks

In addition to the number of reported vulnerabilities and subsequent patching and resolution of vulnerabilities, we believe our research has much value in the practical approach of coping with embedded code and packages that may or may not be known about. We believe all vendors benefit in creating and maintain databases of embedded code and package-level clones in their package repository and our research fills a gap when the manual task of auditing in excess of 10,000 packages per distribution is too time consuming to be practical. There is much work as a consequence that could be applied to current practice to aid operating system security and we feel our work is a good step towards this goal.

Chapter 4: Wire - A Formal Intermediate Language for Binary Analysis

Wire is a intermediate language to enable static program analysis on low level objects such as native executables. It has practical benefit in analysing the structure and semantics of malware, which is a key topic in software similarity and classification. In this chapter we describe how an executable program is disassembled and translated to the Wire intermediate language. We define the formal syntax and operational semantics of Wire and discuss our justifications for its language features. We use Wire in Malwise, our malware variant detection system described in Chapter 5. We also examine applications for when a formally defined intermediate language is given. Our results include showing the semantic equivalence between obfuscated and non obfuscated code samples and identifying similarity between software programs . These examples stem from the obfuscations commonly used by malware and the areas of software theft detection, plagiarism detection, and code clone detection.

4.1 Introduction

Static program analysis is a useful tool that provides many benefits and applications. In summary, static analysis identifies the runtime behaviour of software. It does this analysis statically, meaning that the program is not executed. Applications of static analysis include detecting plagiarism of software code, optimising code during compilation, verifying software by proving the absence of certain bug classes, or in a weakened form, to identify software bugs. Static analysis is generally performed at the source level, but applications exist when we only have access to low level object code. The applications of low level static analysis include the analysis and detection of malware, detecting the theft of proprietary or licensed software, or detecting bugs in binaries which are the result of compilation or link-time conditions.

4.1.1 Motivation

Malware analysis and detection is a large motivation for why low level static analysis is required. Traditional static malware detection employed in commercial Antivirus has ignored program structure and semantics. Instead, pattern recognition on the raw byte-

level content has been the dominant technique in signature based detection. However, program structure such as that exhibited by the static control and data flow of the malware results in more robust and predictive characteristics. These characteristics or fingerprints are often invariant in large malware families and strains. Thus, by employing static analysis techniques, signature based detection is much more robust in the detection of variants such as polymorphic and metamorphic malware. Moreover, the use of program structure and semantics to extract robust features allows machine learning to detect novel samples of malware that we can predict as being malicious, but not belonging to known families of malicious software. Malware is almost always in binary form so a low level static analysis system that examines the binary form of executables content is required.

Software theft detection is another motivation for why low level static analysis is needed. Detecting unauthorized use of software code is desirable to protect industry investment. Similar to the malware variant detection problem, software theft detection extracts program structure and semantics and identifies unauthorized software copies by finding those same features in illegitimate software. It is necessary then to be able to examine closed source software by using low level static analysis.

More motivation is that of detecting the presence of software bugs in binaries. The purpose of this form of bug detection is not to replace traditional source level analysis, but complement it by providing an increased level of assurance. Source level analysis by definition is the unfinished form of a software that is lacking detail of how the program will be physically executed after assembly and linking. Bug detection in binaries by nature has access to the final form of the program where assembling and link time editing has been performed. This also provides additional assurance that the compiler has done what it was designed to do. This type of assessment is not only useful for development and quality assurance; it is also beneficial to system auditors who by requirements do not have access to software source.

Analysing binaries is hard. Many simple problems such as separating code from data are undecidable. Our first motivation stems from the desire of representing a binary in a manner that makes analysis easier. The native assembly in a binary is unfavourable for analysis. The reasons that native assembly is difficult to use are:

- Native CISC assemblies such as x86 have hundreds of instructions which requires significant and duplicate efforts to model for each class of static analysis.

- Native assemblies have instructions with side effects which make analyses require hidden information and assumptions.

- Native assemblies are platform dependent which requires separate static analysis implementations for each architecture.

This motivates us to use an intermediate language to represent native assembly. The intermediate language should be low level enough so that translation from assembly is not complex. It should also be high level enough so that traditional static analysis techniques can be applied.

We have implemented Wire and use it as the intermediate representation in performing static analysis on binaries and to detect malware variants in our research system Malwise. This chapter represents the formal description of the intermediate language we have implemented.

4.1.2 Innovation

The contributions of this chapter are as follows:

- We propose a new low level intermediate language and define its formal operational semantics.

- We propose methods to translate native assembly into our intermediate language.

- We propose applications of a formally defined intermediate language and demonstrate operational semantics can be used to show equivalence between metamorphic malware codes.

- We use our language as the basis for Malwise – our malware variant detection system.

4.1.3 Structure of the Chapter

The structure of this chapter is as follows: Section 4.2 explains how to translate native code into our intermediate language. Section 4.3 defines the formal syntax and operational semantics of our language. Section 4.4 demonstrates applications of our language to semantic equivalence. Section 4.5 demonstrates applications in software similarity and classification. Finally we present our concluding remarks.

4.2 Translating Native Code

The input to our system is an object file. The most typical case is an x86 binary. For Windows this is a portable executable (PE) object or an Executable and Linking format object under Linux. The system can also partially process Java class files, and C source code for the GNU compiler (GCC), however these aspects are experimental and not described in this chapter. Our system is designed as modular software that allows plugin extensions to inspect or modify the object file or the results of a static analysis. An XML configuration file determines which plugins will be loaded, the order in which they are processed, and at which stage of object file processing and static analysis they will be called.

The first stage is object file parsing. PE and ELF binaries contain information on how to access the object code and the dynamic linking information such as imported and exported functions. The object code is extracted and code is processed. For x86 binaries, a disassembly is performed.

The native representation contains instruction level information. These native instructions are translated to an intermediate language. All further static analyses operate on the intermediate language which by its construction is easier to analyse. Our implementation consists of 10,000 lines of C++ code for the disassembly to be translated to the intermediate language.

4.2.1 Disassembly

Disassembly is the process of translating machine code to assembly language [44]. This is the first stage of a static analysis. We employ the use of speculative disassembly in our framework as described in Section 2.4.1.

The set of addresses for a machine is defined by *A*. A native instruction in an executable is located in memory and is defined by the ordered pair. A disassembly is the set of ordered pairs.

$$D = \{(address, native\_assembly\_instruction)\}$$

Execution transfers from one instruction to another and is identified using speculative disassembly in Wire. There are two types of control transfers. The first type is the when execution transfers from one instruction to the subsequent or fall through instruction without following a branch or a call. The second type is when a branch or call is taken.

$$F_{fallthrough} = \{(u, v) | u, v \in D\}$$

$$F_{branch} = \{(u, v) | u, v \in D\}$$

4.2.2 Abstract Machines

The intermediate language used for the intermediate code runs on an abstract machine that has a correspondence to the actual machine. Typical models of computation for the abstract machine are register machines or random access machines. In Wire we use a register machine which has the following components:

- An unlimited number of uniquely labelled registers (in practice this number is limited by a 32 bit representation).
- A small number of instructions roughly into divided into arithmetic and control.
- An instruction pointer.
- A sequence of labelled instructions.
- A random access memory.
- An entry point.

4.2.3 Intermediate Code Generation

As described in Section 2.4.2, one approach to transform assembly into an intermediate language is to translate each instruction without maintaining intermediate state. We use

this approach also and in our framework we translate native assembly into three address code. This part of our system is not formally verified and we assume the translation is correct. The generated three address code is a list of ordered intermediate instructions.

$$native\_assembly\_instruction \rightarrow \{(\mathrm{n}, (Opcode, Operand_1, Operand_2, Operand_3))|\mathrm{n} \in \mathbb{N}\}$$

4.2.4 Register Mapping between Native Architectures and Wire

Wire assigns registers labels using a 32 bit number. Wire's registers overlap the native registers for the x86 architecture. That is, the 8 x86 registers numbered 0 to 7 in the native disassembly are reserved and map to the first 8 registers of the Wire intermediate language.

4.2.5 Label Generation

Native assembly memory addresses are not used in the intermediate language. Nor do all instructions have a memory location. Instead, a label is assigned at the beginning of a basic block. The labels contain an address to identify the location of a basic block. We make two passes over the assembly to generate label addresses. In the first pass, all branch targets are identified, and then a Wire label address is assigned to each native address. Finally, the native addresses are eliminated and labels are used to replace them.

$$targets = \{t|(x,t) \in F_{allthrough}\} \cup \{t|(x,t) \in F_{branch}\}$$

$$label: A \rightarrow L$$

Like the execution flow in disassembly, labelled basic blocks in the intermediate language have an execution flow.

$$L_{allthrough} = \{(u,v)|u,v \in L\}$$

$$L_{branch} = \{(u,v)|u,v \in L\}$$

4.2.6 Condition Code Generation

Condition codes represent arithmetic conditions. For example, an arithmetic instruction performing an assignment may store the fact that the operand is zero. In x86 assembly, arithmetic instructions such as subtraction also store information on inequalities such as one operand being less, greater, or equal to the other. In Wire, each possible condition is stored in a separate register. That is, there is a register storing equality, less than, zero

status etc. Each arithmetic instruction sets the set of these registers based on the operands of the instructions. These registers are set using Wire's *mkbool* instructions which can assign a register a Boolean value (a numeric 1 or 0) based on an inequality and its parameters.

4.2.7 Decompilation

Native instructions are translated into the Wire intermediate language, but after construction, the intermediate code is analysed to generate additional or replacement code. For example, Wire uses the PUSHARG instruction to give procedure calls arguments, however this requires decompilation to generate this information. Decompilation is used for the following components:

- Local variable reconstruction
- Procedure argument reconstruction
- Condition code elimination

The use of decompilation to generate IL instructions enables high level static analysis to be employed. Traditional source level analyses such as bug detection can use the decompiled results. This feature distinguishes itself from most other intermediate languages for reverse engineering except those specifically used for decompilation.

Local variable reconstruction transforms stack based memory access into much simpler register based variables. Procedure argument reconstruction extends the stack based memory analyses to identify arguments which are on the stack at call sites. This is done by reconstructing what the stack looks like at a call site and unwinding values from it. Condition code elimination transforms explicit use of condition codes and a branch on a condition code into a simpler branch on condition. The approach is to look at the reaching definition of the condition code at a branch on condition code and then to propagate the definition and transform the branch into the branch on condition.

4.2.8 Intermediate Code Optimisation

The generation of the intermediate language produces a very verbose and inefficient code. We transform this into a simpler code by using compiler style optimisations. The optimisations we employ are:

- Dead code elimination
- Constant propagation
- Constant Folding
- Copy Propagation

Dead code elimination or more correctly dead store elimination removes stores which are never subsequently read before they are redefined. Constant propagation and constant folding simply expressions and assignments using constants such that their result is calculated when possible during the optimization pass. Copy propagation eliminates extraneous copies/assignments that are often used to has temporary placeholders for further expressions.

4.3 Formal Syntax and Semantics

In this section we define our intermediate language's syntax, the abstract machine it runs on, and its operational semantics. We believe formally defining Wire is important because it allows formal reasoning about the assembly language it represents. One application that becomes possible is the ability to prove semantic equivalence between two different syntactical representations. The problem of semantic equivalence is central to the problem of metamorphic malware detection. We give a detailed description of the Wire language to make these proofs and to also give insight into the language features required to represent assembly language.

4.3.1 Syntax

Program p ::= p i | i

Instruction	i	::=	m| m t
Type	t	::=	u8_t
			| u16_t
			| u32_t
			| s8_t
			| s16_t
			| s32_t
Instructions	m	::=	*(r3) := r1
			| r3 := (*r1)
			| r3 := r1
			| r3 := n
			| r3 := uop r1
			| r3 := r1 bop r2
			| r3 := r1 bop n
			| mkbool r1 ucond
			| mkbool r1 bcond r2
			| nop
			| halt

```
                    | label l
                    | jmp l
                    | ijmp r
                    | if r1 cond1 jmp l
                    | if r1 cond2 r2 jmp l
                    | lcall s
                    | cast(r1, t)
                    | r3 := getpc()
                    | r3 := returnaddress()
                    | pusharg(n, r)
                    | r3 := malloc(r)
                    | free(r)
                    | r3 := alloca(r)
```

Operations	uop	::=	-|~|!
	bop	::=	+,-,*,/,%,>>,<<,|,&,^
Conditions	ucond	::=	== 0|!= 0
	bcond	::=	==|!= | >|>=|<|<=
Operands	v	::=	n (an integer literal)
			r (a register)

l (a label)

s (a symbol)

4.3.2 Functions

Instructions	I	::=	n → i
Heap	H	::=	nxn → n
Memory	M	::=	n → n
Register	R	::=	r → n
Labels	L	::=	l → pc
AllocAMemory	V	::=	nxn→n

Instructions: (maps instruction number to instruction)

Heap: (maps heap address and memory size to non overlapping memory addresses)

Register: (maps register name to numeric value)

Memory: (maps address to numeric value)

Labels: (maps label to instruction address pc)

AllocAMemory: (maps alloca address and memory size to non overlapping memory addresses)

Note that we assign each instruction a unique program counter address that is used internally to describe the semantics.

4.3.3 Abstract Machine State

Call Stack	C	::=	stack of (I,pc,A,V)

Argument Stack	A	::=	stack of (n,r)
Process State	P	::=	(I,L,H,M,C,A,V,pc)

CallStack: Where I is the current function label, pc is the return address, A is the argument stack for function I, and V is the alloca memory mappings for function I.

ArgumentStack: (argument stack for callee of current function) Where n is the argument index and r is the register argument.

4.3.4 Operational Semantics of Core Instructions

Operational semantics [34] describe the state transitions that occur from execution of a program. We follow the following format:

$$\frac{\begin{array}{c} premise\ 1 \\ . \\ . \\ . \\ premise\ n \end{array}}{(i, P) \Rightarrow P'} NAME$$

Where i is the current instruction, P is the current state and P' is the next state following execution of the instruction i.

For simplicity, in this section we only show instructions of a single typing. In practice we have separate instructions for 8, 16, and 32 bit types.

4.3.4.1 Control Flow Instructions

The control flow instructions handle conditional and unconditional branches.

$$\frac{L(l) \rightarrow pc'}{(jmp\ l, P) \Rightarrow P[pc = pc']} JMP$$

The JMP instruction implements an unconditional branch. It simply changes the program counter to the target of the branch. In the case above, it is a direct branch to a label.

$$\frac{R(n) \rightarrow pc'}{(ijmp\ r, P) \Rightarrow P[pc = pc']} IJMP$$

The IJMP instruction also implements an unconditional branch, but uses register contents as the branch target.

$$\frac{\begin{array}{c} R(r1) \rightarrow n1 \\ cond(n1) \rightarrow 0 \\ L(l) \rightarrow pc' \end{array}}{(if\ r1\ ucond\ jmp\ l, P) \Rightarrow P[pc = pc']} UCJMP - T$$

$$\frac{\begin{array}{c} R(r1) \rightarrow n1 \\ cond(n1) \neq 0 \end{array}}{(if\ r1\ ucond\ jmp\ l, P) \Rightarrow P[pc = pc + 1]} UCJMP - F$$

$$\frac{\begin{array}{c} R(r1) \rightarrow n1 \\ R(r2) \rightarrow n2 \\ cond(n1, n2) \rightarrow 0 \\ L(l) \rightarrow pc' \end{array}}{(if\ r1\ bcond\ r2\ jmp\ l, P) \Rightarrow P[pc = pc']} BCJMP - T$$

The CJMP-T instruction implements a conditional branch on a true condition to a branch target specified by a label. There are a number of possible conditions including less than, greater than, less than or equal to and so forth.

$$\frac{\begin{array}{c} R(r1) \rightarrow n1 \\ R(r2) \rightarrow n2 \\ cond(n1, n2) \neq 0 \end{array}}{(if\ r1\ bcond\ r2\ jmp\ l, P) \Rightarrow P[pc = pc + 1]} BCJMP - F$$

The CJMP-F implements a conditional branch on condition false.

$$(label\ l, P) \Rightarrow P[pc = pc + 1] LABEL$$

The LABEL instruction specifies a location in the instruction sequence. Wire does not assign individual addresses to instructions to specify locations, so whenever an instruction is the target of a branch a label must be specified.

$$(nop, P) \Rightarrow P[pc = pc + 1] NOP$$

The NOP instruction implements a no operation.

4.3.4.2 Arithmetic Instructions

The arithmetic instructions handle unary and binary operations. The binary operation instructions have a version where one of the arguments is a constant.

$$\frac{\begin{array}{c} R(r1) \to n1 \\ n3 = uop\ n1 \end{array}}{(r3 \coloneqq uop\ r1, P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n3]]}\ UOP$$

$$\frac{\begin{array}{c} R(r1) \to n1 \\ R(r2) \to n2 \\ n3 = n1\ bop\ n2 \end{array}}{(r3 \coloneqq r1\ bop\ r2) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n3]]}\ BOP$$

The OP instruction implements the arithmetic instructions. It is a function that takes 3 operands and modifies those operands as necessary. In practice, the 3rd operand is kept as a destination register when possible.

$$\frac{\begin{array}{c} R(r2) \to n2 \\ n3 = n1\ bop\ n2 \end{array}}{(r3 \coloneqq n1\ bop\ r2, P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n3]]}\ BOPC$$

The OPC instructions implements the OP instruction except 2 of the operands are registers and the 3rd operand is a constant.

4.3.4.3 Boolean Instructions

$$\frac{\begin{array}{c} R(r1) \to n1 \\ cond(n1) = 0 \end{array}}{\begin{array}{l}(r3 \coloneqq mkbool\ cond\ r1, P) \Rightarrow \\ P[pc = pc + 1, R[r3 \mapsto 1]]\end{array}}\ UMKBOOL - T$$

$$\frac{\begin{array}{c} R(r1) \to n1 \\ cond1(n1) \neq 0 \end{array}}{\begin{array}{l}(r3 \coloneqq mkbool\ cond1\ r1, P) \Rightarrow \\ P[pc = pc + 1, R[r3 \mapsto 0]]\end{array}}\ UMKBOOL - F$$

$$\frac{\begin{array}{c} R(r1) \to n1 \\ R(r2) \to n2 \\ cond(n1, n2) = 0 \end{array}}{\begin{array}{l}(r3 \coloneqq mkbool\ r1\ cond2\ r2, P) \Rightarrow \\ P[pc = pc + 1, R[r3 \mapsto 1]]\end{array}} BMKBOOL - T$$

$$\frac{\begin{array}{c} R(r1) \to n1 \\ cond2(n1, n2) \neq 0 \end{array}}{\begin{array}{l}(r3 \coloneqq mkbool\ r1\ cond\ r2, P) \Rightarrow \\ P[pc = pc + 1, R[r3 \mapsto 0]]\end{array}} BMKBOOL - F$$

4.3.4.4 Transfer Instructions

The transfer instructions handle assignments of either registers or constants.

$$\frac{R(r1) \to n1}{(r3 \coloneqq r1, P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n1]]} ASSIGN$$

$$(r3 \coloneqq n1, P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n1]] ASSIGNC$$

4.3.4.5 Memory Access Instructions

The memory access instructions handle reading and writing to memory.

$$\frac{\begin{array}{c} R(r1) \to n1 \\ M(n1) \to n2 \end{array}}{(r3 :=* (r1), P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n2]]} LOAD$$

The LOAD instruction implements a memory read.

$$\frac{\begin{array}{c} R(r1) \to n1 \\ R(r3) \to n2 \end{array}}{\big((* r3) := r1, P\big) \Rightarrow P[pc = pc + 1, M[n2 \mapsto n1]]} STORE$$

The STORE instruction implements a memory write.

4.3.4.6 Casting Instructions

The CAST instruction is an assignment instruction between operands of different types.

$$\frac{\begin{array}{c}R(r1) \rightarrow n1 \\ n3 = c\_cast(r1, src\_type, dst\_type)\ )\end{array}}{(r3 := cast(r1, t), P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto n3]]}\, CAST$$

4.3.4.7 Procedural Instructions

$$(lcall\ s, P) \Rightarrow P[pc = pc + 1, A = \emptyset] LCALL$$

The LCALL instruction implements an API or library call.

$$\frac{\begin{array}{c}C' = push(C, (l, pc + 1, A, 0)) \\ L(l) \rightarrow pc'\end{array}}{(call\ l, P) \Rightarrow P[pc = pc', C = C', A = \emptyset]}\, CALL$$

The CALL instruction implements a procedure call instruction to a label target. The return address (pc+1) is pushed onto the call stack.

$$\frac{\begin{array}{c}R(r3) \rightarrow pc' \\ L(l) \rightarrow pc' \\ push\big(C, (l, pc + 1, A, 0)\big) \rightarrow C'\end{array}}{(call\ r3, P) \Rightarrow P[pc = pc', C = C', A = \emptyset]}\, ICALL$$

The ICALL instruction implements an indirect procedure call to a register target.

$$\frac{\begin{array}{c}(l', pc', A', V') \rightarrow top(C) \\ pop(C) \rightarrow C'\end{array}}{(return, P) \Rightarrow P[pc = pc', C = C', A = \emptyset]}\, RETURN$$

The RETURN instruction implements a return from a procedure. The return address is stored at the top of the call stack. The memory allocated by ALLOCA instructions becomes freed after a return. Likewise, the argument stack is emptied.

4.3.5 Operational Semantics of Decompiled Instructions

A number of instructions in Wire are only generated after a stage that decompiles the specified object file.

4.3.5.1 Address Instructions

$$(r3 \coloneqq getpc(), P) \Rightarrow P[pc = pc + 1, R[r3 \mapsto pc]] GETPC$$

The GETPC instructions returns the address of the current instruction in the binary being analysed.

$$\frac{(l', pc', A', V') \rightarrow top(C)}{\begin{array}{c}(r3 \coloneqq returnaddress, P) \Rightarrow \\ P[pc = pc + 1, R[r3 \mapsto pc']]\end{array}} RETURNADDRESS$$

The RETURNADDRESS returns the return address of the current procedure.

4.3.5.2 Memory Allocation Instructions

$$\frac{\begin{array}{c}[h, h + r1) \notin H, h \in \mathbb{N} \\ H' \rightarrow H \cup [h, h + r1) \\ H'(h, r1) \rightarrow m \\ M(m) \rightarrow n\end{array}}{\begin{array}{c}(r3 \coloneqq malloc(r1), P) \Rightarrow \\ P[pc = pc + 1, H = H', R[r3 \mapsto n]]\end{array}} MALLOC$$

The MALLOC instruction implements dynamic memory allocation. It stores the allocation information on the heap (H).

$$\frac{\begin{array}{c}H(h, x) \rightarrow m \\ M(m) \rightarrow r \\ H - [h, h + x) \rightarrow H'\end{array}}{(free(r), P) \Rightarrow P[pc = pc + 1, H = H']} FREE$$

The FREE instruction frees dynamically allocated memory.

$$\frac{\begin{array}{c}top(C) \rightarrow (l', pc', A', V) \\ [v, v + r1) \notin V, v \in \mathbb{N} \\ V' \rightarrow V \cup [v, v + r1) \\ V'(v, r1) \rightarrow m \\ M(m) \rightarrow n \\ (l', pc', A', V') \rightarrow C'\end{array}}{\begin{array}{c}(r3 \coloneqq alloca(r1), P) \Rightarrow \\ P[pc = pc + 1, H = H', C = C', R[r3 \mapsto n]]\end{array}} ALLOCA$$

The ALLOCA instruction performs dynamic memory allocation for the current procedure. The memory does not require freeing and will be done so automatically when the procedure returns.

4.3.5.3 Procedural Instructions

$$\frac{push\big(A,(n,r)\big) \rightarrow A'}{(pusharg(n,r),P) \Rightarrow P[pc = pc + 1, A = A']}\, PUSHARG$$

The PUSHARG instruction pushes the contents of a register onto the argument stack. The argument stack is passed into the next called procedure. The PUSHARG instructions are generated as a result of decompilation to identify procedure arguments.

4.3.6 Three Address Code

The high level syntax we have described is not used internally by Wire. For that we employ a three address code. The semantic equivalence between the high level syntax and three address code is shown using the semantic function A for the high level syntax and the semantic function B for the three address code.

$A[\![*\,(r3) \coloneqq r1]\!] = B[\![STORE\ r1, -, r3]\!]$

$A[\![r3 \coloneqq *\,(r1)]\!] = B[\![LOAD\ r1, -, r3]\!]$

$A[\![r3 \coloneqq r1]\!] = B[\![ASSIGN\ r1, -, r3]\!]$

$A[\![r3 \coloneqq n1]\!] = B[\![ASSIGNC\ n1, -, r3]\!]$

$A[\![r3 \coloneqq f1(r1)]\!] = B[\![UOP_{f1}\ r1, -, r3]\!]$

$A[\![r3 \coloneqq f2(r1, r2)]\!] = B[\![BOP_{f2}\ r1, r1, r3]\!]$

$A[\![r3 \coloneqq f3(r1, n1)]\!] = B[\![BOPC_{f2}\ r2, n1, r3]\!]$

$A[\![nop]\!] = B[\![NOP -, -, -]\!]$

$A[\![label\ l]\!] = B[\![LABEL\ l, -, -]\!]$

$A[\![jmp\ l]\!] = B[\![JMP\ -, -, l]\!]$

$A[\![ijmp\ r]\!] = B[\![IJMP\ -, -, r]\!]$

$A[\![if\ r1\ cond1\ jmp\ l]\!] = B[\![UCJMP_{cond1}\ r1, -, l]\!]$

$$A[\![if\ r1\ cond2\ r2\ jmp\ l]\!] = B[\![BCJMP_{cond2}\ r1, r2, l]\!]$$

$$A[\![lcall\ s]\!] = B[\![LCALL\ -, -, s]\!]$$

$$A[\![r3 \coloneqq cast(r1, t1)]\!] = B[\![CAST_{t1}\ r1, -, r3]\!]$$

$$A[\![r3 \coloneqq getpc()]\!] = B[\![GETPC\ -, -, r3]\!]$$

$$A[\![r3 \coloneqq returnaddress()]\!] = B[\![RETURNADDRESS\ -, -, r3]\!]$$

$$A[\![pusharg(n, r)]\!] = B[\![PUSHARG\ n, r, -]\!]$$

$$A[\![r3 \coloneqq malloc(r)]\!] = B[\![MALLOC\ r, -, r3]\!]$$

$$A[\![free(r)]\!] = B[\![FREE\ r, -, -]\!]$$

$$A[\![r3 \coloneqq alloca(r)]\!] = B[\![ALLOCA\ r, -, r3]\!]$$

$$A[\![r3 \coloneqq mkbool\ r\ ucond)]\!] = B[\![UMKBOOL_{ucond}\ r, -, r3]\!]$$

$$A[\![r3 \coloneqq mkbool\ r1\ bcond2\ r2)]\!] = B[\![BMKBOOL_{bcond}\ r1, r2, r3]\!]$$

4.4 Applications in Semantic Equivalence

One application of a formally defined language is to prove properties of its programs. One type of proof that can be performed is an equivalence proof. Equivalence proofs are useful and we will examine the particular case of equivalence between obfuscated codes which is a commonly seen occurrence in malware. Our proofs work on the intermediate code only and assume the intermediate code generation has been performed correctly.

4.4.1 Semantic Equivalence of Obfuscated Code

A syntactic metamorphic malware technique is a method that changes the syntactic structure of the malware [17]. Though the syntactic structure changes in polymorphic malware, the malware semantically remains identical. The technique is predominantly used to evade byte level signature based detection and classification that is routinely employed by traditional Antivirus. Metamorphism borrows many of the techniques from the field of program obfuscation.

4.4.1.1 Dead Code Insertion

Dead code is also known as junk code and a semantic nop [17]. Dead code is semantically equivalent to a nil operation. Insertion of this type of code has no semantic impact on the malware. The insertion increases the size of the malware and modifies the byte and instruction level content of the malware.

An example of dead code insertion is shown below. The intermediate code is also shown. For simplicity we assume that the condition codes are not required as is the case when a future arithmetic instruction overrides earlier ones.

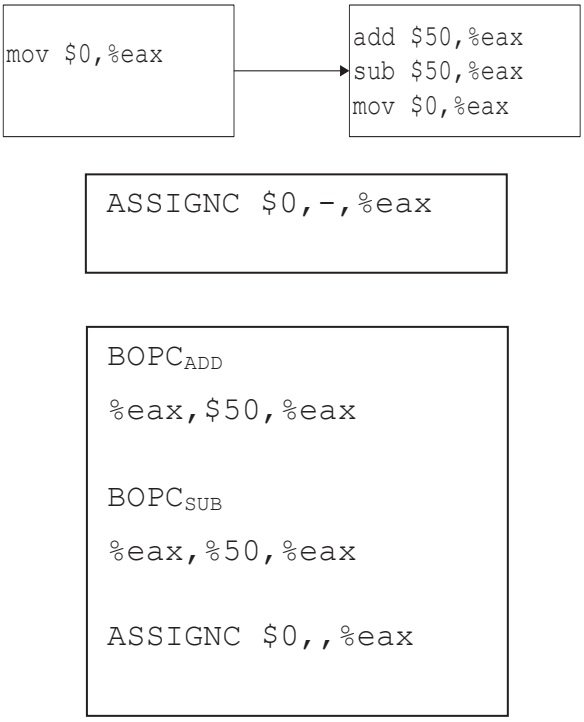

Fig. 35. Dead code insertion.

In the proof that we perform we show the equivalence between code using dead code and code that is not using dead code. The proof is carried out by simulating execution of each code sample and showing that the program states for both sequences are the same once complete.

Firstly, we map register names to register indices that will be used in all proofs in this section of the paper.

Reg_name("eax") = 0

Reg_name("ebx") = 1

Reg_name("zf") = 100

In the first part of the dead code equivalence proof we execute the instructions without the dead code.

$$\frac{n1 = 0}{("ASSIGNC\ 0, -,0", s) \Longrightarrow s'}$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto n1]\big]$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto 0]\big]$$

In the second part of the proof we execute the instructions with the dead code.

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = n1 + \ 50 \end{array}}{("BOPC_{ADD}\ 0, \$50,0", t) \Longrightarrow t'}$$

$$t' = P[pc = pc + 1, R[0 \mapsto n3]]$$

$$t' = P[pc = pc + 1, R[0 \mapsto n1 + 50]]$$

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = n1 - \ 50 \end{array}}{("BOPC\ _{SUB} 0, \$50,0", s') \Longrightarrow s''}$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto n3]\big]$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto (n1 + 50) - 50]\big]$$

$$\frac{\begin{array}{c} R(0) \rightarrow n1 \\ n3 = 0 \end{array}}{("ASSIGNC\ 0, -,0", t'') \Longrightarrow t'''}$$

$$t''' = P\big[pc = pc + 2, R[0 \mapsto n1]\big]$$

$$t''' = P\left[pc = pc + 2, R[0 \mapsto 0]\right]$$

Now we can see that t'''-pc = s'-pc which means they are semantically equivalent when ignoring the effect the code has on the program counter. We also note that s' and s'' are semantically equivalent. We have thus proven the obfuscated and deobfuscated code samples are equivalent.

This approach to proving semantic equivalence between code samples is useful to a malware researcher who wants to identify malware instances and variants.

4.4.1.2 Code Reordering

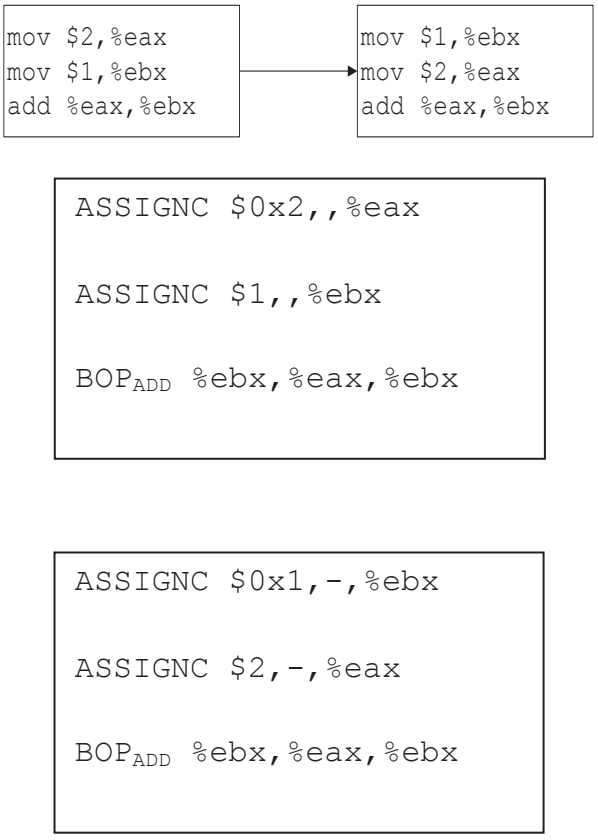

Fig. 36. Code reordering

Code reordering [18] changes the syntactic order of the code in the malware [17]. The actual or semantic execution path of the program does not change. However, the syntactic order as present in the malware image is altered..

We show an example of code reordering and the intermediate code generated from each sequence above.

For the first part of the proof we execute the first instruction sequence.

$$\frac{n1 = 2}{("ASSIGNC\ 2, -,0", s) \Longrightarrow s'}$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto n1]\big]$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto 2]\big]$$

$$\frac{n1 = 1}{("ASSIGNC\ 1,0,1", s') \Longrightarrow s''}$$

$$s'' = P\big[pc = pc + 1, R[0 \mapsto 2,1 \mapsto n1]\big]$$

$$s'' = P\big[pc = pc + 1, R[0 \mapsto 2,1 \mapsto 1]\big]$$

$$\frac{\begin{array}{c} R(0) \to n1 \\ R(1) \to n2 \\ n3 = n1 + n2 \end{array}}{("BOP_{ADD}1,0,1", s'') \Longrightarrow s'''}$$

$$s''' = P[pc = pc + 1, R[0 \mapsto 2,1 \mapsto n3]]$$

$$s''' = P[pc = pc + 1, R[0 \mapsto 2,1 \mapsto 3]]$$

For the second part of the proof we execute the second instruction sequence.

$$\frac{n1 = 1}{("ASSIGNC\ 1,0,1", t') \Longrightarrow t''}$$

$$t' = P\big[pc = pc + 1, R[1 \mapsto n1]\big]$$

$$t' = P\big[pc = pc + 1, R[1 \mapsto 1]\big]$$

$$\frac{n1 = 2}{("ASSIGNC\ 2,0,0", t') \Longrightarrow t''}$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto n1, 1 \mapsto 1]\big]$$

$$t'' = P\big[pc = pc + 1, R[0 \mapsto 2, 1 \mapsto 1]\big]$$

$$\frac{\begin{array}{c} R(0) \to n1 \\ R(1) \to n2 \\ n3 = n1 + n2 \end{array}}{("BOP_{ADD}1,0,1", t'') \Longrightarrow t'''}$$

$$t''' = P[pc = pc + 1, R[0 \mapsto 2, 1 \mapsto n3]]$$

$$t''' = P[pc = pc + 1, R[0 \mapsto 2, 1 \mapsto 3]]$$

Thus we see that t'''-pc = s'''-pc and therefore the two instruction sequences are semantically equivalent.

4.4.1.3 Opaque Predicate Insertion

An opaque predicate [19] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in the malware's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis. The opaque predicate we are examining is shown on the next page in Fig. 23.

In the first part of the proof we execute the first code sequence.

$$\frac{\begin{array}{c} R(0) \to n1 \\ R(0) \to n2 \\ n3 = n1 \; xor \; n2 \end{array}}{("BOP_{XOR}0, -, 0", t) \Longrightarrow s'}$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto n3]\big]$$

$$s' = P\big[pc = pc + 1, R[0 \mapsto 0]\big]$$

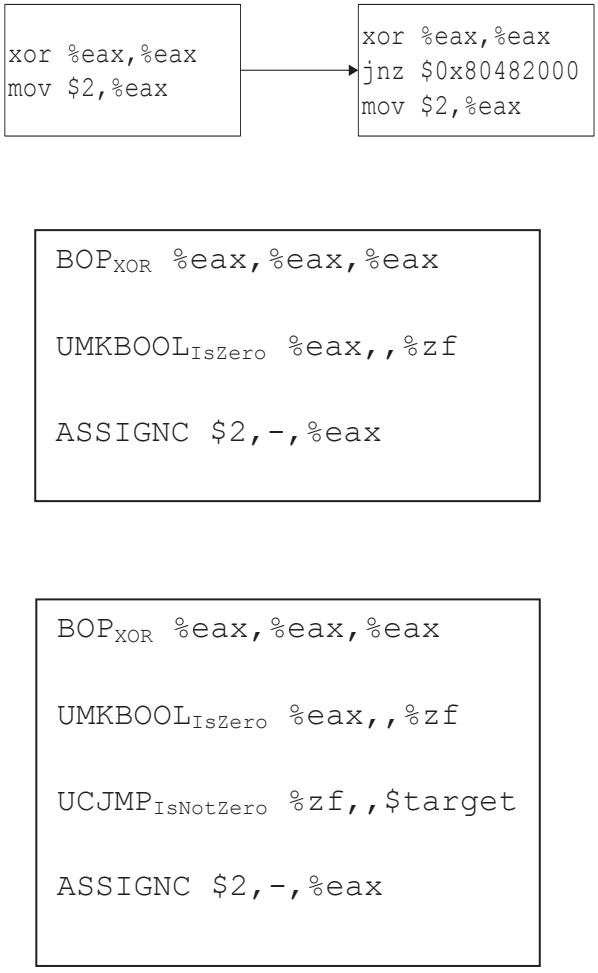

Fig. 37. An opaque predicate.

$$\frac{R(0) \to n1}{("UMKBOOL_{IsZero}\ 0{,}0{,}100", t') \Longrightarrow s''} MB_{ISZERO} - T$$

$$s'' = P\left[pc = pc + 2, R[0 \mapsto 0{,}100 \mapsto 1]\right]$$

$$\frac{n1 = 2}{("ASSIGNC\ 2, -{,}0", s) \Longrightarrow s'}$$

$$s' = P\left[pc = pc + 1, R[0 \mapsto n1{,}100 \mapsto 1]\right]$$

$$s' = P\left[pc = pc + 1, R[0 \mapsto 2{,}100 \mapsto 1]\right]$$

In the second part of the proof we execute the second code sequence.

$$\frac{\begin{array}{c}R(0) \to n1 \\ R(0) \to n2 \\ n3 = n1\ xor\ n2\end{array}}{("BOP_{XOR}0,-,0",t) \Longrightarrow t'}$$

$$t' = P\big[pc = pc + 1, R[0 \mapsto n3]\big]$$

$$t' = P\big[pc = pc + 1, R[0 \mapsto 0]\big]$$

$$\frac{R(0) \to n1}{("UMKBOOL_{IsZero}\ 0,0,100",t') \Longrightarrow t''} MB_{ISZERO} - T$$

$$t'' = P\big[pc = pc + 2, R[0 \mapsto 0,100 \mapsto 1]\big]$$

We see that register 100 is set which makes the conditional branch in the following instruction use a false condition.

$$\frac{R(100) \to 0}{("UCJMP\ _{IsNotZero}100,0,target",t'') \Longrightarrow t'''} JNZ - F$$

$$t''' = P\big[pc = pc + 3, R[0 \mapsto 0,100 \mapsto 1]\big]$$

$$\frac{n1 = 2}{("ASSIGNC\ 2,-,0",t''') \Longrightarrow t''''}$$

$$t'''' = P\big[pc = pc + 4, R[0 \mapsto n1,100 \mapsto 1]\big]$$

$$t'''' = P\big[pc = pc + 4, R[0 \mapsto 2,100 \mapsto 1]\big]$$

Thus we see that s''-pc=t''''-pc and this proves semantic equivalence.

4.4.2 Assisted and Automated Theorem Proving

The manual proofs shown in the previous section are useful. However, a more automated approach is beneficial. Algebraic specification [43] has been used in previous research to combine algebraic semantics [33] and theorem proving. Our work is different and uses operational semantics. Proof assistants may be used by an analyst. An alternative is to use automated theorem provers such as those for Satisfiability over Modulo Theories (SMT). These solvers can solve 1st order logic problems in a number of theories including bit

vectors. Public solvers are freely available [166]. SMT solvers have been used in the past to perform semantic NOP detection [17] and show equivalence between the code in basic blocks of two programs [167]. Our work gives a semantic basis and theory for these solvers to be used.

4.5 Applications in Software Similarity and Classification

Another application of our intermediate language is the detection of similar software. This has uses in malware variant detection, plagiarism detection, and software theft detection.

4.5.1 Software Isomorphism

L is an intermediate language representing a translation of assembly code. For each three address code, the label associated with its basic block is also maintained.

$$L = \{(\mathrm{l}, (Opcode, Operand_1, Operand_2, Operand_3)) | \mathrm{l} \in \mathbb{N}\}$$

4.5.1.1 Interprocedural Control Flow Graph (ICFG)

The interprocedural control flow graph (ICFG) represents both control flow graphs of each procedure (the intraprocedural control flow) and the call graph (the interprocedural control flow). In our intermediate language we can define it as:

$$ICFG = (V, E)$$

$$V = \{c | (n, (o, a, b. c)) \in L \wedge o = LABEL\}$$

$$E = \{(s, d) | (s, (o, a, b, d)) \wedge o \in \{CALL, JMP, UCJMP, BCJMP\}\}$$

This definition of the ICFG is not conservative since we ignore indirect calls and branches. However, it is suitable for the purposes of software similarity detection.

To detect if two programs are equal, we can approximate this by testing for isomorphism between their interprocedural control flow graphs.

An isomoprhism of graphs G and H is a bijection between the vertex sets of G and H

$$f: V(G) \rightarrow V(H)$$

such that any two vertices u and v of G are adjacent in G if and only ig f(u) and f(v) are adjacent in H.

No polynomial time algorithm has been constructed for isomoprhism testing, however it has not been formally proven that the algorithm is exclusively outside of complexity class P. In practice, graph invariants can be used to speed up testing. A graph invariant is a property of a graph that remains constant amongst its isomorphisms. One graph property of interest is the degree sequence of a graph. The degree sequence of an undirected graph is the non-increasing sequence of its vertex degrees. Using this graph invariant we can apply set equality testing to definitely show two graphs are non isomorphic. This allows us in the software isomorphism problem to show that two programs are not equivalent with respect to their interprocedural control flow graphs.

4.5.1.2 Call Graph

The call graph shows the control flow between procedures and intraprocedural control flow. Formally, we can define an approximation of the call graph using our intermediate language.

$$CallGraph = (V, E)$$

$$V = \{c | (n, (o, a, b.c)) \in L \wedge o = CALL\}$$

$$E = \{(s, d) | (s, (o, a, b, d)) \wedge o \in CALL\}$$

We can then apply our graph isomorphism testing to show equivalence between programs based on their call graphs.

4.5.2 Software Similarity and Classification

The software similarity problem extends the software isomorphism problem to show that two programs are approximately equal to each other to some degree. The similarity between two programs is typically represented as a real number [0,1] where 0 identifies the programs as being totally dissimilar and 1 shows that the programs are isomorphic.

There are a number of approaches to showing the similarity between two programs. The general idea is to use specific features of the program and then to construct a signature out

of these features. Once a signature has been constructed, a similarity function solves the pairwise similarity problem between these signatures.

4.5.2.1 Instructions

For this problem we use the instructions of our intermediate code as a feature to construct a signature.

The intermediate language disassembly is:

$$\mathrm{D} = \{(\mathrm{address}, \mathrm{n}, \mathrm{Opcode}, \mathrm{Op\_1}, \mathrm{Op\_2}, \mathrm{Op\_3}) | \mathrm{n} \in \mathbb{N}\}$$

The 'birthmark' of the program is a fingerprint or signature. In this case we use a bag of opcodes of the instruction listing as our birthmark. The birthmark is represented mathematically as a vector.

$$V_i = \left| \begin{array}{c} S = \{s| \\ s \in D, \\ s = (address, n, (Opcode, Op\_1, Op\_2, Op\_3)), \\ i = Natural(Opcode) \end{array} \right|$$

where Natural is a bijective mapping between the opcodes and a natural number.

The similarity between two birthmarks can be defined in terms of the Euclidean distance between two vectors representing the programs.

$$similarity(P, Q) = 1 - \frac{|P - Q|}{max(|P|, |Q|)}$$

4.5.2.2 The Small Primes Product

Sometimes it is useful to represent a sequence of instructions and compare them to another sequence of instructions irrespective of the instruction ordering. This is beneficial when facing malware which reorders its code without changing the semantics. Although we earlier showed a semantic method to identify equivalence, we note that there are syntactic methods as well.

The Small Primes Product [113] was proposed to tackle this problem. Each possible opcode is represented by a unique prime number. Given an instruction sequence, the

primes associated with each opcode are multiplied together. The resulting prime product is unique for a given set of unique instructions.

Formally, for a sequence of instructions I, the Small Primes Product (SPP) is:

$$p: i \rightarrow Prime(i)$$

$$SPP(I) = \prod_{i \in I} p(Opcode(i))$$

We can use the small primes product to create a set of basic blocks, where each basic block is represented by the small primes product of the basic block's instructions. We can then use all of the set similarity measures described in Chapter 2 to show the similarity between two programs.

4.5.2.3 API Calls

The API calls made by a program are another type of feature that can be used when creating birthmarks to show similarity between programs. API Calls can also be represented using vectors and the similarity between two programs determined by the Euclidean distance as is the same when using opcodes.

The birthmark is thus represnted as:

$$V_i = \left| \begin{array}{c} S = \{s\} | \\ s \in D, \\ s = (address, n, (Opcode, Op\_1, Op\_2, Op\_3)), \\ Opcode = LCall, \\ i = Natural(Op\_3) \end{array} \right|$$

where Natural is a bijective mapping between the API call target and a natural number.

4.5.2.4 Control Flow

We can define the similarity the similarity between two interprocedural control flow graphs in terms of their edit distance. We can also apply the same to the call graph.

The graph edit distance (GED) between two graphs is defined as the minimum number of graph edit operations to transform one graph to the other.

For two graphs G=(V1,E1) and H=(V2,E2), the similarity between those graphs can b e defined as:

$$similarity(G,H) = 1 - \frac{ged(G,H)}{\max(|V1| + |E1|, |V2| + |E2|)}$$

4.5.2.5 Classification and Clustering

Representing programs by feature vectors allows for the traditional application of machine learning techniques. Classification is the task of assigning a class to an object, after a period of training with a labelled data set. Classification can be used for instance to detect the difference between malicious and non malicious programs. Clustering is an unsupervised machine learning method which groups together similar objects according to some definition of closeness or similarity. The Euclidean distance is one such measure of similarity.

The typical input to a classification or clustering algorithm is a feature vector. The feature vectors described earlier in this section is exactly the kind of input that these algorithms work on. Therefore, machine learning can be directly applied to these objects.

4.5.3 Software Embedding

Another software similarity related problem is to determine if one program is embedded in another. A typical example of this is in the virus detection problem. A formal approach to tackle this problem is by using the maximum common subgraph.

Formally, given two graphs G and H, the maximum common subgraph (MCS) is the largest subgraph, S, of G that is isomorphic is a subgraph of H.

To determine if V is a viral infection in G, we can test if V is the maximum common subgraph of G where the graphs represent call graphs or interprocedural control flow graphs. Formally, V is a viral infection of G if V=MCS(G). We can allow for approximate solutions potentially accounting for mutations of V using the graph edit distance. V is a viral infection of G if ged(V,MCS(H)) > e.

Concluding Remarks

Wire is an intermediate language that enables analysis of executable programs. Wire has unique features including the ability to integrate the results of decompilation into the core language. While this makes the translation possibly unsound, for the majority of programs the translation is effective and useful for analysis. A formal definition of the operational semantics of the language enables researchers to formally reason about assembly code. We demonstrated proofs of program equivalence between obfuscated and non obfuscated code samples. We also demonstrated that using the syntax and semantics lets us formally model software similarity problems. These applications reinforce our belief that a formal approach to describing Wire has practical benefits.

Chapter 5: Malwise II - Control Flow-based Malware Variant Detection

Static detection of polymorphic malware variants plays an important role to improve system security and is an important area in software similarity and classification. Control flow has shown to be an effective characteristic that represents polymorphic malware instances and construction of which was shown formally in the last chapter. In this chapter, we propose a similarity search of malware using novel distance metrics of malware signatures based on control flow. We describe a malware signature by the set of control flow graphs the malware contains. We first experiment with string based signatures. We then try using vector and set of strings based signatures. We propose two approaches and use the first to perform pre-filtering. Firstly, we use a distance metric based on the distance between feature vectors. The feature vector is a decomposition of the set of graphs into either fixed size k-subgraphs, or q-gram strings of the high-level source after decompilation. We also propose a more effective but less computationally efficient distance metric based on the minimum matching distance. The minimum matching distance uses the string edit distances between programs' decompiled flow graphs, and the linear sum assignment problem to construct a minimum sum weight matching between two sets of graphs. We implement the distance metrics in a complete malware variant detection system. The evaluation shows that our approach is highly effective in terms of a limited false positive rate and our system detects more malware variants when compared to the detection rates of other algorithms.

5.1 Introduction

Malware classification and detection can be divided into the tasks of detecting novel instances of malware, and detecting copies or variants of known malware. Both tasks require suitable feature extraction, but the class of features to be extracted is often dependant on which problem is trying to be solved. Detecting novel samples primarily uses statistical machine learning. On the contrary, malware variant detection uses the concept of similarity searching to query a database of known instances. These similarity queries or nearest neighbour searches are known in machine learning as instance-based learning.

Instance-based learning uses distance functions to show dissimilarity and hence similarity between objects. If the distance function has the mathematical properties of a metric, then algorithms exist that enable more efficient searching than an exhaustive set of queries over the database.

Traditional and commercial malware detection systems have predominantly utilised static string signatures [95, 96] to query a database of known malware instances. Static string signatures capture sections of the malwares' raw file content that uniquely identifies them. String signatures have been employed because they have desirable performance characteristics that enable real-time use [168]. However, string signatures perform poorly when faced with polymorphic malware variants. Exact string matching also ineffectively handles closely related but non-identical signatures.

Polymorphic malware variants have the property that the byte level content of the malware changes between instances. This can be the result of source code modifications or self mutation and obfuscation to the malware. Signatures that rely on fixed byte level content are unable to capture the invariant characteristics between these polymorphic instances.

Efficient real-time systems have been proposed that examine the run-time behaviour of programs to identify malicious behaviour [107]. Malicious behaviour can either conform to a policy of malicious intent, or reassemble the behaviour of a program instance, known in advance to be malicious. However, static detection of malware has advantages - it does not require conditional, untrusted or sandboxed execution of malware once the original contents of the malware are visible. Unpacking is the processing of revealing that code and typically occurs before the malware performs its malicious intent. Many Antivirus products implement static unpacking for known packers, and this accounts for the majority of samples. However, for novel packing techniques unpacking is often a dynamic process making effective static analysis against novel malware a hybrid approach. Additionally, snapshots of process images can be taken at runtime, thus avoiding the most common packing issues and can be used to statically identify if those processes belong to a known malware family.

A variety of algorithms have been employed to statically detect malware variants with superior classification compared to string based approaches. An n-gram is one of all possible fixed sized substring extracted from a larger string. Our work is directly related to the n-gram concept. N-grams of byte level, or instruction level content, utilising machine learning and classification has been proposed. However, n-grams are ineffective with polymorphic malware because of the changes the instruction level content.

More detailed program analysis techniques have been employed on the instruction level content to extract high level features. Data flow analysis reveals useful high level features that are more invariant than instruction content alone. Likewise, abstract interpretation using specific domains reveals desirable features. Efficiency still remains a concern for industrial usage.

Control flow has also been used to overcome the limitations of byte level and instruction level classification [110]. Control flow has the desirable property that instruction level changes do not affect the resulting flowgraphs. Control flow is observed to be more invariant in polymorphic malware [115].

Our work is based on the set of control flow graphs of the program. In some literature, the individual control flow graphs are merged together into a single interprocedural control flow graph (ICFG). However, for our work, we represent each procedure with a separate graph and therefore consider the set of graphs problem. In contrast, most malware analysis using control flow has focused on analysing a single call graph. The advantage of considering each control flow graph individually is that we can apply the decompilation technique of structuring which is not possible with the ICFG.

The challenge of using graphs to show similarity is that accurately measuring similarity such as when using the graph edit distance does not perform in polynomial time. Therefore, research must investigate methods that make using graphs feasible for large scale malware detection. The real or near real-time constraints of Antivirus software make this challenge even more significant. The challenge increases again when complex graph based objects are considered such as the set of graphs signature our research investigates.

5.1.2 Motivation

This work is motivated by several real-world applications that would benefit from control flow-based malware variant detection.

5.1.2.1 A replacement to traditional Antivirus

Traditional AV suffers from the inability to detect malware variants efficiently from large databases. Control flow is effective and our system makes such a system practically efficient when using large databases. Moreover, it would reduce the size of the database required on the end host due to requiring fewer samples to recognise a large malware family.

5.1.2.2 To cluster interesting samples

AV vendors need to know which malware families are significant enough that they require manual analysis. Our system could be used to identify variants and group them to their family. If many instances of a family are identified, then that family may require human analysis to determine what the real impact of the malware is. Moreover, our system could be used to avoid redundancy of work. In this case, a human analyst would not perform more work on already analysed family.

5.1.2.3 Incident Response

An accurate system that identifies what family of malware a sample belongs to could be used in incident response. An analyst could attribute authors of malware to the family it belongs to or identify what disinfection procedures are required and what impact a sample has on an infected system.

5.1.3 Innovation

Our work is based on control flow classification but we make the following contributions:

We propose a system that performs similarity searching of sets of control flow graphs. We perform the search in close to real-time in the expected case. No other system has demonstrated near real-time performance for this use of control flow based signature.

We propose using the Levenshtein distance, the NCD and the BLAST algorithms to perform similarity comparisons using novel string based malware signatures.

We propose using fixed size k-subgraphs to construct a feature vector approximating a set of graphs. Using a vector representation improves efficiency significantly and has not been used before.

We also propose the novel use of a polynomial time algorithm to generate q-gram features of decompiled control flow graphs to construct a feature vector. These features are shown to have more accuracy than k-subgraphs and can be constructed faster than k-subgraphs. K-subgraph feature construction is not known to take polynomial time. Q-grams of decompiled graphs have not been used before for malware classification.

We propose a distance metric between two sets of graphs based on the minimum matching distance. The minimum matching distance uses the linear sum assignment problem. It has been used previously with sets of vectors, but not sets of graphs. The minimum matching distance has not been used before in malware classification.

We implement these ideas in a complete prototype system and perform an evaluation on a set of benign binaries and on real malware, including those malware that are packed and polymorphic. The evaluation demonstrates the system is effective and fast enough for potential desktop adoption.

5.1.4 Structure of the Chapter

The structure of this chapter is as follows: Section 5.2 defines the malware classification problem and our approach. Section 5.3 describes the unpacking and general static analysis component of the system. Section 5.4 examines string based signatures. Section 5.5 describes the vector based pre-filtering stage used in classification. This is a course grained classification process. Section 5.6 describes the fine grained classification algorithms. Section 5.7 describes distance metrics and the nearest neighbour similarity search. Section 5.8 performs an evaluation using benign and malicious samples. Section 5.9 examines limitations and discusses points of interest. Section 10 looks at future work. Finally, we present some concluding remarks.

5.2 Problem Statement and Our Approach

5.2.1 Problem Statement

New programs that are discovered on the host system are inspected to determine if they are malicious or benign. Unknown malware are detected by calculating their similarity to existing malware. A high similarity identifies a malicious variant. Existing malware are collected from honeypots and other malicious sources to construct a database of malware signatures. The described malware variant detection problem is equivalent to the software similarity search problem.

The software similarity problem is to determine if program p is a copy or derivative of program q and is defined in Section 1.4. The software similarity problem is extended to operate over a database of programs. We use the nearest neighbour search.

To recap Section 2.8, the nearest neighbour range search is defined as:

Given a set of objects P and a query q, and a range r > 0, the range nearest neighbours (rNN) query is to find a result set rNN that consists of objects such that for any $p' \in rNN, p' \in P, dist(p', q) \leq r$.

A slight variation is to find any nearest neighbour in range. This variation can improve performance.

Definition 58. *Given a set of objects P and a query q, and a range r > 0, the any range nearest neighbours (rNN) query is to find any object p, such that* $p \in P, dist(p, q) \leq r$.

The distance function used in the nearest neighbour search is d(p,q) =1-s(p,q).

5.2.2 Our Approach

Our approach builds a signature or birthmark of a malware based on the set of control flow graphs it has. We compare signatures using distance metrics to show similarity. In our experiments we evaluate constructing strings to represent signatures and then use a variety of string metrics to show signature similarity. We also use a vector based signature which we observe is more effective and efficient than our string signatures. Finally, we add a set of strings signature which we observe as more accurate and is used to refine the vector based result.

Malware is first unpacked to remove obfuscations. Control flow is reconstructed and the control flow graphs decompiled and structured into strings. Malware variants are detected by identifying existing malware the query programs are related to. Pre-filtering is used to provide a list of potentially related malware. The pre-filtering algorithm is based on constructing a feature vector to represent the query programs and malware. Either of two algorithms can be used to extract features. Firstly, subgraphs of size k are used to represent features. Alternatively, q-grams are extracted from the strings representing the structured graphs. Q-grams our equivalent to n-grams when using strings from decompiled control flow graphs. Using either algorithm for feature extraction, the most relevant features are used to construct a feature vector. The pairwise similarity between two feature vectors employs a distance function on the pair of vectors. Vectors that are close to each other are indexed to the same bucket. To identify candidates with high similarity to existing malware, a metric similarity search is performed using Vantage Point trees [81].

To compare the query to the candidate malware, a more accurate pairwise distance function is used. Each control flow graph from one program is assigned a unique mapping to a flowgraph from the other program. This mapping intuitively shows the flowgraphs represent the same procedure. The mapping is assigned a weight and the mappings chosen by considering it as an optimization problem. The mappings are chosen to minimize the sum of all weights associated with the mappings. The weight is the distance between flowgraphs and is based on the string distance between structured graphs. This sum weight is known as the minimum matching distance and is known to be metric. Metric Access Methods using DBM trees [82] are used to perform a similarity search.

5.3 Unpacking and Static Analysis

5.3.1 Unpacking

The query program may have its real contents hidden using the code packing transformation [20]. Code packing encrypts, compresses, or obfuscates the code by dynamically generating the original program at runtime. This obfuscation layer is removed using automated unpacking. The unpacking process employs application level emulation as proposed in previous research [60].

```
Procedure           ::= StatementList
StatementList       ::= Statement | Statement StatementList
Statement           ::= Return | Break | Continue | Goto
                      | Conditional | Loop | BasicBlock
Goto                ::= 'G'
Return              ::= 'R'
Break               ::= 'K'
Continue            ::= 'C'
BasicBlock          ::= SubRoutineList
SubRoutineList      ::= 'S' | 'S' SubRoutineList
Condition           ::= | ConditionTerm
                      | ConditionTerm NextConditionTerm
NextConditionTerm   ::= '!' Condition | Condition
ConditionTerm       ::= '&' | '|'
IfThenCondition     ::= Condition | '!' Condition
Conditional         ::= IfThen | IfThenElse
IfThen              ::= 'I' IfThenCondition StatementList 'H'
IfThenElse          ::= 'I' IfThenCondition StatementList
                      'E' StatementList 'H'
Loop                ::= PreTestedLoop | PostTestedLoop
                      | EndlessLoop
PreTestedLoop       ::= 'W' Condition 'StatementList '}'
PostTestedLoop      ::= 'D'  StatementList '}' Condition
EndlessLoop         ::= 'F' StatementList '}'
```

Fig. 38. The grammar of a structured string.

5.3.2 Dissasembly and Control Flow Reconstruction

In our system, an unpacked program is disassembled using speculative disassembly [44]. The disassembly is translated to an intermediate language using the Wire static analysis framework. The control flow is reconstructed into control flow graphs for each procedure [60] based on the intermediate code. This and the remaining components of the static analysis are architecture independent.

The control flow graphs are normalized to eliminate unnecessary jumps such as when an unconditional branch is used to divide a basic block in two. This is typically done by a malware for the purpose of changing its byte level content and static string signature. The result after control flow reconstruction is the set of control flow graphs associated with each identified program procedure

5.3.3 Structuring

Structuring is a reverse engineering and decompilation technique to transform a control flow graph into its high level source code representation. We use a structuring algorithm to transform the control flow graphs into strings. The intuition is that similar control flow graphs are structured into similar strings [60]. This effectively forms a locality sensitive hash. The structuring algorithm we use is based on the algorithm used in the DCC decompiler [41]. The grammar for the resulting string is shown in Fig. 38 using alphabet Σ. Formally, for program P and for a control flow graph c, let $P = \{c \in G\}$. A structuring function for a control flow graph is defined as s and a structuring algorithm for program P is defined as S.

$$S : P \to M$$
$$c \mapsto s(c)$$

5.4 String Based Signatures

We first experimented with string based signatures. This approach was eventually discarded in favour of representing signatures using vectors, however insight into the malware detection problem is gained by examining these novel techniques.

5.4.1 Feature Extraction

There is an associated string representing control flow for each procedure identified in the binary. These strings and ordered and concatenated to form a single string to represent the control flow of the entire binary. The substrings are deliminated by a specific character (eg 'Z'), The novelty of our approach is to order and concatenate the control flow graph strings into a single unified string based signature, which allows us to use traditional string similarity metrics for malware classification. The order of the concatenated strings is determined by features of the procedure, which are used as sort keys. Procedures that have duplicate sets of keys are removed from the analysis. The keys in order of importance are:

- Number of IL instructions in procedure
- Length of string representing decompiled control flow graph
- Number of basic blocks in procedure
- Number of edges in control flow graph
- Number of procedure's callers
- Number of procedure's callees

5.4.2 Indexing Using String Metric Access Methods

String metrics are proposed to show the similarity between a query signature and malware signatures. A similarity search over the malware database enables the malware variant classification. The string metric we propose is the Levenshtein or edit distance. The Levenshtein distance between two strings gives the minimum number of insertions, deletions and substitutions to transform one string to the other. The run-time complexity is O(nm) where n and m are the lengths of the strings. The Levenshtein distance forms a metric. A metric allows efficient indexing and searching of objects. Sequence alignment algorithms also provide suitable string distances. The Smith-Waterman algorithm is an optimal local alignment algorithm. We propose using Metric Access Methods to perform a range similarity search. The similarity search finds all malware signatures similar to the

query with at most r edit operations to transform the query signature to the malware signature.

String metrics may also be used on the byte-level content of the unpacked malware. We evaluate the effectiveness of using byte-level content in Section 5.9, and compare it to our proposed signature of using decompiled control flow graphs.

5.4.3 Indexing Using Genome Strings and Blast

The Smith-Waterman algorithm gives the optimal local sequence alignment between two strings. The local sequence alignment seeks to provide an alignment between two strings taking into account the alignment of substrings. Local sequence alignment is used often in the field of Bioinformatics to identify similarity between genome sequences. It forms a metric allowing for Metric Access Methods for indexing and searching. The Smith-Waterman algorithm has quadratic run-time complexity like the Levenshtein distance. A quadratic running time has poor efficiency when the length of the strings becomes moderately large. The Basic Local Assignment Search Tool (BLAST) [169] approximates the Smith-Waterman algorithm using a heuristic search. BLAST is used frequently to improve the efficiency of genome searches. We propose using off-the-shelf BLAST software to perform similarity searches of our malware signatures. To do this, we translate our control flow graph signatures to a protein string in the FASTA format to be used as input to the BLAST software. To construct a protein sequence, the decompiled string is translated character by character to a genome identifier. The BLAST algorithm does not employ distance metrics for the similarity search, but uses the notion of an expected value, which describes the statistical probability of the occurrence of a random signature.

The use of off-the-shelf genome similarity search software is a novel aspect used by our approach, and to the best of our knowledge has not been proposed in earlier research.

5.4.4 Indexing Using the NCD Metric Access Method

We propose using the normalized compression distance (NCD) [84] to perform a similarity search. The NCD utilises the notion of compressed objects being related to Kolmorogov complexity. The NCD takes note that when two objects are related, compressing the concatenated objects results in a blob of similar length to compressing only one of the

objects. The NCD provides a measure of dissimilarity or distance between objects without explicit knowledge or representation of the internal structure of the objects in question. It is able to provide a distance measure using many existing compression algorithms without modification. To recap Chapter 2, the NCD is defined as:

$$NCD(x,y) = \frac{C(xy) - \min\{C(x),(C,y)\}}{\max\{C(x),C(y)\}}$$

where C(x) is the length of the compressed object, and C(xy) is the length of the compressed concatenated objects.

For the NCD to perform effectively, the size of the objects must be less than the compressors window size. The NCD is a metric and so can employ the use of Metric Access Methods to index and search the signatures. To the best of our knowledge, Metric Access Methods have not been used in conjunction with the NCD and malware indexing by previous research.

5.5 Vector Based SIgnatures – Pre-filtering

To reduce the search space for potentially related malware, we use an initial similarity search to select candidate malware variants. We chose a vector based approached because during our evaluation we observed that this approach was more efficient and effective. Note that in our final system we use both vector based signatures for pre-filtering and the set of strings based signatures for refinement.

We construct and search for feature vectors that are associated with malware. We propose two methods to extract features for the feature vector using either k-subgraphs or q-grams of structured control flow. Q-grams are more efficient and evaluation shows that they generate more accurate results. The use of approximating a set of graphs by a vector is a novel contribution of this thesis.

5.5.1 The K-Subgraph Feature

Using subgraphs of size k to characterize control flow has been investigated in previous literature [115]. Subgraphs of size k are those subgraphs in the control flow graph which have k nodes. We use each possible subgraph of size k in the control flow graphs as features of the program. Our novel contribution is the use of these features in the construction of a feature vector which is subsequently used in a similarity search.

For each control flow graph, we construct a depth first spanning tree to eliminate cycles. We then perform a traversal of all possible paths in the tree where the traversal is terminated when k nodes have been visited [115].

Given a subgraph of size k, the graph is transformed into a unique and canonical representation using the Bliss open-source toolkit [170]. A canonical graph labeling is formed and the adjacency matrix of the resulting graph is stored as a string. This string represents a feature of the malware. Graph canonization is not known to take polynomial time in the general case. An example of possible k-subgraph features from a control flow graph when k is 7 is shown in Fig. 39.

5.5.2 The Control Flow Q-Gram Feature

Q-grams can be employed to represent control flow if the control flow graph is modeled as a string. We use structuring to generate the strings. A q-gram is any character sequence in the string of length q, constructed as a sliding window. For q-grams to be an effective feature, the strings must satisfy the property that similar control flowgraphs have similar strings. Each possible q-gram in the string represents a feature. Constructing the strings

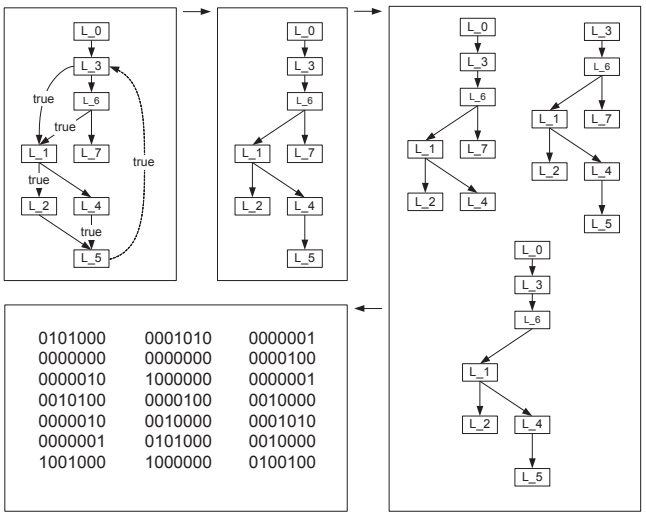

Fig. 39. The k-subgraph feature.

and the q-grams can be done in polynomial time and is more efficient than using k-subgraphs. The use of q-grams on the structured control flow graphs is a novel contribution of this thesis.

5.5.3 Feature Selection

The number of possible and distinct features in a program is large. To reduce the number of distinct features to a feasible number, the set of the 500 most frequent features are selected from a training set of malicious and benign programs. Feature selection works by counting the number of times each feature occurs in the training set and then ranking them in descending order. The top 500 were our selected features. Frequency of features forms our feature selection and is reasonable considering we are performing a nearest neighbour search. If we were performing malware detection using binary classification then another form of feature selection would be more suitable, for example, Mutual Information. The number of features, 500, was chosen to replicate previous work used in n-gram classification. We noted no significance to the accuracy of the system when this number was increased further. We did notice that decreasing this number using dimensionality reduction did decrease the accuracy as explained in section 5.6.4. Both program classes are used because it is our intuition that there is no significant classification difference in control flow between malicious and benign programs. This intuition forms the basis for our instance-based learning approach to classification. These features represent dimensions in a program's feature vector, and the frequency of a particular feature represents the dimension's magnitude. For the remaining features not in the 500 most frequent, they are ignored when constructing the feature vector of a program.

5.5.4 Dimensionality Reduction

To reduce the dimensionality of the feature vector obtained from the previous stage, Principal Component Analysis (PCA) [171] can be employed. Our pilot studies performed more effectively when PCA was not used. We do not consider dimensionality reduction any further. Fig. 8 illustrates the process of feature selection and dimensionality reduction.

5.5.5 Feature Vector Distance

To calculate the pairwise similarity between two feature vectors, a distance metric is employed. Many distance metrics are possible including the Euclidean distance. We use

the Manhattan distance because of its efficiency when compared to the more traditional Euclidean distance. This distance is also reportedly more robust for high dimensional data when compared to the Euclidian distance. The more familiar cosine similarity measure is not used in our work because it is not a metric distance function and therefore does not allow for efficient database indexing. The Manhattan distance (Section 2.7.3) is also known as the city block or L_1 distance. To recap Chapter 2, for n-dimensional vectors *p* and *q*, the Manhattan distance is:

$$d_1(p,q) = \|p-q\|_1 = \sum_{i=1}^{n} |p_i - q_i|$$

5.5.6 Indexing and Searching the Feature Vectors

We group the feature vectors into buckets. To group the feature vectors, the neighbours of each feature vector that are equal to or exceed the similarity of 0.6 are placed in the same bucket. This threshhold was chosen empirically through experimentation.

We pre-filter malware variants by performing a range nearest neighbour similarity search to our query feature vector. Given database D, query q, and threshold t, the set of nearest neighbours R is:

$$R = \{r \in D\} : 1 - \frac{d(r,q)}{|q|} \geq t$$

The results of the similarity search are candidate matches that can be used in the subsequent stage of comparing programs using the assignment problem. The nearest neighbours of the query enable us to determine if those neighbours are variants of the query. All samples available are typically used for the queries once the database is created. The nearest neighbours of those queries, as described in the introduction, can identify polymorphic variants, group samples by their family, or enable incident response to identify clusters of infection. More discussions of the algorithms and implementation of the similarity search are given in Section 8.

5.6 Set of Strings Based Signatures – Malware Classification

We propose a more accurate distance function to be applied to candidate malware variants after their identification in the pre-filtering stage. This improved distance is based on the distance between the control flow graphs' structured strings and is a variant of the minimum matching distance.

5.6.1 A Distance Function for Programs Based On the Linear Sum Assignment Problem

The linear sum assignment problem is to match distinct pairings of elements between two sets. Each match or assignment has an associated weight. The assignments are made such that, the sum of the weights are minimized. The linear sum assignment problem is also known as a minimum weight perfect matching.

The linear sum assignment problem is formally defined as:

Given two sets, A and T, of equal size, together with a weight function C: $A \times T \rightarrow$ R. Find a bijection f: $A \rightarrow T$ such that the cost function:

$$\sum_{a \in A} C(a, f(a))$$

is minimized.

For each program examined by our malware classification system, there exists an associated set of control flow graphs. Each set is represented as a set of structured strings. The assignment problem is used to match control flow graphs between sets. The intuition is that these matched control flow graphs are shared characteristics between malware variants. The weight of the assignment is the string metric or distance between those strings. We use the Smith-Waterman algorithm. We construct a matrix containing the weights of all possible pairings between two programs' sets of structured strings. If the number of elements in each set is not identical, then the elements that cannot be paired to existing elements are paired to the null element. The weight of this pairing is equivalent to the size of the element's string.

We define the distance between programs as the minimal cost function generated by a solution to the assignment problem using the matrix of weights. The use of the assignment problem is a novel contribution used in our system to show the distance between programs. This cost is a variation of the minimum matching distance [85] which is known to be metric.

Formally, let two programs P_1 and P_2 be defined as sets of control flow graphs and let S be a structuring function.

We first normalize the size of the sets making them equal. The additional elements, b_j, used in the normalization process are place holders and not used for any other purpose.

$$M_1 = S(P_1)$$

$$M_2 = S(P_2)$$

$$M_1' = \{a_i \in M_1\} \cup \{b_j\}: 1 \leq i \leq |M_1| < j \leq |M_2|$$

$$M_2' = \{a_i \in M_2\} \cup \{b_j\}: 1 \leq i \leq |M_2| < j \leq |M_1|$$

The function ed(a,b) is defined as the distance between strings. The distance, d, between the programs is found as follows:

$$C: M_1{}' \times M_2{}' \to \mathbb{R}$$

$$C(a,b) = \begin{cases} |a|, & if\ a \in M_1, b \notin M_2 \\ |b|, & if\ b \in M_2, a \notin M_1 \\ ed(a,b), & if\ a \in M_1, b \in M_2 \end{cases}$$

Find a bijection $f: M_1{}' \to M_2{}'$ such that the distance, d is minimized.

$$d = \sum_{a \in M_1{}'} C(a, f(a))$$

5.6.2 Solutions to the Assignment Problem

The assignment problem can be solved optimally using the Munkres or Hungarian algorithm [158] in time $O(N^3)$. Although an optimal solution is available, for programs that have a large number of control flow graphs, the time complexity required of $O(N^3)$ is impractical. In these cases when the number of nodes is greater than 300, we use a

heuristic solution based on a greedy assignment. The greedy assignment matches an element from one set by selecting the element from the other set with the lowest associated weight. The time complexity is *O*(N(N+1)/2). The greedy solution performs more efficiently, but the program distance it identifies is often significantly higher than the optimal solution.

5.6.3 Similarity Search of Malware

The similarity between two objects is given by:

$$s(p,q) = 1 - \frac{d(p,q)}{|q|} \quad \textit{iff} \quad d(p,q) < |q|$$

We scale the distance relative to our query so we can perform a range search relative to only the query using an efficient metric access method. Because we scale to the query and not max(|p|,|q|) we have cases where d(p,q) > |q|. In this case our similarity function would give us a negative result. To simplify interpretation of this result, we say that it is not at all similar and discard it.

A threshold for similarity, t, is chosen as 0.6. The threshold was chosen manually after an empirical evaluation. We then use this to identify any nearest neighbour p to the query q in the set of malware, E, returned by our pre-filtering process.

$$\exists p : p \in E, 1 - \frac{d(p,q)}{|q|} \geq t, d(p,q) < |q|$$

5.7 Nearest Neighbour Similarity Searches

5.7.1 Metric Distance Functions

The distance between two objects shows their dissimilarity. If the distance function has the properties of a distance metric then indexing and searching a database can be performed more efficiently. The formal definition of a metric distance function is given in Section 2.7.1. Given metric distance functions can enable efficient database access, it is beneficial to compare objects or birthmarks (software fingerprints) using distance functions that are metric. Examples of metric access methods are in [81-83].

5.7.2 Similarity Search Using Metric Access Methods

To search for malware that are similar to our query in both the pre-filtering and classification stages, a metric access method is employed. Metric trees encapsulate data structures including BK Trees [87], VP Trees [81] or dynamic indexing structures such as M-Trees [83] and Slim-Trees [88]. Our implementation uses the GBDI metric access method library [172].

In our prototype, we use a Vantage Point Tree [81] for indexing the feature vectors used in the pre-filtering stage. Then, the final classification process uses DBM-Tree [82] to perform a similarity search. Note that our classification system uses two levels of indexing and has different metric access methods for each.

In our prototype we can configure the similarity search to return either any similar objects, or all similar objects. We use the any range search for classification, and the all range search for pre-filtering. By performing a similarity search to find any similar object, the performance is significantly improved when there are many near duplicate malware stored in the database. The any range search was implemented by us into the GDBI Arboretum library [172].

5.8 Implementation and Evaluation

5.8.1 Implementation

Our implementation is built as a set of modules in the Malwise malware and static analysis framework. Malwise consists of approximately 100,000 LOC of C++ and its features include unpacking using application level emulation and static analysis. The modules we developed to perform malware classification consist of approximately 3,000 LOC of C++. Emulation is used to perform unpacking. However, the classification process uses only static analysis and that is the focus of our current work.

5.8.2 Effectiveness of String Signatures

The first evaluation we performed was to examine the similarity matrices for our string based signature classification algorithms on a known family of related malware. We also compared these methods to the q-gram approach. The system ideally identifies high similarity between malware that belongs to the same family. The malware chosen was the

Roron family of malware to replicate previous research [60, 61, 110]. The family of malware variants was identified by an Antivirus vendor and may not necessarily have been entirely trustworthy. We obtained the malware from the Offensive Computing malware database [173]. Identified malware variants have similarities exceeding or equal to 0.6. Identified variants additionally have their table cells highlighted. The more cells highlighted the more effective each approach is. We also evaluate using only the byte-level content for similarity comparisons. The Roron malware family is not stringly polymorphic, so byte-level content is still somewhat effective. The most important observation is that comparing string based signature approaches in Table 9 to the q-gram approach in Table 10 shows that the q-gram vector based signature detects more malware variants. It is also noted that q-grams are theoretically more efficient. It is for these reasons we decided to focus on vector based signatures.

5.8.3 Evaluation Setup

To perform more evaluations of the classification system, 17,430 real malware with unique MD5 hashes were collected between 02-01-2009 and 8-12-2009 from honeypots in the mwcollect Alliance [174] network. From these malware, 15,398 were found to be valid object files for Windows Vista – the remaining binaries were invalid, specific to Windows XP, and not able to be processed by our prototype's unpacking system. In addition to the malware, we employed the use of 1601 benign binaries, which were obtained from the Windows system directory and the Cygwin [175] executable directories. The system we used to evaluate the prototype classification system was an Intel Q6600 Quad Core 2.4GHz PC with 4G of memory running 32-Bit Windows Vista Home Premium, Service Pack 1.

The prototype system requires training to select the 500 most common q-grams and k-subgraphs. 1769 malware and 1601 benign binaries were used in the training set to generate features.

5.8.4 Evaluation of False Positives in Pre-filtering

To evaluate the accuracy of the q-gram and k-subgraph classification algorithms we first constructed a database of 10,000 malware signatures. Then, we found the similarities between each of 10,000 malware and 280 benign binaries from the windows system

TABLE 9. SIMILARITY MATRICES FOR RORON MALWARE.

	ao	b	d	e	g	k	m	q	a
ao	1.00	0.60	0.35	0.38	0.45	0.74	0.60	0.60	0.73
b	0.60	1.00	0.46	0.50	0.37	0.73	0.95	0.96	0.73
d	0.35	0.46	1.00	0.64	0.59	0.36	0.46	0.46	0.35
e	0.38	0.50	0.64	1.00	0.61	0.42	0.49	0.50	0.40
g	0.45	0.37	0.59	0.61	1.00	0.47	0.37	0.37	0.46
k	0.74	0.73	0.36	0.42	0.47	1.00	0.73	0.72	0.86
m	0.60	0.95	0.46	0.49	0.37	0.73	1.00	0.96	0.72
q	0.60	0.96	0.46	0.50	0.37	0.72	0.96	1.00	0.72
a	0.73	0.73	0.35	0.40	0.46	0.86	0.72	0.72	1.00

Levenshtein String Metric on Byte-level Content

	ao	b	d	e	g	k	m	q	a
ao	1.00	0.70	0.42	0.42	0.44	0.72	0.70	0.70	0.70
b	0.70	1.00	0.47	0.47	0.48	0.94	1.00	1.00	0.93
d	0.42	0.47	1.00	0.71	0.80	0.48	0.47	0.47	0.48
e	0.42	0.47	0.71	1.00	0.72	0.47	0.47	0.47	0.47
g	0.44	0.48	0.80	0.72	1.00	0.49	0.48	0.48	0.50
k	0.72	0.94	0.48	0.47	0.49	1.00	0.94	0.94	0.96
m	0.70	1.00	0.47	0.47	0.48	0.94	1.00	1.00	0.93
q	0.70	1.00	0.47	0.47	0.48	0.94	1.00	1.00	0.93
a	0.70	0.93	0.48	0.47	0.50	0.96	0.93	0.93	1.00

Levenshtein String Metric

	ao	b	d	e	g	k	m	q	a
ao	0.94	0.80	0.50	0.52	0.52	0.82	0.80	0.80	0.82
b	0.80	0.93	0.51	0.54	0.53	0.88	0.93	0.93	0.89
d	0.50	0.51	0.93	0.77	0.83	0.52	0.51	0.51	0.52
e	0.52	0.54	0.77	0.94	0.85	0.54	0.54	0.54	0.54
g	0.52	0.53	0.83	0.85	0.93	0.53	0.53	0.53	0.53
k	0.82	0.88	0.52	0.54	0.53	0.94	0.88	0.88	0.92
m	0.80	0.93	0.51	0.54	0.53	0.88	0.93	0.93	0.89
q	0.80	0.93	0.51	0.54	0.53	0.88	0.93	0.93	0.89
a	0.82	0.89	0.52	0.54	0.53	0.92	0.89	0.89	0.93

Normalized Compression Distance (NCD) Metric

directory. This evaluation is to identify how effective the pre-filtering stage is at filtering non matching samples. We expect that similarity found should be generally quite low, and any similarity found above or equal to 0.6 identifies a false positive. The size of the q-gram was 4. The size of the k-subgraph was 10 as recommended in the existing literature. Better selections of the size k were not investigated. The threshold of 0.6 was chosen empirically through experimental testing.

The evaluation shown in Table 11 demonstrates that false positives, or collisions, occur using this pre-filtering algorithm with either feature. The q-gram feature is shown to generate considerably less collisions and false positives compared to using k-subgraphs of size 10. For this reason, we excluded using k-subgraphs as part of the classification process in further evaluations.

TABLE 10. SIMILARITY MATRICES FOR RORON MALWARE.

	ao	b	d	e	g	k	m	q	a
ao		0.44	0.28	0.27	0.28	0.55	0.44	0.44	0.47
b	0.44		0.27	0.27	0.27	0.51	1.00	1.00	0.58
d	0.28	0.27		0.48	0.56	0.27	0.27	0.27	0.27
e	0.27	0.27	0.48		0.59	0.27	0.27	0.27	0.27
g	0.28	0.27	0.56	0.59		0.27	0.27	0.27	0.27
k	0.55	0.51	0.27	0.27	0.27		0.51	0.51	0.75
m	0.44	1.00	0.27	0.27	0.27	0.51		1.00	0.58
q	0.44	1.00	0.27	0.27	0.27	0.51	1.00		0.58
a	0.47	0.58	0.27	0.27	0.27	0.75	0.58	0.58	

Exact Matching

	ao	b	d	e	g	k	m	q	a
ao		0.70	0.28	0.28	0.27	0.75	0.70	0.70	0.75
b	0.74		0.31	0.34	0.33	0.82	1.00	1.00	0.87
d	0.28	0.29		0.50	0.74	0.29	0.29	0.29	0.29
e	0.31	0.34	0.50		0.64	0.32	0.34	0.34	0.33
g	0.27	0.33	0.74	0.64		0.29	0.33	0.33	0.30
k	0.75	0.82	0.29	0.30	0.29		0.82	0.82	0.96
m	0.74	1.00	0.31	0.34	0.33	0.82		1.00	0.87
q	0.74	1.00	0.31	0.34	0.33	0.82	1.00		0.87
a	0.75	0.87	0.30	0.31	0.30	0.96	0.87	0.87	

Heuristic Approximate Matching

	ao	b	d	e	g	k	m	q	a
ao		0.86	0.53	0.64	0.59	0.86	0.86	0.86	0.86
b	0.88		0.66	0.76	0.71	0.97	1.00	1.00	0.97
d	0.65	0.72		0.88	0.93	0.73	0.72	0.72	0.73
e	0.72	0.80	0.87		0.93	0.80	0.80	0.80	0.80
g	0.69	0.77	0.93	0.93		0.77	0.77	0.77	0.77
k	0.88	0.97	0.67	0.77	0.72		0.97	0.97	0.99
m	0.88	1.00	0.66	0.76	0.71	0.97		1.00	0.97
q	0.88	1.00	0.66	0.76	0.71	0.97	1.00		0.97
a	0.87	0.97	0.67	0.77	0.72	0.99	0.97	0.97	

Q-Grams

	ao	b	d	e	g	k	m	q	a
ao		0.86	0.49	0.54	0.50	0.87	0.86	0.86	0.86
b	0.87		0.57	0.63	0.62	0.96	1.00	1.00	0.96
d	0.61	0.64		0.85	0.91	0.64	0.64	0.64	0.64
e	0.64	0.69	0.85		0.90	0.68	0.69	0.69	0.68
g	0.62	0.68	0.91	0.91		0.68	0.68	0.68	0.68
k	0.88	0.96	0.58	0.62	0.61		0.96	0.96	0.99
m	0.87	1.00	0.57	0.63	0.62	0.96		1.00	0.96
q	0.87	1.00	0.57	0.63	0.62	0.96	1.00		0.96
a	0.87	0.96	0.58	0.62	0.61	0.99	0.96	0.96	

Optimal Distance Using Assignment Problem

5.8.5 True Positives of the System Compared to Previous Research

The next evaluation we performed was to examine the similarity matrices for our complete classification algorithms on a known family of related malware. This evaluation incorporates all elements of our system and is the main evaluation we performed on the true positive detection rate of the system. The system ideally identifies high similarity between malware that belongs to the same family. We compared the q-gram classification

TABLE 11. FALSE POSITIVES USING K-SUBGRAPHS AND Q-GRAMS.

Similarity	K-Subgraphs	QGrams
0.0	1302161	2334251
0.1	463170	413667
0.2	356345	40055
0.3	285202	7899
0.4	200326	3790
0.5	129790	327
0.6	46320	11
0.7	10784	0
0.8	5883	0
0.9	19	0
1.0	0	0

algorithm and the assignment problem classification algorithm. Additionally, we made comparison to algorithms proposed in previous research. We compared our system to a real-time flowgraph based classification system that uses exact or isomorphic testing of control flow graphs in [61]. We expect our approximate matching algorithm to detect more variants than the exact matching system. The second comparison was to a previously proposed system that uses an approximate control flow graph matching algorithm in [60]. The previously proposed system uses an alternative heuristic algorithm based on greedy matching and string metrics of the structured control flow graphs. The system we compared against does not employ the assignment problem or a program distance metric.

TABLE 12. MALWARE DETECTION

Classification Algorithm	Klez	Netsky	Roron	Frethem
Maximum	*36*	*49*	*81*	*289*
Exact	20	29	17	139
Heuristic Approximate	20	27	43	144
Q-Grams	20	31	79	226
Optimal Distance	22	46	73	220
Q-Grams + Optimal Distance	20	43	73	217

The results are shown in Table 12. The results show that our prototype detects more malware variants in this family of malware than existing systems.

The Netsky, Roron, and Klez, and Frethem malware were chosen to continue the evaluation of variant detection. For each malware family, the maximum number of possible variants is listed, along with the detection results of our algorithm and existing algorithms [60, 61]. Table 4 lists the results. Ideally, the number of variants detected would be the maximum, meaning all variants were related to each other. Our system detects many but not all variant relationships. It may be that some of the variants are quite distinct. The classification algorithms we proposed in this research are shown to be highly effective and detect more malware than previous systems. Looking at the Frethem malware family, our algorithm detects 217 variant relationships, while the next best system detects 144 variants.

5.8.6 Evaluation of the System's False Positives

We next evaluate the number of false positives generated by the system. The number of false positives gives indication of how the distance functions perform using non similar programs. In our first test we simply aggregated the families of malware from our true

TABLE 13. FALSE POSITIVES.

Classification Algorithm	False Positives	FP Percentage
Q-Grams	10	0.62
Q-Grams + Optimal Distance	7	0.43

positive testing. Our system did not report any samples as belonging to incorrect families. We then implemented a more thorough test of our system. We performed an evaluation using a much larger malware database size of 10,000. We classified the set of 1601 benign programs and expected that any identified malware would be a false positive. The evaluation demonstrates false positives when using the q-gram as is also demonstrated in Table 13. For a database size of 10,000 the false positive rate is shown to be less than 1%. We also show that using the assignment problem solution in conjunction with the q-gram classification results in fewer false positives.

We suspect the reason for the remaining false positives is because we do not eliminate statically linked functions from the analysis. Programs that share the same statically linked objects have a high similarity, even when the programs are generally unrelated.

TABLE 14. ALGORITHMIC COMPLEXITY COMPARISONS

Algorithm	Complexity
SMIT	$O(N^{3)}$
Exact Matching	O(NlogN)
Graph Edit Distance	NP
Graph Isomorphism	NP
String	$O((k*N)^{2)}$
Vector	O(1)
Set of Strings-Optimal	$O(N^3)$
Set of Strings-Approximate	O(N(N+1)/2)

5.8.7 Algorithmic Complexity Analysis

The algorithmic complexity of comparing malware signatures is shown in Table 14. We examine our string based signature based on using the optimal edit distance, the vector-based signature using the Manhattan distance, and the set of strings-based signature using the optimal and greedy approach to solving the assignment problem. We also compare our approach with previous work in SMIT, exact control flow graph matching in [61], and traditional graph algorithms. Our vector-based signature is the most efficient and a distance between signatures can be performed in O(1) relative to the size of the programs in terms of number of procedures. This is why our system performs so efficiently. The string-based signature performs quite slowly because each procedure incurs a cost, k, relative to the size of the procedures' decompiled control flow graphs. Our set of strings-based distance can be performed in $O(N^3)$ which is comparable to the previous research in SMIT [112] which uses an approximation to the graph edit distance on the programs' call graphs. For large graphs we can use the approximate algorithm in our approach which

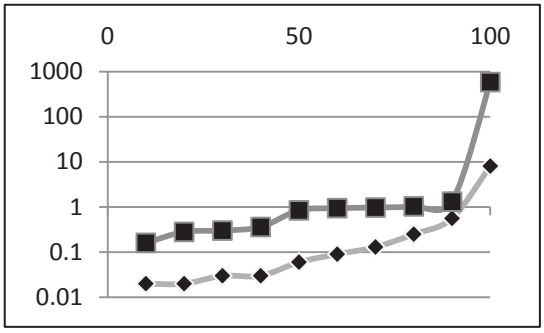

Fig. 40. Malware and benign sample processing times

performs in *O*(N(N+1)/2) and is more efficient than SMIT. The exact matching algorithm was proposed in our previous research but does not perform approximate matching of control flow graphs which our current work does. The classical single graph based distance and equality algorithms are either in NP (graph edit distance) or believed to be in NP (graph isomorphism) making control flow intractable when used for a signature.

5.8.8 Efficiency

To evaluate the efficiency of our system, we record the execution time to classify each of 1601 benign programs and 15,398 malware. The malware database is pre-populated with 10,000 malware signatures. We evaluate the complete processing time of the system including unpacking, disassembly, control flow reconstruction and analysis. The processing times for the malware and benign programs are shown in Fig. 40. The malware processing times are higher in general. The median time for processing malicious samples is 0.84 seconds. 90% of the samples could be processed in under 1.31 seconds. The maximum time taken is 585 seconds and may have resulted from excessive memory consumption causing thrashing. Some candidate buckets were large due to a high number of related malware variants, resulting in higher than average pairwise comparisons using the less efficient distance function. Unpacking binaries using emulation may also cause significant overhead in some cases. If applied in a desktop environment, the analysis may need to flag such binaries that impact performance and whitelist known benign programs that would otherwise cause false positives. In practice, we do not see these edge cases as reducing the effectiveness if they are handled in these ways. The median time for processing the benign binaries is 0.06s. 90% of samples could be processed in under 0.56

seconds. Classifying only the Windows system programs has a median time of 0.15s. Processing benign programs is the expected case and performs more quickly than classifying malware due to the extra overhead of unpacking. The slowest time is 8.06 seconds which is still reasonable for industrial deployment. Our system improves the performance in classifying benign programs compared to the less effective exact matching algorithm proposed in [61] which has a median Windows system directory processing time of 0.25s. This is not due to classification performance, which is almost identical, but due to improvements to efficiency in the static analysis component. The general results indicate that the speed of classification may warrant the system suitable for real-time use for desktop Antivirus or on an Email gateway system.

5.9 Limitations and Discussion

5.9.1 Code Packing

A malware obfuscation technique commonly employed to resist static analysis is packing. Malware packing that encrypts, compresses, or obfuscates the code contents and then later regenerates the original program needs to be removed. The majority of packed samples can be automatically unpacked, but there exist binaries which evade this analysis. Instruction virtualization [21, 176] is resistant to an entirely automated static analysis. Instruction virtualization implements an emulator which interprets bytecode representing the hidden code. Therefore, the hidden code in its original unpacked form is never revealed. If unpacking cannot be achieved by a malware classification system, then the packing tool may be classified instead of the packed contents. It is probably advantageous for Antivirus to blacklist programs that cannot be unpacked. Manually written static unpackers can be developed on a case by case basis and this is what is traditionally employed by commercial Antivirus. A better approach is to detect packed programs and flag them as suspicious. Benign programs that are packed can be whitelisted. The scope of our system is limited to malware that can be unpacked using the approach of application level emulation. Application level emulation is fast but because of its limited use of a faithful emulator, malware can detect its presence and therefore change its behaviour. Unpackers such as Renovo [29] employ whole system emulation and are more resistant to detection. The current problem is that such systems have poor performance in terms of real-time

constraints. Another approach is to unpack on the fly during program run-time by monitoring memory access, as is done by OmniUnpack [177]. This system claims real-time performance suitable for Antivirus. Such a system could be combined with our work to make a real-time malware classification capable of unpacking most or all non instruction virtualization based malware.

5.9.2 Obfuscation

For the most part, code packing is the obfsucating process employed by malware authors. Therefore, once a sample has been unpacked, analysts have access to the original unobfuscated image. This is becoming prevalent as malware becomes more like traditional software development and malware authors employ high level languages to implement their works.

Control flow can be obfsucated but this is typically not present in most malware today. Code insertion, deletion, substitution, and reordering within a basic block does not affect the structured control flow that our system uses. This makes control flow a more invariant program representation than traditional byte-level signatures. If control flow is modified, then our system can perform an approximate match. The changes to the decompiled strings should show the changes locally. The global view of the strings should still retain similarity. Through the normal process of software development and evolution, decompiled strings of control flow graphs can identify those changes while still identifying them as variants.

Obfuscations such as opaque predicates which add conditional branches which always evaluate to the same path but are hard to determine statically present a bigger problem. Unless opaque predicates account for the majority of the control flow, our system should still detect the malware as a variant. Other obfsucations including negating conditions and swapping the branches resulting in different decompiled strings. A solution to this could involve using an unordered Abstract Syntax Tree (AST) instead of a string. If malware in the future obfuscates control flow like this, we may consider using k-subtrees of the AST instead of q-grams of the decompiled strings.

Concluding Remarks

Malware can effectively be characterized by its control flow. We proposed a malware classification system using approximate matching of control flow graphs. We first tried using string signatures to describe malware. We then used techniques to extract q-grams and k-subgraphs of sets of control flow graphs and created feature vectors. From these feature vectors we were able to construct an efficient distance metric and similarity search. We also used the assignment problem and the string distance to construct a distance metric between programs. We implemented these algorithms in a prototype and performed an evaluation of the system. Our evaluation showed that our work more effectively detected malware than previous comparable systems. The number of false positives was low, and the efficiency of the prototype demonstrated that the system could be used on a desktop system or Email gateway.

Chapter 6: Software Similarity and Classification in the Cloud

Simseer and Clonewise are online web services that perform the software similarity and classification applications proposed in the previous chapters. There are two services available - Simseer and Clonewise. Simseer exposes the software implemented in Chapter 3 and can identify similarity between submitted executables based on the similarity in the control flow of each binary. Simseer extends Malwise II from Chapter 5 by providing a search service, a clustering service, and an evolutionary tree visualisation service. Clonewise exposes an online version of the system in Chapter 4. Clonewise takes a tar ball of a software system and identifies and reports any embedded package-level clones in that software. Both Clonewise and Simseer are built on a scalable cloud infrastructure hosted by Amazon's elastic compute cloud (EC2).

6.1 Introduction

Cloud services offer the ability for people to use the applications developed throughout this thesis in an easy to use manner. The processing is offloaded to behind the scene servers that can be scaled up and out easily. These cloud services could potentially integrate with client based software, thus affording the best of cloud and traditional software services.

6.1.1 Services

We have implemented multiple services based on Malwise and Clonewise. These services are described below.

6.1.1.1 Simseer

This service takes as input a ZIP archive of 32-bit x86 executables. Using Malwise, the similarities between each sample is identified. These similarities are then passed to phylogenetics software to graphically visualize an evolutionary tree of relationships between the samples.

6.1.1.2 Simseer Search

This service takes as input a threshold of similarity and a 32-bit x86 executable. The executable is used as query to search a database of samples. The result is all samples in

the database that are at least the threshold of similarity similar to the query. Each sample submitted to the service is stored in the database for future comparison and each sample is scanned with traditional Antivirus. This system allows users to take an unknown sample and identify if it is related to any previously identified malicious samples. Additionally, code packing detection is performed as a heuristic to identify obfuscated samples that cannot be deobfuscated by Malwise.

6.1.1.3 Simseer Cluster

This service takes as input a number specifying the number of clusters and a a ZIP archive of 32-bit x86 executables. Using the feature vector approach of Malwise, the vectors representing each sample are clustered using hierarchical clustering and the cosine distance as a notion of dissimilarity. The results show the samples clustered into groups of similar samples. Each sample is also scanned with traditional AntiVirus which enables a user to identify the family name of potential malware. Like Simseer Search, if an unknown sample is in the same family of known malicious samples, the sample is likely to be malicious also.

6.1.1.4 Clonewise

This service takes as input an email address and a tar ball of source code. The source code can be of any language. Clonewise reports via E-Mail any library of 420 identified possible clones that are present in the tar ball. The results show which files are shared between the tar ball and identified clones and the importance of each file in respect to clone determination.

6.1.2 Structure of the Chapter

The structure of this chapter is as follows: Section 6.2 discusses the design and implementation of our system as a cloud service. Section 6.3 gives details on how to access our service. Section 6.4 looks at future work. Finally, Section 6.5 gives our concluding remarks.

6.3 System Design and Implementation

The system uses multiple Virtual Private Servers (VPS) in the cloud and could potentially be scaled to operate on large server farms. Both Simseer and Clonewise run on the same infrastructure. The servers can be divided into serving the frontend of the system, those

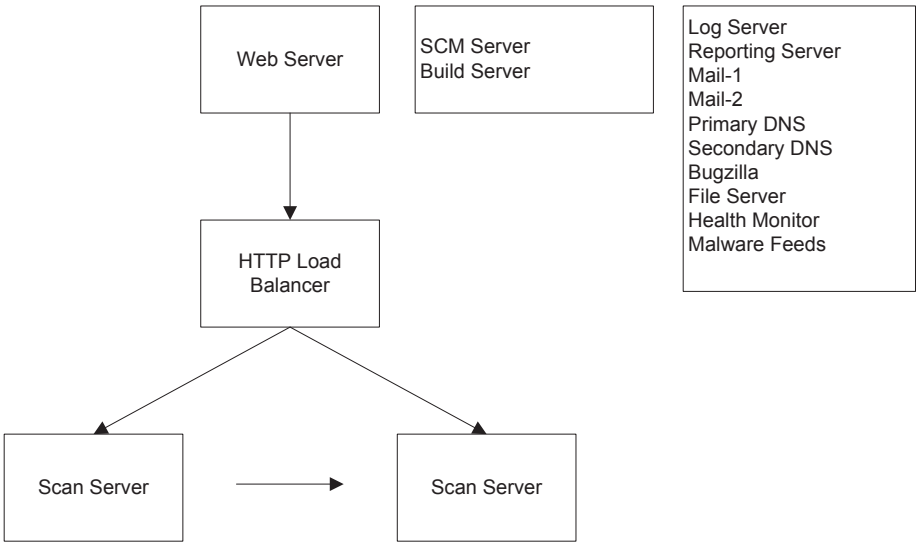

Fig. 41. The cloud services infrastructure.

serving the backend of the system, and those supporting the network infrastructure. All servers run on the Amazon EC2 elastic compute cluster cloud infrastructure and in our work use the Linux Ubuntu 12.10 operating system distribution. The frontend and supporting infrastructure use a 64-bit platform and the backend uses a 32-bit platform.

6.3.1 The Web Frontend

The frontend of the system provides infrastructure to support serving the web content and accepting submissions to the services. Chiefly, this part of the system uses the Apache web server. This node is a micro instance and has 615M of memory, 1 core, and is specified as having up to 2 EC2 compute units for small bursts.

The web frontend is the user interface to the Simseer and Clonewise cloud service and the landing pages and the final results are shown in Fig. 42 - Fig. 48. A user can submit executables and file archives to Simseer or tar balls to Clonewise. Our frontend implementation is coded in the server side PHP programming language and uses the Twitter Bootstrap CSS (Cascading Style Sheets) to implement the presentation.

Both Simseer and Clonewise are currently implemented on the same web server to reduce the operational costs of Amazon EC2. This is achieved by separating each service on its own virtual host. The virtual host for Simseer is www.simseer.com and the virtual host for

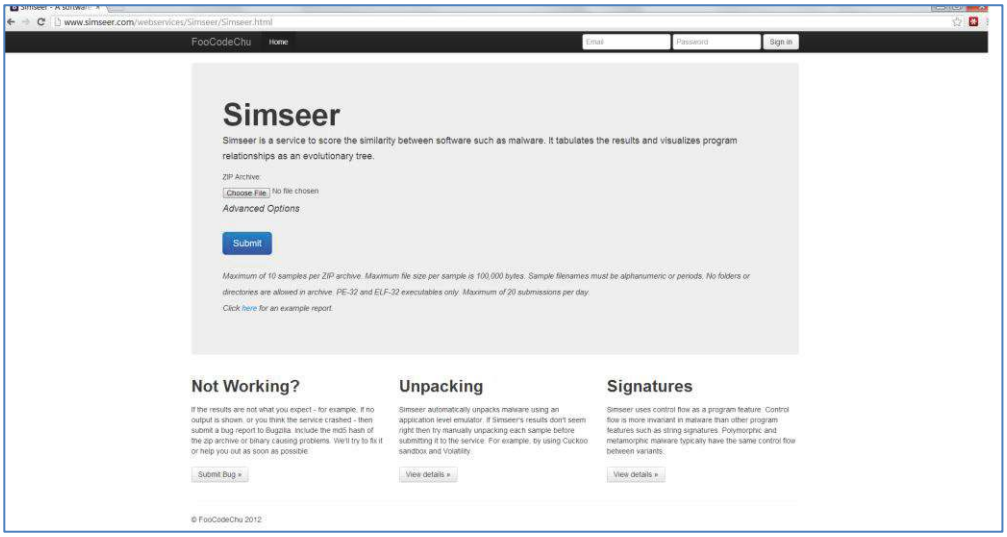

Fig. 42. Simseer landing page.

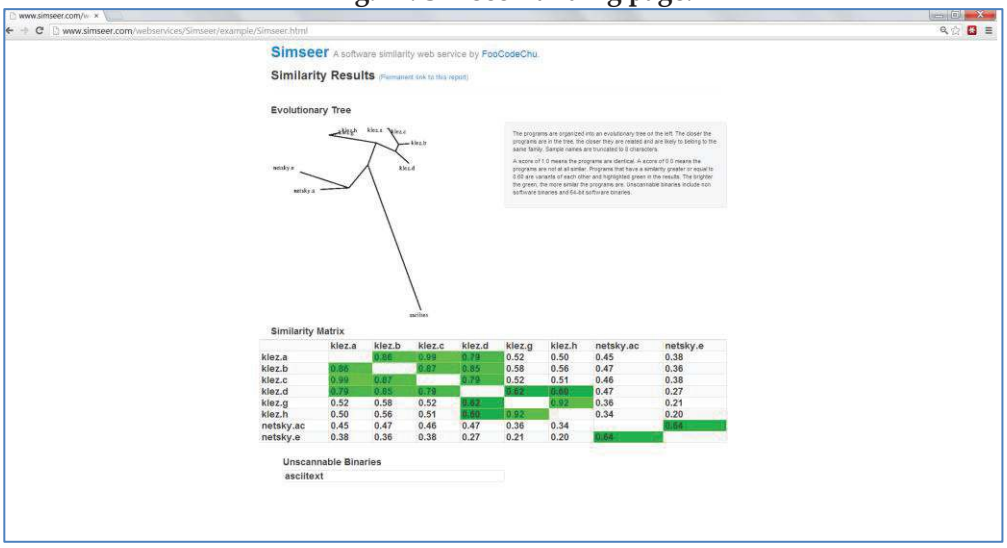

Fig. 43. Simseer results.

Clonewise is www.codeclones.com. Potentially, if the system requires to be scaled, each service could be placed on its own isolated host. Likewise, these nodes could be placed behind a load balancer for high availability and scalability.

The PHP code in the frontend examines the files submitted to it and performs sanity checking. For example, Simseer for each archive submitted, it will check that the ZIP archive is valid, does not contain an excessive number of samples, does not contain symbolic links as archive members, and does not contain archive member names using

special characters. Clonewise performs an equivalent amount of sanity checking on tar balls.

Logging is performed for each submission and a copy of the submission is made to a directory that is stored on the network's file server. The web submission is then relayed to a HTTP load balancer via a Python script which will distribute the job to a scan server. The scan server will then report the results as an XML document and the frontend will make a copy of this report on the file server and present it in a suitable form to the user. Additionally, all previously generated reports can be retrieved via another wbe-based request given their MD5 hashes.

6.3.2 Cluster-based Load Balancing

As described in the previous section, the frontend web server resubmits each job to a separate node that distributes the work. The load balancer is implemented with an Apache web server. The node is a micro instance and has 615M of memory, 1 core, and up to 2 EC2 compute units for small bursts. The load balancing distributes jobs to a cluster so that each node in the cluster receives approximately the same number of jobs as every other node. The jobs to the cluster nodes are sent as web requests.

6.3.3 Backend Clustering and Work Scheduling

The scan nodes in the cluster backend perform the backend work for each job. There are currently 3 nodes in the backend cluster that perform job requests. Each node is a small instance and has 1.7G of memory, 1 core, and 1 EC2 compute units.

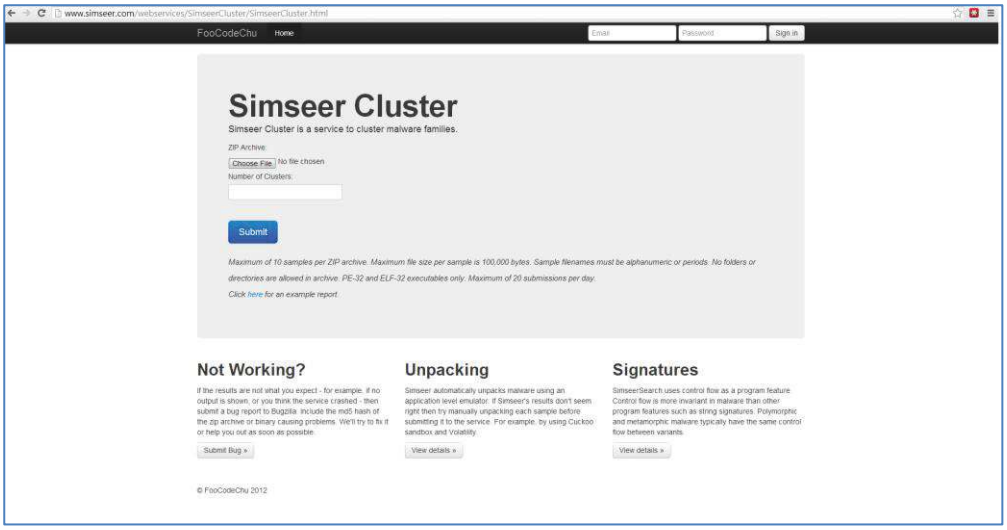

Fig. 44. Simseer Cluster landing page.

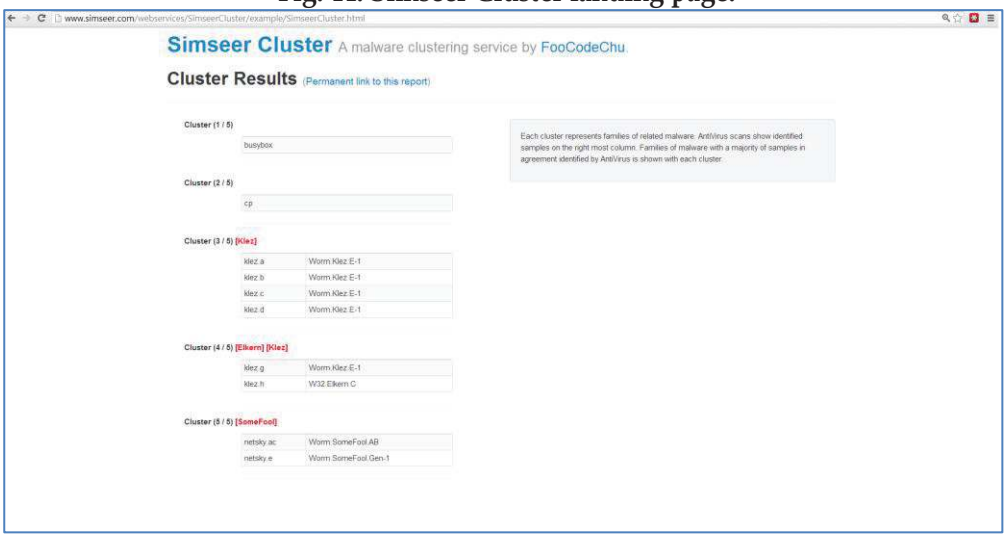

Fig. 45. Simseer Cluster results.

The backend cluster nodes run the Apache web server and accept requests via a PHP web interface. When a job is received, it is passed to a script on the node. This script launches a network client which submits the job to a network daemon listening on a local TCP port.

Each scan node listens locally on a TCP port to accept scan jobs. The network daemon and client are implemented in C++. This daemon queues and schedules jobs. Currently, 1 job can run on 1 node at any one time so that the server does not consume excessive

resources. Running multiple jobs in parallel places more pressure on memory usage per instance which we wanted to avoid.

Once a job has been scheduled by the network daemon a script is launched to process the file and launch the Malwise or Clonewise systems.

6.3.4 Network Infrastructure

Simseer and Clonewise require supporting infrastructure outside of the frontend, load balancer and backend cluster. These requirements include a file server, syslog server (using rsyslog) to collect logs from all servers on the network, a mail server (using postfix), a domain name server (using bind9), an Apache web based reporting server for system process usage (using munin), web and load balancer usage, and a server to monitor the health of the network by submitting known jobs to the services and checking that the results are correct.

6.3.5 DevOps Infrastructure

To develop Simseer and Clonewise requires such things as source code management and build management. All source code is maintained in the Git source code management software. A build server is implemented that clones the Git repository, builds the software and packages it into Debian DEB archives. These can then be deployed with the use of deployment scripts to each necessary server.

6.3.5 Service Specific Processing

Although all services run on the same infrastructure, each service has its own specific capabilities and implementation requirements.

6.3.5.1 Simseer Evolutionary Tree Visualization

Simseer visualizes program relationships using an evolutionary tree. A phylogenetic or evolutionary tree is a visual representation of the evolutionary relationships between species based on similarity between features or characteristics. Species closer to the tree in relation to the number of branches or branch lengths are more closely related. Simseer uses evolutionary trees to visualize the relationships between programs and their variants. This visualization is useful because program variants are typically derivatives and modified versions of their upstream source.

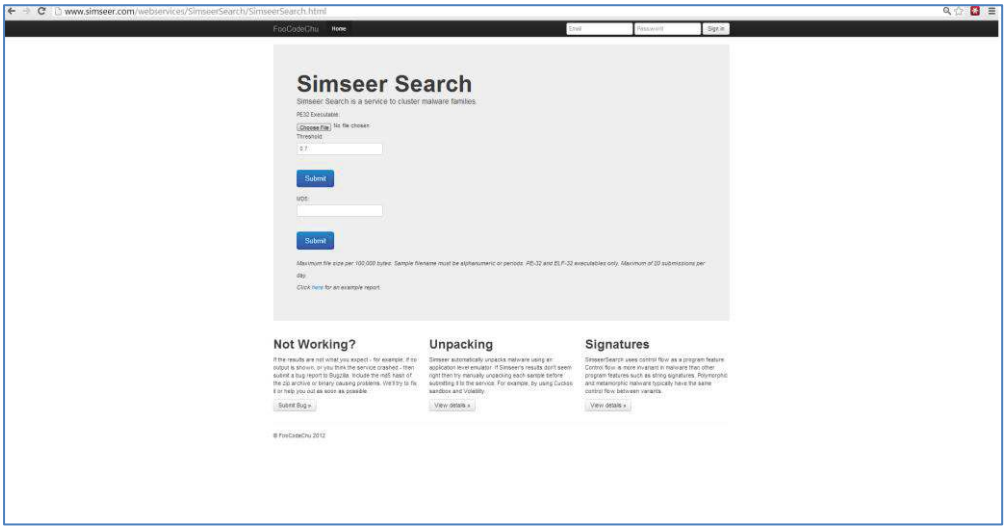

Fig. 46. Simseer Search landing page.

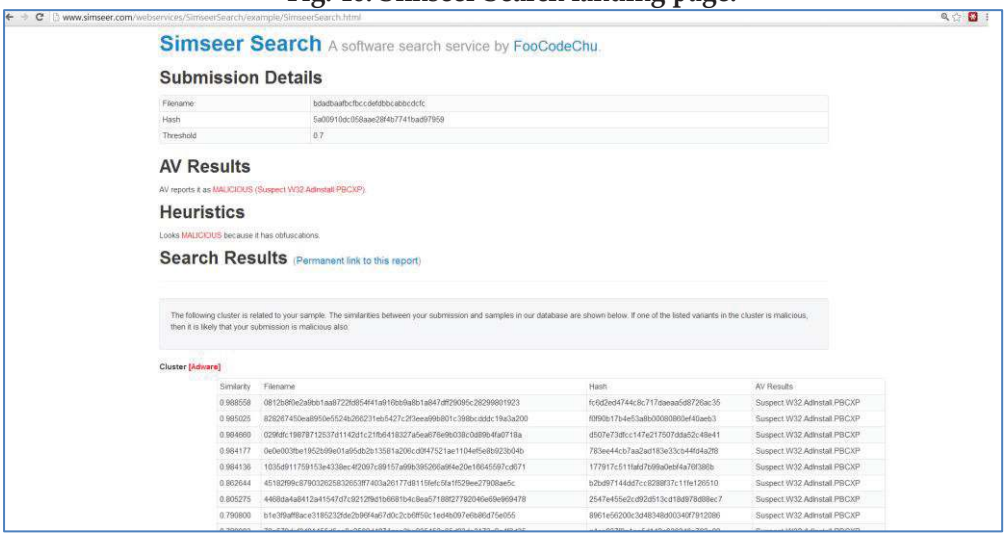

Fig. 47. Simseer Search results.

The web frontend host is responsible for processing the XML results returned by Malwise. The XML returned by Malwise scores the similarity between each sample. The frontend transforms the XML into a distance matrix. Distance is calculated as 1 – similarity. This distance matrix is then analysed to create an evolutionary tree using the PHYLIP software package [178]. The PHYLIP package uses the neighbour joining method [179] to construct an evolutionary tree. The evolutionary tree is described by the Newick tree format which gives such information as branch lengths in the tree. The Newick tree file is processed to

render a figure suitable for display. The figure is then transformed to a PNG image and stored on the web host. An example of the tree visualization is shown in Fig. 4.

To display the results, the Malwise XML similarity results are displayed as an HTML table. The background colour of the table cells are proportional to how similar the samples are. The lighter the colour, the more similar the programs are. If the programs are not variants of each other, the table cell is left unshaded. The evolutionary tree image of the programs is shown on the same page.

6.3.5.2 Simseer Search

SImseer Search uses a backend database that must be accessible by all nodes in the backend cluster. We met this requirement by using a network file server implementing NFS. To synchronize writing and reading from the database we use file locking. Simseer Search uses traditional AntiVirus to scan samples. We use Clam AntiVirus (ClamAV) which runs on Linux.

6.3.5.3 Simseer Cluster

Simseer Cluster uses the WEKA machine learning toolkit to perform hierarchical clustering. WEKA does not by default allow the use of the cosine distance in its hierarchical clustering algorithm. We implemented a custom distance function to achieve this capability.

6.3.5.4 Clonewise

Clonewise uses a database for the 420 common clones it checks for that may be potentially embedded. This database was generated offline on an Amazon EC2 cluster and is used for the Clonewise web service.

6.3.6 Updating the Malware Database

The Simseer Search service is entirely dependent on the quality of the database that it uses. Some of the samples in the database come from user submissions, but for the majority of samples in the database, they are uploaded automatically from an internet based malware feed, VirusShare. Each night the daily malware feed that VirusShare provides is downloaded as a ZIP archive. The ZIP archive ranges in size from 600M to 16G. The archive consists of a variety of file types. The Windows 32-bit executables are extracted and passed to Simseer's web interface via a python script similar to how a user submission is made. This process allows the full use of the backend cluster to process

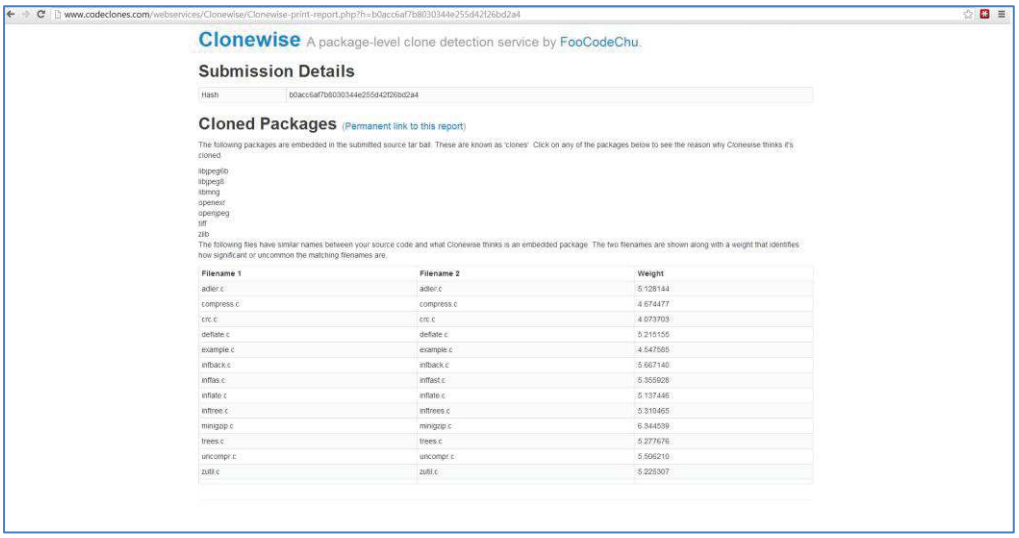

Fig. 48. Clonewise results.

samples and enter them into its database. The difference between user submissions and submissions from the malware feed is that the requests from the feed are marked so that a copy of the sample's binary is not made on the servers. This is a requirement to limit the disk usage. Otherwise, disk space on the servers would quickly reach capacity.

6.4 Availability

The Simseer and Clonewise servics are free to use. Simseer can be accessed on the web at http://www.simseer.com. The Clonewise service can be accessed on the web at http://www.codeclones.com.

Concluding Remarks

In this chapter we have demonstrated novel services to detect and analyse malware variants and to detect package-level clones in software. The Simseer and Clonewise services are deployed as cloud services and are free to use. We are the first to make a public service that analyses executable binaries and software tar balls in these contexts and see the area of cloud based software analysis and similarity detection as having future growth.

Chapter 7: Future Work and Conclusion

In this section we discuss potential areas of future work for each of our systems. FInally, we conclude the thsis.

7.1 Future Work

7.1.1 Clonewise

In Clonewise, although we decided not to use the original set theory approach to perform package-level clone detection, some interesting problems can still be examined. For example, given a set of packages, one can build a signature of at least k filenames by finding k-cliques (k-bicliques) in a bipartite graph where nodes in one partition represent packages, nodes in the other partition represent filenames, and an edge exists when a package contains a filename. Another research direction could be to consider a package as a directory tree. Finding the maximum common subtree between two packages identifies common code and could be used as a signature.

Using our classification approach, there are several ways we could see it applied to improve current practice. We could apply our system to more source code, including other Linux distributions, BSD vendors and also online source code repositories such Sourceforge [180]. It is conceivable that source code repositories could offer services to find package clones. Our system could be integrated into a package build system to automatically update the embedded database information or ask for validation from a package maintainer. Debian Linux would like our Clonewise tool to run constantly in the background and scan the source code repository to update a live database of clones. If we did this, we could enforce build recommendations that aim for avoidance of embedded code. The Debian Linux security team has asked us to perform this integration into their distribution as part of a standard operating procedure for when a vulnerability is found in a package and this is a focus of our current work.

7.1.2 Wire

An important aspect of Wire that we would like to implement is 64-bit support for x86. Most malware is still 32-bit so this does not present an immediate concern, but 64-bit would be

required at some point in the future. Another aspect we would like to work on is the data flow analysis framework we implemented using Wire. We see this as giving us the potential to go outside of the field of software similarity and classification and into bug detection. We have done some initial work on this and already have had some interesting results and found real world bugs [181]. Future work may also see theorem proving added to the system including weakest precondition and verification condition generation. Using proof assistants may also help analysts show equivalence between malware codes.

7.1.3 Malwise II

Malwise could be extended by using any-time, incremental, or stream clustering in addition to the similarity search it currently uses. In stream clustering, malware could be added to existing clusters in an online process as submissions are made to the system. Experimenting with clustering may help our system. Another aspect we have considered is the use of distance metric learning. A distance metric could be trained given a small set of labelled data.

7.1.4 Cloud Services

One thing we would like to do in our cloud services is replace our custom scheduling work queue with an enterprise messaging system such as RabbitMQ. Enterprise-level messaging systems have guarantees on reliabilities in the case of transmission or network failures. Using such a system would improve our reliability.

An option to improve the clustering service is using any-time clustering on the stream of samples that are given to Simseer. In this approach, cluster analysis is performed incrementally as objects are given to the system sequentially. An any-time phylogenetic tree analysis could follow on from any-time clustering. Any-time clustering could provide intelligence into new families of malware that are given to Simseer. This could benefit analysts in determining if a new sample relates to an existing family is something never seen before or relatively new.

7.2 Conclusion

In this thesis, we surveyed the state-of-the art in software similarity and classification. The thesis made disparate literature become a cohesive whole by showing that a number of

areas were very related. In some areas of software similarity and classification, theory is more developed and in other areas different algorithms and analysis techniques have been proposed. For example, the software similarity problem definition presented in the introduction stems directly from the area of software theft. This theory, which employs the birthmark concept, had not been used in malware similarity. This cohesive presentation of literature gave the foundation for this thesis to extend and propose new ideas, algorithms, and complete systems, all while significantly contributing to knowledge. To recap the major contributions of this thesis, we:

- Proposed the concept of package-level clones and automated some of their applications.
- Proposed using pattern classification to detect package-level clones
- Proposed and formally defined a new intermediate language that combines low level semantics with high level information recovered from decompilation.
- Proposed new types of graph-based malware signature that allows for efficient comparison, indexing, and searching.
- Proposed and implemented a complete infrastructure for malware and clone detection in the cloud.

In Clonewise, we evaluated our system using real-world data including an entire Linux distribution, and over 15,000 malware found in the wild. This system improves current practice by automating the tedious and manual practice in current use. Clonewise was shown to perform effectively, which was demonstrated by finding previously unknown clones, bugs, and vulnerabilities.

In Wire, we presented a formal intermediate language suitable for low level binary analysis. We demonstrated that this language could be used in a purely theoretic context to detect code equivalence, software similarity and classification. We used Wire in a practical context when we applied it to our malware detection system to extract intermediate representations of programs that enabled malware variants to be detected.

Malwise II was shown to be effective and demonstrated to be efficient. Typical graph based comparisons perform in NP complexity, yet using our novel birthmark representations, we were able to improve the efficiency using vector-based signatures.

We extended all the systems implemented to execute in the cloud. The cloud-based systems make this work available to many potential users. We used a scalable cluster-based infrastructure allowing us to grow the services as use increases. We automated almost all of our builds and deployment, making our infrastructure reproducible and resilient.

In summary, our systems found real bugs and vulnerabilities in Linux, gave analysts the capability to identify malware strains or families, and gave researchers new tools and techniques in software similarity and classification. The algorithms we presented were demonstrated to work in real environments and contributed significantly to knowledge.

References

[1] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR,* vol. 541, p. 115, 2007.

[2] Symantec, "Symantec internet security threat report: Volume XII," Symantec2008.

[3] Symantec, "Internet Security Threat Report," vol. 16, 2011.

[4] F-Secure. (2007, 19 August 2009). F-Secure Reports Amount of Malware Grew by 100% during 2007. Available: http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2007/fs_news_20071204_1_eng.html

[5] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," presented at the Proceedings of the 16th ACM conference on Computer and communications security, Chicago, Illinois, USA, 2009.

[6] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95)*, 1995, p. 86.

[7] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research (CASCON '93)*, 1993, pp. 171-183.

[8] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. Matsumoto, "Dynamic software birthmarks to detect the theft of windows applications," in *International Symposium on Future Software Technology (ISFST 2004)*, 2004.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Reading, MA: Addison-Wesley, 1986.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 9, pp. 319-349, 1987.

[11] J. R. Levine, *Linkers and loaders*: Morgan Kaufmann Pub, 2000.

[12] M. Pietrek, "Inside Windows-An In-Depth Look into the Win32 Portable Executable File Format," *MSDN Magazine,* pp. 80-92, 2002.

[13] T. I. Standard, "Executable and Linking Format (ELF) Specification Version 1.2," *TIS Committee, May,* 1995.

[14] T. Lindholm and F. Yellin, *Java virtual machine specification*: Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand 1173-3500, 1997.

[16] M. Joy and M. Luck, "Plagiarism in programming assignments," *Education, IEEE Transactions on,* vol. 42, pp. 129-133, 1999.

[17] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, "Malware normalization," University of Wisconsin, Madison, Wisconsin, USA Technical Report #1539, 2005.

[18] C. Mihai and J. Somesh, "Testing malware detectors," presented at the Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA, 2004.

[19] L. Cullen and D. Saumya, "Obfuscation of executable code to improve resistance to static disassembly," presented at the Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA, 2003.

[20] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *Computer Security Applications Conference*, 2006, pp. 289-300.

[21] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Rotalume: A Tool for Automatic Reverse Engineering of Malware Emulators," 2009.

[22] Panda Research. (2007, 19 August 2009). Mal(ware)formation statistics - Panda Research Blog. Available: http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx

[23] A. Stepan, "Improving proactive detection of packed malware," in *Virus Bulletin Conference*, 2006.

[24] J. Oberheide, M. Bailey, and F. Jahanian, "Polypack," in *USENIX Workshop on Offensive Technologies (WOOT '09)*, Montreal, Canada, 2009.

[25] T. Graf, "Generic unpacking: How to handle modified or unknown PE compression engines," presented at the Virus Bulletin Conference, 2005.

[26] (2010, 6 April 2010). *UPX: the Ultimate Packer for eXecutables*. Available: http://upx.sourceforge.net/

[27] (2010, 6 April 2010). *Themida*. Available: http://www.themida.com/

[28] S. Cesare, "Fast automated unpacking and classification of malware," Masters Thesis, Central Queensland University, 2010.

[29] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Workshop on Recurring Malcode*, 2007, pp. 46-53.

[30] L. Boehne, "Pandora's Bochs: Automatic Unpacking of Malware," University of Mannheim, 2008.

[31] H. G. Rice, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society,* vol. 74, pp. 358-366, 1953.

[32] S. S. Muchnick, *Advanced compiler design and implementation*: Morgan Kaufmann, 1997.

[33] J. Goguen and G. Malcolm, *Algebraic semantics of imperative programs*: The MIT Press, 1996.

[34] H. R. Nielson and F. Nielson, *Semantics with applications: an appetizer*: Springer Verlag, 2007.

[35] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM,* vol. 12, pp. 576-580, 1969.

[36] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM,* vol. 18, pp. 453-457, 1975.

[37] J. C. King, "Symbolic execution and program testing," *Communications of the ACM,* vol. 19, pp. 385-394, 1976.

[38] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security TISSEC (2008),* vol. 12, pp. 10:1-10:38, 2008.

[39] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, D. Song, and H. Yin, "BitScope: Automatically dissecting malicious binaries," Technical Report CMU-CS-07-133, School of Computer Science, Carnegie Mellon University2007.

[40] E. Clarke, "Model checking," 1997, pp. 54-56.

[41] C. Cifuentes, "Reverse compilation techniques," Queensland University of Technology, 1994.

[42] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977, pp. 238-252.

[43] M. Webster and G. Malcolm, "Detection of metamorphic computer viruses using algebraic specification," *Journal in Computer Virology,* vol. 2, pp. 149-161, 2006.

[44] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX Security Symposium*, 2004, pp. 18-18.

[45] T. Dullien and S. Porst, "REIL: A platform-independent intermediate representation of disassembled code for static code analysis," ed: CanSecWest, 2009.

[46] N. Nethercote and J. Seward, "Valgrind A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science,* vol. 89, pp. 44-66, 2003.

[47] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," presented at the Information Systems Security, 2008.

[48] K. Daniel, stner, and W. Stephan, "Generic control flow reconstruction from assembly code," *SIGPLAN Not.,* vol. 37, pp. 46-55, 2002.

[49] H. Theiling, "Extracting safe and precise control flow from binaries," presented at the Proceedings of the Seventh International Conference on Real-Time Systems and Applications, 2000.

[50] K. Johannes, Z. Florian, and V. Helmut, "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries," presented at the Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, Savannah, GA, 2009.

[51] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," *Algebraic Methodology and Software Technology,* pp. 81–95, 2006.

[52] D. Brumley and J. Newsome, "Alias analysis for assembly," Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 20062006.

[53] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, "Wysinwyx: What you see is not what you execute," *Verified Software: Theories, Tools, Experiments,* pp. 202-213, 2007.

[54] F. Leder, B. Steinbock, and P. Martini, "Classification and Detection of Metamorphic Malware using Value Set Analysis," in *Proc. of 4th International Conference on Malicious and Unwanted Software (Malware 2009)*, Montreal, Canada, 2009.

[55] K. C. S. Debray and T. K. G. Townsend, "Automatic Static Unpacking of Malware Binaries," presented at the Working Conference on Reverse Engineering - WCRE, 2009.

[56] M. J. Van Emmerik, "Static single assignment for decompilation," The University of Queensland, 2007.

[57] S. Hex-Rays, "IDA Pro Disassembler," ed, 2008.

[58] E. Moretti, G. Chanteperdrix, and A. Osorio, "New algorithms for control-flow graph structuring," presented at the Software Maintenance and Reengineering, 2001.

[59] T. Wei, J. Mao, W. Zou, and Y. Chen, "Structuring 2-way branches in binary executables," presented at the International Computer Software and Applications Conference, 2007.

[60] S. Cesare and Y. Xiang, "Classification of Malware Using Structured Control Flow," in *8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010)*, 2010.

[61] S. Cesare and Y. Xiang, "A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost," in *IEEE 24th International Conference on Advanced Information Networking and Application (AINA 2010)*, 2010.

[62] A. Mycroft, "Type-based decompilation," *Lecture notes in computer science,* pp. 208-223, 1999.

[63] R. N. Horspool and N. Marovac, "An approach to the problem of detranslation of computer programs," *The Computer Journal,* vol. 23, pp. 223-229, 1979.

[64] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Annual Computer Security Applications Conference (ACSAC)*, 2007.

[65] G. Hunt and D. Brubacher, "Detours: binary interception of Win32 functions," presented at the Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3, Seattle, Washington, 1999.

[66] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," presented at the Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005.

[67] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," presented at the Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 2000.

[68] W. Guizani, J. Y. Marion, and D. Reynaud-Plantey, "Server-side dynamic code analysis," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, 2009, pp. 55-62.

[69] D. Quist and Valsmith, "Covert Debugging Circumventing Software Armoring Techniques," in *Black Hat Briefings USA*, 2007.

[70] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*, 2008, pp. 51-62.

[71] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," *Lecture notes in computer science,* vol. 4779, p. 1, 2007.

[72] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference*, 2005, pp. 41–46.

[73] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A tool for analyzing malware," in *European Institute for Computer Antivirus Research (EICAR 2006)*, 2006.

[74] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*: Elsevier Science Inc., 1977.

[75] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins, "Text classification using string kernels," *The Journal of Machine Learning Research,* vol. 2, pp. 419-444, 2002.

[76] K. Grauman and T. Darrell, "The Pyramid Match Kernel: Efficient Learning with Sets of Features," *J. Mach. Learn. Res.,* vol. 8, pp. 725-760, 2007.

[77] R. Kondor and T. Jebara, "A kernel between sets of vectors," in *Proceedings of ICML'2003*, 2003, p. 361.

[78] M. Collins and N. Duffy, "Convolution kernels for natural language," *Advances in neural information processing systems,* vol. 1, pp. 625-632, 2002.

[79] H. Kashima and A. Inokuchi, "Kernels for graph classification," 2002, p. 25.

[80] K. M. Borgwardt and H. P. Kriegel, "Shortest-path kernels on graphs," presented at the Data Mining, 2005.

[81] N. Y. Peter, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, Austin, Texas, United States, 1993, pp. 311-321.

[82] M. R. Vieira, F. J. T. Chino, C. Traina, Jr., and A. J. M. Traina, "DBM-Tree: A Dynamic Metric Access Method Sensitive to Local Density Data.," in *Brazilian Symposium on Databases*, Brazil, 2004, pp. 163-177.

[83] C. Paolo, P. Marco, and Z. Pavel, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," presented at the Proceedings of the 23rd International Conference on Very Large Data Bases, 1997.

[84] R. Cilibrasi and P. M. B. Vitányi, "Clustering by compression," *Information Theory, IEEE Transactions on,* vol. 51, pp. 1523-1545, 2005.

[85] S. Brecheisen, "Efficient and Effective Similarity Search on Complex Objects," Ludwig-Maximilians-Universität München, 2007.

[86] P. Bille, "A survey on tree edit distance and related problems," *Theoretical Computer Science,* vol. 337, pp. 217-239, 2005.

[87] R. Baeza-Yates and G. Navarro, "Fast approximate string matching in a dictionary," in *South American Symposium on String Processing and Information Retrieval (SPIR'98)*, 1998, pp. 14-22.

[88] Caetano Traina, Jr., J. M. T. Agma, S. Bernhard, and F. Christos, "Slim-Trees: High Performance Metric Trees Minimizing Overlap Between Nodes," presented at

the Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology, 2000.

[89] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," 1998, pp. 604-613.

[90] D. Novak and P. Zezula, "M-Chord: a scalable distributed similarity search structure," presented at the Proceedings of the 1st international conference on Scalable information systems, Hong Kong, 2006.

[91] M. Batko, C. Gennaro, P. Savino, and P. Zezula, "Scalable similarity search in metric spaces," 2004, pp. 213-224.

[92] M. Batko, C. Gennaro, and P. Zezula, "A scalable nearest neighbor search in p2p systems," *Databases, Information Systems, and Peer-to-Peer Computing,* pp. 79-92, 2005.

[93] P. Haghani, S. Michel, and K. Aberer, "Distributed similarity search in high dimensions using locality sensitive hashing," presented at the Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, Saint Petersburg, Russia, 2009.

[94] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning,* vol. 20, pp. 273-297, 1995.

[95] K. Griffin, S. Schneider, X. Hu, and T. Chiueh, "Automatic Generation of String Signatures for Malware Detection," in *Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009*, Saint-Malo, France, 2009.

[96] J. O. Kephart and W. C. Arnold, "Automatic extraction of computer virus signatures," in *4th Virus Bulletin International Conference*, 1994, pp. 178-184.

[97] G. Wicherski, "peHash: A Novel Approach to Fast Malware Clustering," in *Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET'09)*, Boston, MA, USA, 2009.

[98] S. Wehner, "Analyzing worms and network traffic using compression," *Journal of Computer Security,* vol. 15, pp. 303-320, 2007.

[99] Y. Zhou and W. M. Inge, "Malware detection using adaptive data compression," in *Proceedings of the 1st ACM workshop on Workshop on AISec (AISec '08)*, 2008, pp. 53-60.

[100] W. Andrew, M. Rachit, R. C. Mohamed, and L. Arun, "Normalizing Metamorphic Malware Using Term Rewriting," presented at the Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, 2006.

[101] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics,* vol. 1, pp. 156-168, 2007.

[102] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology,* vol. 1, pp. 13-23, 2005.

[103] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, 2008, pp. 301-310.

[104] J. Z. Kolter and M. A. Maloof, "Learning to detect malicious executables in the wild," in *International Conference on Knowledge Discovery and Data Mining*, 2004, pp. 470-478.

[105] M. Gheorghescu, "An automated virus classification system," in *Virus Bulletin Conference*, 2005, pp. 294-300.

[106] Y. Ye, D. Wang, T. Li, and D. Ye, "IMDS: intelligent malware detection system," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007.

[107] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X. F. Wang, and U. C. Santa Barbara, "Effective and efficient malware detection at the end host," in *18th USENIX Security Symposium*, 2009.

[108] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005)*, Oakland, California, USA, 2005.

[109] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," presented at the Proceedings of the 12th USENIX Security Symposium, 2003.

[110] E. Carrera and G. Erdélyi, "Digital genome mapping–advanced binary malware analysis," in *Virus Bulletin Conference*, 2004, pp. 187-197.

[111] I. Briones and A. Gomez, "Graphs, Entropy and Grid Computing: Automatic Comparison of Malware," in *Virus Bulletin Conference*, 2008, pp. 1-12.

[112] X. Hu, T. Chiueh, and K. G. Shin, "Large-Scale Malware Indexing Using Function-Call Graphs," in *Computer and Communications Security*, Chicago, Illinois, USA, pp. 611-620.

[113] T. Dullien and R. Rolles, "Graph-based comparison of Executable Objects (English Version)," in *SSTIC*, 2005.

[114] G. Bonfante, M. Kaczmarek, and J. Y. Marion, "Morphological Detection of Malware," in *International Conference on Malicious and Unwanted Software, IEEE*, Alexendria VA, USA, 2008, pp. 1-8.

[115] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," *Lecture notes in computer science,* vol. 3858, p. 207, 2006.

[116] H. Park, S. Choi, H. Lim, and T. Han, "Detecting code theft via a static instruction trace birthmark for Java methods," 2008, pp. 551-556.

[117] G. Myles and C. Collberg, "K-gram based software birthmarks," presented at the Proceedings of the 2005 ACM symposium on Applied computing, Santa Fe, New Mexico, 2005.

[118] H. Tamada, M. Nakamura, A. Monden, and K. I. Matsumoto, "Java Birthmarks-Detecting the Software Theft," *IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS E SERIES D,* vol. 88, p. 2148, 2005.

[119] H. Lim, H. Park, S. Choi, and T. Han, "Detecting theft of java applications via a static birthmark based on weighted stack patterns."

[120] H. Park, H. Lim, S. Choi, and T. Han, "A Static Java Birthmark Based on Operand Stack Behaviors," in *Proceedings of the 2008 International Conference on Information Security and Assurance (ISA 2008)*, 2008, pp. 133-136.

[121] H. Lim, H. Park, S. Choi, and T. Han, "A Static Java Birthmark Based on Control Flow Edges," in *Computer Software and Applications Conference (COMPSAC '09)*, 2009, pp. 413-420.

[122] H. Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology,* vol. 51, pp. 1338-1350, 2009.

[123] S. Choi, H. Park, H. Lim, and T. Han, "A static API birthmark for Windows binary executables," *Journal of Systems and Software,* vol. 82, pp. 862-873, 2009.

[124] S. Choi, H. Park, H. Lim, and T. Han, "A static birthmark of binary executables based on API call structure," *Advances in Computer Science–ASIAN 2007. Computer and Network Security,* pp. 2-16, 2008.

[125] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A software birthmark based on dynamic opcode n-gram," in *Proceedings of the International Conference on Semantic Computing (ICSC '07)*, 2007.

[126] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," *Information Security,* pp. 404-415, 2004.

[127] O. MORIYAMA, T. FURUE, T. TOOYAMA, and T. MATSUMOTO, "A Method of Software Dynamic Birthmarks Using History of API Function Calls," *IEIC Technical*

Report (Institute of Electronics, Information and Communication Engineers), vol. 106, pp. 77-84, 2006.

[128] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," presented at the Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, 2007.

[129] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Design and evaluation of dynamic software birthmarks based on api calls," *Nara Institute of Science and Technology, Technical Report,* 2007.

[130] D. Schuler and V. Dallmeier, "Detecting software theft with API call sequence sets," in *Proceedings of the 8th Workshop Software Reengineering*, Bad Honnef, Germany, 2006.

[131] E. L. Jones, "Metrics based plagarism monitoring," *Journal of Computing Sciences in Colleges,* vol. 16, pp. 253-261, 2001.

[132] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science,* vol. 8, pp. 1016-1038, 2002.

[133] M. J. Wise, "YAP3: improved detection of similarities in computer program and other texts," *SIGCSE Bull.,* vol. 28, pp. 130-134, 1996.

[134] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," *SIGCSE Bull.,* vol. 39, pp. 73-77, 2007.

[135] J.-W. Son, S.-B. Park, and S.-Y. Park, "Program Plagiarism Detection Using Parse Tree Kernels," in *PRICAI 2006: Trends in Artificial Intelligence*. vol. 4099, Q. Yang and G. Webb, Eds., ed: Springer Berlin / Heidelberg, 2006, pp. 1000-1004.

[136] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," presented at the Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, Philadelphia, PA, USA, 2006.

[137] Christoph Biedl, Mark Adler, and F. Weimer. (2011). *Discovering copies of zlib*. Available: http://www.enyo.de/fw/security/zlib-fingerprint/

[138] H. A. Basit and S. Jarzabek, "A Data Mining Approach for Detecting Higher-Level Clones in Software," *IEEE Trans. Softw. Eng.,* vol. 35, pp. 497-514, 2009.

[139] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," presented at the Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Dubrovnik, Croatia, 2007.

[140] Y. Dang, S. Ge, R. Huang, and D. Zhang, "Code Clone Detection Experience at Microsoft," in *Proceedings of the 5th International Workshop on Software Clones*, 2011.

[141] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, 2011.

[142] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," presented at the Proceedings of the 29th international conference on Software Engineering, 2007.

[143] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," 1999, p. 109.

[144] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering,* pp. 654-670, 2002.

[145] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, 2007, pp. 106-115.

[146] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI '04)*, 2004, pp. 20-20.

[147] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering,* pp. 176-192, 2006.

[148] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," 1998, p. 368.

[149] J. Krinke, "Identifying similar code with program dependence graphs," 2001, p. 301.

[150] J.-l. Gailly and M. Adler. (2011). *zlib*. Available: http://zlib.net

[151] (2011). *Debian Linux*. Available: http://www.debian.org

[152] Red_Hat. (2011). *Fedora Linux*. Available: http://fedoraproject.org

[153] A. Z. Broder, "On the resemblance and containment of documents," 1997, pp. 21-29.

[154] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proceedings of the National Academy of Sciences,* vol. 99, pp. 7821-7826, June 11, 2002 2002.

[155] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *SIGKDD Explor. Newsl.,* vol. 11, pp. 10-18, 2009.

[156] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation,* vol. 3, pp. 91-97, 2006.

[157] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management,* vol. 24, pp. 513-523, 1988.

[158] H. Kuhn, W., "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly,* 1955.

[159] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. Data Anal.,* vol. 6, pp. 429-449, 2002.

[160] T. Dacheng, T. Xiaoou, L. Xuelong, and W. Xindong, "Asymmetric bagging and random subspace for support vector machines-based relevance feedback in image retrieval," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 28, pp. 1088-1099, 2006.

[161] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE,* vol. 5, pp. 46-55, 1998.

[162] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation

Recent Advances in Parallel Virtual Machine and Message Passing Interface." vol. 3241, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 353-377.

[163] G. H. John and P. Langley, "Estimating continuous distributions in Bayesian classifiers," presented at the Proceedings of the Eleventh conference on Uncertainty in artificial intelligence, Montréal, Qué, Canada, 1995.

[164] J. R. Quinlan, *C4.5: programs for machine learning*: Morgan Kaufmann Publishers Inc., 1993.

[165] L. Breiman, "Random Forests," *Machine learning,* vol. 45, pp. 5-32, 2001.

[166] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," presented at the Proceedings of the 19th international conference on Computer aided verification, Berlin, Germany, 2007.

[167] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security*, 2008, pp. 238–255.

[168] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM,* vol. 18, p. 340, 1975.

[169] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology,* vol. 215, pp. 403-410, 1990.

[170] T. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in *Ninth Workshop on Algorithm Engineering and Experiments*, SIAM, 2007.

[171] L. I. Smith, "A tutorial on principal components analysis," *Cornell University, USA,* vol. 51, p. 52, 2002.

[172] (2010, 26 March 2010). *GDBI Arboretum*. Available: http://gbdi.icmc.usp.br/arboretum

[173] (2009, 21 September 2009). *Offensive Computing*. Available: http://www.offensivecomputing.net

[174] (2009, 21 September 2009). *mwcollect Alliance*. Available: http://alliance.mwcollect.org

[175] (2010, 26 March 2010). *Cygwin*. Available: http://www.cygwin.com

[176] R. Rolles, "Unpacking Virtualization Obfuscators," in *USENIX Workshop on Offensive Technologies (WOOT)*, Montreal, Canada, 2009.

[177] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, generic, and safe unpacking of malware," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 431-441.

[178] J. Felsenstein, "PHYLIP (phylogeny inference package), version 3.6," ed: Joseph Felsenstein., 2005.

[179] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," *Molecular biology and evolution,* vol. 4, pp. 406-425, 1987.

[180] Geeknet. (2011). *Sourceforge*. Available: http://sourceforge.net/

[181] S. Cesare, "Detecting Bugs Using Decompilation and Dataflow Analysis," in *Ruxcon Breakpoint*, 2012.