Malware Variant Detection

Khalid Mohamed Abdelrahman Y Alzarooni

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of the
University College London.

Department of Computer Science
University College London

March 2012

To my parents for their love and encouragement.

To Fatima for her endless support.

Declaration of Authorship

I, Khalid Mohamed Abdelrahman Y Alzarooni, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Signed: ____________________________________

// Abstract

Malware programs (e.g., viruses, worms, Trojans, etc.) are a worldwide epidemic. Studies and statistics show that the impact of malware is getting worse. Malware detectors are the primary tools in the defence against malware. Most commercial anti-malware scanners maintain a database of malware patterns and heuristic signatures for detecting malicious programs within a computer system. Malware writers use semantic-preserving code transformation (obfuscation) techniques to produce new stealth variants of their malware programs. Malware variants are hard to detect with today's detection technologies as these tools rely mostly on syntactic properties and ignore the semantics of malicious executable programs. A robust malware detection technique is required to handle this emerging security threat.

In this thesis, we propose a new methodology that overcomes the drawback of existing malware detection methods by analysing the semantics of known malicious code. The methodology consists of three major analysis techniques: the development of a semantic signature, slicing analysis and test data generation analysis. The core element in this approach is to specify an approximation for malware code semantics and to produce signatures for identifying, possibly obfuscated but semantically equivalent, variants of a sample of malware. A semantic signature consists of a program test input and semantic traces of a known malware code.

The key challenge in developing our semantics-based approach to malware variant detection is to achieve a balance between improving the detection rate (i.e. matching semantic traces) and performance, with or without the effects of obfuscation on malware variants. We develop slicing analysis to improve the construction of semantic signatures. We back our trace-slicing method with a theoretical result that shows the notion of correctness of the slicer. A proof-of-concept implementation of our malware detector demonstrates that the semantics-based analysis approach could improve current detection tools and make the task more difficult for malware authors. Another important part of this thesis is exploring program semantics for the selection of a suitable part of the semantic signature, for which we provide two new theoretical results. In particular, this dissertation includes a test data generation method that works for binary executables and the notion of correctness of the method.

Acknowledgements

I would like to sincerely thank my supervisor, David Clark, for his guidance and patience. He is the person whom I learned a lot from; the person whose guidance and expertise were instrumental in making this research possible. For this, I am indebted to him.

I am very grateful to Laurent Tratt and Kelly Androutsopoulos for their continued support and constructive comments all through my thesis. It is much appreciated.

Thanks to CREST group members for their support and encouragement. Many of them have provided helpful discussions and criticisms about my research over the years. A special thanks to Mark Harman, Kiran Lakhotia, Bill Langdon and former CREST members: Zheng Li and Youssef Hassoun for their valuable advice and insights that they kindly shared with me.

In several master student projects, I worked with Luri, Eman and Stefanos to develop and implement ideas related to this work. I have benefited from their feedback, and discussion over the years. I appreciate their work.

I am also very fortunate to have a loving and supportive family. My wife, Fatima, has been very supportive of me over the years. Without her, all of this would be meaningless. Thanks to my parents, brothers, and sisters for all of the love, encouragement, and support that they provided me over the years. A hug of thanks to my kids: Abdelrahman, Omar and Maitha for being so nice and cooperative during this journey.

Finally, I would like to thank the UAE financial support and all the staff at the UAE Academic Education Department without their support this thesis would not have been written.

Contents

Declaration of Authorship 3

Abstract 4

Acknowledgements 5

List of Tables 10

List of Figures 11

List of Algorithms 14

1 Introduction 15
- 1.1 Motivation 15
- 1.2 Problem Definition and Challenges 17
- 1.3 Proposed Solution 18
- 1.4 Contributions 20
- 1.5 Scope and Limitations 21
- 1.6 Thesis Overview and Structure 22

2 Background 25
- 2.1 The Malware Problem 25
  - 2.1.1 Basic Terms 25
  - 2.1.2 Malware Categories and Behaviour 27
- 2.2 Background on Code Obfuscation 28

2.2.1 The Notion of Code Obfuscation 29
2.2.2 Common Malware Obfuscating Techniques 29
2.3 Theoretical Limitations 35

3 Literature Review 36
3.1 Input Representations and Analysis Types 38
3.1.1 Input Representations 38
3.1.2 Analysis Types 39
3.2 Detection Approaches 40
3.2.1 Signature-based Detection 42
3.2.2 Behaviour-based Detection 45
3.2.3 Heuristic-based Approaches 48
3.2.4 Model Checking 50
3.2.5 Semantics-based Malware Detection 52
3.3 Malware Analysis Techniques 54
3.4 Discussion and Conclusion 56

4 Trace Slicing 60
4.1 Programming Language 61
4.1.1 Syntax 62
4.1.2 Semantics 62
4.2 Overview of Slicing Approaches 66
4.3 The Trace-Slicing Algorithm 67
4.3.1 Capturing Memory References and Assignments 68
4.3.2 Preliminary 71
4.3.3 Overview of the Trace-Slicing Algorithm 75
4.3.4 Description of the Trace-Slicing Algorithm 77
4.3.5 Implementation 81
4.4 Correctness of the Trace-Slicing Algorithm 86
4.5 Strengths and Limitations of the Trace Slicing Algorithm 92
4.6 Review of Related Work 94
4.7 Conclusion 96

5 Semantic Trace-based Detector 97
5.1 Overview of the Detection System 99

5.1.1 Defining Semantic Signatures 99
5.1.2 The Architecture of the Detection System 102
5.2 Signature Matching 103
5.2.1 Mapping Semantic Traces 103
5.2.1.1 State Matching 105
5.2.1.2 Trace Mapping 107
5.2.2 Program Variant Comparison 109
5.3 Input Extraction 113
5.3.1 Binary Code Extraction 113
5.3.2 Code Disassembly 114
5.3.3 `asm2aapl`: a Translator 115
5.4 Semantic Simulator (SemSim) 119
5.5 Prototype Evaluation 127
5.5.1 Prototype Setup 127
5.5.2 Evaluation 128
5.5.2.1 Signature Extraction 129
5.5.2.2 Detection of In-The-Wild Variants 130
5.5.2.3 Detection of Obfuscated Variants 135
5.5.2.4 False Positives 136
5.5.2.5 Classification of Malware Variants 137
5.5.3 Prototype Limitations 137
5.6 Conclusion 138

6 Test Data Generation for Malware Executables 140
6.1 Overview 141
6.2 Preliminaries 143
6.3 A Test Data Generation Problem: DDR Approach 148
6.3.1 Description of DDR Analysis 149
6.3.2 DDR Procedures 156
6.4 Description of the Extended DDR Algorithm 159
6.4.1 Expression Domain Evaluation Procedure 160
6.4.2 Update Domains Procedure 162
6.4.3 Examples 172
6.5 The Correctness Proof of Extended DDR 180

6.6 Related Work 188
6.7 Conclusion 190

7 Conclusion and Future Work 191

Bibliography 194

List of Tables

3.1 Malware detection techniques, their analysis types, input representations and year published. . . . 43

4.1 Program samples used in the evaluation of the trace slicing algorithm. 83

4.2 Summary of trace slicing algorithm results. . . . 84

5.1 Obfuscating transformations. . . . 108

5.2 Application of the translation rules (partial list) of assembly instruction syntax to AAPL code. . . . 119

5.3 Supported addressing modes in the semantic simulator. . . . 123

5.4 A sample of simulated system calls. . . . 125

5.5 Malware variants in the wild. . . . 133

5.6 Results of evaluating our detector on real-world malware variants. . 134

5.7 Comparison of the similarity and running time (in msec) for detecting malware variants using semantic signatures and *sig-wo-slice* signatures. . . . 134

5.8 The detection results of a set of obfuscated variants of four malware families. . . . 136

5.9 The similarity rates of randomly selected variants from different malware families. . . . 137

List of Figures

2.1 x86 assembly language code fragment of the Mobler worm, developed for Win32 operating systems (a). Code reorder obfuscation applied to a code variant of Mobler (b). 31

2.2 Garbage insertion obfuscation applied to the fragment of the Mobler code in Fig. 2.1 on page 31 (a). 32

2.3 A variant produced by applying equivalent code replacement obfuscation to the fragment of the Mobler code in Fig. 2.1 on page 31 (a). 33

2.4 A variant produced by applying variable renaming obfuscation to the fragment of the Mobler code in Fig. 2.1 (a). 34

3.1 A three-tier hierarchy diagram of malware-detection research. . . . 37

4.1 Instruction syntax and value domain of the Abstract Assembly Programming Language (AAPL). 63

4.2 Semantics of AAPL. 64

4.3 A sample program in AAPL (a) and its simulation trace (b). 65

4.4 *DDG* of the program P with respect to $t_{\mathtt{x}}$ in Figure 4.3. t'' and t' are computed by Algorithm 4.5 on page 80 for $r0$ and $*r1$ at positions 8 and 13, respectively (b). 81

4.5 Instrumentation tool infrastructure. 82

4.6 A simulation trace configuration t and its slice t'. 85

4.7 The program P' is produced by extracting the command sequence from the trace slice t' in Figure 4.4 on page 81; a simulation trace $t'_{\mathtt{x}}$ of P' on input $\mathtt{x} : n=1, m=2$ (b). 92
4.8 An obfuscated code variant of program P in Fig. 4.3 and its *DDG* after applying data obfuscations. 94

5.1 The architecture of the Trace-based Malware Detection System. The outcome of the system is either "yes" for a successful detection of a malware variant or "no" otherwise. 104
5.2 A sample program (a) and its variant (b), after applying program obfuscation techniques from Table 5.1. 109
5.3 $t_{\mathtt{x}}$ and $t'_{\mathtt{x}}$ are simulation traces of programs p and p', respectively, (in Fig. 5.2). 110
5.4 The semantic traces t_p and $t_{p'}$ are generated from $t_{\mathtt{x}}$ and $t'_{\mathtt{x}}$, respectively, of Fig. 5.3. 111
5.5 Mapping program states of trace variants in Fig. 5.4. 111
5.6 Implementation of the input extractor module. Solid-line boxes represent off-the-shelf tools. 113
5.7 Implementation of `asm2aapl` translator module. 115
5.8 A fragment of an assembly .asm file and its AAPL file from the Bho (win32) virus code. 117
5.9 Architecture of SemSim 120
5.10 The extraction process of a fragment of the semantic traces (part of signature) of the Binom family. 131
5.11 A fragment of the semantic trace in the *sig-wo-slice* signature of the Binom family from the simulation trace in Fig. 5.10(a). 132

6.1 A sample of AAPL code and its control flow graph. The dotted regions represent basic blocks of the code. 143
6.2 Syntactic categories and Syntax of the AAPL path language. 144
6.3 Semantics of the AAPL path language. 145

6.4 The flow diagram for the DDR algorithm. 151
6.5 A sample of AAPL code and its control flow graph revisited in Example 6.1. 152
6.6 A sample of AAPL code and its control flow graph for Example 6.2. 154
6.7 A code fragment in AAPL and its control flow graph for Example 6.3. 173
6.8 A code fragment in AAPL and its control flow graph for Example 6.4. 177

List of Algorithms

4.1 Algorithm to find reference data manipulators in a trace. 72
4.2 Algorithm to find definition data manipulators in a trace. 73
4.3 The construction of *DDG* in TSAlgo. 78
4.4 The computation of the slice in TSAlgo. 78
4.5 Trace-Slicing Algorithm (TSAlgo). 80
5.1 Algorithm to map semantic traces of two program variants. 112
5.2 Algorithm to compute program inputs during a program simulation. 124
5.3 Algorithm to find current values of program inputs. 125
5.4 SemSim. 126
6.1 Dynamic domain reduction. 157
6.2 Algorithm to evaluate domains of program expressions. 161
6.3 Algorithm to evaluate a new domain for an arithmetic expression. . 163
6.4 Algorithm to evaluate a new domain for a bitwise expression. 165
6.5 Algorithm to update domains of program input variables. 166
6.6 Algorithm to update the domain of a program data manipulator in a bitwise expression. 168
6.7 Algorithm to update the domain of a program data manipulator. . 170

Chapter 1

Introduction

1.1 Motivation

Nowadays a large number of personal computers, which are used by businesses, government agencies and individuals, are connected to the Internet. Most of these personal computers run commercial operating systems, such as Microsoft Windows and Mac OS, and free operating systems, such as Linux; studies have shown that these systems are an attractive target for computer hackers and criminals who develop malware [Sym03]. Malware is a generic term that describes all types of malicious executable programs (e.g. viruses, spyware, Trojans and worms). Malicious software poses a serious threat to the integrity and security of personal data and computer systems. Unfortunately, malware has turned into a profitable business for malware authors and their customers. Malware authors often sell malicious software toolkits to their inexperienced customers, who can quickly create new customised malicious code variants. For instance, in 2009, nearly 90,000 malicious files were identified to be unique variants of malicious files produced by the Zeus family toolkit [Sym10]. These tens of thousands of new malicious variants are then used to launch attacks, where each variant may only be targeted at a single machine. Thus, the problem of the wide spread of malicious software is likely to continue to grow in the future, as malware writers use new techniques to create more variants of their malicious software.

Many organisations around the globe suffer significant financial losses due to the rise in malware distribution and the weakness of security tools in place. A new study of 45 business corporations revealed that the cost of coping with malware

attacks ranges from $1 million to $52 million per year per company [Pan10]. According to Sophos [Sop11], the “Stuxnet” worm, which targeted Iran’s sensitive nuclear program computers, was one of the most advanced pieces of malware code. Before it performed its malicious functionality, Stuxnet was able to copy its own code into the machine’s system and to hide itself from Anti-virus (AV) scanners by making code alterations. Variants of the Stuxnet malware could be a possible threat to other nations’ infrastructure, as stated by the Congressional Research Service (CRS) [KRT10]:

> “A successful broad-based attack on the US, using new variants of the Stuxnet weapon, could do enough widespread damage to critical infrastructure.”

Malware detectors use a combination of anti-malware techniques, such as virus signature scanners and heuristic methods, to defend against malicious software. Most current commercial AV tools rely on a database of syntactical patterns or regular expressions that characterise known malware variants. Anti-virus companies very often update their databases whenever an unknown malware variant is encountered in the wild. From 2006 to 2009, the number of signatures created for new malware variants doubled every year [Sym10]. This figure is consistent with the overall observation that new malware variants are created using techniques that change the appearance but preserve the underlying functionality or behaviour of each malware variant. Recent studies conducted by AV companies [Pan10], demonstrate that current commercial AV products using traditional approaches such as pattern or heuristic-based detection are no longer effective in defending against malware threats. Thus, extracting and using semantic features that are preserved across variants of malware is the key to a robust malware detector.

In the next section (Section 1.2), we present the research problem we attempt to tackle in this thesis and discuss the research challenges we face. In Section 1.3, we present the proposed solution for the research challenges. In Section 1.4, we present our contributions. In Section 1.5, we discuss the scope and limitations of our solution. In Section 1.6, we outline the structure of this thesis.

1.2 Problem Definition and Challenges

Recent AV security reports [Sym10, Sop10, Sop11] show that malware continues to grow and replicate at alarming rates. AV lab experiments [Pan10] demonstrate the weakness of commercial anti-malware software in detecting new variants of known malware programs. Existing countermeasures that use pattern signatures, behavioural heuristics or the reputation approach, analyse known malware instances and extract properties such as syntactical and usage patterns. The extracted properties are used as signatures for detection. For instance, Norton examines captured malware variants and includes byte sequences of malicious instructions in its huge database of malicious programs [Nac10]. Each signature represents the variant code of a single malware class. Thus, most new variants with a different appearance (fingerprint) are undetected by current detection tools until new signatures are developed for them.

Malware writers improve their tactics in circumventing commercial security products with advanced variants of malicious programs. In the last few years, it is evident that after a new malware family is created and distributed for infection, many different variants of the same family are generated by applying *syntactic* code transformations such as packing and obfuscation techniques. Program transformation involves transforming program statements or instructions into semantically equivalent code with different syntax [Mor01, Nac97, SF01]. Therefore, malware writers with the aid of code transformation techniques are able to produce several new instances of malicious software, which are undetectable by most current detection methods. Therefore, major improvements in current malware detection techniques are required to tackle new transformed variants of malware. One new tactic to improve malware detection is to capture a semantic property of the malware and to detect different malware variants using that property.

In this thesis, we attempt to develop a new method to automatically detect (possibly) obfuscated variants of a malware using properties of *code semantics* as signatures from the known malware sample. The advantage of using a semantics-based detection approach is that semantic signatures are more generic and, thus, a single signature can be used to detect multiple variants, manually or automatically produced from the same malware program.

To tackle the detection of malware variants using a semantics-based approach, we face the following challenges:

- The first fundamental challenge is that the correct *selection of a semantic signature* for describing the behaviour of a malicious program has a great impact on the strength and efficiency of a semantics-based malware detector. Current techniques for malware signature generation extract syntactical patterns from the malware code as features to look for in suspicious files. Anti-malware detectors that incorporate these techniques have a fast detection phase but they suffer from high numbers of false positives – identifying benign programs as malicious – and as new malware variants alter their syntax these detectors are prone to higher false negative rates.

- The second challenge is the construction of a semantics-based detector that is resilient to code obfuscation. Because most new variants are created using various common code obfuscating techniques, a detector must be robust in dealing with a possibly obfuscated malware variant. Dealing with code obfuscations, by using de-obfuscation methods for handling different effects of obfuscations on code variants, may lower the performance of a detector and strengthen malware writers' evasion techniques. A more generic method is required in which a detector focuses on the core semantics of a malicious program that are not associated with obfuscation effects.

- The third fundamental challenge is the automation and the effectiveness of a semantics-based detector. A detector that relies on semantic signatures of code as its main form of detection of unknown instances of a malware class, has to be *automatic* and *fast* in creating semantic signatures and in identifying and classifying unknown variants of the malicious code with *minimal false alarms* (i.e., false positive and false negative rates).

1.3 Proposed Solution

In an attempt to deal with the above mentioned research challenges for developing a semantics-based approach, our solution tackles each challenge and takes the following form.

- Challenge 1: In an attempt to address the first challenge, we select a semantic signature of a known malware program that doesn't depend on irrelevant details in the program syntax (which may be introduced by code obfuscating transformations). A semantic signature of a malware class must contain

the necessary semantic features that allow a detector to identify and classify syntactically transformed variants of the malware. That is, a semantic signature has to describe unique semantic characteristics that exist across variants of a malware class. In Chapter 5 we propose a novel type of malware semantic signature, which is created based on the information of the code evaluation. A semantic signature of a malware program represents the approximate behaviour of the malware, which may be similar across semantically equivalent variants of the same malware. Moreover, we introduce *test data generation for malware executables* as a technique to identify the approximate behaviour from a known malware sample and to extract semantic signatures. Our technique works for executables and it explores the control flow graph of a program to identify a set of test inputs that guarantees to traverse a particular set of program execution paths, called *feasible program paths*. The test inputs can then be used to improve the construction of semantic signatures. Our conjecture is that no matter how new variants of a malware program alter their code, as long as they behave in a similar way, their semantic signatures do not change. Chapter 6 presents the technique and the developed algorithms backed by the correctness proof of the technique.

- Challenge 2: Our idea is for a semantics-based detector that can use the existing semantic signatures of known malware samples to perform its analysis of a suspicious executable program regardless of whether the program is obfuscated or not. To this end, we extract a test input for a known malicious code using a random test input generator, which may describe the malicious functionality of the code. Then, during the detection phase, the semantics-based detector simulates the execution of the candidate malware variant, using a random test input from the semantic signature, to capture the semantic details, i.e. *trace semantics*, and determine whether the semantic details of the code contain a known semantic signature. This approach is generic in the sense that it can handle any variant of a known malware program as long as it preserves the code semantics. Chapter 5 presents our semantic simulator for evaluating code variants using the semantic signature of known malware. Using experimental results, the chapter further illustrates that our approach is resilient to a common set of malware obfuscating transformations.

- Challenge 3: In an attempt to tackle the challenge of the automation and the effectiveness of a malware detector, we introduce a method that may improve both the generation of semantic signatures and the detection of new malicious code variants in the presence of code obfuscations. In Chapter 4 we introduce *trace slicing* as an automated method for extracting *fine-grained* sub-traces (semantic characteristics) from the simulation traces of malware code as part of the signature. Chapter 4 also presents a trace-slicing algorithm and its correctness proof. This step may produce smaller traces, closer to traces of the original (or unobfuscated) malware, improving both the speed and rate of the detection phase as the experimental results of Chapter 5 demonstrate. Moreover, to have a practical detector, we implement the detection phase as a separate step from the signature generation phase, which deals with malware variant candidates and implements a mapping algorithm. Chapter 5 presents our method of matching semantic signatures. The chapter further presents a proof-of-concept prototype system for the semantics-based approach backed by results that highlight its performance and detection rates.

By tackling the three challenges for semantics-based malware detection, this dissertation takes an important step in developing a robust and effective approach for malware variant detection.

1.4 Contributions

We introduce methods to improve the automatic detection and analysis of malicious program variants via the semantic analysis of code. Towards this goal, we make the following contributions in this dissertation:

1. **Specification of a semantic signature for malware.** We define an abstract machine language and its syntax as a target language for malware code. We present a definition of trace semantics for malware programs (Chapter 4). This specification allows the construction of semantic signatures and the automatic detection of malware variants (Chapter 5).

2. **Trace-slicing algorithm.** We describe and prove an algorithm for computing correct sub-traces from an executable program trace (Chapter 4). Trace

slices may improve the construction of semantic signatures of a malware variant by handling a class of malware code obfuscating transformations. We set up an implementation to show its practical use as a trace slicer for Intel x86 binaries. Then we incorporate an implementation of the algorithm into our semantics-based detector prototype (Chapter 5).

3. **Semantics-based malware variant detection algorithm.** We present a general architecture for detecting variants of a known malware sample (Chapter 5). It consists of a static analyser, trace slicer and a semantic trace-matching algorithm. The static analyser, called *Semantic Simulator*, is developed to construct semantic signatures for known malware and to capture semantic traces for candidate malware variants. The matching algorithm compares semantic signatures to detect subsequent malware variants. An experimental evaluation of the prototype on a collection of malware samples (in the presence of obfuscated code) shows the effectiveness of our approach in detecting variants of a malware sample. The experimental results confirm that automatically generated semantic signatures using the trace-slicing method may enhance the performance of the detection phase in terms of speed and accuracy. Also, the results highlight the capabilities of our detector as a classification tool for malware samples.

4. **Test data generation method.** We present a general testing method for extracting a set of test inputs in executable programs (Chapter 6). An algorithm, developed for an abstract machine language code, approximates a program by identifying a set of feasible program paths with program inputs that can be used to generate semantic signatures for program variant detection. A correctness proof is presented for the algorithm, which guarantees that a given feasible program path is traversed via the computed program test input.

1.5 Scope and Limitations

Since malware attacks are targeted at many different types of computer systems, a single panacea that can handle all malware threats in every environment may be impossible. We develop an approach, presented in this dissertation, under the following limitations:

- **New malware variants on personal computers.** Our approach for detecting malware programs cannot be the solution to the malware detection problem, as this problem is undecidable and malware writers increasingly armour their new malware generations with stronger evasion techniques. However, our approach focuses on handling the problem of detecting new, zero-day variants of already analysed malware programs. Also, this research aims to develop a detection system that examines suspicious files on one important environment, a personal (local) computer.

- **Dynamic code generation techniques.** Our approach cannot handle malicious programs that incorporate dynamic code generation techniques. If a malicious binary, for example, contains encrypted commands within its data segment or commands received at runtime via network communication with the malware master (e.g. a command-and-control server), our tool would fail to capture its semantics and, hence, would be unable to identify correctly its semantic signature.

- **Current static analysis tools.** Our approach operates under the assumption that most malicious executables can be handled by off-the-shelf static tools (e.g. packers and disassemblers) to unpack and disassemble the code and extract an approximate control flow graph. The robustness and efficiency of our approach directly depends on the static analysis techniques that are integrated into it [MKK07b]. In other words, our semantics-based technique to extract and detect malware signatures is as effective as the static analysis methods it is developed upon.

1.6 Thesis Overview and Structure

The core idea of the proposed method of malware variant detection is that the semantic signature reflects the unique characteristic of a known malware. A semantic signature is a pair consisting of a program input and a set of semantic traces. A semantic trace represents a sequence of execution contexts (i.e. register and memory states). The program input of a known malware can be used to simulate the execution of a program and to generate a simulation trace. The approach attempts to identify malware variants by reasoning about their behaviour during the simulation rather than how the syntactic structure of the code is represented.

We implemented a semantics-based detector exploiting the semantic signature to justify this idea. The goal of the detector is to see if the set of semantic traces in the malware signature is contained within the generated semantic traces of the program under test. The first phase of the detector produces a random input and generates a simulation trace for a known malware. With respect to different program registers and memory locations that have been defined in the trace, a set of backward slices of the trace is produced at the end of the trace. As code obfuscation may introduce some irrelevant operations before the malware performs its intended operations, all the registers and memory locations that are defined in the trace are selected as the slicing criteria at the end of trace. Two abstraction steps are implemented on the trace to produce a set of semantic traces. The first abstraction removes states that contain duplicate execution contexts and the second abstraction discards information about the command syntax. The second phase of the detector simulates a suspicious code with the same input as a known malware to produce a semantic trace. The detector searches, through a trace-matching step, the semantic trace of the suspicious code to see if these semantic slices can be found. That is, for each slice in the signature, the matching step measures the similarity between the slice and the semantic trace.

Note that our technique only generates trace slices for a known malware sample and compares them against the semantic trace of a suspicious program. This may be computationally more efficient than producing and comparing dependence graphs as exact graph or subgraph matching, i.e., graph or subgraph isomorphism, is computationally expensive when comparing a large number of graphs; both graph and subgraph isomorphism are NP problems (and subgraph isomorphism has been proven to be NP-complete)[GJ90].

Finally, with our test data generation algorithm, a set of test data can be produced for an executable code. The idea behind this method is that a set of test inputs may represent different program paths through a malware sample and hence, produce a more representative malware signature.

The rest of this thesis is organised as follows.

- **Chapter 2.** We present the malware problem and its definitions. We present code obfuscation theory, recalling several code obfuscation techniques used in generating malware variants. Also, the theoretical limitation of the malware detection problem is discussed.

- **Chapter 3.** We discuss related work in the areas of malware detection and analysis. The chapter provides the background from which this thesis is developed.

- **Chapter 4.** We present our abstract machine language (AAPL) and define trace semantics. Also, we describe the design, the correctness proof and implementation of the trace-slicing algorithm. We discuss the strengths and limitations of the trace slicer with respect to malware code obfuscating techniques.

- **Chapter 5.** We present our matching algorithm for identifying similar semantic details for (possibly obfuscated) malware variants using semantic signatures. We present our architecture for a semantics-based detector, which incorporates a semantic simulator (SemSim). SemSim takes an abstract machine code (AAPL), evaluates its commands using a given test input and collects program traces. As a proof-of-concept malware detection tool, we implement a malware variant detector and conduct four different experiments to evaluate our approach. The experiments are based on the construction of semantic signatures for a single test input of each malware sample.

- **Chapter 6.** We describe our testing method for exploring program behaviour through identifying program inputs (test data) for feasible program paths within the control flow graph of a program. We extend an existing test data generation method, called the dynamic domain reduction technique. The extended method automatically identifies test data for input variables of AAPL programs. We prove that our algorithm for computing test data for executables is correct.

- **Chapter 7.** We summarise our work, and highlight directions for future work.

Chapter 2

Background

In this chapter, we introduce the basic terms and definitions of the malware problem that we address in the thesis. In Section 2.1 we introduce the malware terms that are used in the research community. Section 2.2 gives the definitions of code obfuscation, recalling several malware obfuscation techniques used in generating new malware variants. In Section 2.3 we discuss the theoretical limitation of the malware variant detection problem.

2.1 The Malware Problem

2.1.1 Basic Terms

The term *malware* refers to any malicious software that could intentionally perform malicious tasks on a computer system or on networked systems. The following covers some basic definitions of the malware problem:

- *A virus* is a program that is designed to replicate itself and to spread from one machine to another using an infected (carrier) host program. That is a malicious program copies itself into a benign program. Once an infected program is executed, the virus starts its functionality, infects and damages the machine. Thus, viruses attempt to spread and infect within the infected machine.

- *A Trojan horse* is a program that is believed to be useful but which has a harmful intention towards the host machine. Some hidden part of this type of malware contain a malicious payload that may exploit or damage the host system. Also, Trojan horses can be spyware because of their malicious actions such as the unauthorised collection of a user's data.

- *A worm* is another type of malware that uses malicious code to propagate from one host to another in a networked system without user intervention – other malware types require external actions. A worm may have greater impact; it can execute harmful actions such as a denial of service attack on a network of computer systems or can use system resources for illegal purposes. A comprehensive description of these malware and other types can be found in [Szö05].

- *Malvertising* is the use of a malicious advertising program that targets a website by placing an advertisement alongside the standard set of ads in the website. A malvertising program conceals the malicious activities of its ads to evade detection. Malware writers use malvertising programs to scam website visitors into buying fake anti-virus software in order to resolve non-existent malware infections on victims' machines [Sop11].

- *A bot* is a malicious program that controls an infected machine; usually a bot is connected through the Internet with a large-scale botnet (i.e. a bot network). A botnet of infected machines can be easily controlled by a single attacker (bot master), who can send malicious commands to launch attacks to other machines or websites connected to the Internet.

- *A False Positive* is generated when a benign file is identified as malware because a match is found.

- *A False Negative* is generated when a scanned file is a malware file because no match is found.

- *A Malware detector* is a program that analyses a file (an executable binary object or a source code) and identifies whether the file is malicious. Thus, a malware detector is a function F that takes a program file as an input p and returns either *yes* if the input is believed to be malicious, or *no* if the input file is believed to be *benign*, i.e. $F : P \rightarrow \{yes, no\}$. Current detectors, including commercial detection tools, use features of malicious programs as signatures to identify malware. Malware features are extracted either from

the byte sequences in the case of malware executable binary files or from the instruction (program command) sequences in the case of malware source code files. The byte signature is constructed from the machine code representation, i.e. a sequence of hexadecimal code. The instruction signature is composed of heuristic information (e.g. the frequency of a sequence of system calls or instructions found in the code) about the malicious operations contained in the analysed code. In Chapter 3 we discuss several approaches proposed for constructing and detecting malware signatures. The main objective of deploying a malware detector is to prevent malware attacks on (local) computer machines by inspecting incoming files before they execute. The main drawbacks of current detection programs are that they are not resilient to malware obfuscation because they are prone to generating false positives and false negatives when tackling obfuscated malware variants. In the remainder of this chapter we present a partial list of common malware obfuscation techniques used for generating new hard-to-detect variants.

2.1.2 Malware Categories and Behaviour

Most malware families have similar behaviour and properties, which the majority of scanners use as signatures to detect malware variants. For instance, one of the properties of a worm is self-replication – a worm tries to spread by simply copying itself to a host machine through the communication channels of other infected machines. On the other hand, a virus will attempt to spread by a carrier such as an infected file or a media drive. In the following we will examine some common environments and the behaviour of malware.

Malware Environments. In order for malware to perform its malicious functionality and to infect other victims, some components or resources should exist. Malware writers usually develop their code for a particular operating system. For instance, Win32 viruses are effective against Microsoft Windows and may not work on other operating systems. Moreover, a malware may require that some particular applications are running on the victim system in order to be effective. For example, some virus attacks are only effective if a scripting language such as Microsoft VB script or JavaScript (.vbs, .js, etc.) files can execute on the local machine.

Means of Infection. Malware uses common methods of transmission between computer systems. One of the traditional methods, and the easiest, of transmitting malicious programs is via external media such as USB devices and memory disks; however, the rate of spreading malware using this method is considered low compared to other methods such as through networked systems. Malware writers find networked computer systems an excellent environment to replicate and spread their viruses and worms; therefore, inadequate security on a network means that a large number of systems are vulnerable to malicious attacks. Another means of malware infection between computer systems is electronic mail (e-mail). Malicious code can spread easily as a file attachment sent with an e-mail message to as many as possible e-mail users. This type of spreading mechanism requires only a little effort from malware writers to make successful attacks. E-mail-based malware falls into two categories: mailer and mass mailer malware. The first category uses mail software such as Microsoft Outlook; the list of e-mail addresses on the host machine is used by the virus to e-mail itself to other users. The second category uses its own SMTP engine to send malicious code to many e-mail addresses.

Malicious Behaviour. Each malware type has its own malicious intention or behaviour towards the infected machine. This behaviour is developed as program code and embedded within the malware payload. Thus, by examining a virus payload we can determine its malicious behaviour and the threat it poses to computer systems. Some common payload types are: Denial of Service (DoS), information theft and bandwidth flooding. DoS attacks make various services on a computer system unavailable for some period of time. The second type of payload compromises the security of the infected machine by stealing sensitive information such as user keystrokes and passing it to its master. Bandwidth flooding attacks occur when a malware payload contains commands to generate a large volume of traffic, which stop the machine from utilising its network bandwidth.

2.2 Background on Code Obfuscation

Obfuscation techniques transform a program into a variant that is hard and time-consuming to understand and reverse engineer with current analysis tools. In general, a code obfuscation technique consists of a set of transformation functions that aims to maximise the obscurity of the new program but preserves the semantics of the original program. In the malware world, there are several transformation

functions that can be applied to obfuscate malicious programs, including dead and irrelevant code insertions, equivalent code replacements and data encodings, etc. Section 2.2.1 covers the general notion of code obfuscation and Section 2.2.2 investigates several existing techniques used by malware writers to produce obfuscated malware variants with some examples.

2.2.1 The Notion of Code Obfuscation

Collberg et al. [CTL97, CTL98] presented the first formal definition of the obfuscation problem. They stated that a transformation function $\mathcal{T}$ maps a program $\mathcal{P}$ to a new program $\mathcal{P}'$ such that $\mathcal{P}'$ is resilient to deobfuscation and $\mathcal{P}'$ contains the same behaviour as $\mathcal{P}$. Collberg et al. put obfuscation transformations into three categories:

- **Lexical Transformations** modify information that is related to the structure of the program, e.g. by renaming identifiers, changing or removing debugging information and comments. This type of obfuscation transformation can also be used to protect the intellectual property rights of software.
- **Control-flow Transformations** make changes to the control flow of the program, e.g. through code reordering and jump insertion, and through the insertion of opaque predicates. The results of these transformations are hard for a deobfuscator to compute.
- **Data Transformations** obfuscate data and data structures in the program, e.g. variable-splitting transformations that split a single variable into two or more variables with new operations to produce an equivalent result to the original variable.

2.2.2 Common Malware Obfuscating Techniques

Malware writers develop their malicious code in such a way that a malware detector may not be able to detect it. Since many commercial anti-virus scanners look for common malicious behaviour and syntactic characteristics, malware authors deploy new evasive techniques to hide the true intentions of their malicious code and to generate new instances of their malicious programs. Each new variant of

a particular malware can be produced automatically (or manually) by applying obfuscating transformation techniques to the current malware code. A survey of malware obfuscation techniques is presented in [YY10]. We first present the types of obfuscated malware and then we discusses common techniques used to generate new malicious variants.

Stealth. The malware hides the actual changes it made to the system and shows clean data to the scanner. For instance, when an anti-virus program scans for infected areas of a disk, the virus provides an ideal state of the system without any infection.

Encryption. This evasion method is used to hide the presence of malware by encrypting the payload. This type of malware usually consists of a constant decryptor, an encryption key and the encrypted payload. Since the actual malware behaviour or functionality is encrypted, malware detectors find it difficult to detect the malicious code. The malware creates a copy of itself by encrypting its payload with a new encryption key, so that the new payload looks different to the original malware payload. However, encrypting malware always uses the same decryptor to create malware variants; malware detectors may use this as a signature to detect this type of malware.

Oligomorphic. A malware that uses this evasion technique encrypts its payload in the same way as encrypting malware but the malware can change its decryptor to create different copies of itself [Szö05]. The detection of oligomorphic malware is very hard when using the decryptor as a signature and it requires a close analysis of the malware decryptor generator.

Polymorphic. Malware of this type is equipped with the same evasion technique as the other malware discussed previously; however, it contains an encrypted body with several copies of the decryptor (polymorphic decryptor). When creating new instances, the malware uses different encryption keys in different instances so in each instance the malware body looks completely different from other variants. For example, some viruses such as Win32/Coke variants contain a polymorphic decryptor, which implements multiple layers of encryption over the body. Other viruses such as Win32/Crypto variants use random encryption techniques to get different versions of the decryptor [SF01]. Also, some polymorphic viruses apply obfuscation techniques to only their decryptor to evade detection. For instance, the decryptor code may be reordered by placing jump instructions or inserting garbage code to change the malware signature whilst preserving its semantics.

```
1 :  push eax
2 :  dec  esi
3 :  add  [eax],al
4 :  or   al,[eax]
5 :  add  [eax],al
6 :  push 0x0
7 :  cmp  al,[eax]
8 :  pop  esp
9 :  add  bl,dh
10 : xchg edi,eax
```

(a)

```
1 :  dec  esi
2 :  push eax
3 :  jmp  7
4 :  cmp  al,[eax]
5 :  pop  esp
6 :  jmp  12
7 :  add  [eax],al
8 :  push 0x0
9 :  or   al,[eax]
10 : add  [eax],al
11 : jmp  4
12 : xchg edi,eax
13 : add  bl,dh
```

(b)

Figure 2.1: x86 assembly language code fragment of the Mobler worm, developed for Win32 operating systems (a). Code reorder obfuscation applied to a code variant of Mobler (b).

The detection of this malware type requires emulating the polymorphic decryptor (PD) behaviour and then generating PD signatures.

Metamorphic. In order to further evade detection, malware writers have extended the above mentioned malware types by applying code obfuscation to the entire malware body. A metamorphic virus does not contain a decryptor and a constant data part, instead it consists of a single program code that can replicate into a completely different variant of the malicious code [Szö05]. It is difficult to detect this type of malware because with every malware evolution the signature used for detection needs to be completely different.

Description of Malware Obfuscations

Several different variants of malware can be generated by using one or several of the following (partial list of) common malware obfuscation techniques:

```
1 :  push eax
2 :  nop
3    dec  esi
4 :  add  [eax],al
5 :  nop
6 :  or   al,[eax]
7 :  jmp  12
8 :  mov [ebp-0x18],esp
9 :  mov [ebp-0x14],0x4010e0
10 : and eax,0x1
11 : mov [ebp-0x10],eax
12 : add  [eax],al
13 : push 0x0
14 : cmp  al,[eax]
15 : nop
16 : pop  esp
17 : add  bl,dh
18 : xchg edi,eax
```

Figure 2.2: Garbage insertion obfuscation applied to the fragment of the Mobler code in Fig. 2.1 on the preceding page (a).

- **Code reordering.** This technique changes the order of code instructions. It can be applied to independent instructions where their order in the malware body do not affect other instructions (i.e. there are no data or control dependencies between other code fragments). Unconditional branches (e.g. jump instructions) may be inserted, if necessary, to preserve the original execution order. Also, an opaque predicate command may be used to indicate the correct flow of control within the code. An opaque predicate is a condition that always returns *True* or *False* but which cannot be determined statically. Thus, code reordering allows the creation of new variants that are syntactically different from but semantically similar to the original malware code. Figure 2.1 on the preceding page shows an example of code reordering obfuscation. Note that each jump instruction in this example will point to the next instruction, thus, the modified code will still run just like the original code. Code reordering makes the task of handling obfuscated code more challenging for syntactic scanners.

```
1 :  push eax
2 :  sub  esi,1
3 :  add  [eax],al
4 :  or   al,[eax]
5 :  add  [eax],al
6 :  xor  ebx,ebx
7 :  cmp  al,[eax]
8 :  mov  esp,ebx
9 :  add  bl,dh
10 : xchg edi,eax
```

Figure 2.3: A variant produced by applying equivalent code replacement obfuscation to the fragment of the Mobler code in Fig. 2.1 on page 31 (a).

- **Garbage insertion.** Malware writers introduce this obfuscation technique as a way of creating new variants from existing malware. The technique inserts irrelevant instructions into a program, which do not affect the original behaviour of the program. For instance, a new variant can be generated by adding a fragment of code (i.e. garbage code) that will not be reachable during the execution of the malware program. Unconditional jump instructions allow the correct execution order to be retained and bypass all fragments of garbage code. Also, a sequence of `nop` instructions (e.g. the x86 NOP command) can be inserted anywhere in the code; they modify the syntactic signature of the variant but keep the semantics unchanged. There are other instruction sequences that are difficult for static analysis to identify as garbage code. For instance, inserting and retrieving dummy values through the stack or memory do not alter the original behaviour of the program but it make harder to reverse engineer the obfuscation. The example in Figure 2.2 illustrates this technique.

- **Equivalent Code Replacement.** This obfuscation technique replaces instructions with other instructions that preserve the semantics of the original code and creates a new program variant. This obfuscation technique involves a set of transformation rules that contain equivalent instruction sequences to replace one instruction sequence with another. Since the x86 assembly language provides a large set of instructions for programmers, a low-level operation can be implemented with several implementations using a different

```
1 :  push ebx
2 :  dec  esi
3 :  add  [ebx],al
4 :  or   al,[ebx]
5 :  add  [ebx],al
6 :  push 0x0
7 :  cmp  al,[ebx]
8 :  pop  esp
9 :  add  bl,dh
10 : xchg edi,ebx
```

Figure 2.4: A variant produced by applying variable renaming obfuscation to the fragment of the Mobler code in Fig. 2.1 (a).

set of instructions. For instance, the Intel x86 memory addressing modes provide flexible access to memory and registers, allowing one to easily manipulate data. This instruction set flexibility allow malware authors to produce new obfuscated malware variants. However, for an automatic detection of this obfuscation, a static analysis tool would need to maintain a set of rewriting rules to transform code variants into a canonical form for signature matching. Figure 2.3 shows an example of equivalent code replacement obfuscations.

- **Variable Renaming.** This obfuscation technique replaces program identifiers (e.g. registers, labels and constant names) throughout the code, but the code is equivalent in terms of semantics. The implementation of this technique is expensive in the sense that it requires static def-use and liveness (identifying which variables are live at each point in a program) analyses of program identifiers. Thus, to lower the cost, malware authors use this technique to generate code variants based on a few selected program registers and branch labels [Szö05]. An example of this technique is shown in Fig. 2.4.
- **Code and Data Encapsulation.** This method is also known, as *code packing* [MKK07b, Kuz07]; malware writers use packer tools such as the UPX packer to reduce the size of their malicious code, generally through compression. Thus, these tools are used to protect and hide malware variants from static analysis tools and scanners when they are distributed. According

to PandaLabs [Pan08], 78% of new malware variants were developed using this new technique in order to evade detection.

2.3 Theoretical Limitations

Fred Cohen first introduced one of the few solid theoretical results in the study of computer virology. In particular, Cohen presented a formal definition of a computer virus based on the Turing machine model of computation, and demonstrated that the problem of detecting viruses is undecidable [Coh87]. That is, no detector can perfectly detect all possible viruses. Cohen proved also that the detection problem due to the evolution of viruses from known viruses is undecidable, meaning that detection of (obfuscated) malware variants of known malware code is undecidable. Chess and White [CW00] applied formal computability theory to viruses and virus detection, showing that there are computer viruses which no algorithm can detect, even under a somewhat more liberal definition of detection.

Despite these theoretical limitations, which show that the problem of detecting malware is, in general, impossible, it could be the case that developing detection systems that handle a class of malware is a possible (partial) solution [YHR89]. Thus, the malware detection research community has begun to propose detection schemes that try to tackle the problem of detecting polymorphic and metamorphic malware variants.

Chapter 3

Literature Review

Current commercial anti-malware tools are constantly challenged by the increased frequency of malware outbreaks. Several malware analysis and detection approaches have been proposed to minimise the distribution of malicious programs. However, malware writers deploy new techniques such as obfuscation and altering program behaviour [KKSV10] in order to create new, undetectable, malicious programs that evade state-of-the-art detectors. We provide, through selective reference to some of the literature, a clearer understanding of the existing malware detection techniques. Malware detection approaches presented in the literature are based on various analysis strategies that are common in computer software analysis, i.e. *static*, *dynamic* and *hybrid*. Moreover, we propose classifying existing malware detection approaches into five broad categories:

- Signature-based
- Behaviour-based
- Heuristic-based
- Model checking-based
- Semantics-based

Before reviewing the development of malware detection research, we categorised it into three tiers. Research into the detection approach is placed at the top level of the hierarchy, followed by research into input representations and analysis types.

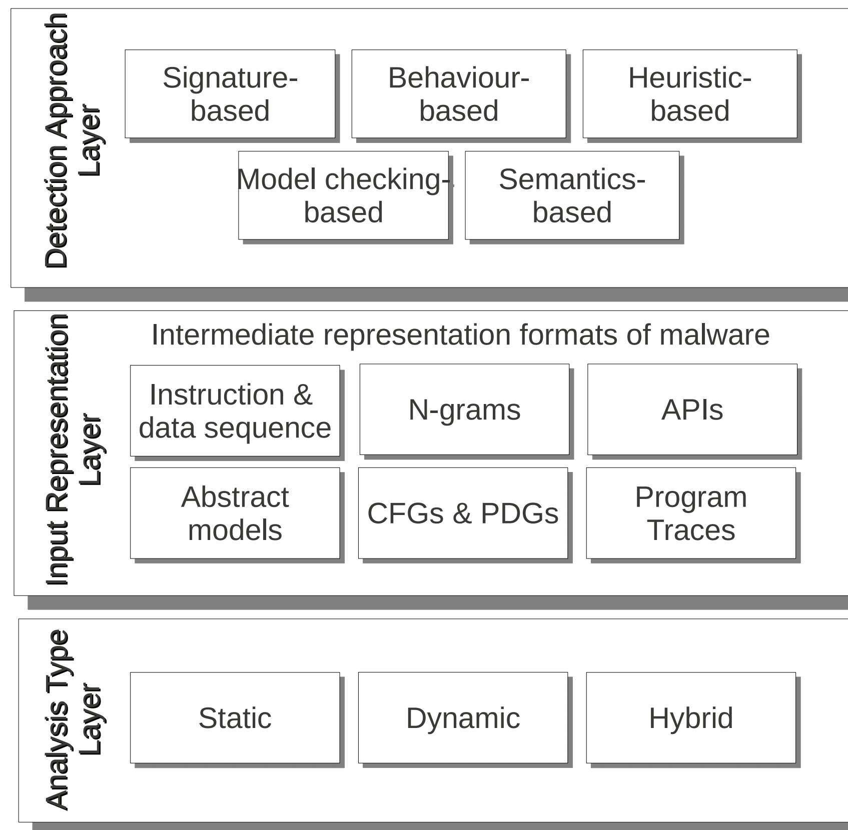


Figure 3.1: A three-tier hierarchy diagram of malware-detection research.

Input representations are the intermediate representation formats of malware programs produced as inputs to the malware detectors. The analysis type divides detection techniques into three groups: static, dynamic and hybrid. Figure 3.1 shows our three-tier hierarchy for malware-detection research. Our analysis in this chapter of the existing malware detection research is based on this hierarchy.

We cover the malware detection research hierarchy in the next two sections: Section 3.1 covers some input representations and the analysis types that are used in research. Then, we discuss several recent techniques in malware detection in Section 3.2. Section 3.3 covers some research into malware analysis techniques. We conclude this chapter, Section 3.4, with a discussion of the surveyed literature.

3.1 Input Representations and Analysis Types

3.1.1 Input Representations

Malware programs and their malicious behaviours are transformed into a suitable form that can be used as the input parameter to an analysis or detection tool. Input representations of malicious programs are used to capture various syntactical or semantic properties of the input code, and can be used to classify and detect malware families and their variants. Various reverse-engineering methods are deployed to perform abstraction of programs at different levels (i.e syntactic and semantic) of malware analysis. We cover some input representations of malware that are widely used in recent malware analysis and detection methods, such as control and data flow graphs, byte and instruction sequences, and system call sequences, abstract models and runtime execution traces.

- **Instruction and Data Sequences.** The byte representation of a specific sequence of instructions and data are extracted from a malware file (referred to as the string signature). This signature is typical of the malware but not likely to occur in benign programs. Most current anti-virus tools use this type of signature to scan and detect malicious programs. The signatures exactly match the corresponding malware sample and, hence, they are prone to a high number of false negatives (for new syntactically transformed variants of the malware).
- **N-grams.** An n-gram is a subsequence of n consecutive tokens in a stream of tokens. N-gram analysis has been applied to many text and speech processing tasks [JM08], and is well understood and efficient to implement. By converting a string of bytes in a malware executable into n-grams, it can be embedded in a vector space to efficiently distinguish between malicious and benign files by comparing the streams of bytes.
- **Application Programming Interfaces.** API calls and system events are interface methods that a host operating system (OS) provides for running processes (programs) to request OS services, such as creating a file, opening a network socket, etc. Thus, a sequence of system or API calls within an executable program could be used to describe its activity or behaviour and, hence, detect malicious programs.

- **Abstract Models.** An abstract model is a set of specifications describing the desired security properties of the system. Usually, specifications in malware analysis and detection systems are produced to represent the malicious behaviour of a malware program. A specification is expressed in a suitable temporal logic formula such as Linear Temporal Logic (LTL) [BM09] or Computation Tree Predicate Logic (CTPL) [KKSV10]. An abstract model allows the description and detection of a large set of known malware variants using a single specification formula.

- **Control Flow and Program Dependence Graphs.** CFGs and PDGs (including data and control dependence graphs) represent the semantic structures of programs. A suspicious program is detected as a malware variant if the malware variant CFG has a subgraph that is isomorphic to the CFG of the suspicious program [GBMKJYM07, AHJD10].

- **Program Traces.** A trace is an abstracted form of program semantics that expresses the malicious behaviour of malware. Program traces are generated at runtime or statically using all possible program paths. A database of abstract traces that represents malicious behaviour can be used to hold signatures for malware detection [PCJD07, BGM10].

3.1.2 Analysis Types

- **Static.** Static analysis discovers information about program control and data flows and other statistical features (executable formats, instruction opcode and library signatures) without actually executing the program. The main method used in the static analysis of malware is reverse engineering. Several reverse-engineering techniques are applied to extract input representations of a malware executable binary.

- **Dynamic.** Dynamic analysis requires executing the program that is being analysed and monitoring its execution in a real or a virtual machine environment. This provides actual information about the control and data flow, which can be used to extract more precise and accurate abstractions of program code. Dynamic analysis suffers from the execution overhead and provides information about a particular execution of a malware program for a given set of inputs. CWSand-box [WHF07] and TTAnalyze [BKK06] are dynamic analysis-based tools developed for analysing malware.

- **Hybrid.** Hybrid analysis is a mixed approach where the abstraction of malware produced by static analysis is refined by executing portions of the malicious code. Hybrid analysis can be performed at the source level by static analysis, which then passes information about the program to the dynamic analysis phase for detection. Hybrid analysis can improve the understanding of malware behaviour and, hence, may reduce false positive rates.

3.2 Detection Approaches

We will focus on the strengths and drawbacks of some new and interesting approaches to malware detection that have recently been proposed in the literature. This section surveys several proposed detection techniques for the five different approaches to malware detection: signature-based, behaviour-based, heuristic-based, model checking and semantic-based. It seeks to gain a general feel for the sort of work that is being undertaken for these five approaches and to see whether there are promising applications or theoretical ideas. Moreover, it is an attempt to identify the trends in this field.

Before we review the detection approaches in the following subsections, we will list and describe survey papers in this field of research. They were selected to provide a good survey of the current state of the research in this area and to give several different facets of the malware detection problem.

An overview of malware detection techniques is presented in [IM07]. This survey paper examines 45 malware detectors and classifies them into three different malware detection approaches: anomaly, signature and specification based. Patcha and Park[PP07] give a comprehensive survey of two malware detection systems: intrusion detection and anomaly detection techniques. The authors describe the generic architectural design of intrusion and anomaly detection systems. They highlight three main detection techniques for malware attacks in computer networks and systems: statical, data-mining and rule-based techniques. The authors suggest that building a hybrid system that consists of both anomaly and signature techniques could reduce the number of false alarms. Filiol et al. [JDF08] describe a new taxonomy of two main families of behavioural malware detectors. The authors classify and describe 29 proposed techniques that use simulation-based and formal verification methods. The survey considers four key points: data-collection

mechanisms, data interpretation, the adopted model and the decision process implemented within a detection system. The main idea of this survey is that the identification of functionality is a common principle in the different techniques examined. [SWL08] surveys data-mining techniques for malware detection using file features. The techniques are examined according to the features extracted from the binary program, an analysis of the technique (i.e. whether it is static or dynamic) and the detection type deployed (i.e. misuse or anomaly detection). The paper gives hierarchical categories for the 19 surveyed techniques. Glezer et al. [SMEG09] proposed a new taxonomy for categorising detection techniques of new malware variants using a machine-learning approach. The authors provide a comprehensive review of malware detectors that use machine-learning classifiers on static file features extracted from malicious executables. Their taxonomy has three dimensions: the file representation method, the feature selection method and the classification algorithm. The static features discussed in the paper are byte n-grams, OpCode n-grams and function-based features. The paper suggests that such a framework for detecting new instances of malware in executable files should combine multiple classifiers for various types of file feature in order to achieve a high accuracy in detection. In [RRZ+09], 48 malware detection techniques were reviewed and analysed for improving the capability of Intrusion Detection Systems (IDS). Furthermore, a new categorisation, called the Hybrid-Malware Detection Technique (Hybrid-MDT), of malware detection techniques was proposed. Hybrid-MDT consists of two hybrid techniques: the hybrid signature-anomaly technique and the hybrid specification-anomaly technique. Both techniques in Hybrid-MDT were proposed in an attempt to overcome the weaknesses found in the signature-based, anomaly-based and specification-based techniques. [FSR09] reviewed detection techniques in the literature for one type of malware: the botnet. This survey paper identifies four classes of techniques: signature-based, anomaly-based, DNS-based and data-mining-based. It highlighted proposed detectors in each class and gave a brief comparison of botnet detection techniques.

A significant amount of research has appeared in the professional literature over the last ten years addressing the problem of malware detection. Almost over 330 research papers have been published since 2001 in 12 computer security conferences (i.e. the IEEE Symposium on Security and Privacy, the Symposium on Recent Advances in Intrusion Detection, the Annual Computer Security Applications Conference, the ACM Conference on Computer and Communications Security, the

Annual Network and Distributed System Security Symposium, the USENIX Security Symposium, the European Symposium on Research in Computer Security, the IEEE Communications Society/CreateNet International Conference on Security and Privacy for Emerging Areas in Communication Networks, the Conference on Detection of Intrusions and Malware and Vulnerability Assessment, the ACM Symposium on Information, Computer and Communications Security, the International Conference on Applied Cryptography and Network Security and the Symposium on Usable Privacy and Security). The papers presented in this overview were selected from various yearly conferences on computer and information security. We will focus on the most relevant (and recent) papers for each approach and aim to provide the background from which we will develop our thesis. Table 3.1 on the following page gives detailed information about the five malware detection approaches.

3.2.1 Signature-based Detection

Signature-based detection is based on investigating suspicious code and gathering information in order to characterise any malicious intent of the malware. The main objective of this approach is to extract specific byte sequences of code as signatures and to look for a signature in suspicious files. An excellent overview of signature-based detection approach is provided in [IM07]. Most of today's commercial anti-virus scanners use a collection of signatures to detect malicious programs [SF01]. That is, the suspicious code is compared with a unique sequence of program instructions or bytes. If the signature is not found in the file, then the file is considered to be not malicious. A database of signatures has to be maintained continuously by manually analysing new variants of malware using static, dynamic or both analysis methods [SF01]. Therefore, one of the limitations of the signature-based detection approach is that it requires human intervention to update the database with new signatures. Moreover, Christodorescu et al. [CJ04] showed that the authors of metamorphic malware can easily defeat signature-based detectors by using obfuscation techniques, as discussed in Section 2.2.2. This leads us to conclude that this detection method is prone to false negatives (failure to raise alerts); also, as more malicious variants become known, the signature database grows in size making the false positive (false alerts) issue even more pervasive.

Chouchane and Lakhotia [CL06] extended the traditional string signature method by using an obfuscation engine signature. This method scans opcode instruction

Technique	**Analysis type**	**Input representation**	**Year**
		Signature-based Approaches	
Chouchane and Lakhotia [CL06]	static	sequence of x86 opcode mnemonics	2006
Karnik et al. [KGG07]	static	instruction opcode mnemonics	2007
Bruschi et al. [BMM07]	static	program instructions	2007
Bonfante et al.[GBMKJYM07]	static	CFG	2007
Cha et al. [CMJ+10]	static	bit vectors of hashes of files	2010
		Behaviour-based Approaches	
Rabek et al. [RKLC03]	hybrid	APIs	2003
Kirda et al. [KKB+06]	hybrid	browser events and APIs	2006
Bailey et al. [BOA+07]	dynamic	system events	2007
Collins [Col08]	dynamic	network protocols events	2008
Aaraj et al. [ARJ08]	hybrid	regular expressions and data invariants	2008
		Heuristic-based Approaches	
Zhang et al. [ZYH+07]	static	n-gram byte sequences	2007
Bose et al. [BHSP08]	dynamic	TLCK of APIs and system events	2008
Moskovitch et al. [MFT+08]	static	n-grame of opcode sequences	2008
Griffin et al. [GSHC09]	static	byte sequence	2009
Shafiq et al. [STMF09]	static	APIs and ..	2009
		Model Checking Approaches	
Singh and Lakhotia [SL03]	static	LTL formulas	2003
Holzer et al. [HKV07]	static	CFG and CTPL model	2007
Beaucamps and Marion [BM09]	dynamic	LTL formulas	2009
Kinder et al. [KKSV10]	static	CTPL formulas	2010
		Semantics-based Approaches	
Christodorescu et al. [CJS+05]	static	control-flow graph template	2005
Moser et al. [MKK07a]	dynamic	input values and CFG paths	2007
Preda et al. [PCJD07]	hybrid	program trace semantics	2007
Feng and Gupta [FG09]	dynamic	instruction output	2009
Lee et al. [LJL10]	static	APIs graph	2010

Table 3.1: Malware detection techniques, their analysis types, input representations and year published.

patterns to detect variants of malware generated by a known engine. They designed an engine-specific scoring procedure targeted at the instruction-substitution technique. Their detector needs only the signature of the specific obfuscation engine and detects malicious variants, which the engine can produce. The method attempts to alleviate the problem of scanning a byte stream by identifying a set of instruction opcode sequences and then matching the instruction stream against the engine signature. However, this technique could be prone to miss true alarms since the malware engine writer only has to change the form of the engine instance, not the semantics.

Karnik et al. [KGG07] extracted a sequence of functions from a disassembled malware code. Each element in this sequence points to an array of opcode instructions, which exist in the function. Thus, the signature of a known malware program represents a sequence of functions and their opcode instruction groups. For a pair of signatures, the detector computes a cosine similarity measure to determine if the programs are obfuscated versions of each other. Their approach does not handle the instruction-substitution technique of code obfuscation and no experiments were presented for evaluating the rate of false positives of the method.

Bruschi et al.[BMM07] proposed a technique that has the ability to take patterns of malicious behaviour and create a normalised malware as a template for detection. The main idea of normalisation is to remove the effects of the most well-known transformation techniques and identify the control flow connections between the malicious parts of the code. The detector works as follows: first, it performs the normalisation step on the suspicious program, which may contain some malware patterns. Second, it generates the inter-procedural control flow graph CFG_P, which specifies all program function connections. Finally, the detector compares the graph CFG_P with the malware template graph CFG_M and decides whether CFG_M is actually a subgraph of CFG_P. The experimental evaluation of their approach shows that the detector can detect variants of malware code that contain specific patterns of instructions in the template. It is obvious that this technique relies on syntactic patterns of template code to detect malicious programs.

Bonfante et al. produced control flow graphs for assembly x86 programs and matched them with graphs of known malware programs [GBMKJYM07]. The graphs represent all paths that might be traversed through a program during its execution. To remove some classic obfuscation techniques, three graph rewriting rules are applied to the extracted graphs: instruction nodes are concatenated if contiguous data instructions have been changed, code is realigned if reordering jumps have been introduced and any existing consecutive conditional jumps are merged into a single conditional jump. Their detection technique is based on two algorithms. The first algorithm searches for isomorphism between the CFGs of known program m and the program under test p. The second algorithm uses the reduced CFG of p and tests if m is the reduction of the CFG of p. The two algorithms' false-positive ratio results against 2278 malware samples were successful with respect to the size of the CFGs. Algorithms 1 and 2 have a similar detection accuracy, with 4.5% and 4.4% false positive rates, respectively [GBMKJYM07].

Metamorphic malware programs are not examined under the proposed detection method.

Cha et al. describe a network-based system to detect distributed malware files that uses only a subset of malware signatures [CMJ+10]. The detection system is based on a text-based pattern matching technique called a Feed-Forward Bloom Filter (FFBF). Their method works in two steps. First, all files are scanned using FFBF. The filter outputs (1) a set of suspect matched files, and (2) a subset of signatures from the signature database needed to confirm that suspect files are indeed malicious. The system then performs a verification step to eliminate Bloom filter false positives. The suspect matched files are rescanned using the subset of signatures and an exact pattern matching algorithm. Two issues are addressed in this work. First, the technique handles the signature distribution challenges by producing a smaller subset of signatures from the verification step for matched malware files. Second, the memory scaling challenge is handled by using a single FFBF bit-vector to pattern-match input files and another bit-vector for generating the set of signatures for matched files. The FFBF-generated signatures proved to be a particularly useful input representation, as opposed to the traditional string signatures, of input files as the scanning throughput increased by over 2x and using half as much memory. However, like any other text-based Bloom filter output, the output pattern subset may contain false positives.

3.2.2 Behaviour-based Detection

Behaviour-based detection techniques aim to reduce the false positive rate generated during the monitoring stage. A behaviour-based technique monitors and checks the system to be protected against a given set of requirements and the security policy. During the learning phase, a behaviour-based detector is provided with a rule set, which specifies all acceptable behaviour any application can exhibit within the protected host. The major drawback of behaviour-based detection is the difficulty of determining the entire set of safe behaviour that a program can exhibit while running under a protected system. Filiol et al presents a survey of behaviour-based detection techniques in [JDF08].

Rabek et al. [RKLC03] present a detection method for obfuscated malware that dynamically injects and generates itself at runtime. The detector uses a static analysis technique to get details of all relevant system calls embedded in the code,

such as function names, addresses, and the address instruction followed by each system call. Also, the detector keeps a record of the return addresses for system calls in the code. Then, when a suspicious program is executed, the detector monitors the behaviour of the executable and ensures that all calls made to the system services at runtime match those recorded in the first step. The authors conclude, by a proof of concept study, that their technique ensures that any injected and generated malicious code can be detected when it makes unexpected system calls. A major drawback of this technique is that when inserting some irrelevant API calls into a malicious code, the detector may fail to match the new malicious behaviour with ones kept in the record.

Bailey et al. propose an approach based on the actual execution of malware samples and observation of their persistent state changes [BOA+07]. The input representation that is generated from malware programs is the set of environment state changes, referred to as a behavioural fingerprint. The behavioural fingerprints are extracted from the raw event logs during program execution. For instance, spawned process names, modified file names and network connection attempts are state changes, which are stored as part of a fingerprint for a known malware program. Classes and subclasses of malware are then formed by clustering groups of fingerprints that exhibit similar behaviour. They produced a distance metric that measures the difference between any two fingerprints, and used this metric for clustering. In particular, they utilised the normalised compression distance (NCD) approach to effectively approximate the overlap in information between two fingerprints. Then they constructed a tree structure based on a single-linkage hierarchical clustering algorithm to show the distance between two malware clusters.

Kirda et al. [KKB+06] proposed a novel method for spyware detection that is based on an abstraction of the characterisation of the behaviour of a popular spyware family. The method handles spyware applications that use Internet Explorer's Browser Helper Object (BHO) and toolbar interfaces to monitor a user's browsing behaviour. The method applies a hybrid analysis to binary objects to characterise and detect specific malicious program behaviour. First, a dynamic analysis technique is used to expose a suspicious component by simulating user activity. That is, they dynamically record both the browser events and the Window APIs that the component calls. Second, a static analysis step is implemented to extract the control flow graph (CFG) from the disassembled suspicious program. This step examines particular code regions in the CFG for the occurrence of operating system

calls. Then their behavioural characterisation is represented as a list of Windows API calls and browser functions. The characterisation is automatically produced by identifying the browser functions and Windows APIs performed by malicious BHOs and toolbars. A total of 51 samples (33 malicious and 18 benign) were used to evaluate their method. Seven (two benign and five malicious) out of 51 samples were used to develop the behavioural characterisation list. The remaining samples were considered unknown and were used to validate the effectiveness of the detection method. All remaining malicious samples were identified malicious and two benign programs were identified as malicious.

Collins [Col08] introduced the Protocol Graph Detector (PGD) to detect the behaviour of a slowly propagating hit-list or topologically aware worms over a network. The detector works by building protocol-specific graphs where each node in the graph is a host, and each edge represents a connection between two hosts with a specific protocol. The input representation used in this method is a set of network protocol events generated by hosts in the graph. The technique is based on the observation that during legitimate operations over short time periods, the number of hosts in the graphs is normally distributed and the number of nodes in the largest connected component of each graph is also normally distributed. PGD continuously measures the distribution numbers and it detects the presence of worms when both numbers are beyond their normal range.

Behavioural-based malware detection methods that incorporate hybrid-based analyses for producing and comparing similar behavioural signatures include Aaraj et al.'s dynamic binary instrumentation-based (DBI) tool [ARJ08]. They used an isolated testing environment, wherein a suspicious executable code is executed using a DBI technique. The runtime behavioural pattern of the program is checked against extensive security policies, which consist of a set of behavioural signatures of malicious programs. The runtime behavioural pattern is collected in the form of a hybrid model that represents the program's dynamic control and data flow execution traces. They used the same model to express the security policies of malicious behaviour. Then a suspicious program is extensively tested with the tool and its behaviour is checked against the model. An automatic input generation technique based on static analysis and symbolic propagation is deployed to generate input values. A formal proof to show the correctness of their input generation technique was not presented.

3.2.3 Heuristic-based Approaches

Heuristic-based approaches to malware detection deploy several well-known experience-based and machine learning techniques to search for specific attributes and characteristics for capturing malware variants. Most malware detection systems that make use of heuristic-based techniques do not need to create, match or maintain signatures. This approach usually detects an abnormality in the program under test or the host system where the program will run.

Detecting malware programs using a heuristic-based technique is accomplished in two phases. The first phase is to train the malware detector. A detector system must be trained with data in order to capture characteristics of interest. The second phase is the monitoring or detection phase where the trained detector makes intelligent decisions about new samples based on training data. Moreover, there are two methods deployed in the training phase. The first method uses two classes of data, i.e. both normal and abnormal data. The second method uses a single class of data, where malware detectors are trained with only one (normal or abnormal) class. This means the system only needs to be trained with normal system activity, allowing it to produce a useful output about what activity is abnormal. The rest of this section briefly outlines some proposed techniques in the literature that utilise this approach to malware detection. [SWL08] presents an in-depth survey of proposed methods in this approach.

Zhang et al. [ZYH+07] detected and classified new malicious code based on n-gram byte sequences extracted from the binary code of a file. They used a selection method called the information-gain to choose the best n-gram byte sequence. Then, a probabilistic neural network (PNN) was used to construct several classifiers for detection. Finally, during the learning step of each classifier, they produced a single set of decision rules consisting of the individual decision results generated from each PNN classifier. Three classes of new malicious program files downloaded from the the VX Heavens computer virus collection website [Hea] and some Windows benign files were used to evaluate their method. The false and true positive rate results show a great improvement of the combined PNNs over the individual PNN classifier of each of the three classes.

Bose et al. [BHSP08] proposed a detection method based on a supervised learning method, Support Vector Machines (SVMs), for malware variants that increasingly target mobile handsets. A lightweight temporal pattern classifier for malware

detection was implemented. The method requires a malicious behaviour signature database to train the classifier. Each type of malicious behaviour is described in terms of API calls and events represented by temporal logic of causal knowledge (TLCK). Thus, a single TLCK signature is used to represent the behaviour of an entire family of malware including their variants. The database is generated at runtime by monitoring the API calls and system events from more than 25 distinct families of mobile viruses and worms. Their evaluation showed that this method results in very high detection rates (over 96%) and is able to detect new malware that contain similar behavioural patterns with existing ones in the database.

Moskovitch et al. [MFT$^+$08] present a machine learning method for detecting unknown malware variants using the operation code (opcode) of the binary code as the input representation. The set of opcodes is converted into several n-grams (1, 2, 3, 4, 5 and 6-byte sequences of opcodes), which are used as inputs to the classifiers. They deployed four commonly used classification engines for detecting unknown malware files: Artificial Neural Networks (ANN), Decision Trees (DT), Nave Bayes (NB) and Adaboost.M1. The classifiers were compared and the results show that DT and ANN outperform NB and Adaboost.M1 by exhibiting lower false positive rates.

Griffin et al. [GSHC09] present a system that automatically generates string signatures. Each generated string signature is a contiguous 48-byte code sequence that potentially can match many variants of a malware family. The system uses two types of heuristics to examine and to find potential candidate signatures. That is, every 48-byte code sequence in unpacked malware files are inspected using probability-based and disassembly-based heuristics to find string signatures. From a large set of benign programs, a pre-computed Markov chain-based model probability threshold is used to filter byte sequences whose estimated occurrence in benign programs is above the limit. Further, they refine their string signature set into a multiple component signature (MCS) set that consists of multiple byte sequences that are potentially disjoint from one another. Each component in the MCS set represents a unique signature sequence found in the longest substring that is common to all malware files that have the sequence. They show that the multiple component signatures are more effective than single-component signatures, but the actual runtime performance impact of MCS is unclear.

Shafiq et al. [STMF09] developed a data mining-based framework that automatically extracts distinguishing features from portable executable (PE) files to detect

previously unknown malware. A PE file is a data structure that encapsulates the information necessary for the Windows OS loader to execute the wrapped executable code. They believe that the structural information contained within PE files such as dynamic-link libraries (DLLs) and PE section headers have the potential to achieve high detection accuracy. The proposed detection method consists of three steps: data extraction (the program input representation), data preprocessing and classification. The input representation is statically extracted as a set of a large number of features from a given PE file. To improve the training and testing of classifiers, three well-known data reduction filters are used in step 2. The filters are Redundant Feature Removal (RFR), Principal Component Analysis (PCA) and Haar Wavelet Transform (HWT). They evaluated their method with five different data-mining algorithms. Their results on two malware collections, the VX heavens and Malfease databases, show that the method achieves more than a 99% detection rate with a less than 0.5% false alarm rate for distinguishing between benign and malicious executables. Their method is robust with respect to different polymorphic techniques in PE files but it is not clear if it handles metamorphic techniques in new malware variants.

3.2.4 Model Checking

Singh and Lakhotia [SL03] introduced a malicious code verifier, which statically verifies binary executables against a property formula for viruses and worms. The malicious behaviour of viruses and worms is manually extracted and encoded using linear temporal logic (LTL) formulas. Each formula is representative of a particular action by the program. A program action is described by a sequence of one or more function system calls, connected through a flow relationship. They specified five functions, which are sufficient to describe the behaviour of a malicious program. These functions are survey, concealment, propagation, injection and self-identification. The verifier takes as input the program and a set of behavioural properties (LTL formula). It explores all possible paths (branches) at each conditional branch instruction in the program. If malicious behaviour is detected, the execution path of the matched property is returned by the tool. Although their method statically detects a malicious action in a program, its detection capability relies on manually formulated LTL formulas.

The work proposed in [HKV07] uses the control flow graph of a program as a model and defines a set of specifications at the level of assembly instructions. The

creation of malicious code specifications requires assistance from a user using Computational Tree Predicate Logic (CTPL). The user selects relevant instructions in the control flow graph that are of particular relevance for program behaviour. The temporal logic formula uses predicates rather than atomic propositions to represent assembly instructions and to quantify an instruction's parameters. In their model each line of code corresponds to a state model and it is uniquely labelled by a location. The model checker returns a report to the user if the input program satisfies the specification. The model-checking tool is PSPACE-complete and it uses several optimisation techniques to reduce the overhead of verifying the number of procedures and computing the number of variable assignments. Also, the disassembled code of a new malware is manually analysed to locate portions of code that exhibit characteristic malicious behaviour.

In [KKSV10], a malware-detection tool is presented. The tool extends the use of the new expressive branching time logic formula CTPL to check a model of a potentially malicious code against known malware specifications. The authors are able to produce precise specifications as signatures that can match a large class of functionally related worms; for instance, they show that with one single CTPL formula, several variants of worms can be detected without false positives. The set of malicious code specifications in this method cover program calls to the system API, the program stack layout, which can be affected by push and pop instructions, and program register values at a particular point in a program execution path. The model produced for an input program consists of a set of Kripke structures, where each structure represents the control flow of one subroutine. The tool was tested using 21 worm variants from 8 different families. Two malicious specifications were developed in CTPL of two portions of known malware. The specifications were used to verify the input samples. The results showed that the model checker is able to categorise all variants of the worm families. However, similar to previous work on model checking, the process of constructing the specifications (i.e. CTPL formulas) from a fragment of malicious code and the identification and the extraction of characteristic code sequences from a set of worms requires a major intervention by a user.

Beaucamps and Marion in [BM09] proposed an approach that considers malware as a concurrent system interacting with an environment. They extracted malware traces from execution runs as infinite words and created a trace automaton, which then compared these using a database of malicious behaviour. The method allows the detection of similar malicious behaviour in a generic way but the behaviour

patterns they define for detection do not consider data flow information when matching trace elements (e.g. system calls). Also, the approach looks at a single execution run during the detection of malicious behaviour and it does not cover all possible execution paths in the control flow graph of a program. [KKSV05] contains an overview of the model checking-based approach to malware detection.

3.2.5 Semantics-based Malware Detection

Semantics-based malware detection is a new approach, which may overcome the weaknesses of heuristic- or signature-based detection methods by incorporating the semantics of program statements (instructions) rather than the syntactic properties of the code. This section investigates some recent preliminary work that handles code obfuscation in malware detection using the program semantics approach. The preliminary work reviewed in this section shows that the semantics-based approach has the ability to identify the malicious behaviour of a program hidden under the cover of obfuscation and can improve the detection of future unknown malware variants. [PCJD08] contains an overview section of related techniques in semantics-based detection.

Christodorescu and Jha [CJS+05] presented a semantic-based method for detecting malicious programs. That is, the malware detector is aware of common malicious behaviour of malware variants. This work mainly presents two key contributions. First, specifying a set of malicious behaviour using a template in order to match it with a malicious code fragment in a program. Second, using a state and an execution context, which allows the representation of the behaviour of a program and abstracts away both from registers and the names of constants. Christodorescu and Jha show that their semantics-based matching algorithm is resilient to some code obfuscations, for instance, register renaming and code reordering. The algorithm looks at the state of the memory after the check in order to determine if the suspicious program code memory segment matches the template's memory segment. This is implemented using def-use pairs; a match can occur if there is a unique def-use pair both in the template and in the disassembled suspicious code. However, as it requires an exact match between the template node and program instructions, obfuscation attacks using equivalent-instruction replacement and reordering are still possible. Moreover, because some memory segments of a suspicious program are more complex than others, the algorithm uses four different decision functions to determine the state of memory before and after the check.

Their results show that the semantics-aware approach has zero false positives in detecting malware.

Moser et al. [MKK07a] introduced a software testing-based tool for discovering multiple execution paths and generating semantic signatures of malware. Their tool helps to improve test coverage in malware analysis systems. For a given set of inputs, the tool executes a malicious program and monitors its runtime behaviour. The input dependency of the program control flow is examined and different input values are generated to alter the execution along a specific path. Then the tool explores the actions generated from the executed path, and if any malicious action is encountered, the input values along with the path are used as a semantic signature for the malware. For each new execution path taken, a snapshot is created of the current process at control flow decision points. The evaluation shows that the tool was able to successfully identify the behaviour of malware variants from different malware families, but no sound arguments were presented to prove the accuracy of their testing coverage technique.

Preda et al. [PCJD07] proposed a formal semantics-based framework for assessing the resilience to obfuscation of malware detectors. The authors presented an abstract interpretation of trace semantics to characterise malware behaviour and to incorporate it into their semantic malware detector (SMD). Since malware writers use obfuscation to generate metamorphic malware variants, the authors consider obfuscation as a transformation of the trace semantics of a malware program where an obfuscating transformation $O : P \to P$ produces a set of transformations. Moreover, the authors define their notion of semantic trace abstraction so as to allow discarding of the details changed by the obfuscation while preserving the maliciousness of the obfuscated program. The authors prove that SMD is complete and sound with respect to obfuscation O under abstraction α if it detects all obfuscated programs $O(P)$ that are infected by a malware M (no false negatives). Preda et al. have two major classifications of obfuscations. The first class is conservative obfuscation, which covers obfuscation techniques such as code reordering, opaque predicate insertion, semantic NOP insertion and substitution of equivalent commands. The process of dealing with this class of obfuscation is straightforward, the abstraction α_c handles common obfuscations listed above by providing environment-memory traces only of the program. The second class is non-conservative obfuscations, which deals with the variable-renaming obfuscating transformation. Since, with the variable-renaming obfuscation technique the names of register in the malicious code are changed, the environment-memory

traces will look different to the original malware traces. Therefore, a canonical abstraction α_v is introduced to deal with this obfuscation. All register names are replaced or mapped with a set of canonical names before applying the abstraction α_e between M and P traces.

Much effort has been made to improve the detection of obfuscated malware variants using dynamic analysis. Feng and Gupta [FG09] presented a malware detector that uses dynamic signatures to detect obfuscated malware variants. Their dynamic signatures are produced from the runtime behaviour of a known malware program. Each dynamic signature can be extracted from the program runtime trace by taking a backward slice of the program trace with respect to an API call. The detector maintains a database of dynamic signatures that are generated from a set of known malware families. Their evaluation show that the detector can handle a set of obfuscation techniques, including variable renaming, instruction reordering, insertion of NOP instructions, control flow alteration and a limited set of instruction substitutions. However, since the signatures are based on API syntax, the method fails with techniques that alter API operation syntax.

Another semantic-based detection mechanism for defeating packing and code obfuscation techniques in malware variants was introduced by Lee et al. [LJL10]. They consider the static API call sequences as program semantic invariants. Then their method abstracts away program instructions and produces a (code) graph using the API call sequence of a known malware. Code graphs are stored as semantic signatures and used to compare with new generated graphs of suspicious programs. To evaluate the detection tool, 300 obfuscated malware programs were generated using the code insertion, code reordering and code replacement techniques. The results showed that the generated samples were not found by anti-virus detection tools, but their method was able to detect all of the samples. However, the method lacks the ability to capture obfuscated malware variants generated by using equivalent API operations and when introducing meaningless API calls.

3.3 Malware Analysis Techniques

Malware analysis techniques help to improve the process of understanding the functionality and the intent of executable code. Anti-malware product developers

and analysts use a variety of reverse-engineering tools to assist with the analysis process. Reducing the amount of time necessary to understand the overall program layout yields large increases in reverse engineering and detection productivity. Obfuscating a program with packers and other code metamorphic tools makes the analysis more difficult. Several recent research studies have introduced and improved the tools and techniques within malware analysis research. This branch of malware problem research is based on two main classes of analysis tools: static and dynamic tools. Static analysis tools include disassemblers (e.g. IDAPro [Res]) and signature scanners (e.g. anti-virus scanners). Dynamic analysis tools include executable trace and operating system environment monitoring (e.g. virtual machines [Ora, VMw] and Pin [LCM+05]) and debugging tools (e.g. OllyDbg [Yus]). Static analysis tools extract and investigate binary code without executing its instructions. However, dynamic analysis tools can handle more sophisticated malware programs by monitoring and understanding program execution behaviour.

Using a visualisation method for monitoring program execution and exploring the overall flow of a program was suggested by Quist and Liebrock [QL09]. They proposed the VERA architecture, which distills large compiled malware programs (over 1 million lines of code) and produces a high-level overview of the overall flow of basic-block portions of a program.

Recent work by Miwa et al. [MME+07] introduced an isolated sandbox for analysing malware programs. The tool handles anti-virtualisation malware that are capable of detecting analysing environments (i.e. virtual machines) and it avoids any impacts to/from the Internet. Mimetic Internet functionality is implemented to fool detecting mechanisms of malware. The proposed isolated sandbox is a closed experimental environment that executes anti-virtualisation malware. To recover from damage caused by executing malware, the tool rebuilds the executing environment using a clean image from a disk-image.

Rieck et al. [RHW+08] proposed an automatic classification method of malware families based on their dynamic behaviour (input representation). Their method consists of two major machine learning steps. Learning the behaviour of labelled malware samples is the first step. The second step constructs models that are capable of classifying unknown variants of sampled malware families while rejecting the behaviour of benign program files and malware families not considered during the first step. A large number of malware samples were dynamically analysed using a sandbox environment and their behaviour (e.g. API calls) collected. Their model

construction step use the vector space model and bag-of-words model techniques to create behaviour patterns, which are used later as training data. The method uses the well-established technique of Support Vector Machines (SVM) as a classifier, which takes training data of two classes and measures the optimal distance from them. Their results show that the proposed method can correctly classify 70% of malware instances that are not detected by anti-virus software.

Purely dynamic behaviour monitoring and analysis methods for malware programs include FuYong et al.'s MBMAS tool [FDJ09]. They developed an automatic tool to execute malware programs in an isolated environment, and to monitor five aspects of the state of a system: process, file system, registry, port and networks. A report is produced for a human analyst. To analyse malware behaviour more accurately, the tool deploys a filtering mechanism, which identifies process information created by the malware, the tool only collects information related to identified processes.

Roundy and Miller [RM10] developed a hybrid malware analysis method to simplify malware analysis by providing a pre-execution analysis of binary code and a post-execution analysis of malware behaviour. Their method combines static and dynamic techniques to construct and maintain the control- and data-flow analyses that form the interface through which the analyst understands and instruments the code. It provides a dynamic instrumentation feature to identify new instructions that are dynamically generated and hidden by obfuscations. Then the CFG is updated with the captured code and presented to the analyst. The method works well on self-modifying, packed and obfuscated programs.

3.4 Discussion and Conclusion

The detection approaches and the list of surveyed research present the plethora of research in the area of malware detection. There are several observations that can be made about the growing trend in malware detection.

- The traditional signature approach lacks the ability to detect new malware variants due to the syntactic proprieties used (e.g. byte and instruction sequences) in their detection methods. The methods work well for extracting and matching signatures of loosely structured data; nonetheless, for binary

executables, these techniques have high overheads in processing structural data in executable files and they are prone to a high number of false positives. Our work addresses this deficiency by automatically generating compact semantic signatures using a trace-slicing algorithm. Our trace-slicing algorithm improves the detection rate (by helping to increase the accuracy and the performance of matching semantic traces).

- Behaviour detection techniques use very restrictive behavioural specifications (i.e. normal and malicious behaviour) to detect a large number of malware. However, this notion of restriction in detection may lead to a high number of false positives. Most benign programs perform similar operations with a host system (e.g. making system calls to access OS resources). Thus, information about a sequence of events (e.g. API calls) of a program is not enough to determine whether the program is malicious. Also, the complexity of the approach (e.g. the intensive monitoring of OS and program activities against a behavioural specification) makes such detectors undesirable for users to run as it tends to slow down the performance of their computer systems. Also, this approach requires a security expert to specify the behavioural specification (of benign or malware programs), as no automatic method for generating such specifications is implemented. Our research addresses this weakness by computing a set of program inputs (as part of the semantic signature) for discovering multiple program execution paths using an automatic test data generation algorithm.

- An issue with the heuristic-based approach to developing a machine-learning malware detector, is that the detector would need to be trained using many malware programs, which could become infeasible, as well as making the detector 'aware' of specific malware instances rather than a broader generalisation of what a malware family is. Thus, detectors which are purely based on heuristics have a very low resilience against new attacks of metamorphic malware variants. Our approach doesn't consider (nor is it affected by) the syntactical structure of a file as it focuses on the semantics of the code to detect malware variants.

- A novel approach to malware detection using program semantics seems a promising approach to improve current detection tools. Semantics-based techniques are very effective in using low-level properties of programs and

are accurate (few false alarms). This has, therefore, motivated the implementation of our semantic signature to capture new variants of known malware and to improve the detection rate.

- The malware analysis approach is a potential method for achieving a complete understanding of new malicious behaviour. Malware analysis techniques may be used primarily as back-end tools that act as good filtering mechanisms through capturing useful information that security experts need for most malware. A key feature of analysis techniques is that they allow security analysts to be selective about the behaviour monitored and about the granularity of results captured. Most proposed techniques are purely dynamic and expensive as they try to analyse malicious code incorporating anti-static analysis features. However, a combination of static and dynamic analysis might lead to cost-effective and high-performance malware analysis tools. Our work takes a step toward a hybrid (static-dynamic) approach in malware analysis by automating malware classification using semantic simulation of malware code.

- Most malware analysis and detection techniques handle the problem of unpacking/decrypting malware executables quite well. These techniques assume that the unpacking and decryption processes are self-contained within malware programs (i.e. decryption and unpacking routines are part of the code). Our malware-detection system uses unpacking tools to extract malicious payloads.

- Malware detection is similar to other related fields, such as clone detection and dynamic software birthmarking techniques. However, there are important differences. First, clone-detection methods [RC07, LRHK10, PTK11] analyse programs for similarities, at the syntax level or at the structural level, between and within large-scale programs. Thus, the clone-detection approach is not robust to code obfuscation. Dynamic software birthmark techniques identify a software application piracy event by comparing the runtime behaviour of programs [WJZL09a, CHY11]. Two methods for creating dynamic birthmarks have been proposed: a whole program path (WPP)[MC04, ZSSY08] and application programming interface (API) [TONM04, WJZL09b]-based birthmarks. The WPP-based birthmark method extracts the dynamic control flow of a program while the API-based birthmark method uses a sequence of recorded API function calls during the execution of a program

to identify similarities. The WPP-based method is not robust to opaque predicate obfuscation. The API-based method is robust to code obfuscation but it is platform dependent and needs to run the program. Second, to our knowledge, all proposed work on clone-detection and dynamic birthmark techniques analyse source code (i.e. to find similarities between high-level software applications). In contrast, our technique analyses malicious binary executables (i.e. low-level programs) and compares the semantic structures of the code.

Chapter 4

Trace Slicing

In this chapter, we describe the improvements to our semantic signature generation and detection approach for malware variants. We introduce trace slicing for malware executables. We propose a trace-slicing algorithm, which aids the construction of a semantic signature for a known malware variant. From a known malware variant, a set of semantic traces (trace slices) are produced by applying our trace slicing. With trace slicing, we may be able to capture slices of the trace semantics and construct a semantic signature that could improve the detection of (possibly obfuscated) variants of malware. The proposed trace-slicing algorithm works on our abstract machine code language AAPL and is based on computing and updating data dependencies in the trace at simulation time. We use our abstract machine code language, AAPL, which is introduced in Section 4.1, to represent malicious code and understand the effects of code obfuscations on both a program code and its trace, and to apply our trace-slicing method to compute trace slices. Moreover, our trace-slicing algorithm can be used as a trace slicer for executable machine code as we show in the implementation section (Section 4.3.5).

Our conjecture is that the trace-slicing method can improve the detection of malware variants in two ways. First, the trace-slicing algorithm handles some code obfuscating techniques and abstracts away their effects. Second, it computes a set of correct trace slices as short semantic signatures for the malware variant detection algorithm in an attempt to efficiently match against semantic signatures of a known malware family. Thus, the slicing approach may improve the speed and accuracy of the malware detector.

In summary, the contributions of this chapter are fourfold:

1. We identify a suitable target low-level language to describe the syntax and semantics of malware code.
2. We define slicing for the traces that occur in the semantics, and provide a slicing algorithm for the abstract machine code (AAPL).
3. We introduce the correctness property for the trace-slicing algorithm and prove the algorithm is correct with respect to the semantics.
4. We provide a prototype implementation of our algorithm, which works on binary executables and evaluate our algorithm on several real-world binary executables. A version of this implementation is developed for the AAPL code and is part of the signature extraction process in Chapter 5.

To fully appreciate this contribution, it is necessary to understand the context in which we propose to apply the slicing algorithm. The remainder of this chapter is structured as follows. Section 4.1 presents our low-level programming language AAPL. Section 4.2 provides an overview of the slicing approach. Section 4.3 gives the details of our trace-slicing algorithm. The correctness proof of the algorithm is presented in Section 4.4. Section 4.5 highlights the strengths and limitations of the algorithm. Section 4.6 describes related research work in the area of dynamic program slicing and slicing binary executables. Section 4.7 concludes the chapter.

4.1 Programming Language

In this section we introduce our simple Abstract Assembly Programming language (AAPL), which is used for reasoning about code obfuscating transformations in malware program variants. Our main objective is to have an intermediate representation of assembly programs that aids in supporting various program analysis approaches such as generating data dependence graphs (DDGs) (presented in Section 4.3) and control flow graphs (CFGs) (presented in Chapter 6); moreover, this approach allows us to investigate the semantic properties of code independently of the target architecture. This means we can employ source code analysis techniques on low-level code. The aim of AAPL is to significantly improve code analysis while preserving code semantics.

4.1.1 Syntax

Programs written in AAPL consist of a sequence of statements. Every program statement contains a command C and, optionally, a label L. We define program registers to be a finite set of assembly registers that represent a small fixed set of word-sized containers used during program simulation. We define PC as the program counter register to hold the memory address of the next command to be executed and SP as the stack pointer register, which points to a region of memory. Our programming-language semantics are similar to those presented in [CC02] and [PCJD07], except that our language treats memory addresses as unsigned integer numbers $\mathbb{Z}_\perp$ and assumes they hold either integer values or commands.

Figure 4.1 on the next page describes the programming syntax of AAPL. A program P is a set of commands $\wp(\mathbb{C})$. There are two types of command in AAPL, actions and conditional jump commands. An action command C_A may perform the following: evaluating an expression to a register ($R := E$), loading the result of an expression into a memory location pointed to by a register, performing the `SKIP` (i.e. `nop`) operation, or branching to another part of a program using unconditional jump commands. An unconditional jump command may perform jumps based on an expression value, a call by expression value and a return to a memory location specified by the stack pointer SP. A conditional jump command C_B performs a jump to a location specified by the value of expression E when the Boolean expression B evaluates to *true* (e.g. $\hat{\mathbf{B}}[\![B]\!] = true$). In Figure 4.1 on the following page, we let ρ describe the *environment* of program registers, including the program counter, during program simulation. An environment $\rho \in \mathcal{E}$ maps a register to its content value, i.e. $\rho : \mathbf{R} \to \mathbb{Z}_\perp$. Moreover, the *memory* in the language describes the actual contents of the program environment. Program labels L hold the locations of program code in the memory. We let API denote the set of system calls in the AAPL language and $\forall$`api` $\in API$, we have an output component `api`.out, which represents the set of registers that are updated when evaluating a system call. We define a function `sys_env` to map the set of registers $R \in$ `api`.out to their new (pre-defined) values n after evaluating an `API` command.

4.1.2 Semantics

The semantics of the programming language is shown in Figure 4.2 on page 64. The semantics of actions describes how the memory and the environment pair (ρ', m')

$\mathbf{R} ::= \{PC,\ SP,\ r0,\ r1, \ldots\}$	(program registers)
$API ::= \{api_1,\ api_2, \ldots\}$	(system calls)
$E ::= n \mid L \mid R \mid *E \mid E_1 \texttt{ op } E_2$	($\texttt{op} \in \{+, -, \times, /, ...\}$)
$B ::= true \mid false \mid E_1 < E_2 \mid \neg B_1 \mid B_1 \texttt{ \&\& } B_2$	(boolean expressions)
$A ::= R := E \mid *R := E \mid *n := E \mid *(E_1 \texttt{ op } E_2) := E \mid \texttt{API}$	(program actions)
$\mid \texttt{CALL } E \mid \texttt{RTN} \mid \texttt{SKIP} \mid \texttt{JMP } E \mid \texttt{POP E} \mid \texttt{PUSH E}$	
$C ::= C_A := A$	(action command)
$\mid\ C_B := B \texttt{ JMP } E$	(conditional command)
$P ::= \wp(\mathbb{C})$	

$\mathbb{B} = \{true, false\}$	(truth values)
$n \in \mathbb{Z}$	(unsigned integers)
$C \in \mathbb{C}$	(commands)
$\rho \in \mathcal{E} = \mathbf{R} \to \mathbb{Z}_\perp$	(environments)
$m \in \mathcal{M} = \mathbb{Z} \to \mathbb{Z}_\perp \cup \mathbb{C}$	(memory)
$\xi \in \mathcal{X} = \mathcal{E} \times \mathcal{M}$	(execution contexts)
$S = \mathbb{C} \times \mathcal{X}$	(program states)

Figure 4.1: Instruction syntax and value domain of the Abstract Assembly Programming Language (AAPL).

of the next command to be executed in the program is evaluated. The simulation of program $P = \wp(\mathbb{C})$ (i.e. a set of commands) starts by evaluating the initial command of P that is specified by the program counter PC in Figure 4.1. PC is a special register that always points to the memory location of the next command, to be executed, in the program. That is, a sequence of program commands stored in the memory are reachable through simulation via memory locations pointed to by PC. The memory location values are computed during program simulation and assigned to PC. Thus, PC should hold a valid memory address. For instance, when executing a call command, the location of the next command in the program is stored in the stack memory indexed by SP. Also, in the semantics of the return command (`RTN`), the program counter retrieves the location of the next command to be executed from the stack.

The behaviour (i.e, the set of traces) of a program during simulation is described using the set of *execution contexts* $\mathcal{X}$, where $\mathcal{X} = \mathcal{E} \times \mathcal{M}$ is a set of pairs each composed of an environment and a memory of the program being simulated. The

Semantics of Arithmetic Expressions:

$\hat{\mathbf{E}} : \mathbf{E} \times \mathcal{X} \to \mathbb{Z}_{\bot}$

$\hat{\mathbf{E}}[\![n]\!]\xi = n$

$\hat{\mathbf{E}}[\![L]\!]\xi = L$

$\hat{\mathbf{E}}[\![R]\!]\xi = \rho(R)$

$\hat{\mathbf{E}}[\![*E]\!]\xi = \text{if } (\exists n\ \hat{\mathbf{E}}[\![E]\!]\xi\ \in \mathbb{Z}) \text{ then } m(n);\ \text{else } \bot$

$\hat{\mathbf{E}}[\![E_1 \texttt{ op } E_2]\!]\xi = \text{if } (\hat{\mathbf{E}}[\![E_1]\!]\xi \in \mathbb{Z} \text{ and } \hat{\mathbf{E}}[\![E_2]\!] \in \mathbb{Z}) \text{ then } \hat{\mathbf{E}}[\![E_1]\!]\xi \texttt{ op } \hat{\mathbf{E}}[\![E_2]\!]\xi;\ \text{else } \bot$

Semantics of Actions:

$\hat{\mathbf{A}} : \mathbf{A} \times \mathcal{X} \to \mathcal{X}$

$\hat{\mathbf{A}}[\![\texttt{SKIP}]\!]\xi = \xi$ where $\xi = (\rho, m)$

$\hat{\mathbf{A}}[\![R := E]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(R \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![*R := E]\!]\xi = (\rho, m')$ where $\xi = (\rho, m)$ and $m' = m(\rho(R) \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![*n := E]\!]\xi = (\rho, m')$ where $\xi = (\rho, m)$ and $m' = m(n \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![*(E_1 \texttt{ op } E_2) := E]\!]\xi = (\rho, m')$ where $\xi = (\rho, m)$ and $m' = m(\hat{\mathbf{E}}[\![E_1 \texttt{ op } E_2]\!]\xi \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![\texttt{JMP E}]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![\texttt{CALL E}]\!]\xi = (\rho', m')$ where $\xi = (\rho, m),\ \rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi, SP \mapsto SP - 1)$ and $m' = m(\rho(SP - 1) \mapsto \rho(PC + 1))$

$\hat{\mathbf{A}}[\![\texttt{RTN}]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(PC \mapsto m(\rho(SP)), SP \mapsto SP + 1)$

$\hat{\mathbf{A}}[\![\texttt{PUSH E}]\!]\xi = (\rho', m')$ where $\xi = (\rho, m),\ \rho' = \rho(SP \mapsto SP - 1)$ and $m' = m(\rho(SP - 1) \mapsto \hat{\mathbf{E}}[\![E]\!]\xi)$

$\hat{\mathbf{A}}[\![\texttt{POP E}]\!]\xi = (\rho', m')$ where $\xi = (\rho, m)$, and

$$(\rho', m') = \begin{cases} \rho' = \rho(SP \mapsto SP + 1), m' = m(\hat{\mathbf{E}}[\![E]\!]\xi \mapsto m(\rho(SP))) & \text{if } E := *E \\ \rho' = \rho(SP \mapsto SP + 1, \hat{\mathbf{E}}[\![E]\!]\xi \mapsto m(\rho(SP))),\ m' = m & \text{otherwise} \end{cases}$$

$\hat{\mathbf{A}}[\![\texttt{API}]\!]\xi = (\rho', m)$ where $\xi = (\rho, m)$ and $\rho' = \rho(R \mapsto n),\ \forall R \in API.out$ and $sys_env : API \to n$

Semantics of Commands:

$\hat{\mathbf{C}} : S \to \Sigma(S)$ (determines transition relation between program states)

$\hat{\mathbf{C}}[\![C_A]\!]\xi = (\xi', C')$ where $\xi = (\rho, m),\ \xi' = \hat{\mathbf{A}}[\![A]\!]\xi$ and

$$C' = \begin{cases} m'(\rho'(PC)) & \text{if } A := \texttt{JMP} \cup \texttt{CALL} \cup \texttt{RTN} \\ m(\rho(PC + 1)) & \text{otherwise} \end{cases}$$

$\hat{\mathbf{C}}[\![C_B]\!]\xi = (\xi', C')$ where $\xi = (\rho, m)$, and

$$(\xi', C') = \begin{cases} \xi' = (\rho', m), \rho' = \rho(PC \mapsto \hat{\mathbf{E}}[\![E]\!]\xi), \text{and } C' = m(\rho(\hat{\mathbf{E}}[\![E]\!]\xi)) & \text{if } \hat{\mathbf{B}}[\![B]\!]\xi = \text{true} \\ \xi' = \xi, C' = m(\rho(PC + 1)) & \text{otherwise} \end{cases}$$

Figure 4.2: Semantics of AAPL.

simulation trace of a program is a sequence of program states representing the evaluation of the binary executable's environment (i.e. registers and memory) during the simulation of the instructions in a program run. More precisely, in AAPL, we refer to the program execution state and execution trace as the program state and simulation trace, respectively.

Definition 4.1 (Program State). The program state $s \in S$ is a pair $s = (C, \xi)$

P :

```
1        r0:=n
2        r1:=w
3 Loop:  (r0 >= 3) JMP Exit
4        *r1:=*r0+4
5        r0:=r0+1
6        r1:=r1+1
7        JMP Loop
8 Exit:  JMP End
```

(a)

$t_{\mathtt{x}}$:

s_0		r0:=1,	(ρ_{s_0}, m_{s_0})
s_1		r1:=2,	$(\rho_{s_1}(\mathtt{r0}\mapsto 1), m_{s_1})$
s_2	Loop:	(r0 >= 3) JMP Exit,	$(\rho_{s_2}(\mathtt{r1}\mapsto 2), m_{s_2})$
s_3		*r1:=*r0+4,	(ρ_{s_3}, m_{s_3})
s_4		r0:=r0+1,	$(\rho_{s_4}, m_{s_4}(2 \mapsto (m_{s_4}(1) + 4)))$
s_5		r1:=r1+1,	$(\rho_{s_5}(\mathtt{r0}\mapsto 2), m_{s_5})$
s_6		JMP Loop,	$(\rho_{s_6}(\mathtt{r1}\mapsto 3), m_{s_6})$
s_7	Loop:	(r0 >= 3) JMP Exit,	(ρ_{s_7}, m_{s_7})
s_8		*r1:=*r0+4,	(ρ_{s_8}, m_{s_8})
s_9		r0:=r0+1,	$(\rho_{s_9}, m_{s_9}(3 \mapsto (m_{s_8}(2) + 4)))$
s_{10}		r1:=r1+1,	$(\rho_{s_{10}}(\mathtt{r0}\mapsto 3), m_{s_{10}})$
s_{11}		JMP Loop,	$(\rho_{s_{11}}(\mathtt{r1}\mapsto 4), m_{s_{11}})$
s_{12}	Loop:	(r0 >= 3) JMP Exit,	$(\rho_{s_{12}}, m_{s_{12}})$
s_{13}	Exit:	JMP End,	$(\rho_{s_{13}}, m_{s_{13}})$

(b)

Figure 4.3: A sample program in AAPL and its simulation trace. A sample program P; A trace on program input $\mathtt{x}$: $n = 1, w = 2$

composed of the next command C to be evaluated in the execution context ξ. The set of states, denoted by $S = (\mathbb{C} \times \mathcal{X})$, describes both the program command and the execution context of the program in each state.

Given a state $s \in S$, the transition function $\hat{\mathsf{C}}(s)$ provides the set of possible successor states of s, that is, $\hat{\mathsf{C}} : S \rightarrow \wp(S)$ specifies the transition relation between states. We let $S[\![P]\!]$ be the set of states of a program P, then we have $S[\![P]\!] = \mathcal{X}[\![P]\!] \times P$. The transitional semantics $\hat{\mathsf{C}}[\![P]\!] \in S[\![P]\!] \rightarrow \wp(S[\![P]\!])$ of a program P is defined as: $\hat{\mathsf{C}}[\![P]\!](c, \xi) = \{(c', \xi') \in \hat{\mathsf{C}}((c, \xi)) \mid c' \in P,\ \xi' \in \mathcal{X}[\![P]\!]\}$.

Definition 4.2 (Simulation Trace). A trace $t_x \in S^*$, where S^* is the set of finite sequences of states over S, consists of a sequence of states $< s_0, \ldots, s_n >$ of length $|t_{\mathtt{x}}| \geq 1$ that has been produced by simulating the program P with a given program input `x` at initial state s_0 (i.e. `x` is the initial execution context ξ_0 in which the first command in P is to be evaluated with) such that for all i, $1 \leq i \leq n : s_i \in \hat{\mathbf{C}}(s_{i-1})$.

That is, for a given state $s = (C_s, \xi_s)$, $\hat{\mathbf{C}}(s)$ provides the next program state s' by evaluating the program command C_s with the current execution context ξ_s at the state s [CC02]. For instance, for the jump command, `JMP` E, the arithmetic expression E in the current command must be evaluated and the result is assigned to the program counter PC that represents the location of the next program command (i.e. $C' = m(\rho(PC))$). Figure 4.3 on the previous page shows a fragment of a malware routine written in AAPL and its single simulation trace $t_{\mathtt{x}}$, which represents the program syntax, environment and memory evolution. Note that $t_{\mathtt{x}}$ begins with the initial state $s_0 = (C_{s_0}, \xi_{s_0})$, where $\xi_{s_0} = (\rho_{s_0}(n \mapsto 1, w \mapsto 2), m_{s_0})$, and ends with the final state $s_{13} = (C_{s_{13}}, \xi_{s_{13}})$.

4.2 Overview of Slicing Approaches

Since code obfuscation is used in most new malware instances, collected traces from malicious code simulation may contain some elements of code mutation, such as irrelevant code instructions. Therefore, it is less effective to construct a semantic signature of a malware family from an entire trace of the program simulation when trying to detect obfuscated variants of the same malware. A possible solution to this problem is to incorporate a dynamic *slicing* approach when constructing a semantic signature from a simulation trace. Dynamic slicing, introduced by Korel and Laski [KL88], has been available for almost two decades. The idea of this approach is that program dependencies that are exercised during a program run are captured precisely and collected in the form of a program dynamic dependence graph. Then, dynamic program slices are produced by traversing the dynamic dependence graph. Since the objective of dynamic slicing is to determine a relevant subset of executed program statements that can potentially contribute to the computation of the value of a variable during a program run, short and precise slices of program traces are desired for producing semantic signatures in malware detection. Although many different techniques have been introduced in various dynamic

slicing algorithms [KY94, BGS+01, Far02, ZGZ03, MM06, HMK06, WR07], there are two main challenges to adopting these algorithms for our case. First, only limited efforts have been made in developing a formal guarantee for the correctness of existing dynamic slicing algorithms [MM06]. Second, conventional algorithms produce dynamic slices that are usually a subset of the original program code; however, when producing semantic signatures for malware, our interest lies in producing trace slices that are a *subtrace* of the simulation trace and not an executable sub program of the machine code. Therefore, an efficient and provably correct *trace*-slicing algorithm for executables that includes dynamic program slicing [KL88, AH90, ADS91b] is required.

4.3 The Trace-Slicing Algorithm

Given a program input $\mathtt{x}$ (i.e. initial execution context) and a program P, a simulation trace $t_{\mathtt{x}}$ is generated by simulating P with the input $\mathtt{x}$. The simulation trace $t_{\mathtt{x}}$ of a program captures the semantic information of the program instructions' evaluations, which can later be used by our slicing algorithm. The information that the trace holds consists of both the command (*the syntax*) trace and the execution context reference (*the semantics*) trace. For example, $t_{\mathtt{x}} = < s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13} >$ is a program simulation trace when the program in Figure 4.3 on page 65 is simulated on the input data (the initial execution context at s_0) $n = 1, m = 2$. A simulation trace is a sequence of program states. Notationally, each program state in a trace is subscripted with its position. We let POS_t denote the set of positions of program states S in a trace t. Also, in order to map a particular position (i.e. a state index) to a state in t and vice versa, we define two auxiliary functions: $\mathit{State} : \mathit{POS}_t \rightarrow S$ and $\mathit{Index} : S \rightarrow \mathit{POS}_t$. The function *State* allows one to produce the program state which POS_t refers to. The function *Index* provides the position index POS_t of a state in the trace. Also, we define functions to extract information from states, i.e. we let $\mathit{Command} : \mathit{POS}_t \rightarrow C$ and $\mathit{Context} : \mathit{POS}_t \rightarrow \mathcal{X}$ denote the mapping from a position index in the simulation trace t to the command and the execution context, respectively, if they exist in that particular state. Our trace-slicing method differs from traditional slicing algorithms found in the literature [AH90, KL88, MM06], where the control flow graph of the program is *statically* analysed. In other dynamic slicing methods [MMS02, MMS03], the program dependence graph (PDG) is constructed as an intermediate program representation

to compute dynamic slices. However, the algorithm that we present, *Trace Slicing*, does not require the computation of either control dependencies or the PDG. Instead, our trace-slicing algorithm involves the following:

- *on-the-fly* computation of data dependence edges from the trace, constructing a *dynamic data dependency* graph (DDG),
- performing the *backward slice* for a given trace-slicing criterion.

Therefore, the trace slice, which is computed from the program simulation trace, is the transitive closure of data dependencies in the *DDG* relevant to the trace-slicing criterion. The details of our trace-slicing algorithm are presented in the following subsections.

4.3.1 Capturing Memory References and Assignments

The possible presence of indirect memory accesses (dereferencing) causes complex data-dependence relationships between states in a simulation trace. In the literature, many studies have addressed static program slicing for high-level languages (e.g. C) in the presence of array or pointer variables [CWZ90, HPR89, OSH01, LB97]. Also, in the area of dynamic program slicing, the approach of Agrawal et al. uses a dynamic dependence graph to resolve *use* and *def* sets in terms of memory cells and composite variables, and to detect inter-statement dependencies in C [ADS91a]. The possibility of accessing memory locations using several methods (i.e. aliasing), in low-level languages such AAPL, makes the computation of variable definition and use in code statements more difficult. For example, unlike high-level programming languages, in AAPL a memory location may be accessed at a given statement using syntactically different indirect access methods, e.g. dereferencing a memory location via an immediate offset, a register or one or two registers with an offset.

Therefore, syntactic information is not sufficient because of these methods, and the set of memory locations that can be accessed through a reference must be determined before the computation of definition-use associations. Moreover, because an assignment or use through the dereference of a memory location can potentially use the value of one or two registers, or assign a value to, or use the value of, a memory location, these memory assignments and uses must be treated differently

from direct (i.e. syntactic) assignments. A reference to $*r$ (the syntax of AAPL shown in Figure 4.2 on page 64.) is a reference to a register r and a possible reference to any memory location that r might point to. To capture such effects, we adopt the technique in [LB97] to develop two cases where memory reference and memory assignment can be determined for each program state in a trace:

Memory Reference (use)

Suppose we have a statement at position i in a trace such as:

$$i : r0 = *D + n$$

where n is a constant and $*D$ is an operand dereferencing to a memory location in a program state indexed by i, i.e. $m_{s_i}(D)$ where $State(i) = s_i = (\rho_{s_i}, m_{s_i})$. Let $ruse(D)$ be a function that returns the set of registers that are used in expression D to compute the memory location value. To compute the references at i, we take a union of the memory address computed at runtime $m(D)$ and the set of registers used in D, $ruse(D)$, that is:

$$use(i) = m_{s_i}(D) \cup ruse(D)$$

Algorithm 4.1 on page 72 shows the pseudocode for computing the set of registers and memory addresses used (the registers or memory values are used) in expression *exp* at position i in a simulation trace of an AAPL program. The algorithm has one routine `find_use`, which accepts two input parameters: i and *exp*. In order to evaluate the expression *exp* at trace position i and compute the set of variables that are used at i, six cases are implemented in the routine. The first two `if` conditions (Lines 7 and 10) handle the base cases where the expression is either a register or a memory address (a dereferenced expression). In the first base case the routine returns the register as the used variable in the expression. For the second case, the routine returns the computed memory address and the set of registers that are used for computing this memory address in the dereferenced expression $*E$. The other cases in the algorithm (Lines 13, 16, 22 and 25) analyse operation, assignment, branch and boolean expressions using recursive calls. Note that for an assignment expression, the left-hand expression LE is only evaluated if it is a dereferenced operand $*E$ where the routine $ruse(E)$ (in line 18) is called to extract the set of registers used in E. Finally, the algorithm returns the list U of

the registers and memory addresses that are used at *exp*. Note that Algorithm 4.5 on page 80 (introduced in Section4.3.3) uses the routine `find_use` for definition-use analysis.

Memory Assignment (define)

Suppose we have a statement at position i in a trace such as:

$$i : *D := r0 + n$$

This statement computes a value by adding the value of $r0$ to the constant n. The result is stored at a memory address specified by D. Thus, the definition in the statement at i is the memory address computed when the assignment is executed:

$$def(i) = m_{s_i}(D)$$

Algorithm 4.2 on page 73 shows the pseudocode for computing the register or the memory address defined (the register or memory location value is assigned a value) in expression *exp* at position i in a simulation trace of an AAPL program. The function in Algorithm 4.2 on page 73 accepts two input parameters, i and *exp*, and returns the program variable (i.e. a register or a memory address) that is defined for the given expression *exp*. The function accepts an assignment expression (in line 4) that consists of a left-hand expression, LE, and a right-hand expression, RE. Then according to the AAPL syntax in Figure 4.1, two cases are implemented for the defined program variable, where LE is either a register R or a memory address $m_{s)i}(E)$. Note that for a memory address, the function returns the computed memory address dereferenced by E at the current state s_i. Both Algorithms 4.1 on page 72 and 4.2 on page 73 will be used to compute the definition-use associations between states in Algorithm 4.5 on page 80.

Example 4.1. *The sets of memory addresses and registers referenced (their values are used) and defined at each program state in the simulation trace in Figure 4.3*

are as follow:

t_x	*def*	*use*
s_0	$\{r0\}$	ϕ
s_1	$\{r1\}$	ϕ
s_2	ϕ	$\{r0\}$
s_3	$\{m_{s_3}(r1)\}$	$\{r0, r1, m_{s_3}(r0)\}$
s_4	$\{r0\}$	$\{r0\}$
s_5	$\{r1\}$	$\{r1\}$
s_6	ϕ	ϕ
s_7	ϕ	$\{r0\}$
s_8	$\{m_{s_8}(r1)\}$	$\{r0, r1, m_{s_8}(r0)\}$
s_9	$\{r0\}$	$\{r0\}$
s_{10}	$\{r1\}$	$\{r1\}$
s_{11}	ϕ	ϕ
s_{12}	ϕ	$\{r0\}$
s_{13}	ϕ	ϕ

4.3.2 Preliminary

We present a few definitions that are necessary for our trace-slicing algorithm (TSAlgo). In these definitions, and throughout the rest of the chapter, we use the term *state* nodes to denote program states in a trace. Since our proposed trace-slicing method could be applied to a program trace that may have been generated by the simulation of the program, we use *simulation trace* and *execution trace* interchangeably. Also, AAPL uses registers, r (i.e. the environment $\rho(r)$) and direct memory locations (i.e. addressing memory locations with an immediate offset, a register, or a register with an offset) for data manipulations during program simulation, such as retrieving and storing data from memory.

We use the term *data manipulator* to denote registers and memory locations that are used to process the program data.

Definition 4.3 (Data Manipulators)**.** In AAPL, a data manipulator dm is a program register or memory location used to perform data definition and manipulation operations. The value of dm is described as either the environment value, $\rho(dm)$ in the case of a register, or the memory value $m(dm)$ in the case of a memory location, as described by the semantics of AAPL shown in Figure 4.2 on page 64.

Algorithm 4.1: `find_use(`i`,`exp`)` finds references in exp at position i in a simulation trace.

1: **Input:** A state node index i and
2: exp_i is an expression at i
3: **Output:** The set of registers and memory addresses that are used at position i
4: **begin** `find_use(`i`,`exp`)`
5: initialise the set to empty:
6: $U_i \rightarrow \{\emptyset\}$
7: **if** exp is register **then**
8: **return** $U \rightarrow \{exp\}$
9: **end if**
10: **if** exp is $*E$ (memory address) **then**
11: **return** $U \rightarrow m_{s_i}(E) \cup ruse(E)$
12: **end if**
13: **if** exp is E_1 *op* E_2 **then**
14: **return** $U \rightarrow$ `find_use(`i`,`E_1`)` $\cup$ `find_use(`i`,`E_2`)`
15: **end if**
16: **if** exp is $LE := RE$ (assignment) **then**
17: **if** LE is $*E$ **then**
18: $U \rightarrow ruse(E)$
19: **end if**
20: $U \rightarrow U \cup$ `find_use(`i`,`RE`)`
21: **end if**
22: **if** exp is `JMP` E or `CALL` E (unconditional jump) **then**
23: $U \rightarrow$ `find_use(`i`,`E`)`
24: **end if**
25: **if** exp is E_1 *bop* E_2 `JMP` E_3 (conditional jump) **then**
26: $U \rightarrow$ `find_use(`i`,`E_1`)` $\cup$ `find_use(`i`,`E_2`)` $\cup$ `find_use(`i`,`E_3`)`
27: **end if**
28: **return** U
29: **end** `find_use(`i`,`exp`)`

During program simulation a data manipulator can be defined or used at any point via a state node (e.g. assignment or memory update operations). We define an auxiliary function $man[\![t]\!] \stackrel{\text{def}}{=} \{$data manipulators occurring in $t\}$ to provide the set of data manipulators that are defined and used in a simulation trace. For instance, the set of data manipulators that occur in $t_{\mathtt{x}}$ in Figure 4.3 on page 65 is $man[\![t_{\mathtt{x}}]\!] = \{r0, r1, m_{s_3}(r0), m_{s_3}(r1), m_{s_8}(r0), m_{s_8}(r1)\}$. In order to capture data dependency information in a simulation trace, the following definitions are introduced.

Definition 4.4 (Definition Position *def*(p))**.** Let *def*(p) be the set of data manipulators whose values are defined at position p in a simulation trace t.

Definition 4.5 (Use Position $use(p)$)**.** Let $use(p)$ be the set of data manipulators whose values are used at position p in a simulation trace t.

Algorithm 4.2: `find_def`(i,exp) finds definitions at position i in a simulation trace.

1: **Input:** A state node index i and an expression exp at i
2: **Output:** a register or a memory address that is defined in exp at position i
3: **begin** `find_def`(i,exp)
4: **if** exp is $LE := RE$ (assignment) **then**
5:   **if** LE is register **then**
6:     **return** LE
7:   **end if**
8:   **if** $Lexp$ is $*E$ (memory address) **then**
9:     **return** $m_{s_i}(E)$
10:   **end if**
11: **end if**
12: **end** `find_def`(i,exp)

Definition 4.6 (Def-clear Path)**.** Given a simulation trace t, and a data manipulator $dm \in man[\![t]\!]$, $\forall i,k \in POS_t$ and $i < k$. The path $< i,...,k >$ is a Def-clear path w.r.t dm iff $\forall j \in< i,...,k >$, $dm \notin def(j)$.

Definition 4.7 (Recent Definition Position $dp_i(dm)$)**.** For a simulation trace t, let $i \in POS_t$ and dm be a data manipulator $dm \in man[\![t]\!]$. The function $dp_i(dm)$ computes the position of the most recent data definition of dm with respect to any given point, i, in t. $dp_i(dm) = k$ iff $\exists < k,...,i >$, $dm \in def(k)$ and $< k+1,...,i >$ is a Def-clear path or $k = 0$ (no definition exists for dm).

The most recent definition of a data manipulator dm can be computed as a program is simulated (i.e. during the simulation of the program). $dp_i(dm)$ allows one to keep track of positions of state nodes that define dm from any given index i in a trace. For instance, consider the trace $t_{\mathtt{x}}$ in Figure 4.3. The recent definition position of the data manipulator $r0 \in man[\![t]\!]$ from state s_{12} is $dp_{12}(r0) = 9$ as $r0$ is defined at position 9 (i.e. $r0 \in def(9)$ in Example 4.1 on page 70), and there has been no subsequent definition up to position 12.

Definition 4.8 (Dynamic Data Dependence $s_i \overset{ddd}{\to} s_j$)**.** In a simulation trace t, let $i,j \in POS$ where $i < j$ and $s_i, s_j \in t$. s_j is (directly) data dependent on s_i iff there exists a data manipulator $dm \in DM$ in t such that:

- when dm is a memory location dereferenced by operation $*(A)$, i.e. $dm = m_j(A)$, then:
  1. $m_j(A) \cup ruse(A) \cap use(j) = \phi$, and

2. $dp_j(m_j(A)) = i$ or $\exists r \in ruse(A)$, $dp_j(r) = i$

- when dm is a register, i.e. $dm = r$, then:

1. $r \in use(j)$, and
2. $dp_j(r) = i$

Example 4.2. *Consider the trace t_x in Figure 4.3; some data dependencies between states can be established using Definition 4.8 as follows:*

- *s_8 is data dependent on s_3, $s_3 \overset{ddd}{\rightarrow} s_8$, because the data manipulator $m_{s_8}(r0)$ at index 8 in t_x is $m_{s_8}(r0) \in use(8)$ and $dp_8(m_{s_8}(18)) = 3$; note that at state s_3 the memory location $m_{s_3}(r1) = m_{s_3}(2) = m(2)$ is defined and then it has been used at state s_8, i.e. $m_{s_8}(r0) = m_{s_8}(2) = m(2)$.*
- *s_3 is data dependent on s_0, $s_0 \overset{ddd}{\rightarrow} s_3$, since $r0 \in ruse(m_{s_3}(r0))$, $r0 \in use(3)$ and $dp_3(r0) = 0$.*
- *s_7 is data dependent on s_4, $s_4 \overset{ddd}{\rightarrow} s_7$, as register $r0 \in use(7)$ and $dp_7(r0) = 4$.*

Note that the second condition in Definition 4.8 ensures that a data manipulator is not redefined after position i in the trace. Due to Definition 4.7, the most recent definitions of program data manipulators are captured during the program's simulation.

During the simulation of a program with program input, data dependence edges and a *dynamic* data dependence graph can be constructed from the trace and information gathered (e.g. the recent definition positions of the data manipulators).

Definition 4.9 (Data Dependence Edge)**.** A data dependence edge is an ordered pair of positions of program states in a trace. A directed edge de is constructed between a pair of positions of state nodes in a trace, s.t. $de = (j, i)$, iff $s_i \overset{ddd}{\rightarrow} s_j$.

Definition 4.10 (Data Dependence Graph (*DDG*))**.** A data dependence graph is a set of data dependence edges that represents the data dependencies between state nodes in a program execution trace.

Since we are interested in slicing simulation traces of assembly code, the definitions below capture the notion of the trace slice.

Definition 4.11 (Trace-Slicing Criterion (tsc))**.** Let t be a simulation trace of an AAPL program P simulated on input x. A trace-slicing criterion is a pair, $tsc = (dm, k)$, where $dm \in man[\![t]\!]$ is a program data manipulator, and $k \in POS_t$ is a state node position in the simulation trace t.

Definition 4.12. DDG_{tsc} is the set of data dependence edges obtained from DDG by computing the backward reachability in DDG from the position specified by $dp_k(dm)$ for dm in the trace-slicing criterion tsc.

Definition 4.13 (Trace Slice)**.** A trace slice of an AAPL program simulation trace t is a trace t' that is a projection of t relevant to the value of the slicing criterion of dm. That is, t' is t less any state nodes not in DDG_{tsc}.

A desired property of a trace slice is that it preserves the effect of the original program trace on the data manipulator chosen at the selected point of interest within the trace. Although any static data slice of a program can be computed by pure static analysis, the computation of a trace slice requires evaluation information. The evaluation information is generated as the program is simulated with a given program input. This information provides the control flow path the program follows (while it is under the simulation) to reach the specific state of the program command in the slicing criterion. Definition 4.13 captures the set of all reachable program states from position k in the trace t that directly or indirectly affect a slicing-criterion data manipulator dm in tsc. Thus, the trace slice preserves the program's behaviour with respect to dm and removes any irrelevant state nodes from t, producing a shorter trace slice (a proof of this property is presented in Section 4.4). This definition will be applied by our algorithm.

4.3.3 Overview of the Trace-Slicing Algorithm

Dynamic slicing algorithms typically first carry out all the static computation of the control dependencies and then construct the dynamic program dependence graph (DPDG) to calculate the slice. The generated slices are program statements that may be a subset of the original program [KL88, AH90]. Our goal is to slice the simulation traces of a program under inspection and to generate a trace slice that is a subsequence of the original trace. Given a simulation trace t (produced by the semantic simulator) and for all data manipulators $man[\![t]\!]$, the semantic trace-based malware-detection system in Chapter 5 makes several calls

to the trace-slicing algorithm to compute trace slices. That is, in each slicing call, a criterion $tsc = (dm, k)$ is passed to the slicing algorithm where $dm \in \text{man}[\![t]\!]$. Then the set of computed trace slices can be used to construct semantic signatures for detecting malware variants. Moreover, we observe that the trace captures the full control flow and data manipulation information of the program's simulation for a given input. Therefore, a program trace abstracts away the effect of control dependencies and contains the complete path followed during the simulation in which the value of the tsc data manipulator is computed. For this reason, we propose a precise trace-slicing algorithm that does not perform any static evaluation of control dependencies or construct a PDG. We refer to our algorithm as the *trace-slicing* algorithm (TSAlgo).

TSAlgo has two main steps (procedures) when producing a trace slice:

1. The DDG computation (procedure `find_data_dep_edge` in Algorithm 4.3 on page 78). This step is performed once and the *DDG* can be used for all slicing calls. Also, the step can be accomplished either during (on-line) or after (off-line) the program simulation. In our system (Chapter 5), the semantic simulator retrieves the simulation trace of a program and the *DDG* information is computed once and the DDG is then available for all slicing calls.

2. The trace-slice computation (procedure `compute_slice` in Algorithm 4.4 on page 78). Once the *DDG* is constructed and a call is made by our detection system, a trace slice is computed from the simulation trace using the *DDG* information with respect to a given slicing criterion (Definitions 4.12 and 4.13).

The objective of the procedure `find_data_dep_edge` in Algorithm 4.3 on page 78 is to establish dynamic data dependence edges between a given state and other states in a given trace $t_{\mathtt{x}}$. The algorithm accepts two parameters as input: a simulation trace $t_{\mathtt{x}}$ and a state index j for which the algorithm computes dependence edges. The algorithm starts, in line 9, with a `for` loop, which extracts the set of data manipulators that are used in the current position j; it then determines the positions of the previous states where the data manipulators are defined. Finally, the algorithm creates a dependence edge, between each state and j, and adds the edge to the set of dynamic dependence edges *DDG* in lines 11 and 12.

The objective of the procedure `compute_slice` in Algorithm 4.4 on the next page is to compute the backward trace slice t' for a given simulation trace $t_{\mathtt{x}}$ with respect to a trace slice *tsc*. The algorithm uses the *DDG* of the simulation trace, which is produced by applying Algorithm 4.3 on the following page. The procedure `compute_slice` sets all states in the trace as not marked and not visited (line 5) and starts processing the *DDG* to find outgoing dependence edges from the slicing criterion position k in $t_{\mathtt{x}}$. The backward traversal in the trace and the creation of the slice are implemented as a `while` loop (lines 7 to 13). For each visited state (through an outgoing data dependence edge) the procedure adds the state to the slice and marks all states (to be visited later) that can be reached from the current state. Finally, in line 14, the procedure adds the state that preserves the final value of the slicing criterion data manipulator in the slice.

TSAlgo employs the dynamic definition-update analysis of the data manipulators to recover dynamic data dependencies between program states. The algorithm (in Algorithm 4.5 on page 80, step 1, line 7) performs a dynamic definition-update of all data manipulators in each state node using the $dp()$ function (Definition 4.7). This allows the algorithm to compute new data dependencies in a dynamic fashion from the trace and then to construct the *DDG*.

The construction of a *DDG* with dynamic definition-update analysis of data manipulators allows the production of a precise trace slice for any data manipulator at any state node position in the trace. For example, if we need a dynamic data slice for a value of a data manipulator dm at position p in the simulation trace, we begin traversing the computed *DDG* from the definition position of dm, which is recovered from the definition-update analysis (i.e. $dp_p(dm)$). Thus, the algorithm needs to traverse the simulation trace only once to compute data dependencies during the computation of any trace slice.

In essence, this algorithm produces a short trace slice that consists of only those program states in the trace $t_{\mathtt{x}}$ that contribute to the computation of the value of the slicing criterion.

4.3.4 Description of the Trace-Slicing Algorithm

The trace-slicing algorithm processes simulation traces of an assembly-level program. For a given trace, it analyses the program state by state and generates the *DDG*. Then a slice is computed with respect to the slicing criterion.

Algorithm 4.3: The construction of *DDG* in TSAlgo.

1: **Input:** a simulation trace $t_{\mathtt{x}}$ and an index j of a state node in the trace
2: **Output:** a set of dynamic dependence edges *DDG* for $t_{\mathtt{x}}$
3: **procedure** `find_use` in Algorithm 4.1 on page 72.
4: **procedure** `use(`i`)`
5: find all uses in command c_i at location i:
6: **return** `find_use(`i`,`c_i`)`
7: **end procedure**
8: **begin** `find_data_dep_edge(`j`)`
9: **for all** $dm \in$ `use(`j`)` **do**
10:    **if** $\exists z = dp_j(dm),\ z \in POS_t$ s.t. $z < j$ **then**
11:       create dynamic data dependence edge $de = (j, z)$
12:       $DDG \rightarrow DDG \cup \{de\}$
13:    **end if**
14: **end for**
15: **end** `find_data_dep_edge(`j`)`

Algorithm 4.4: The computation of the slice in TSAlgo.

1: **Input:** a simulation trace $t_{\mathtt{x}}$, a set of dynamic dependence edges *DDG* and a trace slicing criterion $tsc = (dm, k)$
2: **Output:** a trace slice t' for $t_{\mathtt{x}}$
3: **begin** `compute_slice()`
4: $t' \rightarrow \emptyset$
5: Set all state nodes in $t_{\mathtt{x}}$ as not marked and not visited
6: Set $s_{dp_k(dm)} \in t_{\mathtt{x}}$ as a marked and not visited state
7: **while** there exists a marked and not visited state in $t_{\mathtt{x}}$ **do**
8:    Select marked and not visited state $s_q \in t_{\mathtt{x}}$
9:    Set s_q as visited in $t_{\mathtt{x}}$ and $t' \rightarrow t' \cup \{s_q\}$
10:    **for all** outgoing dep. edges from s_q to some state s_i in *DDG* s.t. $de = (q, i)$ **do**
11:       find and mark $s_i \in t_{\mathtt{x}}$
12:    **end for**
13: **end while**
14: Include the state $s_{dp_k(dm)+1}$ that contains the final value of slicing criterion dm
15: $t' \rightarrow t' \cup \{s_{dp_k(dm)+1}\}$
16: **end** `compute_slice()`

During the simulation (in Chapter 5) of the program with program input `x`, the execution contexts and commands are stored as state nodes in a simulation trace $t_{\mathtt{x}}$. As the simulation trace is captured, TSAlgo uses only data dependence relations that are established between state nodes in the trace for identifying a trace slice. The data dependence associated with each command in a state is determined when a state node of the command is processed in the *DDG* construction step. New dependence edges between state nodes, are only established when the associated dynamic data dependence exists. That is, as dependence edges are established,

the *DDG* for a particular $t_{\mathtt{x}}$ is created. Let us assume that a dynamic outgoing dependence edge, *de*, is established from a state node s_j at position j with the existing state node s_i in the trace $t_{\mathtt{x}}$ (i.e. $i < j$). Then the updated *DDG* after identifying *de* is $DDG \rightarrow DDG \cup \{de\}$, where $de = (j, i)$. After constructing the *DDG* for the trace $t_{\mathtt{x}}$, our algorithm (in step 2, line 14) computes the *backward* reachable subgraph with respect to any given *tsc*, and all state nodes that appear in the reachable subgraph are contained in the trace slice. That is, the trace slice is computed by traversing only the relevant dynamic dependence edges in the *DDG*.

Algorithm 4.5 on the next page shows the pseudocode for the trace-slicing algorithm. It constructs the *DDG* for a simulation trace $t_{\mathtt{x}}$ by computing data dependence edges between state nodes in $t_{\mathtt{x}}$. Then the algorithm computes the trace slice for a given slicing criterion (e.g. $tsc = (dm, k)$). We consider that the simulation trace is constructed (by our semantic simulator) and it is provided as an input to TSAlgo. In the first step of Algorithm 4.5 on the following page, data dependence edges are computed for each state in the trace. This part of the algorithm is a while loop (lines 9 to 13). On each iteration of the while loop, a new state node s_j is selected and *de* is computed for j. In line 11, the procedure `find_data_dep_edge(`s_j`)` identifies data dependence edges by finding the state node at position $dp_j(dm)$ that defines data manipulator *dm*, which is used at position j. The pseudocode of `find_data_dep_edge` is listed in Algorithm 4.3 on the previous page. If there exists a definition position node $dp_j(dm)$ in the trace $t_{\mathtt{x}}$ such that $dp_j(dm) < j$ then the procedure creates a dependence edge $de = (j, dp_j(dm))$ and includes it in the set *DDG*. The process of identifying data dependencies for state nodes in a trace and creating dependence edges in the *DDG* continues until all states in the trace are processed. Finally, the procedure `compute_slice`, line 15, performs a backward slice and produces the sequence of state nodes in $t_{\mathtt{x}}$ that are reachable from the slicing criterion via *de* in the *DDG*. In order to include the final execution context of the last evaluated command in the trace slice, the procedure includes (in the trace slice) the state whose position in the original trace is one more than the recent definition position of the slicing criterion data manipulator, i.e. $dp_k(dm) + 1$ (given that $tsc = (dm, k)$). This ensures that the last state in the trace slice preserves the final execution context of the original trace (the program's behaviour) with respect to the slicing criterion data manipulator. The pseudocode of `compute_slice` is in Algorithm 4.4 on the preceding page.

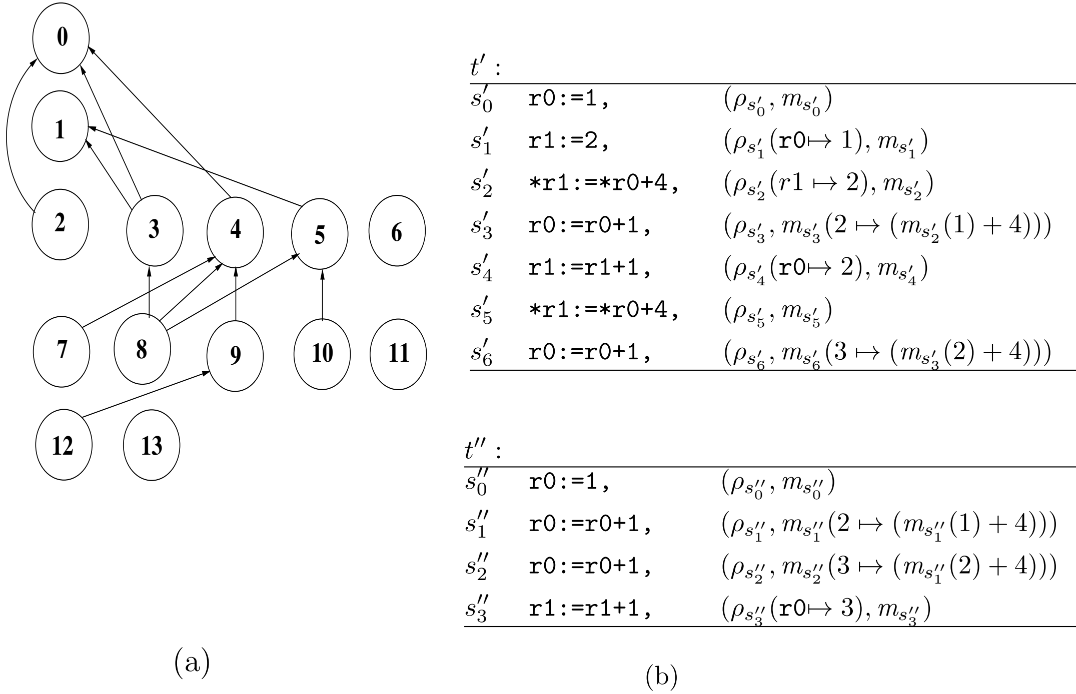
Algorithm 4.5: Trace-Slicing Algorithm (TSAlgo).

1: **Input:** a simulation trace $t_{\mathtt{x}}$ and a trace slicing criterion $tsc = (dm, k)$
2: **Output:** a trace slice t'
3: **procedure** `find_data_dep_edge(`j`)` in Algorithm 4.3 on page 78.
4: **procedure** `compute_slice()` in Algorithm 4.4 on page 78.
5: *DDG*: a set of dynamic dependence edges
6: **begin** `TSAlgo`
7: step 1: construct *DDG* only once.
8: start at index $j = 0$ in $t_{\mathtt{x}}$:
9: **while** there exists a state s_j in $t_{\mathtt{x}}$ **do**
10: compute data dependence edges for s_j:
11: `find_data_dep_edge(`j`)`
12: j++
13: **end while**
14: step 2: for each call (a slicing query made by our detection system) w.r.t $tsc = (dm, k)$, compute the slice:
15: `compute_slice()`
16: **end** `TSAlgo`

Example 4.3. *We illustrate the working of TSAlgo with the aid of the sample* AAPL *program in Figure 4.3 on page 65 and its simulation trace t in Figure 4.3 on page 65. When Algorithm 4.5 is applied to t_x for $dm = r0$ at position $k = 13$ and for $dm = *r1$ at position $k = 8$ (in a different slicing query), the DDG, and the trace slices t' and t'' are computed (w.r.t. $*r1$ and $r0$, respectively) as shown in Figure 4.4 on the following page. The trace slice t'' for $r0$ at position 13 in t_x is computed in the following way:*

After the initialisation steps in `compute_slice` (lines 4–6 of Algorithm 4.4 on page 78), all state nodes in the trace are set as not marked and not visited, the slice set is empty and the algorithm marks the most recent definition node of $r0$ (i.e. $dp_{13}(r0) = 9$). After the first iteration of the while loop, $t'' = s_9$ and the following state is set as marked and not visited in t_x in line 7: this is $\{s_4\}$ because it is reachable from s_9 in the DDG of Figure 4.4 on the following page. After the second iteration of the while loop in line 7, the slice contains s_9; s_4 is set as marked and as a not visited state in t_x so far. The following is the outcome of the remaining while loop iterations of Algorithm 4.4 on page 78:

- *After the third iteration: $t'' = < s_4, s_9 >$ and marked and not visited state nodes are $t_x = \{s_0\}$.*



t' :

s'_0	`r0:=1,`	$(\rho_{s'_0}, m_{s'_0})$
s'_1	`r1:=2,`	$(\rho_{s'_1}(\texttt{r0}\mapsto 1), m_{s'_1})$
s'_2	`*r1:=*r0+4,`	$(\rho_{s'_2}(r1 \mapsto 2), m_{s'_2})$
s'_3	`r0:=r0+1,`	$(\rho_{s'_3}, m_{s'_3}(2 \mapsto (m_{s'_2}(1) + 4)))$
s'_4	`r1:=r1+1,`	$(\rho_{s'_4}(\texttt{r0}\mapsto 2), m_{s'_4})$
s'_5	`*r1:=*r0+4,`	$(\rho_{s'_5}, m_{s'_5})$
s'_6	`r0:=r0+1,`	$(\rho_{s'_6}, m_{s'_6}(3 \mapsto (m_{s'_3}(2) + 4)))$

t'' :

s''_0	`r0:=1,`	$(\rho_{s''_0}, m_{s''_0})$
s''_1	`r0:=r0+1,`	$(\rho_{s''_1}, m_{s''_1}(2 \mapsto (m_{s''_1}(1) + 4)))$
s''_2	`r0:=r0+1,`	$(\rho_{s''_2}, m_{s''_2}(3 \mapsto (m_{s''_1}(2) + 4)))$
s''_3	`r1:=r1+1,`	$(\rho_{s''_3}(\texttt{r0}\mapsto 3), m_{s''_3})$

(b)

Figure 4.4: *DDG* of the program P with respect to $t_{\texttt{x}}$ in Figure 4.3 on page 65. t' and t'' are computed by Algorithm 4.5 on the preceding page w.r.t. $tsc = (*r1, 13)$ and $tsc = (r0, 8)$ (i.e. $*r1$ and $r0$ at positions 8 and 13 in $t_{\texttt{x}}$, respectively) (b). Note that the state indices in both slices are renumbered to reflect the new sequencing in the trace slices.

- *After the fourth iteration:* $t'' = < s_0, s_4, s_9 >$ *and marked and not visited state nodes are* $t_{\texttt{x}} = \{\phi\}$.

Finally, to show the final value of the slicing criterion data manipulator $r0$ *is preserved in* t''*, the state which is at position* $dp_{13}(r0) + 1 = 9 + 1 = 10$ *(in the original trace) is included in* $t'' = < s_0, s_4, s_9, s_{10} >$.

4.3.5 Implementation

The objective of the algorithm is to evaluate the trace-slicing algorithm using binary executables. In Chapter 5, a version of this implementation is developed for slicing traces of an AAPL program in which the trace-slicing algorithm is incorporated into our trace-based malware detection system. In this experiment, we implemented trace generation, *DDG* construction (step 1 of TSAlgo) and trace-slice

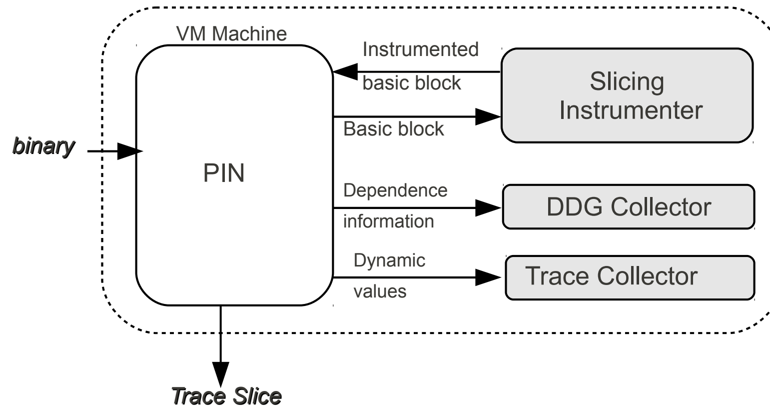


Figure 4.5: Instrumentation tool infrastructure.

computation (step 2 of TSAlgo) presented in this chapter for malicious binary executables. In addition, we also developed implementations of several slicing queries for the slicing criterion data manipulators in the captured traces.

PIN DBI framework. There are many DBI frameworks; Nethercote [15] discusses eleven in detail. To carry out this experiment, we use Pin [LCM+05], a dynamic instrumentation tool, to instrument the input binary programs and collect execution traces. We selected the Pin framework because it is the best known of the currently available DBI frameworks, and the one that provides the most support for virtual registers or register re-allocation and memory values. Pin makes instrumentation tools, which are robust, relatively easy to write and has powerful instrumentation mechanisms with reasonable performance. Pin is more suitable for lightweight dynamic binary analysis (DBA) than other popular frameworks, such as DIOTA [MRD02], Valgrind[Net04] and DynamoRIO [BGA03].

Tool Infrastructure. The experiment was carried out on a Core 2 Duo CPU 2.10 GHz machine with 4 gigabyte RAM and a 120 gigabyte hard disk, running Ubuntu Linux (kernel 2.6.32-22). Since our algorithm is designed to handle executable malicious programs the malicious code must be instrumented and executed. However, the execution of malicious code may cause damage to the host machine. To make our approach practical, we use a virtual machine (VM) [VMw] running on the host machine. A virtual machine is a closed environment, so that the untrusted code is kept in an isolated operating system. The instrumentation tool analyses a malicious executable in the isolated environment. Thus, the actual host machine will not be damaged by running the programs. Figure 4.5 shows the

Program	Description
print_tokens	lexical analyser
gzip	compression utility
bzip2	compression utility
flex	lexical analyser generator
gap	discrete algebra function
lychan	virus
rst	virus
telf	virus
xone	virus
binom	virus

Table 4.1: Program samples used in the experiment.

main components of our tool. The dynamic profiling component of our system, based on the Pin profiler, runs an executable program (the input) and collects the sequence of program states, i.e. program commands that are executed during the instrumentation and information about the environment of program registers and memory locations (that are modified by the program commands). With Pin, we can code our slicing algorithm in a single .cpp file then feed it to Pin to generate our customised instrumentation tool. The slicing instrument accepts input code from Pin, instruments it and returns the instrumented binary back to the Pin framework. The instrumented binary is executed with the support of the slicing instrumenter. Pin executes the input binary by calling the instrumentation routine in the instrumenter. The instrumentation routine instruments the basic block provided and returns the instrumented code to Pin. Then Pin executes the instrumented code instead of the original one. Our tool intercepts output system calls to collect dynamic information, and these are used to augment the execution trace and to update the dynamic data dependence graph (*DDG*). Our tool [Alz10b] can be run under Pin and accepts application binaries (currently, in order to use our tool with Pin, the binaries have to be produced and run under Linux). The slicing tool can then perform slicing on the collected trace.

Table 4.1 shows the programs we used for our experimentation. The first five programs are medium-sized Linux utility programs and the remaining five are executable variants of malware programs (Linux-based viruses) downloaded from the VxHeavens [Hea] website. The instrumentation program ran the executable programs and collected the sequences of program states, i.e. program commands that are executed during the instrumentation and information about the environment of program registers and memory locations (that are modified by the program

Program	Slicing Queries	Slice Size			Execution Time	
		MIN	MAX	AVG	inst_t	trace_t + slice_t
print_tokens	20	2	32	638	5.98	2.23
gzip	20	1	499	25	3.66	2.58
bzip2	20	2	818	41	6.05	4.15
flex	20	1	679	34	9.04	3.02
gap	20	2	1098	55	12.02	5.07
lychan	20	1	699	35	5.03	3.34
rst	20	3	437	22	5.76	2.51
telf	20	1	139	7	1.81	1.40
xone	20	1	299	15	2.89	1.67
binom	20	1	359	18	3.36	1.93

Table 4.2: Summary of overhead results.

commands). With Pin, we can code our slicing algorithm in a single .cpp file then feed it to Pin to generate our customised instrumentation tool. Our tool [Alz10b] can be run under Pin and accepts application binaries (currently, in order to use our tool with Pin, the binaries have to be produced and run under Linux). The slicing tool can then collect dynamic data dependence information from the instrumented application. For all programs tested in this experiment, the slicing criteria we chose, at the end of each trace, are the set of defined program registers and memory addresses (i.e. variables) in the trace. We limited the number of slicing queries to 20 for each program run.

Improving Execution Time Performance. To reduce the overhead of collecting trace information and the slicing computation, the tool removes program states (and their instructions) from the trace if they do not contribute to the data dependency. That is, the instrumentation executes these instructions to update the flow of the execution and to find the next instruction to be executed, but it does not include them within the trace and the DDG. Also, due to the presence of loops, the trace data collection step may require significant space and time to run. To reduce the time and space overheads, a termination condition is set. Once the execution of the instrumented program reaches the termination threshold (e.g. the number of program states recorded is over a threshold, h), the instrumentation tool stops the instrumentation analysis of the running program. With the current computing power, we specified the termination threshold to be around $h = 1.5$ million program state nodes (with their instructions) to be collected and analysed by the slicing algorithm.

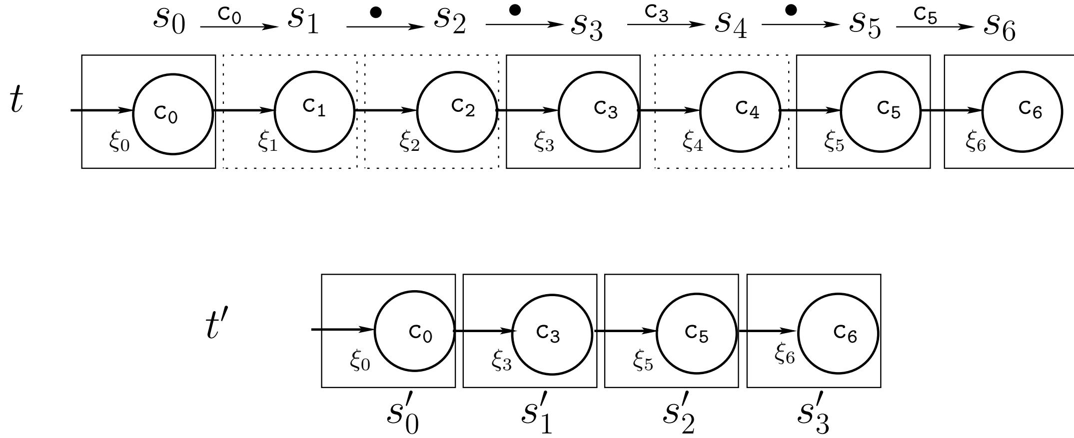


Figure 4.6: A simulation trace configuration t and its slice t'. Each state consists of an execution context and a command. States in t with solid and dotted-line squares represent sliced states and non-sliced states, respectively.

Execution Time Overheads. Table 4.2 on the preceding page shows the overhead of the implementation on the program examples. For each of the programs, the slicer had computed 20 distinct trace slices at the end of the program's execution. The minimum (MIN), maximum (MAX) and average (AVG) trace slice sizes that were observed are given in the slice size column. The slice sizes are measured in terms of program instructions (statements) sliced from the execution trace. The key components in the implementation that contribute to the execution time are: *inst_t*, the time required by instrumentation analysis (i.e. executing program commands and capturing the trace), *trace_t*, the time required by step 1 of TSAlgo (Algorithm 4.5 on page 80) to construct the DDG graph and *slice_t*, the time required by step 2 of TSAlgo to compute a trace slice. The average total time required by all three components was around 8.35 seconds, based on an average of 20 executions consisting of 1.5 million program state nodes. In fact, the average total *inst_t* time for producing the same number of state nodes without *trace_t* and *slice_t* was 5.57 seconds. Thus, running all three components increases the average total (execution) time by 50% compared to running only the first component (*inst_t*). We believe that this incurred time is acceptable and does not impose a severe limitation on a module within a semantic malware detector.

4.4 Correctness of the Trace-Slicing Algorithm

Given a trace t of a program P simulated with a program input `x`, the trace slice t' is generated by applying TSAlgo on t with respect to a slicing criterion tsc. The conjecture of the correctness property states that our trace-slicing algorithm produces a correct trace slice (i.e. t' is a correct slice of t with respect to tsc) if the observable behaviour in the simulation of program commands in the trace slice is similar to the observable behaviour in the simulation of the original program for the slicing-criterion data manipulator dm. That is, the semantic values of dm in t and t' during the simulation are consistent.

The correctness proof of Algorithm 4.5 on page 80 is based on a given trace $t \in S^*$, and its slice t'. We present some definitions for the correctness proof:

Definition 4.14 (n-sequence Directed Trace)**.** The n-sequence directed trace t is a trace that consists of ordered program states, with $n > 1$. $t = (\{s_0, ..., s_{n-1}\}, \rightarrow)$ where $t \vdash s_i \rightarrow s_{i+1}$ for any $i \in [0, n-2]$.

That is, within a trace t, only one transition '$\rightarrow$' can occur from one state to another such that each state in the n-sequence directed trace has only one outgoing transition edge and one incoming transition edge. Thus, a transition edge has one *source* state and one *destination* state. Figure 4.6 on the previous page depicts a trace configuration and the transitions between its states. Also, we let $s_i \Rightarrow s_k$ be the reflexive transitive closure of multiple state transitions '$\rightarrow$' taken between states s_i and s_k in t, i.e. $s_i \Rightarrow s_k$ if there exists $s_j \in t$ s.t. $s_i \rightarrow s_j \rightarrow s_k$. Also, in order to distinguish between state transitions that are produced by sliced states (t') and the ones produced by non-sliced states in t, we use the following two labels:

- $t \vdash s \xrightarrow{\mathsf{c}} m$ if $t \vdash s \rightarrow m$, $\mathsf{c} = cmd(s)$ and $s \in t'$
- $t \vdash s \xrightarrow{\bullet} m$ if $t \vdash s \rightarrow m$ and $s \notin t'$
- $t \vdash s \overset{\bullet}{\Rightarrow} s'$ if for all $s_i \in \, < s \ldots s' >$ such that $s_i \xrightarrow{\bullet} s_{i+1}$ (reflexive transitive closure of all *non-sliced* state transitions)
- $t \vdash s \overset{c}{\Rightarrow} m$ if there exists $s' \in t$ such that $t \vdash s \overset{\bullet}{\Rightarrow} s'$, and $t \vdash s' \xrightarrow{c} m$.

Example 4.4. *Given a simulation trace configuration and its slice t' in Figure 4.6 on page 85, then the state transition from s_0 to s_1 is labelled with the instruction command c_0, $t \vdash s_0 \overset{c_0}{\rightarrow} s_1$ since s_0 is included in the slice. Also, there exists a multiple state transition $t \vdash s_1 \overset{\bullet}{\Rightarrow} s_3$ between non-sliced states s_1 and s_3. However, the multiple transition state between s_1 and s_4 is labelled with c_3 since $s_3 \in t'$ and $t \vdash s_3 \overset{c_3}{\rightarrow} s_4$.*

Definition 4.15 (Sliced State Successor *SSuc*()). Let t be an n-sequence directed trace, and $i, q \in POS_t$. Let $s_i, s_q \in t$ where $s_i = (c_i, \xi_i)$ and $s_q = (c_q, \xi_q)$. Let t' be a trace slice of t. s_q is the successor of s_i in the trace t, $SSuc(s_i) = s_q$ iff $\exists < i, .., q > \in t$ where $i \leq q$ such that $s_q \in t'$ and for all $j \in i, ..., q-1$, $s_j \notin t'$.

Example 4.5. *Consider the trace t in Figure 4.6 on page 85. $SSuc(s_1) = SSuc(s_2) = \{s_3\}$, $SSuc(s_4) = \{s_5\}$ and $SSuc(s_6) = \{\emptyset\}$. But for states that are in the slice, $SSuc(s_3) = \{s_3\}$, $SSuc(s_0) = \{s_0\}$ and $SSuc(s_5) = s_5$.*

Note that every state in the trace slice $s \in t'$ is the sliced state successor of itself, i.e. $SSuc(s) = s$.

Definition 4.16 (Weak Simulation). Given two simulation traces t and t', a binary relation $\diamond$ is a weak simulation of t by t' if $\exists s_i \in t$ and $\exists s'_g \in t'$ and whenever $s_i \diamond s'_g$ and $SSuc(s_i) = s_j$ then $\exists s'_q \in t'$ s.t. $s_j \diamond s'_q$ and $SSuc(s'_g) = s'_q$.

Definition 4.16 describes a relation between states in a trace t and its slice t' with respect to a slicing criterion. If a state has a sliced state successor in t then it corresponds to the same sliced state successor in t'.

Definition 4.17 (Relevant Data Manipulators). In a simulation trace t, let $i, k \in POS_t$, and $i < k$. Also, let $< i, ..., k >$ be a state index sequence in t where $s = (c_i, \xi_i) \in t$, and $n = (c_k, \xi_k) \in t$ are program states in t. Moreover, let t' be a trace slice of t. $\forall s_j$ in t where $i \leq j \leq k$, we define $RDM(s_j)$, the set of relevant data manipulators of state s_j, such that $dm \in RDM(s_j)$ iff there exists a state $m \in t'$, $z = Index(m)$ and $c_z = c_k$ s.t. $dm \in use(k)$, but $\forall j \in i, ..., k-1, dm \notin def(j)$.

Example 4.6. *Consider the trace t_x in Figure 4.3 on page 65 and its slice t' in Figure 4.4 on page 81, the sets of relevant data manipulators R_t and $R_{t'}$ are*

computed for each state in t *and* t'*, respectively, as follows:*

t_x	R_t	t'	$R_{t'}$
s_0	$\{m_{s_0}(1)\}$	s'_0	$\{m_{s'_0}(1)\}$
s_1	$\{m_{s_1}(1), r0\}$	s'_1	$\{m_{s'_1}(1), r0\}$
s_2	$\{m_{s_2}(1), r0, r1\}$	s'_2	$\{m_{s'_2}(1), r0, r1\}$
s_3	$\{m_{s_3}(1), r0, r1\}$	s'_3	$\{m_{s'_3}(2), r0, r1\}$
s_4	$\{m_{s_4}(2), r0, r1\}$	s'_4	$\{m_{s'_4}(2), r0, r1\}$
s_5	$\{m_{s_5}(2), r0\}$	s'_5	$\{m_{s'_5}(2), r0, r1\}$
s_6	$\{m_{s_6}(2), r0, r1\}$	s'_6	$\{r0, r1\}$
s_7	$\{m_{s_7}(2), r0, r1\}$		
s_8	$\{m_{s_8}(2), r0, r1\}$		
s_9	$\{r0, r1\}$		
s_{10}	$\{\phi\}$		
s_{11}	$\{\phi\}$		
s_{12}	$\{\phi\}$		
s_{13}	$\{\phi\}$		

Next Lemma 4.1 shows that whenever $SSuc(s) = SSuc(s')$ where $s \in t$ and $s' \in t'$ then both states s, s' have the same set of relevant data manipulators.

Lemma 4.1. *Assume that trace slice* t' *produced by TSAlgo (Algorithm 4.5 on page 80) on* t *(in Figure 4.6 on page 85) is closed under* $\stackrel{ddd}{\rightarrow}$*, and that each state* s *in a trace has at most a single slice successor, i.e.* $0 \leq |SSuc(s)| \leq 1$*. Let states* $s_i \in t$ *and* $s_v \in t'$ *be such that* $SSuc(s_i) = SSuc(s_v)$*, then we have* $RDM(s_i) = RDM(s_v)$*.*

Proof *When* $SSuc(s_i) = \emptyset$*, there is no sliced state successor in* t *and, hence, no sliced state in* t'*, and, thus,* $RDM(s_i) = RDM(s_v) = \emptyset$*. Otherwise, there exists a command* $\{c\}$ *such that* $SSuc(s_i) = SSuc(s_v) = \{s\}$ *such that* $cmd(s) = c$*.*

First let $dm \in RDM(s_i)$ *be given;* $\exists s_o \in t$ *such that* $s_o \in t'$ *with* $dm \in use(o)$*, and a subsequence* $<i, ..., o> \in t$ *such that if* s *exists (with* $k = Index(s)$*) in the path* $<i, ..., o>$ *in* t *and* $s \neq s_o$ *then* $dm \notin def(k)$*. Since* $SSuc(s_i) = \{n\}$*,* n *must occur somewhere in* $<i, ..., o>$ *(Definition 4.15); we now infer that* $dm \in RDM(n)$*.*

Next, let $dm \in RDM(n)$ *be given (where* $j = Index(n)$*);* $\exists s_o \in t$ *such that* $s_o \in t'$ *with* $dm \in use(o)$*, and a path* $<j, ..., o>$ *in* t*, such that if* n *occurs in* $<j, ..., o>$ *and* $n \neq s_o$ *then* $dm \notin def(j)$*. Since* $SSuc(s_i) = \{n\}$*,* $\exists$ *a path* $< i, ..., j >$ *such that if* s *exists in* $< i, ..., j >$ *and* $s \neq n$ *then* $s \notin t'$*. To establish* $RDM(s_i)$*, it is sufficient to show that if* s *occurs in* $< i, ..., j >$ *and* $s \neq n$ *then* $dm \notin def(k)$*.*

Assume the contrary, then the path $< i, ..., j >$ contains at least s, but for that s we would have $s \overset{ddd}{\rightarrow} s_o$ and, thus, $s \in t'$, which is a contradiction. ∎

Example 4.7. *Consider the states $s_6 \in t_x$ in Figure 4.3 on page 65 and $s'_5 \in t'$ in Figure 4.4 on page 81. $SSuc(s_6) = \{s_8\}$ and $SSuc(s'_5) = \{s'_5\}$ where $cmd(s_8) = cmd(s'_5)$. We have $RDM(s_8) = RDM(s'_5) = \{m(2), r0, r1\}$. Note that $m(2) = m_{s_6}(2) = m_{s_6}(2)$*

Next, in Lemma 4.2, we want to show that for any non-sliced state transition $\overset{\bullet}{\rightarrow}$ in a trace t, the values of the relevant data manipulators before and after the transition are unchanged. Since a data manipulator is either a register or a memory location, we define an auxiliary function $val_{s_i}(dm)$ to represent the value of dm, where s_i is a state at index i in a simulation trace:

$$val_{s_i}(dm) = \begin{cases} \rho_{s_i}(dm) & \text{if } dm \text{ is a register} \\ m_{s_i}(dm) & \text{if } dm \text{ is a memory location} \end{cases}$$

Lemma 4.2. *Assume that t' produced by TSAlgo on a trace t is closed under $\overset{ddd}{\rightarrow}$, and that each state in t has at most one sliced state successor. If there exists a state transition $t \vdash s_i \overset{\bullet}{\rightarrow} s_v$ and $SSuc(s_v) \neq \emptyset$, then for all $dm \in RDM(s_i)$, $val_{s_i}(dm) = val_{s_v}(dm)$.*

Proof *From $t \vdash s_i \overset{\bullet}{\rightarrow} s_v$ and $SSuc(s_v) \neq \emptyset$ we infer that $s_i \notin t'$ and there exists $n \in t$ such that $SSuc(s_i) = SSuc(s_v) = \{n\}$. By Lemma 4.1, $RDM(s_i) = RDM(s_v)$. Next, let $dm \in RDM(s_i)$; since $s_i \notin t'$ we infer that $dm \notin def(u)$, where $u = Index(n)$ in t, which shows that $val_{s_i}(dm) = val_{s_v}(dm)$.* ∎

Example 4.8. *Consider states $s_6, s_7 \in t_x$ in Figure 4.3 on page 65 where $t \vdash s_6 \overset{\bullet}{\rightarrow} s_7$. Also, from Example 4.7 we have $RDM(s_6) = \{m(2), r0, r1\}$ and $\forall dm \in RDM(s_6)$ we have $val_{s_6}(dm) = val_{s_7}(dm)$.*

Definition 4.18 (Relation r). Let t, t' be execution traces, $i \in POS_t$, $j \in POS_{t'}$ where $s_i = (c_i, \xi_i) \in t$ and $s'_j = (c'_j, \xi'_j) \in t'$. We define $s_i \; r \; s'_j$ to hold iff:

1. $SSuc(s_i) = s$, $SSuc(s'_j) = s'$ such that $s = s'$, i.e. $cmd(s) = cmd(s')$
2. $\forall dm \in RDM(s_i), val_{s_i}(dm) = val_{s'_j}(dm)$

Example 4.9. *Consider states $s_9 \in t_x$ in Figure 4.3 on page 65 and $s''_2 \in t''_x$ in Figure 4.4 on page 81, where t''_x is a trace slice of t_x w.r.t $tsc = (r0, 8)$, $s_9 =$*

(ξ_{s_9}, c_{s_9}) *and* $s''_2 = (\xi_{s''_2}, c_{s''_2})$. *We have* $c_{s_9} = c''_{s_2}$ *and* $val_{s_9}(r0) = val_{s''_2}(r0)$ *and hence* $s_9\ r\ s''_2$.

The following lemma allows us to extend the relation r to cover the non-sliced states in t.

Lemma 4.3. *Let* t *be a simulation trace and assume that* t' *is closed under* $\overset{ddd}{\rightarrow}$. *Also, let* $s_i, s_j \in t$ *and* $s'_q \in t'$. *If* $s_i\ r\ s'_q$ *and* $t \vdash s_i \overset{\bullet}{\rightarrow} s_j$, *where* $SSuc(s_j) \neq \emptyset$, *then* $s_j\ r\ s'_q$.

Proof *By assumption, there exists a state from* t *that is included in the sliced trace, i.e.* $s \in t'$, *such that* $SSuc(s_j) = s$; *from* $t \vdash s_i \overset{\bullet}{\rightarrow} s_j$ *we infer that* $s_i \notin t'$ *and that* $SSuc(s_i) = s$. *From* $s_i\ r\ s'_q$ *we also infer that* $SSuc(s'_q) = s'_q$, *and that* $RDM(s_i) = RDM(s_j) = RDM(s) = RDM(s'_q)$ *(by Lemma 4.1). Also, from* $s_i\ r\ s'_q$, *for all* $dm \in RDM(s)$ *we have* $val_{s_i}(dm) = val_{s'_q}(dm)$ *by Lemma 4.2, which shows that* $s_j\ r\ s'_q$. ∎

Example 4.10. *Consider states* $s_6, s_7, s_8 \in t_{\boldsymbol{x}}$ *in Figure 4.3 on page 65 and* $s'_5 \in t'$ *in Figure 4.4 on page 81. Let* $s_6\ r\ s'_5$ *(Definition 4.18) and* $t \vdash s_6 \overset{\bullet}{\rightarrow} s_7$. *Since* $SSuc(s_7) = \{s_8\}$, $cmd(s_8) = cmd(s'_5)$ *and* $\forall dm \in RDM(s_7) = \{m_{s_7}(2), r0, r1\}$ *we have* $val_{s_7}(dm) = val_{s'_5}(dm)$, *and thus* $s_7\ r\ s'_5$.

The following lemma shows that whenever there is a relation r between a sliced state in t and a state in the trace slice of t, their destination states have a relation r. That is, whenever we have a transition from a sliced state to a state in t and the source state is in relation r with a state in the slice, there will be a similar transition in the slice in which the destination states of both transitions have the same relation r.

Lemma 4.4. *Let* t *be a simulation trace and assume that* t' *is closed under* $\overset{ddd}{\rightarrow}$. *Also, let* $s_i, s_j \in t$ *and* $s'_q \in t'$. *If* $s_i\ r\ s'_q$ *and* $t \vdash s_i \overset{c}{\rightarrow} s_j$, *where* $c = cmd(s_i)$, *then there exists* s'_v *such that* $s_j\ r\ s'_v$, $SSuc(s'_v) = s'_v$ *and* $t' \vdash s'_q \overset{c}{\rightarrow} s'_v$.

Proof *From* $t \vdash s_i \overset{c}{\rightarrow} s_j$ *we infer that* $s_i = s \in t'$ *and thus* $SSuc(s_i) = s$. *From* $s_i\ r\ s'_q$ *we infer that* $SSuc(s'_q) = s'_q$, *and* $cmd(s_i) = cmd(s'_q)$ *and by Lemma 4.1 we have* $RDM(s_i) = RDM(s'_q)$. *Also for all* $dm \in RDM(s_i)$, $val_{s_i}(dm) = val_{s'_q}(dm)$ *and we also infer that for all* $dm \in use(i)$, $val_{s_i}(dm) = val_{s_q}(dm)$. *Thus, the sliced state* s'_q *contains the same values of* dm *as the original trace state* s_i, *and since*

$cmd(s_i) = cmd(s'_q) = c$ then the outcome of their commands, c, is the same. Next we show that given $dm \in RDM(s_j)$, it holds that $val_{s_j}(dm) = val_{s'_v}(dm)$. This can be shown in two cases:

- *If $dm \in def(i)$, and since $cmd(s_i) = cmd(s'_q) = c$ then $\exists E$ s.t. $c = (dm{:=}E)$, thus $s_{i+1} = \hat{\mathbf{C}}[\![dm := E]\!]\xi_i$ and $s'_{q+1} = \hat{\mathbf{C}}[\![dm := E]\!]\xi_q$. From $s_i \xrightarrow{c} s_j$ and $s'_q \xrightarrow{c} s'_v$, we infer that $val_{s_j}(dm){=}val_{s_{i+1}}(dm)$ and $val_{s'_v}(dm){=}val_{s'q+1}(dm)$, and thus $val_{s_j}(dm){=}val_{s'_v}(dm)$.*
- *If $dm \notin def(i)$, then $dm \in RDM(s_i)$, and the claim follows from $\forall dm \in RDM(s_i) : val_{s_i}(dm){=}val_{s'_q}(dm)$ since $val_{s_j}(dm){=}val_{s_i}(dm){=}val_{s'_q}(dm){=}val_{s'_v}(dm)$.*

Example 4.11. *Consider the states $s_1, s_2 \in t_x$ in Figure 4.3 on page 65 and $s'_1, s'_2 \in t'$ in Figure 4.4 on page 81. Let $s_1\ r\ s'_1$ (Definition 4.18) and $t \vdash s_1 \xrightarrow{c_1} s_2$ (since s_1 is a sliced state) and $t' \vdash s'_1 \xrightarrow{c_1} s'_2$. We have $SSuc(s_2) = s_3$, $SSuc(s'_2) = s'_2$ and $cmd(s_3) = cmd(s'_2)$. From Example 4.6, $RDM(s_2) = \{m(1), r0, r1\}$ and for all $dm \in RDM(s_2)$, the values produced in dm in t_x and t' in Figures 4.3 on page 65 and 4.4 on page 81, respectively, are similar, i.e. $val_{s_2}(dm) = val_{s'_2}(dm)$ and, thus, $s_2\ r\ s'_2$.*

The following theorem proves the correctness of the slicing algorithm.

Theorem 4.1. *Let $s_i, s_t \in t$ and $s'_q, s'_v \in t'$ where $i, l \in POS_t$, $q, v \in POS_{t'}$ and t' is the slice produced by TSAlgo on t with w.r.t. a slicing criterion. Assume that t' is closed under $\xrightarrow{ddd}$. Whenever $s_i\ r\ s'_q$, and $t \vdash s_i \overset{c}{\Rightarrow} s_t$, the relation r is a weak simulation (Definition 4.16).*

Proof *From $t \vdash s_i \overset{c}{\Rightarrow} s_t$ we infer that there exists $< i \ldots k >$ $(k \geq i)$ such that $s_k \xrightarrow{c} s_l$, and for all $j \in < i \ldots k-1 >$ we have $s_j \xrightarrow{\bullet} s_{j+1}$, $s_j \notin t'$, and thus $SSuc(s_j) = s_k \in t'$. By Lemma 4.4 and from $s_k \xrightarrow{c} s_l$ we infer that there exists $s'_v \in t'$ where $q \leq v$ such that $s_t\ r\ s'_v$. Then with $t \vdash s_j \xrightarrow{\bullet} s_{j+1}$, we can apply Lemma 4.3 to infer that for all states at j from $<i \ldots k-1>$ we have $s_j\ r\ s'_q$.* ∎

Example 4.12. *The trace slice computed in Figure 4.4 on page 81 for data manipulator $*r1$ in the trace in Figure 4.3 on page 65 is not 'executable' in the sense that it does not correspond to an execution, but we can produce an 'executable' program P' from the trace slice via extraction of the set of commands P' from the trace slice*

P' :

1	`r0:=n`
2	`r1:=m`
3	`*r1:=*r0+4`
4	`r0:=r0+1`
5	`r1:=r1+1`
6	`*r1:=*r0+4`
7	`r1:=r1+1`

(a)

$t'_{\mathtt{x}}$:

s'_0	`r0:=1,`	$(\rho_{s'_0}, m_{s'_0})$
s'_1	`r1:=2,`	$(\rho_{s'_1}(\mathtt{r0}\mapsto 1), m_{s'_1})$
s'_2	`*r1:=*r0+4,`	$(\rho_{s'_2}(\mathtt{r1}\mapsto 2), m_{s'_2})$
s'_3	`r0:=r0+1,`	$(\rho_{s'_3}, m_{s'_3}(2 \mapsto (m_{s'_2}(1)+4)))$
s'_4	`r1:=r1+1,`	$(\rho_{s'_4}(\mathtt{r0}\mapsto 2), m_{s'_4})$
s'_5	`*r1:=*r0+4,`	$(\rho_{s'_5}(\mathtt{r1}\mapsto 3), m_{s'_5})$
s'_6	`r1:=r1+1,`	$(\rho_{s'_6}, m_{s'_6}(3 \mapsto (m_{s'_2}(1)+8)))$

(b)

Figure 4.7: The program P' is produced by extracting the command sequence from the trace slice t' in Figure 4.4 on page 81 (a); a simulation trace $t'_{\mathtt{x}}$ of P' on input $\mathtt{x} : n=1, m=2$ (b). Note that at final state s' $m_{s'_7}(3 \mapsto (m_{s'_6}(2)+4))$ where $m_{s'_6}(2) = m_{s'_5}(2) = m_{s'_4}(2 \mapsto (m_{s'_3}(1)+4))$.

in Figure 4.4 on page 81. Then, we execute the program P', shown in Figure 4.7, with the same program input ($n=1, m=2$) used to run the original program and generate t_x, as shown in Figure 4.7. The generated simulation trace (a projection) t'_x agrees with the original trace (i.e. t_x in Figure 4.3 on page 65) for the values of the slicing criterion. Thus, we claim that we have the correct sub-trace if we can execute the program projection from the given input and the sub-trace agrees with the original trace for the values of the slicing-criterion data manipulator at the corresponding program point. Therefore, the observable behaviour of the trace of program P' is similar to the observable behaviour of the original trace of Figure 4.3 on page 65, with respect to the slicing criterion.

4.5 Strengths and Limitations of the Trace Slicing Algorithm

The main motivation for our TSAlgo is to counter the effects of code obfuscations on simulation traces of malware variants and to improve the detection rate using short trace slices in the malware signature. Thus, the construction and matching of semantic signatures is enhanced and a malware detector will have fewer false-negative results in detecting obfuscated malware variants. This can be accomplished by removing the effects of the obfuscation techniques (deobfuscation)

and capturing the true semantics of program traces using slicing. Thus, the power of our TSAlgo algorithm relies on its ability to handle malware-obfuscating transformations. We discuss below the set of obfuscating transformations that are used to generate new malware variants and which the TSAlgo algorithm can handle; we call this set *TSAlgo-handled* obfuscations. *TSAlgo-handled* obfuscations are code transformations that add new (syntax) code lines to create new program variants while preserving the data dependence structure of the original program. *Code reordering*: this obfuscation technique is commonly applied on independent commands where their order in the code does not affect other commands. The execution order of commands can be maintained using unconditional jumps. Thus, new variants of the program can be created with the same semantics but different syntax. *Garbage insertion*: this transformation technique introduces commands that have no semantic effect on the program execution. The main objective of the technique is to create new program variants that preserve the original program semantics but contain different syntax. *Equivalent functionality*: this obfuscation technique replaces commands with other equivalent commands that perform the same operations as the original code. *Variable renaming* is an obfuscation technique used by malware writers to obfuscate their code and to produce new malware variants by simply changing registers and variable names in the program. To deal with variable-renaming obfuscation, TSAlgo can be applied to all possible data manipulators in a given simulation trace to produce a set of trace slices. *Opaque predicate*: a predicate whose value (True or False) is known, by the malware writer, a priori to a code transformation but is hard to determine by examining the obfuscated code [CTL97]. This technique obfuscates the program control flow and makes it difficult to analyse statically.

Limitations. The TSAlgo algorithm has some limitations. The slicing algorithm is not resilient with respect to data obfuscation techniques. Introducing new data dependencies between program registers and memory locations is an obfuscation technique that cannot be handled by our TSAlgo algorithm (TSAlgo-unhandled obfuscation class) and thus, generated trace slices may not be efficient in improving the detection rate. This transformation technique obfuscates a program by creating dependencies between variables by rewriting assignments or introducing new ones [CTL97, MDT07]. For instance, malware writers may use this technique to split a register into two registers or to transform a register $r0$ into the expression $r1 * r0 + r2$ where $r1$ and $r2$ contain dummy constant values. Thus, this technique increases the number of data dependencies in the obfuscated variant so that the

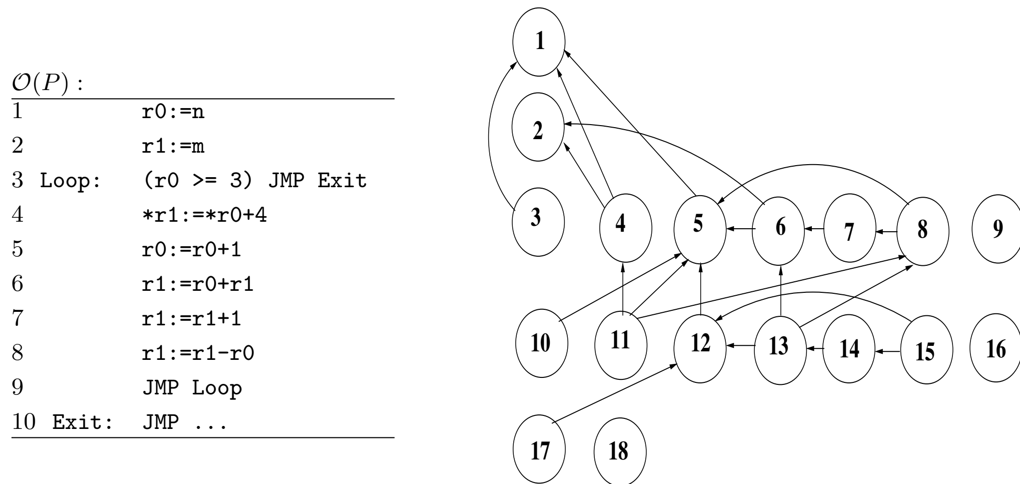


Figure 4.8: An obfuscated code variant of program P in Figure 4.3 on page 65 and its *DDG* after applying data obfuscations.

trace slices have different semantics compared with the trace slices of the malware parent. The example in Figure 4.8 illustrates this transformation technique.

4.6 Review of Related Work

Dynamic slicing has been extended from the traditional slicing techniques for debugging programs [AIP04] to a wider set of applications such as dynamic slicing for concurrent programs [MKM+06, RLG02] and software testing [KY94]. The dynamic slicing approach takes into consideration only one execution history of a program when computing a slice. Thus, it may significantly reduce the size of the slice as opposed to the approach of static slicing. To present all of the dynamic program slicing approaches would be beyond the scope of this chapter. A survey of dynamic program slicing techniques and applications can be found in [XQZ+05, Tip94].

For dynamic slicing techniques that depend on an execution trace, the computed dynamic slice is a subset of the original program. Korel et al. [KL88, KL90] extended Weiser's static slicing algorithm [Wei81] to the dynamic approach. They incorporated the execution history of a program as a trajectory to find the statements that actually affect a variable at a program point. Thus, the resulting

slices are more compact and precise than the program slices proposed by Weiser. Agrawal and Horgan [AH90] provided a novel approach for computing dynamic program slices via program dependence graphs (PDGs). Their algorithm uses the reduced dynamic dependence program (RDDP) where a new node is created if it introduces a new dependence edge with other existing nodes in the RDDP. However, different occurrences of the same node cannot be distinguished in RDDP. None of the above mentioned slicing methods provide a way to capture the dynamic values of variables in a program slice without at least re-executing the program slice. Instead, our approach extracts a trace slice from an simulation trace. The computed slice preserves the semantics of the execution trace of the original program.

Zhang et al. [ZGZ03, ZGZ05] present a dynamic slicing technique that depends on a recorded execution history. Their limited preprocessing (LP) algorithm performs some preprocessing to first augment the record with summary information and then it uses demand driven analysis to extract dynamic control and data dependencies from the augmented record. In this sense, our approach is similar to the approach of Zhang et al. In our approach, the data dependence information can be computed on-the-fly during program simulation and is not used to find control dependencies or to augment the execution trace, but it is mainly used to construct the DDG.

In terms of slicing binary executables, it is hard to find practical slicing solutions for binary executable programs in the literature. The existing techniques proposed in the literature perform static slicing only. Cifuentes and Fraboulet use intraprocedural slicing for handling indirect jumps and function calls in their binary translation framework [CF97]. Debray et al. [DEMDS00] and Kiss et al. [KJLG03] presented methods for the interprocedural static slicing of binary executables. However, these approaches require the extraction of static data dependence information from a control-flow graph (CFG). Instead, our slicing algorithm does not rely on a CFG but it computes information from a simulation trace. Bergeron et al. [BDEK99] propose a static slicing technique for analysing assembly code to detect malicious behaviour. Their approach compares program slices against behavioural specifications (e.g. a set of API signatures) to detect potentially malicious code. However, since their method is purely based on signatures of function calls and the sequence of commands, it lacks the ability to handle certain obfuscation techniques such as code reordering and equivalent functionality. Probably the most similar approach to ours is the work presented in [FG09]. Feng and Gupta

developed a virus detector based on dynamic backward slices of system API traces. Their algorithm uses dynamic data dependence graphs of a system function call as the intermediate representation of the program. However, the slices produced by their algorithm are subsets of the program. Also, they use a statement-based graph as their virus signature that depends on the syntax of the instructions and the data dependency information between the instructions. Instead, our slicing algorithm computes a trace slice that is a subsequence of a simulation trace and preserves the semantic details of the trace. Also, we prove that our trace-slicing algorithm produces a correct trace slice.

4.7 Conclusion

In this chapter we have presented a trace-slicing algorithm for machine code. The method supports the process of capturing semantic details of trace slices as part of a malware signature for detecting (obfuscated) malware variants. Equally importantly, a correctness property is developed and our slicing method is proved to be correct with respect to the property. Trace slicing in this context has two roles: to reverse engineer the effect of obfuscations on the trace and to produce a small trace slice of the simulation trace of a malicious program for efficient signature construction. As a result, detection of malware variants using precise semantic signatures can be improved in terms of speed and accuracy (i.e. fewer false negatives). Our implementation has shown that our TSAlgo may be efficient in computing trace slices within the context of generating and matching signatures of program variants.

Chapter 5

Semantic Trace-based Detector

The malicious behaviour of a malware family is the fundamental qualifying characteristic of all its variants [Coh87, Szö05]. Malware variants generated via code obfuscation can easily change the original syntactical structure of the code and evade syntactic malware detectors. This chapter introduces the approach and implementation of a *semantic trace-based* malware detector. A semantic trace-based detector is based on the idea of incorporating semantic information about the simulation traces of executables to detect malware variants. The detector consists of a static analyser and a semantic trace-matching algorithm. The semantic-based detector uses semantic traces of malicious code as a semantic signature to detect code variants. A static analyser evaluates program instructions and generates semantic traces of executables; we call it the *Semantic Simulator* (SemSim). A semantic trace-matching algorithm uses the SemSim architecture to match semantic traces of malware.

We use the observation that the semantics of program traces represent the effects of malware behaviour. The malicious functionalities of a particular malware code are implemented in its variants, each of which has a different syntactical appearance in the binary code. Our malware detector uses the semantics of the trace to determine whether a given program is a variant of known malware. That is, for a specific malware program to be detected as a variant of a malware, the semantic characteristics of the malware family are consistently presented in their semantic trace information. Our hypothesis is that trace semantics matching is an excellent basis for a malware detector to successfully detect obfuscated variants that belong to the same malware family. We describe a matching algorithm for

identifying similar semantic details for malware variants using semantic traces. We present SemSim for processing program instructions and generating semantic information of the program simulation as a trace. Furthermore, our trace-based malware detector uses trace-slicing algorithm (presented in Chapter 4) and simplification functions (presented in Section 5.1) to construct semantic signatures of known malware variants.

The contributions of this chapter include:

- **A specification of the semantic signature.** A semantic signature of a known malware variant consists of a set of semantic traces and a test input. A semantic signature is used to match variants of the malware. The construction of a semantic signature requires the use of the trace-slicing method and trace simplification functions.
- **A method to match the semantic traces in signatures of code variants.** The approach is a program algorithm `TraceMapping` that uses the program environment and memory domains (i.e. $(\mathcal{E}, \mathcal{M})$, defined in Chapter 4), a semantic simplification step and a program state matching method.
- **A static analyser for simulating the execution of code.** The SemSim architecture statically evaluates a program's execution, computes the instructions and collects a finite simulation trace. The architecture consists of two main steps. First, the malicious code is translated into our intermediate language AAPL and we refer to this step as "input extraction". Second, the known malware code is evaluated and a pair of semantic trace and program input is produced. The outcome of the simulation analysis is an approximation of the program execution behaviour with respect to a program input, which is used to build a signature for code-variant detection.
- **A prototype of a semantic trace-based malware detection system for binary executables.** We have designed and implemented a prototype of a malware detector based on trace semantics. The prototype malware variant detector uses a single signature of a known malware, which consists of a pair of a program test input and a set of semantic traces. The evaluation of the system on real-world and obfuscated malware samples shows that our static analyser (SemSim) together with the semantic trace matching algorithm are effective in detecting real-world and new obfuscated variants of malicious code and in avoiding false positives.

When generating a semantic signature for a known malware sample, SemSim produces a random program input to evaluate the malware program and generate a simulation trace. The trace-based malware detection system implements TSAlgo (Chapter 4) to slice a simulation trace and produce a set of traces for the construction of a malware signature. The rest of this chapter is organised as follows. An overview of the detection system architecture is given in Section 5.1. In Section 5.2, a method of matching semantic traces with code variants and their algorithms are discussed. The discussion of our static analyser architecture is divided into two sections; Section 5.3 presents the input extraction of malicious executables and Section 5.4 presents the semantic simulator. Section 5.5 details the implementation prototype and the experimental results. Section 5.6 concludes the chapter.

5.1 Overview of the Detection System

To determine whether a given executable program is a variant of a malware family (or a known malware program), we must extract the semantic signatures of the program and the known malware. Once a semantic signature is produced, a detector can then identify the malware variant. This section presents the definitions for the *semantic signature* for malware detection and the *semantic trace-based* malware detector (Section 5.1.1). Also, the architecture of the detection system is introduced (Section 5.1.2).

5.1.1 Defining Semantic Signatures

For an AAPL program P, a simulation trace (Definition 4.2 on page 66) can be produced by simulating P. We present our semantic simulator in Section 5.4 which takes a program P (a candidate malware variant) and a program input $\mathtt{x}$ to generate a simulation trace: $\mathit{semsim} : (P, \mathtt{x}) \rightarrow t$. During a simulation of a program, most instructions in the code are responsible for producing data and assigning it to specific registers or memory locations. The produced data is important for the malware program to accomplish its functions (e.g. copying a piece of code into a memory region, calling system functions, etc.). Thus, any manipulation can reveal some of the behaviour (i.e. the semantics) of the malicious program [YHR89]. Our matching method, presented in Section 5.2, looks for the semantic information (of the evaluated code) in the produced trace without considering the syntactical

changes introduced by some obfuscations. In other words, most of the superfluous instructions that are introduced in a malware variant to thwart the static signature- or pattern-based detection techniques, do not affect the semantics of the malicious code. Thus, our detection technique extracts semantic signatures from simulation traces of *known* malware variants and uses them to detect unknown variants.

The semantic signature we develop is in the form of a pair consisting of a set of semantic traces and a corresponding test input. We use the test input from a known malware signature to simulate a suspicious malware variant and to produce a simulation trace. Matching between a semantic trace and an entire simulation trace of a suspicious binary executable will be an expensive task. However, to have an efficient and acceptable detection method, we extract a more compact representation of the trace semantics of malware using the trace-slicing method and trace simplifications (presented in Definitions 5.1 and 5.2), which we call *semantic traces*. A semantic trace is a simple representation of a simulation trace that is produced by simulating program instructions with a given program input. As we will show in Section 5.2, the detection method, which we refer to as *semantic trace matching*, benefits from the trace representation and it can match variants of malware under the presence of code obfuscation transformations. The following are the definitions that are used in the steps we take to generate our semantic signatures of known malware programs.

We define a simplification function α_{sem} that removes any state from a given trace t that does not change the environment or memory of program variables (i.e. the environment $\mathcal{E}$ and memory $\mathcal{M}$ of program registers and memory locations).

Definition 5.1 (Semantic simplification α_{sem}). Given a simulation trace $t \in S^*$, $t =< s_0, \dots, s_i >$ of a program P where $0 \leq i < |t|$, $s_i = (c_{s_i}, \xi_{s_i})$ and $\xi_{s_i} = (\rho_{s_i}, m_{s_i})$, the function $\alpha_{sem} : S^* \rightarrow S^*$ removes program states s_i from t that have no semantic effects during the simulation of P:

$$\alpha_{sem}(t) = \begin{cases} s_i \alpha_{sem}(t') & \text{if } i \geq 0,\ t = s_i t',\ \rho_{s_i} \neq \rho_{s_{i+1}},\ m_i \neq m_{s_{i+1}} \\ \alpha_{sem}(t') & \text{if } i \geq 0,\ t = s_i t',\ \rho_{s_i} = \rho_{s_{i+1}},\ m_i = m_{s_{i+1}} \\ < > & \text{if } t =< > \qquad \text{(empty trace)} \end{cases}$$

The following simplification function, α_e, takes a simulation trace and retains only the information about the execution contexts of the trace:

Definition 5.2 (Execution Context simplification α_e). Given a simulation trace $t =< s_0, \ldots, s_i >$ where $0 \leq i < |t|$ and $s_i = (c_{s_i}, \xi_{s_i})$, the function α_e removes all information about commands that are simulated and produces only execution contexts of t:

$$\alpha_e(t) = \begin{cases} \xi_{s_i}\alpha_e(t') & \text{if } t = (c_{s_i}, \xi_{s_i})t' \\ < > & \text{if } t =< > \qquad \text{(empty trace)} \end{cases}$$

Definition 5.3 (Semantic Traces). A semantic trace of a program P is produced by applying the simplification α_e to a simulation trace t of P, i.e. $t' = \alpha_e(t)$. We use the term semantic states to denote the elements in the semantic trace (i.e. the execution contexts).

Now we present our semantic signature, which is produced by applying the trace-slicing algorithm and the simplification functions on a simulation trace. Given a simulation trace of length k and for all data manipulators that have been defined in the trace (i.e. $\forall dm \in man[\![t']\!],\ dp_k(dm) \neq \emptyset$ (not empty) (Chapter 4)), our trace-based malware detector uses TSAlgo to compute trace slices with respect to the data manipulators. Thus, the trace-slicing algorithm, given in Chapter 4, accepts several slicing queries from the detector where each query consists of a slicing criterion $tsc = (dm, k)$ and produces a trace slice, $TSAlgo : (dm, k) \rightarrow slice$. Then the command information are removed from the set of trace slices to form the semantic traces. We create a semantic signature for a known malware program by including the set of semantic traces, denoted by τ, of *slices* with the program input $\mathtt{x}$ of the simulation trace t.

Definition 5.4 (Semantic Signature). Given a known malware variant M and a program input $\mathtt{x} \in I$, a semantic signature is a pair of simplified, sliced traces and a program input that is produced by the following steps:

1. Generate a simulation trace t of the malware M with $\mathtt{x}$, $semsim : (M, \mathtt{x}) \rightarrow t$, and $k = |t| - 1$.

2. Apply the simplification α_{sem} on the trace t, $\alpha_{sem} : t \rightarrow t'$.

3. Slice the simulation trace and produce a set of trace slices, *slices*: $\forall dm \in man[\![t']\!],\ dp_k(dm) \neq \emptyset,\ tsc = (dm, k),\ TSAlgo : (t', tsc) \rightarrow slices$

4. Apply the simplification α_e on the set *slices* to produce semantic traces, $\alpha_e : slices \rightarrow \tau$.

5. Then a semantic signature of M is

$$sig = (\tau, \mathtt{x})$$

Mapping Semantic Traces Given a known malware program M, its semantic signature $sig = (\tau, \mathtt{x})$, a suspicious program P and the semantic trace of P, t_p, that is produced using the program input $\mathtt{x}$ from sig, we say that P is a variant of M if τ is contained in t_p. In particular, for the execution context updates (states) in each semantic trace in the known malware program signature we look for corresponding semantic states in t_p, $\forall t \in \tau$, where t is mapped to t_p such that $t \subseteq t_p$; i.e. a sub-trace inclusion match in which for each t we identify whether the sequence of nodes (the evolution of execution contexts) in τ exists in t_p. Moreover, to show the effectiveness of the semantic signatures, we use a single semantic trace (i.e. produced only by applying the abstraction functions on the simulation trace) for a known malware to match against semantic traces of suspicious programs. We developed an algorithm that takes a pair of semantic traces and determines if one trace of a suspicious program corresponds semantically to the other trace (of a known malware program). We use the algorithm to iteratively map the semantic traces (the slices) in the set τ to t_p. The method is discussed in Section 5.2.

Malware Variant Detector Our malware detection system, which includes three phases: simulation, signature generation and mapping semantic traces, acts as a malware variant detector $\mathcal{MD}$. The detector takes as its input a signature $sig = (\tau_m, \mathtt{x})$ of a known malware program M and a suspicious program P and it determines whether P may be a variant of M:

$$\mathcal{MD}(sig, P) = \begin{cases} \text{yes} & \text{if } \forall t \in \tau_m,\ t \text{ is contained in } t_p \\ \text{no} & \text{otherwise} \end{cases}$$

5.1.2 The Architecture of the Detection System

In developing the trace-based detection system, which can process executable binaries, extract semantic signatures and identify malicious code variants, we built an architecture with three main components: an input extractor, semantic simulator and signature analyser. Figure 5.1 on page 104 shows the architecture of the trace-based malware variant detection system.

Signature Analyser. This component determines whether the semantic signature, generated by the simulator, contains the signature of a malware family. Thereby, the analyser determines if the input program is a variant of a previously known malware. The analyser maintains and uses a database of malware signatures. This component runs a signature-matching algorithm based on semantic trace mapping and program variants comparison. Details can be found in Section 5.2

Input Extractor. This component transforms an executable binary object into an AAPL code representation. An AAPL program is an intermediate form of the extracted assembly program of the input. We define the syntax and semantics of AAPL in Chapter 4 (Section 4.1). The binary executable is disassembled into assembly code using off-the-shelf disassembler tools. We implemented a program called `asm2aapl`, which translates an assembly program into an AAPL program. Different assembly code syntax such as in Intel and AT&T assembly languages can be represented in AAPL. Section 5.3 discusses the details of our translator.

Semantic Simulator. This component takes AAPL code, evaluates its instructions and generates semantic traces. The simulator evaluates the program based on a set of states (a random program input) for the program environment (i.e. initial memory and register values, system call return values). We include a procedure to generate a random program input for each known malware signature. Section 5.4 describes the simulator in detail.

5.2 Signature Matching

The *signature analyser* in Figure 5.1 on the next page implements the signature-matching mechanism. Trace mapping and program variant comparison are two algorithms that we developed for signature matching.

5.2.1 Mapping Semantic Traces

The objective of the mapping process is to automatically identify a correspondence between program states (nodes) of two semantic traces. The two semantic traces are produced by collecting the simulation traces of two program variants.

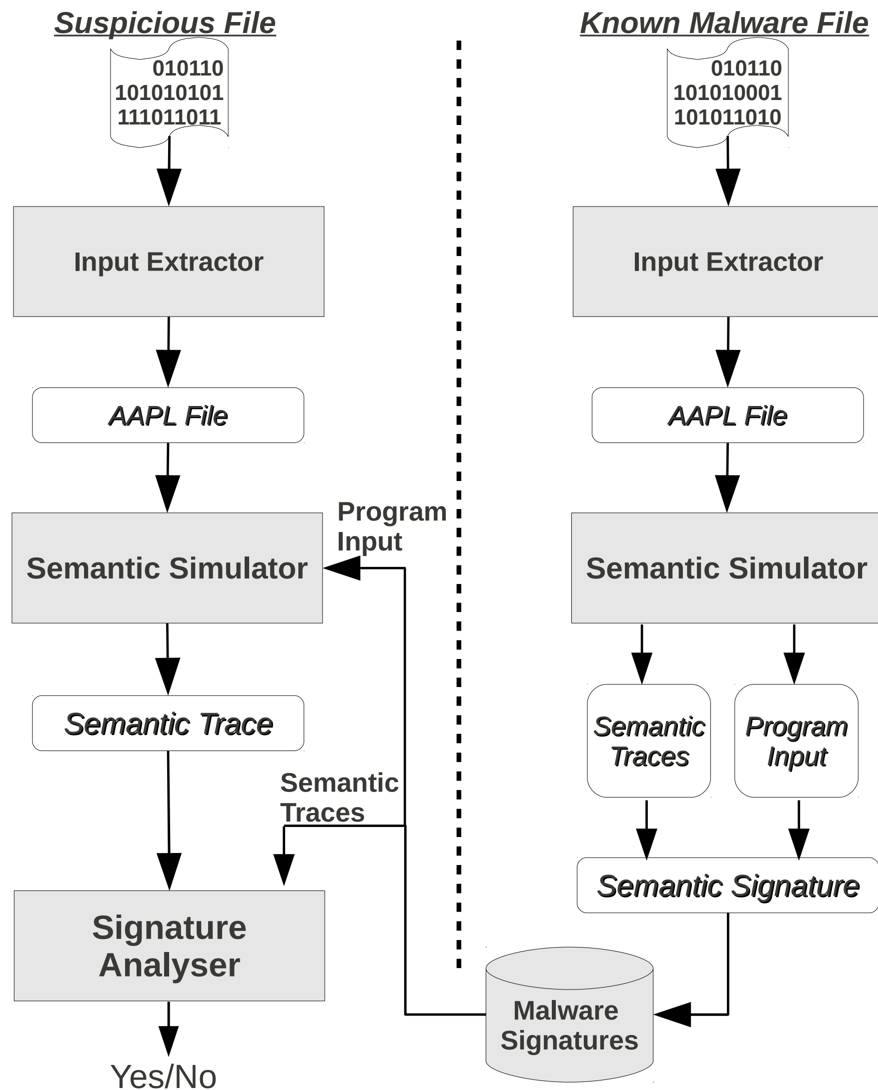


Figure 5.1: The architecture of the Trace-based Malware Detection System. The outcome of the system is either "yes" for a successful detection of a malware variant or "no" otherwise.

We assume that the variants of the program were created by applying semantics-preserving program transformations [CTL98]. In establishing a map between a pair of semantic traces it is our objective to provide an algorithm that produces *complete* and *correct* results. By complete, we mean that our algorithm finds as many true mappings as possible and by correct, we mean the algorithm finds only true mappings as we show later in the evaluation section (Section 5.5), that our detector reports no false positives (i.e. no benign programs are detected as malicious) and few false negatives (i.e. a file from a malware family is misidentified).

Our method uses the known malware signature and a semantic trace produced from a suspicious code variant. The method establishes a correspondence between the program states by examining the semantic details of individual states in both traces. The mapping process has three main steps:

Semantic simplification. Our matching method begins by using the simplification function α_{sem} (Definition 5.1) to remove redundant program states (i.e. execution contexts) in a semantic trace of a suspicious program. During the simplification step, if a state (e.g. s_i) is found to contain similar semantic details (i.e. execution contexts) as the next state (i.e. s_{i+1}) in the sequence (i.e. trace) then the state s_i will be abstracted away from the trace.

State matching. For each given program state in the trace, the semantic value is produced, i.e. the execution context. The semantic values are used to compare two program states, so as to identify potential mappings or exclude mappings of states.

Trace mapping. Given a pair of ordered sets of unique program states, we introduced an iterative algorithm to establish mappings between the states (nodes). For each state in the first set (the semantic trace of a known malware), the algorithm identifies a correspondence candidate state in the other set (the semantic trace of a suspicious program). We consider that the semantic trace of a known malware m is contained in the trace of a suspicious program m' if a large number of semantic states (nodes) in the trace of m are matched with states in the trace of m'. We use the outcome of the trace-mapping step to measure the similarity percentage between the pair of traces. With the similarity measure, we can distinguish between matched and unmatched traces during the signature-matching phase.

Next, we will discuss the details of how the semantic values of program states are used in the state-matching step. Then we discuss the details of the trace-mapping algorithm.

5.2.1.1 State Matching

Before we present our algorithm for mapping a pair of semantic traces from two program variants, we introduce the matching step between a pair of program execution contexts, which we call semantic states. A semantic state is a simplification

of a program state that contains only the information about the execution context (after applying the simplification function α_e to a simulation trace), i.e. a semantic state $s = \xi_s \in \mathcal{X}$ where $s = \xi_s = (\rho_s, m_s)$. The mapping algorithm establishes mappings between two semantic traces based on the successful matches of states. When the state-matching step matches a pair of semantic states, it essentially compares the semantic values produced by both states. The semantic values produced by a state can either represent an environment value ρ or memory value m. Since our mapping step deals with semantic traces of obfuscated program variants, program syntax, i.e. *commands*, may be altered and also some program variables may be replaced with different ones. Thus, establishing an exact match between semantic states is unlikely to succeed. Therefore our state-matching step uses the results computed from individual instructions and ignores command syntax such that the derived semantic results can be easily matched even if program obfuscations have affected the corresponding instructions. For semantic traces with long state sequences, it is unlikely to map traces based on semantic results of states that do not correspond to each other. However, there is a chance of false (i.e. coincidental) mappings between a pair of semantic traces with very short state sequences. Thus, to avoid such false mappings, our state-matching method consists of the following semantic components of the execution context:

Environment values ($\rho \in \mathcal{E}$). To match the environments of semantic states, the environment values are extracted from states and represented as single values $\rho \in \mathcal{E}$. Each environment of a semantic state s returns a single value (ρ_s), which represents the evaluated data value of a data manipulator (i.e. the output) at that particular state. For instance, the output of the semantic state which corresponds to the evaluation of the instruction `POP r7` is the environment value of $r7$, i.e. $\rho(r7)$; the pop command retrieves the data from the stack into the $r7$ register. When matching a pair of semantic states, we look for a match in the evaluated data values of both states. Given two states s_1 and s_2 with their output data values ρ_1 and ρ_2, respectively, we consider that s_1 matches s_2 if the value of ρ_1 is equal to the value of ρ_2.

Memory values ($m \in \mathcal{M}$). When we match memories of semantic states, we are unlikely to find *true* matches of memory addresses between states of semantic traces of both variants. That is because the memory locations of two program variants may vary at runtime. Also, matching the offsets of the memory address of both variants may not be effective in finding matches because we assume that

programs might incorporate dynamic code generation and code reordering techniques to execute new code with a different memory layout (i.e. offset). Therefore, memory values $m \in \mathcal{M}$ are used to establish matches between corresponding semantic states instead. For instance, the output of the instruction `*r1:=r2+m` is the data $m(r1)$ stored at the memory address specified by the operand `r1`. Memory addresses are only used to obtain the memory values of data manipulators in states where memory updates have been performed. The memory match step is performed between a pair of semantic states that have updated (outputted) memory values. The comparison generated values in $\mathcal{M}$ can be performed in the same fashion as that for values in $\mathcal{E}$.

5.2.1.2 Trace Mapping

This section describes the trace-mapping algorithm and how the algorithm establishes mappings between a pair of semantic traces of a known malware and a candidate malware variant. As stated at the start of the chapter, the goal is to map two trace variants of a malware where another variant may have been produced via some code transformations (obfuscation). Semantics-preserving program obfuscations can have significant effects on program syntax. In particular, obfuscating transformations may *rename* program registers, *add* irrelevant commands to the original program, e.g. garbage, system call and opaque predicates, or some transformations may *split*, *reorder* or *merge* commands. Table 5.1 on the following page contains some code transformation techniques deployed in creating new program variants.

Figure 5.2 on page 109 illustrates obfuscation effects on program syntax. In this figure each program command is labelled by a letter. New commands that have been introduced in the program variant are labelled with the obfuscation that was used to create them. Corresponding commands in the original and obfuscated variants are labelled, for example, as g and g', respectively. We use subscripts to show the correspondence between one command in one variant and multiple instructions in the other variant. Figure 5.3 on page 110 shows the generated simulation traces of the program variants p and p' with respect to program input $\mathtt{x} = (5, 6)$. Note that due to the obfuscation effects on p', the simulation trace $t'_{\mathtt{x}}$ of p' contains more states than the trace $t_{\mathtt{x}}$ of the original program. However, with our trace-mapping method, a match could be established between a pair of semantic traces of two program variants.

Label	Category	Obfuscation
gi	Garbage insertion	$\{\} \to \{C\}$
sc	System call	$\{\} \to \{\text{api}\}$
op	Opaque predicate	$\{\} \to \{P^{T/F}\}$
ec	Equivalent command	$\{op\} \to \{\bar{op}\}$
rr	Register renaming	$\{Rx\} \to \{Ry\}$
cs	Command split	$\{C\} \to \{C_x, C_y\}$
cm	Command merging	$\{C_x, C_y\} \to \{C_{xy}\}$
cr	Command reorder	$\{(C_x, C_y)\} \to \{(C_y, C_x)\}$

Table 5.1: Obfuscating transformations.

Algorithm 5.1 on page 112 is our trace-mapping algorithm (`TraceMapping`), which was developed to identify mappings between the semantic traces of two variants of a program that use malware obfuscating transformations. Given a pair of semantic traces $(t_m, t_{m'})$ for two program instances m and m', the algorithm begins by first generating two ordered lists of semantic states *worklistA* and *worklistB* by applying the semantic simplification α_{sem} on t_m and $t_{m'}$, respectively. For instance, Figure 5.4 on page 111 shows the semantic traces t_p and $t_{p'}$ that are produced by applying the simplification functions α_e and α_{sem} on the simulation traces in Figure 5.3 on page 110. Note that after applying α_{sem} on $t'_{\mathtt{x}}$ of the obfuscated program variant, the program states cr_1, cr_2 and op_1 are removed from $t_{p'}$. Then the algorithm begins to establish a correspondence between the semantic states (elements in the lists) by examining the semantic information in the semantic traces of the two malware variants. The output values of the semantic states in each trace, t_p and $t_{p'}$, are extracted and shown in Figure 5.4 on page 111. The matching in Algorithm 5.1 on page 112 consists of a single pass over *worklistA* and for each element in *worklistA* a corresponding matching state in *worklistB* is determined, and the *Mappedlist* is appended with the matched pair. A semantic check between a pair of elements is performed by calling the `state-matching` procedure in step 15 in Algorithm 5.1 on page 112, which is an implementation of the state-matching algorithm presented in Section 5.2.1.1. The trace-mapping (`TraceMapping`) algorithm used is a form of sub-trace inclusion match, in which for the known malware program's semantic trace t_m, the algorithm identifies whether the sequence of semantic states in t_m occurs in the candidate malware variant trace $t_{m'}$, i.e. $t_m \subseteq t_{m'}$, possibly with the interpolation of irrelevant semantic

p :	
a	`r0:=n`
b	`r1:=m`
c	`r2:=r1` $\oplus$ `r1`
d	`r3:=r1+r0`
e	`r2:=r1+9`
f	`r4:=r3-r2`
g	`JMP` ...

(a)

p' :	
rr_1	`r10:=n`
cr_1	`JMP` rr_2
gi_1	`r22:=r11+1`
op_1	P^T `JMP` cm
rr_2	`r11:=m`
gi_2	`r22:=r11+1`
ec	`r2:=0`
cr_2	`JMP` op_1
d'	`r3:=r11+r10`
e'_1	`r2:=9`
e'_2	`r2:=r11+r2`
rr_2	`r4:=r3-r2`
g'	`JMP` ...

(b)

Figure 5.2: A sample program (a) and its variant (b), after applying program obfuscation techniques from Table 5.1.

states introduced by obfuscation artefacts. Figure 5.5 on page 111 shows the mappings between the corresponding program states in semantic traces t_p and $t_{p'}$ of the program variants (in Figure 5.2). Since a malware signature contains a set of semantic traces (slices) τ_m, the trace-mapping algorithm is applied to determine if each trace in τ_m is contained in the semantic trace of a suspicious code t_p. Next (in Section 5.2.2) we define a function to measure the degree of similarity between two semantic traces and determine if a program is a variant of a known malware with respect to a single semantic trace.

5.2.2 Program Variant Comparison

Given a program input `x`, we assume that for two semantically equivalent program variants m and m' (possibly an obfuscated variant of m), that $m \equiv m'$ if m' exhibits the same behaviour as m and m' has already produced more program states than m for the input `x`. Thus, the semantic trace t'_m of m' contains more states than the semantic trace $t_m \in \tau_m$ of m. We consider that the semantic trace of m is a subsequence of m', i.e. a large number of semantic states (nodes) in t_m are matched with states in $t_{m'}$. We define the term *similarity percentage* function between two

$t_{\mathtt{x}}$:		
a	`r0:=5,`	(ρ_a, m_a)
b	`r1:=6,`	$(\rho_b(r0 \mapsto 5), m_b)$
c	`r2:=r1`$\oplus$`r1,`	$(\rho_c(r0 \mapsto 5, r1 \mapsto 6), m_c)$
d	`r3:=r1+r0,`	$(\rho_d(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto 0), m_d)$
e	`r2:=r1+9,`	$(\rho_e(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto 0, r3 \mapsto 11), m_e)$
f	`r4:=r3-r2,`	$(\rho_f(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto \mathtt{0xf}, r3 \mapsto 6+5), m_f)$
g	`JMP ...,`	$(\rho_g(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto \mathtt{0xf}, r3 \mapsto \mathtt{0xb}, r4 \mapsto 4), m_g)$

$t'_{\mathtt{x}}$:		
rr_1	`r10:=5,`	(ρ_{rr_1}, m_{rr_1})
cr_1	`JMP` rr_2`,`	$(\rho_{cr_1}(r10 \mapsto 5), m_{cr_1})$
rr_2	`r11:=6,`	$(\rho_{rr_2}(r10 \mapsto 5), m_{rr_2})$
gi_2	`r22:=r11+1,`	$(\rho_{gi_2}(r10 \mapsto 5, r11 \mapsto 6), m_{gi_2})$
ec	`r2:=0,`	$(\rho_{ec}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7), m_{ec})$
cr_2	`JMP` op_1`,`	$(\rho_{cr_2}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0), m_{cr_2})$
op_1	P^T `JMP` cm`,`	$(\rho_{op_1}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0), m_{op_1})$
d'	`r3:=r11+r10,`	$(\rho_{cm}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0), m_{cm})$
e'_1	`r2:=9,`	$(\rho_{e'_1}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0, r3 \mapsto \mathtt{0xb}), m_{e'_1})$
e'_2	`r2:=r11+r2,`	$(\rho_{e'_2}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 9, r3 \mapsto \mathtt{0xb}), m_{e'_2})$
rr_2	`r4:=r3-r2,`	$(\rho_{rr_2}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto \mathtt{0xf}, r3 \mapsto \mathtt{0xb}), m_{rr_2})$
g'	`JMP ...,`	$(\rho_g(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto \mathtt{0xf}, r3 \mapsto \mathtt{0xb}, r4 \mapsto 4), m_g)$

Figure 5.3: $t_{\mathtt{x}}$ and $t'_{\mathtt{x}}$ are simulation traces of programs p and p', respectively, (in Figure 5.2 on the preceding page); both traces are generated using program input $\mathtt{x} = (5, 6)$ for n and m, respectively.

given semantic traces to determine the percentage of matched nodes between two given semantic traces as

$$\textit{similarity percentage} = |\textit{mappedlist}|/|t|$$

where *mappedlist* is the set of mapped states between two given semantic traces in Algorithm 5.1 on page 112, and t is the semantic trace of the known (unobfuscated) program variant, e.g. m. For instance, for the mappings established between the pair of semantic traces in Figure 5.5 on the following page, the number of matched state pairs, i.e. $|\textit{mappedlist}|$, is 6 and the length of t_p of the (unobfuscated) program variant is $|t_p| = 7$. Thus, the similarity percentage of mapping semantic traces of p and p' is 85.7%. In the case of using a single semantic trace in the signature of a known malware (see Section 5.5.2.1), we consider a program to be a variant of a malware program if the similarity percentage of mapped states

t_p	semantic state	output:(ρ, m)
a	(ρ_a, m_a)	$(5, -)$
b	$(\rho_b(r0 \mapsto 5), m_b)$	$(6, -)$
c	$(\rho_c(r0 \mapsto 5, r1 \mapsto 6), m_c)$	$(0, -)$
d	$(\rho_d(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto 0), m_d)$	$(0\text{xb}, -)$
e	$(\rho_e(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto 0, r3 \mapsto \texttt{0xb}), m_e)$	$(0\text{xf}, -)$
f	$(\rho_f(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto \texttt{0xf}, r3 \mapsto \texttt{0xb}), m_f)$	$(4, -)$
g	$(\rho_g(r0 \mapsto 5, r1 \mapsto 6, r2 \mapsto \texttt{0xf}, r3 \mapsto \texttt{0xb}, r4 \mapsto 4), m_g)$	$(-, -)$

$t_{p'}$	semantic state	output:(ρ, m)
rr_1	(ρ_{rr_1}, m_{rr_1})	$(5, -)$
rr_2	$(\rho_{rr_2}(r10 \mapsto 5), m_{rr_2})$	$(6, -)$
gi_2	$(\rho_{gi_2}(r10 \mapsto 5, r11 \mapsto 6), m_{gi_2})$	$(7, -)$
ec	$(\rho_{ec}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7), m_{ec})$	$(0, -)$
d'	$(\rho_{cm}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0), m_{cm})$	$(0\text{xb}, -)$
e'_1	$(\rho_{e'_1}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 0, r3 \mapsto \texttt{0xb}), m_{e'_1})$	$(9, -)$
e'_2	$(\rho_{e'_2}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto 9, r3 \mapsto \texttt{0xb}), m_{e'_2})$	$(0\text{xf}, -)$
rr_2	$(\rho_{rr_2}(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto \texttt{0xf}, r3 \mapsto \texttt{0xb}), m_{rr_2})$	$(4, -)$
g'	$(\rho_g(r10 \mapsto 5, r11 \mapsto 6, r22 \mapsto 7, r2 \mapsto \texttt{0xf}, r3 \mapsto \texttt{0xb}, r4 \mapsto 4), m_g)$	$(-, -)$

Figure 5.4: The semantic traces t_p and $t_{p'}$ are generated by applying α_e and α_{sem} to $t_{\texttt{x}}$ and $t'_{\texttt{x}}$, respectively, of Figure 5.3 on the preceding page. An output of each semantic state (ρ, m) is extracted for mapping program states.

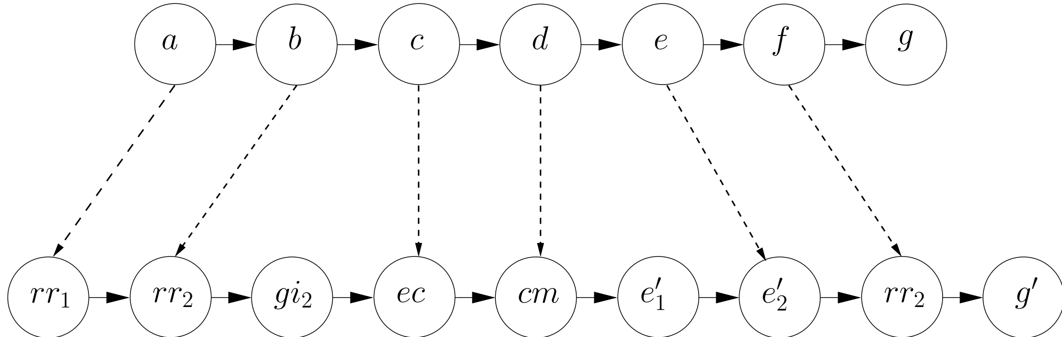


Figure 5.5: Mapping program states of trace variants in Figure 5.4. Since there are no values produced at program state g, a map with its corresponding program state in $t_{p'}$ could not be established.

between their semantic traces in the semantic trace mapping is above a certain similarity threshold k,

$$k \leq \textit{similarity percentage} \leq 100$$

Algorithm 5.1: `TraceMapping`(t,t') maps semantic traces of two program variants.

1: **Input:** a pair of semantic traces t and t' of a known malware program and a suspicious program, respectively.
2: **Output:** a list of pairs of mapped program states *mappedlist*
3: **procedure** `state-matching` is described in Section 5.2.1.1
4: α_{sem} is presented in Definition 5.1
5: `first_index`(l) returns the first index in list l
6: *worklistA*: an ordered list of unique program states
7: *worklistB*: an ordered list of unique program states

8: **begin** `TraceMapping`(t,t')
9: perform semantic simplification process on both traces t and t':
10: $worklistA \rightarrow \alpha_{\text{sem}}(t)$
11: $worklistB \rightarrow \alpha_{\text{sem}}(t')$
12: set all elements in *worklistA* as unvisited:
13: $i =$ `first_index`($worklistA$)
14: **while** $worklistA[i] \neq empty$ **do**
15: $\quad j =$ `first_index`($worklistB$)
16: $\quad$ **while** $worklistB[j] \neq \perp$ **and** match = false **do**
17: $\quad\quad$ **if** `state-matching`($worklistA[i]$,$worklistB[j]$) **then**
18: $\quad\quad\quad$ match = true
19: $\quad\quad\quad$ $mappedlist \rightarrow mappedlist \cup \{(worklistA[i], worklistB[j])\}$
20: $\quad\quad$ **end if**
21: $\quad\quad$ j++
22: $\quad$ **end while**
23: $\quad$ **if** match = false **then**
24: $\quad\quad$ set index i to unmapped in *worklistA*:
25: $\quad$ **end if**
26: $\quad$ i++
27: **end while**
28: **end** `TraceMapping`(t,t')

where k represents a large percentage of (desired) state mappings between a pair of semantic traces. Moreover, when comparing a suspicious program using a malware signature that contains a set of semantic traces, we apply `TraceMapping` and calculate the similarity percentage between each trace in the signature and the semantic trace t'_m of a suspicious program. That is, we consider m' is a variant of m if $\forall t_m \in \tau_m$, the similarity percentage of t_m (w.r.t t'_m) is above threshold k. The prototype implementation and results section (Section 5.5) discusses the selection process for k. In the ideal case, we would expect to have a 100% mapping (similarity) from t_m to $t_{m'}$ where ($|t_m| \leq |t_{m'}|$) for two semantically equivalent program variants. However, this case is rarely achievable and it is illustrated by the example in Figure 5.5 on the previous page and demonstrated by the experimental

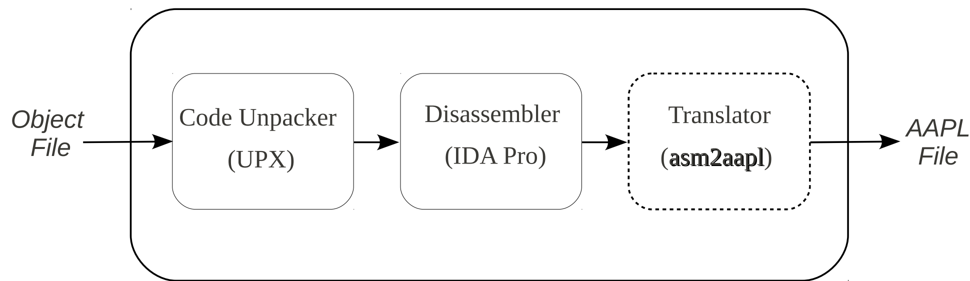


Figure 5.6: Implementation of the input extractor module. Solid-line boxes represent off-the-shelf tools.

results in Section 5.5. The reason is that a program variant might be altered by obfuscation techniques that introduce new instructions.

5.3 Input Extraction

Figure 5.6 shows the basic blocks of our automatic input extractor module (IEM), which is a three-stage process. First, the program has to be prepared for disassembly (e.g. by removing dynamic packing or decrypting the code) so that the potentially malicious binary code is exposed. Then, the resulting binary code is passed to a disassembler to produce assembly code. Third, the assembly code detail is abstracted and an AAPL representation of the code is generated. The first two steps of IEM are implemented using existing tools whereas for the third step, we developed a translator (called `asm2aapl`), which transforms assembly instructions into a simple form of AAPL commands. This step abstracts away some simple obfuscation operations and creates a file in a format that is acceptable for the semantic simulator.

5.3.1 Binary Code Extraction

Malware writers use executable packing tools to hide their malicious code from anti-malware scanners. According to some anti-virus (AV) reports [Symantec, BitDefender], code-packing techniques (i.e. compression and encryption) are used in 75% of all malware executables. Executable packing tools such as Ultimate Packer for eXecutable (UPX) [upx10] and FSG [fsg] compress the binary program in order to save bandwidth or memory space. These tools compress/encrypt

a binary executable and append an extraction/decryption routine to the compressed/encrypted object file. The packer routine is activated at runtime when it decompresses/decrypts the original object code. With a packed binary file, we need to unpack it first and get hold of the original binary object representation. A packed file can be processed by either of two methods: the first method is to execute the packed file and, thus, allow the packer routine to unpack and to output the original executable file. The second method is to use packing tools. We prefer to use the second method as it is a more efficient and reliable way to unpack malware samples. We have selected the UPX packer tool, as it is one of the most commonly used packers in wild malware variants [Res08, SVBY10].

5.3.2 Code Disassembly

Once the binary file has been unpacked and the original suspicious executable is obtained, the machine-level code can be extracted (i.e. disassembled). In general, there are two main approaches for disassembling binary executables. The first approach is a *linear sweep* where the disassembler decodes every sequence of binary code in the file in a strict sequence, starting at the entry point of the program, assuming that each sequence of binary code (i.e. instruction) is aligned to the next. The second approach, which is called *recursive traversal*, decodes each basic block of instructions and determines the control flow of the program by decoding the target address of each branch instruction. In our implementation, IDAPro [Res] is used to disassemble the binary executable of the suspicious file. IDAPro is a state-of-the-art recursive traversal disassembler, which deploys several heuristics and library (API) signatures to resolve imported kernel calls.

Note that extracting a correct syntactical representation of disassembled code can heavily impact upon any malware detection methods based on static analysis. This is because some code obfuscation techniques introduced at this level can hinder successful disassembly. For instance, Linn and Debray [LD03] introduced various *anti-disassembling* obfuscation methods that are specifically designed to make the disassembly of executable code more difficult. Handling such anti-disassembly obfuscations is possible but our method does not. Moreover, from our observations of existing malware variants, we believe the use of these advanced obfuscations in malware is currently minimal. The majority of executable virus, trojan and worm variants are not distributed with any anti-disassembling obfuscation techniques

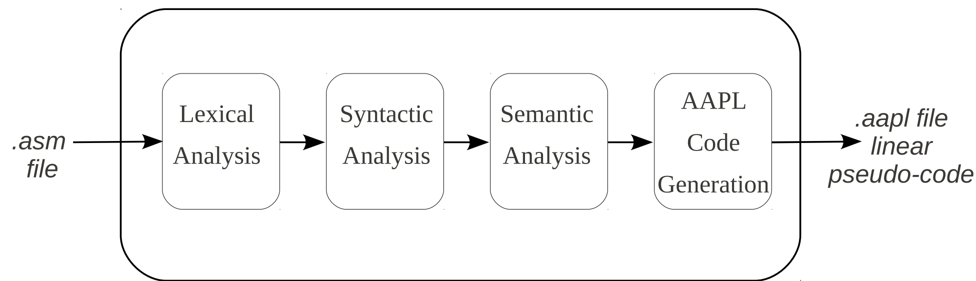


Figure 5.7: Implementation of `asm2aapl` translator module.

besides the use of executable packers. Thus, we assume that malware code is produced either by compiling a high-level code using an industry standard compiler or by writing the program in a standard assembly language, and, thus, in both cases we assume current disassemblers are able to process the produced code successfully. Thus, our approach (and any static approach) is limited to what existing static analysis tools (e.g. disassemblers) provide.

5.3.3 `asm2aapl`: a Translator

The `asm2aapl` translator is an important step in our IEM process, it is designed so that it gives a semantically equivalent simple code of a disassembled executable. The objective of the translator is to analyse assembly code and generate AAPL code, which represents the machine-level operations in a simple and intuitive form. We view this step as a way to transform a given assembly code, which may include simple superfluous commands, into a simple, canonical representation that captures the semantics of program operations and is expressive of a large set of assembly programming instructions. By having malicious executables with different syntactic formats translated into normal form representations, we believe that our detection approach can be leveraged to capture semantic traces and to detect malware variants. Figure 5.7 shows the components of the translator. The source code of the translator can be found in [Alz10a]. The `asm2aapl` translator consists of four stages: lexical analysis, syntactic analysis, semantic analysis and AAPL code generation. All four stages are written in Python and extensive use is made of the Pyparsing Python class library, version 1.5.5 [McG]. We describe our implementation of the four stages in detail as follows.

Lexical and Syntactic Analyses. The first and second stages are a standard process of scanning the input file, generating a sequence of tokens and an abstract syntax tree (AST). Both stages are implemented using the Pyparsing module, which features a set of classes to construct grammar rules. The Pyparsing classes are straightforward to use as an alternative to the lex/yacc approach or the use of regular expressions. During the lexical analysis stage, each line of the source file is tokenised with the help of Pyparsing into two token types: instruction and label. Comments are stripped out and labels and assembler directives are used to build a symbol table. The output of the first stage is a list of line tokens. Each element in the list represents a sequence of tokens describing the types and values for the corresponding line of the input file. The list is then processed during the second stage (syntactic analysis) to resolve instructions, labels, data and register formats and to produce an abstract syntax tree of the parsed code. The syntactic analysis stage performs a “top–bottom” approach to parse the token list and retains syntactic information, such as procedure labels, and the start and end boundaries of data. During this stage, the translator can generate error messages for unrecognised syntax, which then can be handled by identifying the syntax and incorporating suitable AAPL syntax. The output of the second stage is an abstract syntax tree (AST).

Semantic Analysis and Code Generation. This component contains a set of *semantics-preserving* translation rules, which are used to examine the AST and extract the semantic details of the assembly instructions. That is, the component applies semantic rules to each instruction and annotates the AST with semantic information, which is used by the next step to generate AAPL code. The purpose of this stage of the translator is to examine the AST produced by the previous stage and to define a canonical form of the input code in AAPL syntax; by canonical we mean a standard representation. Specifically, we are interested in producing an AAPL representation of the machine-level code with the following information:

- Registers, variables and datatype information (e.g. base and pointer types)
- Data assignment operations
- Instructions for arithmetic and bitwise operations
- Instructions for operations over the stack
- Control flow operations (e.g. call, return, loop, conditional, unconditional jump commands)

Assembly Instructions	AAPL Instructions
`401438: inc  edx`	`401438: r6 = r6 + 1`
`401439: cmp  edx,0x64`	`401439:`
`40143c: jne  0x40142a`	`40143c: ( r6 != 0x64 ) jmp 0x40142a`
`40143e: mov  eax,DWORD PTR [esi]`	`40143e: r17 = *r4`
`401440: mov  edx,DWORD PTR [eax]`	`401440: r6 = *r17`
`401442: mov  DWORD PTR [esi],edx`	`401442: *r4 = r6`
`401444: pop  esi`	`401444: pop r4`
`401445: pop  ebx`	`401445: pop r3`
`401446: ret`	`401446: rtn`
`401447: nop`	`401447: skip`
`401448: mov  DWORD PTR [eax],eax`	`401448: *r17 = r17`
`40144a: mov  DWORD PTR [eax+0x4],eax`	`40144a: *(r17 + 0x4) = r17`
`40144d: ret`	`40144d: rtn`
`40144e: mov  eax,eax`	`40144e: r17 = r17`
`401450: push ebx`	`401450: push r3`
`401451: push esi`	`401451: push r4`
`401452: mov  esi,edx`	`401452: r4 = r6`
`401454: mov  ebx,eax`	`401454: r3 = r17`
`401456: call 0x4013f8`	`401456: call 0x4013f8`
`40145b: test eax,eax`	`40145b:`
`40145d: jne  0x401464`	`40145d: ( r17 != r17 ) jmp 0x401464`
`40145f: xor  eax,eax`	`40145f: r17 = r17 ^ r17`

Figure 5.8: A fragment of an assembly .asm file and its AAPL file from the Bho (win32) virus code.

- Instructions for operations over the environment (e.g. system calls)

Special instructions (i.e. system instructions) for optimisation and control operations over the processor (CPU) are not handled by our tool. System instructions, such as `halt` – halting processor, are implemented to manage and control the processor's functionalities [Int90, Int]. Also, the multimedia (MMX) instruction set (and MMX registers) are not covered in our AAPL syntax. The MMX is an extension to Intel's standard instruction set and it is implemented to greatly increase performance of the execution of programs related to digital signal processing and graphics processing applications. Our tool deals with malicious programs that are possibly variants of known malware code which do not in general incorporate MMX instruction set. However, our tool can be amended to handle MMX instructions as malware authors could use this technology to enhance their future malware variants [Sop09].

Figure 5.8 shows a fragment assembly code of the Bho virus variant in x86 Intel format (i.e. IA-32) and its translation into AAPL. Note that each line in the assembly code may represent a valid instruction. Each line of code is divided into three distinct columns or fields, as in the fragment in the figure. The first, second

and third columns (from left to right) in the assembly code represent labels, operations and the operands of the instructions, respectively. Labels are optional and are used to represent the locations (or names) for a section of code or data. Labels are preserved in the AAPL code. The translation process of the assembly code that appears in Figure 5.8 on the previous page seems intuitive, but generating the AAPL form of an assembly code requires the construction of a set of formal rules. We implemented translation rules based on the Intel Architecture (IA-32, Intel Syntax) [Int]. Table 5.2 on the following page gives a subset of the translation rules for assembly instructions.

We map assembly registers (33 registers for Intel Architecture 32-bit (x86) [Int]) into a table of AAPL registers. For instance, we map the general-purpose register `eax` to `r17` where number 17 corresponds to `eax` in the register table. The datatype information of AAPL instruction operands are designed to accommodate 32 bit unsigned integers (i.e. `DWORD` unsigned values) as contemporary Intel processors are oriented toward operations over 32-bit numbers. Note that we could have one-to-one, many-to-one and one-to-many instruction translations. A one-to-one translation occurs when a single AAPL instruction corresponds to one assembly instruction. A many-to-one translation occurs when several (more than one) instructions in the AAPL code correspond to one instruction in the assembly code. For instance, in Figure 5.8 on the previous page, two assembly instructions at labels `401439` and `40143c` are translated to a single "normal" AAPL instruction i.e.

{`cmp edx,0x64; jne 0x40142a`} → {`( r6 != 0x64 ) jmp 0x40142a`}

Since the above two assembly instructions (`cmp`, `jne`) together subtract the operands without changing their values and branch to the target location `0x40142a` if the flag register is not set, the translator generates the AAPL conditional instruction that provides equivalent semantics to the jump operation.

The code generation stage takes the processed AST, the generated semantic information and stores the AAPL instructions in a file. For the code and data segments of a program, all data are stored in the same sequence as they appeared in the source code.

Rule	Semantics	asm	→	AAPL
r_1	Load the effective address in `m` to `reg`.	`lea reg,m`	→	`reg:=m`
r_2	Load the data to or from the register, memory or immediate operand.	`mov dst,src`	→	`dst:=src`
r_3	Add data to or from `reg` or `m` or `imm`.	`add dst,src`	→	`dst:= dst + src`
r_4	Sets each bit of `dst` to the result of an exclusive `or` operation of the corresponding bits of the two operands.	`xor dst,src`	→	`dst:=dst & src`
r_5	Load the result of multiplying `al` by `reg` or `m` to `ax`.	`mul src`	→	`r40:=r10×src`
r_6	Compare and jump if equal.	`cmp dst, src` `je target`	→	`(dst==src) jmp target`

r_7	Exchange data between registers or between a register and the memory.	`xchg dst,reg`	→	`r18:=reg`
				`reg:=dst`
				`dst:=r18`

r_8	Divide the accumulator and its extension (edx,eax) by the divisor src (32-bit). Load the quotient and the remainder to eax and edx's extension, respectively, where `edx` is `r6` and `eax` is `r17`.	`div src`	→	`r6:=r6≪32`
				`r6:=r6 | r17`
				`r17:=r6/src`
				`r6:=r6-r17`

Table 5.2: Application of the translation rules (partial list) of assembly instruction syntax to AAPL code. Note that `reg`, `m` and `imm` denote a register, operand located in memory and immediate operand (constant), respectively.

5.4 Semantic Simulator (SemSim)

The semantic simulator (SemSim) is a program which *statically* evaluates AAPL code, and produces the effects of the code evaluation in a form of a semantic trace. The simulator is a software tool that simulates the execution of a malicious program and captures the outcome of a simulated execution without harming or contaminating the host system. The simulator tool is based on AAPL operations. The abstract machine-level (AAPL syntax is presented in Figure 4.1 on page 63)

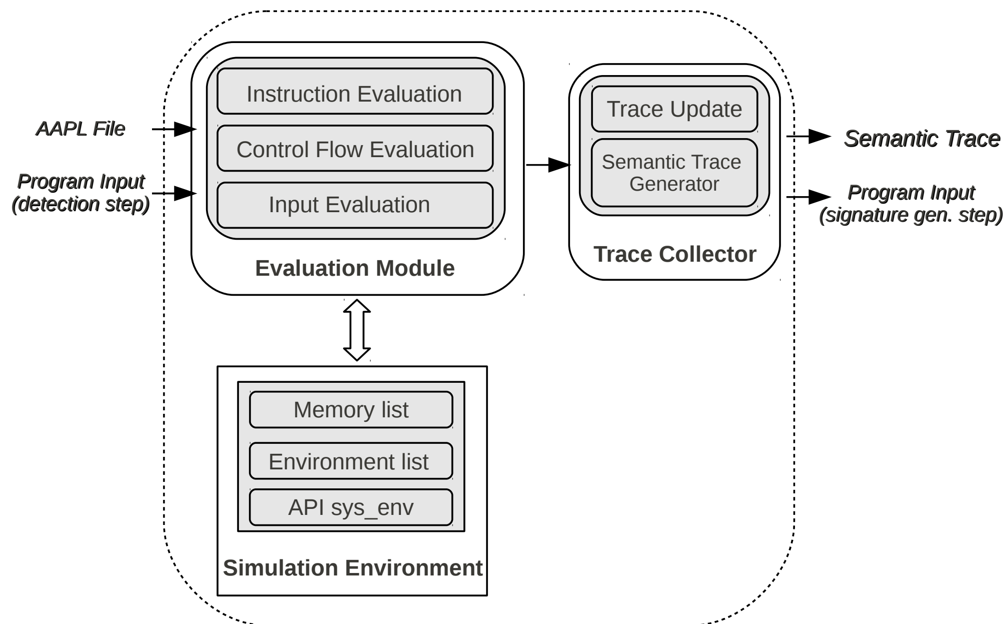


Figure 5.9: Architecture of SemSim. In the signature-generation step, SemSim takes an AAPL file as an input and outputs a trace and a program input; In the detection step, SemSim takes an AAPL file and a program input as an input and outputs a trace as an output.

approach is chosen for the development of the semantic simulator because it particularly fits the purpose of evaluating executable malicious code and capturing its approximate semantics. Most malware variants are reproduced and distributed as executable objects; thus, an effective approach in detecting malicious code variants would be to produce and examine suspected machine-level code. Since AAPL code is an approximate representation of the malicious executable program under test, the goal of our semantic simulation approach is to handle the intermediate representation (e.g. AAPL) of executables and to perform simulations of the program execution and to generate semantic traces from the simulations. Our semantic simulator takes a known malware program during the signature generation step (i.e. AAPL code) as input and simulates the code based on a random configuration of program variables. At the end of the simulation process of a known malware, SemSim generates a set of semantic traces and a program input as a semantic signature of the known malware program. However, during the detection phase, SemSim accepts, as an input, a candidate malware variant and the program input that is part of the signature of a known malware. Figure 5.9 shows the architecture of SemSim and its input and output parameters for the signature-generation and detection steps.

In order to process the AAPL program input and compute semantic traces, the semantic simulation (SemSim) tool comprises three main modules: the evaluation module, a simulation environment and trace collection.

The Evaluation Module (EM) EM includes a set of evaluative procedures, which process AAPL program commands, and a procedure to generate program inputs. The evaluative procedures implement the evaluation rules of the AAPL commands; the semantics are shown in Figure 4.2. The AAPL command set that is handled by our semantic simulator includes commands for data assignment, arithmetic, boolean and program control operations. Moreover, a variety of addressing modes typically available for use in an assembly language are also implemented for AAPL command evaluation in the tool. Table 5.3 on page 123 lists the formats of the addressing modes that can be used for AAPL code. For each AAPL command to be evaluated, EM performs the following steps:

1. Find the values of program data manipulators which are used (read) by the command (Algorithm 5.2 on page 124)
2. Identify the command type (i.e. conditional or action command)
3. Evaluate the command according to its semantics (Figure 4.2 on page 64)
4. Update the program environment and memory lists
5. Update the program input list (Algorithm 5.3 on page 125)
6. Update the PC register to contain the location of the next command to be evaluated

The objective of Algorithm 5.2 on page 124 is to compute the current values of program input data manipulators that are required to simulate program commands in EM. The main procedure in the algorithm is `find_dm_value`; it accepts four input parameters: a command c (to be simulated), the current program environment list, the program memory list and program input list `x` of the simulation trace t. The procedure is invoked each time a command is to be evaluated (in Step 5 in Algorithm 5.4 on page 126). For each given program command c (instruction), the procedure `find_dm_value` computes the value of each data manipulator dm that is used (read) in c. First, the procedure finds all the program input data manipulators that are referenced in the command c and store them in the list use_list

(line 9). Then the algorithm (in a `for` loop) computes the current value for each data manipulator in *use_list* and stores these values in the list *value_list* (lines 12 to 26). If *dm* has already been defined during the simulation, then the procedure returns the value of *dm* from the current program environment or memory in the case of a register or a memory address, i.e. in lines 13 and 19, respectively. However, if a program input data manipulator has no previous defined value, i.e. *dm* has not been defined since the start of the program simulation, then the procedure gets a new generated value (by procedure `compute_value`, in line 29) and initialises *dm* in the current environment or memory with the new value, lines 16 and 22, respectively.

In procedure `compute_value` (in Algorithm 5.2 on page 124), the computation of a new value for an undefined program input data manipulator can be performed under two cases: first (in line 30), if the simulation is performed during the signature-generation step (i.e. constructing a semantic signature for a known malware program) then the new value is generated by calling a random data generator, `get_random_value`, in Algorithm 5.3 on page 125. Second (in line 33), if the simulation is performed during the detection step, then the new value is extracted from the program input `x` of the given malware signature (procedure `get_program_input()` in Algorithm 5.3 on page 125). The simulation terminates if list `x` is empty and no values can be provided to evaluate the candidate program variant. The program under the detection step is considered a benign file with respect to the malware signature used.

The objective of Algorithm 5.3 on page 125 is to provide the value of a program input data manipulator for Algorithm 5.2 on page 124. The algorithm consists of two main procedures: `get_random_value` (in line 3) and `get_program_value` (in line 11). The algorithm takes and updates a list. A program input `x` is treated as a list, consisting of a sequence of values. The procedure `get_random_value` generates a random 32-bit unsigned integer (stored as hexadecimal) value n and appends n to the program input list `x` (e.g. `x`.$append(n)$). Thus, at the end of the simulation run of a known malware program, SemSim outputs the program input list `x` as a part of the semantic signature of the malware. During the detection step, the procedure `get_program_value` in Algorithm 5.3 on page 125 extracts (and removes) the first input value n, as required by `find_dm_value`, from `x` (e.g. `x`.$remove(n)$). As we discuss potential approaches that could defeat our approach in Section 5.5.3, creating a new malware variant that contains new irrelevant program variables could consume the signature program input values. Thus, during the detection

Addressing mode	**Example**
Immediate addressing	`r1=4`
Memory addressing	`r2=*(r3+8)`
Indirect addressing	`r4=*r5`
Register addressing	`r6=r7+r8`

Table 5.3: Supported addressing modes in the semantic simulator.

phase the detector may not be able to extract the correct simulation trace for the variant as the input values are used by irrelevant variables.

The Simulation Environment Module (SE) The simulation environment module consists of the program memory list $\mathcal{M}$, the program environment list $\mathcal{E}$ and a function to simulate API system calls. SE maintains the lists $\mathcal{M}$ and $\mathcal{E}$ and interacts with the evaluation module by providing EM with specific semantic information (values) related to the operands (e.g. program registers, stack pointer and memory locations) and commands being evaluated. The system environment (`sys_env`) function is implemented to simulate the system (kernel) calls (APIs) and their outputs. That is, `sys_env` takes a system call command as an input, checks its input parameters and returns a set of default values *Val* for the command output parameters. Let *API* denote the set of system calls in the AAPL language. For all system calls `api`, we have an output component `api`.*out*, which represents the set of registers that are modified when evaluating an `api` command. The semantics of an `api` command is described in Chapter 4 in Figure 4.2 on page 64. SE produces a set of possible output values for pre-specified system services. That is, when the semantic simulator evaluates a system call, SE checks the corresponding input parameters (registers) that are associated with the system call and returns outputs to EM. A partial list of system calls and their input and output parameters that are implemented in SE are shown in Table 5.4 on page 125.

The Trace Collection Module (TC) TC monitors the simulation process and captures information produced by the evaluation module. When a code sample is simulated in SemSim, the evaluated information (i.e. a pair of command and execution context) is captured and added to the simulation trace. Figure 5.4 illustrates how the above components are combined into an iterative algorithm that allows us to provide semantics-based static simulation of program executables. The key feature of this algorithm is that program commands in the program path

Algorithm 5.2: `find_dm_value`(c) finds the current values of program data manipulators used in the program command c with respect to current execution context (ρ, m).

1: **Input:** command c to be evaluated, current program environment list $\rho \in \mathcal{E}$, current program memory list $m \in \mathcal{M}$, program input list `x` of simulation trace t
2: **Output:** returns the current values of dm to be used (read) in c
3: **procedure** `find_use` is presented in Algorithm 4.1 on page 72
4: **procedure** `get_random_value` is presented in Algorithm 5.3 on the next page
5: **procedure** `get_program_input` is presented in Algorithm 5.3 on the following page
6: **begin:** `find_dm_value`(c)
7: for a given program command c (to be evaluated) at position i in t:
8: find all data manipulators (use_list) that are used (read) in c at i:
9: $use_list =$ `find_use`(i,c)
10: initialise the list of program input values to empty:
11: $value_list \rightarrow \emptyset$
12: **for all** $dm \in use_list$ **do**
13:    **if** dm is a register **then**
14:       **if** $\rho(dm) = \phi$ **then**
15:          compute a value for dm in the program environment:
16:          $\rho(dm \rightarrow v \mid v =$ `compute_value()`$)$
17:       **end if**
18:       $value_list \rightarrow value_list \cup \rho(dm)$
19:    **else if** dm is a memory address **then**
20:       **if** $m(dm) = \phi$ **then**
21:          compute a value for dm in the program memory:
22:          $m(dm \rightarrow v \mid v =$ `compute_value()`$)$
23:       **end if**
24:       $value_list \rightarrow value_list \cup m(dm)$
25:    **end if**
26: **end for**
27: **return** $value_list$
28: **end:** `find_dm_value`(c)
29: **procedure** `compute_value()`
30: **if** signature generation step **then**
31:    generate a random value and append `x`
32:    $v =$ `get_random_value()`
33: **else if** detection step **then**
34:    extract a program input value from `x`
35:    $v =$ `get_program_input()`
36: **end if**
37: **return** v
38: **end procedure**

are semantically evaluated, the control flow is updated, and only updates to the trace are recorded.

System calls	Input	Output
`sys_write`	`r3,r6,r17`	`r6`
`sys_sethostname`	`r3,r15,r17`	`r17`
`sys_munmap`	`r3,r15,r17`	`r17`
`sys_ftruncate`	`r3,r15,r17`	`r17`
`sys_read`	`r3,r6,r15,r17`	`r6`

Table 5.4: A sample of system calls and their I/O parameters (AAPL registers) implemented in SE for simulating system interactions in a program.

Algorithm 5.3: `get_random_value()` and `get_program_value()` updates program input list `x` with input values for Algorithm 5.2 on the preceding page.

1: **Input:** program input list `x` is used to append and remove a program input value during signature generation and detection step, respectively.
2: **Output:** update `x` and return a value n

3: **procedure** `get_random_value()`
4: this procedure is invoked during the signature generation step to select a random value for a new program input in `x`.
5: a data manipulator dm has no value in the current program environment.
6: generate a random value for dm and append the value to `x`:
7: $n \rightarrow$ a random hexadecimal number
8: `x`.$append(n)$
9: **return** n
10: **end procedure**

11: **procedure** `get_program_value()`
12: this procedure is invoked during the detection step.
13: the procedure extracts (removes) the first input value n from the program input list `x`:
14: **if** $\exists n \in$ `x` **then**
15:    `x`.$remove(n)$
16:    **return** n
17: **else**
18:    there exist no values in `x` and the simulation (the detection step) terminates.
19: **end if**
20: **end procedure**

It is possible that the program under simulation may contain a loop or conditional jump operation that may force the simulation to evaluate the program for a very long time (or infinite time). This will increase the overhead of collecting trace information. To handle this situation, the TC module triggers the simulator

Algorithm 5.4: Sketch of the AAPL code simulation algorithm.

```
1. Load the program into the environment memory
2. Set the label _start as the entry point of the program
3. Load the (valid) address of an instruction line, else
   terminate the simulation
4. Parse the instruction line from the memory
5. Evaluate the command at the current line
6. Update the simulation trace with the current program state
7. Compute the valid address of the next point in the program
8. Return to step 3
```

to terminate the evaluation process if the number of program instructions evaluated so far exceeds a certain limit; we discuss the selection of the threshold for terminating a program simulation in Section 5.5.1.

For each known malware program that has already been captured and identified, the TC module generates a semantic signature from the recorded trace. Semantic signature generation is realised by simulating the execution of the sample program with a random program input and generating a simulation trace. Then this trace is sliced by the trace slicer with respect to the set of program data manipulators defined in the trace. For each data manipulator, the slicer performs a backward slice from the recent definition position (Definition 4.7 on page 73 in Chapter 4) of the data manipulator in the trace. The trace slicer applies the trace-slicing algorithm (Algorithm 4.5 on page 80 in Chapter 4) and produces a set of compact traces as part of the semantic signature of the malware. To avoid identifying a benign executable program as a malicious variant of a malware family, i.e. to minimise the number of false positives in the detection process, the set of traces within the signature is reduced by removing ineffective traces. An *ineffective* trace is a sub-trace that contains fewer semantic states and, hence, is likely to produce false positive matches. Trace slices with a length above a predefined threshold are considered to be part of the semantic signature (e.g. we set the acceptable trace length to 20 program states as a minimum sequence of states in a trace).

5.5 Prototype Evaluation

Four experiments were conducted with the semantic trace detection system. The objective of the experiments was to evaluate the effectiveness of the proposed technique to produce semantic signatures and use them to detect program variants in a malware family. The semantic simulator and the signature analyser are modules implemented in Python, along with an input extraction component, forming our malware detector tool. In this section, we discuss details of the prototype implementation, experimental results and the limitations.

5.5.1 Prototype Setup

The implementation and deployment of the system prototype is realised through Python classes. The semantic simulator (SemSim) [Alz11] was developed under Python 2.4.3. The detection system (i.e. `asm2aapl`, SemSim and the signature analyser) can be flexibly installed on various operating systems. During the experimental evaluation of the prototype, it was successfully deployed over diverse distributions of Linux, namely Debian, Ubuntu and CentOS. The semantic simulator is a two-pass AAPL program simulator. During the first pass, each instruction line of the source code is tokenised with the help of Pyparsing [McG]. The Pyparsing module classes are used to construct the AAPL grammar directly in the simulator module. Comments are stripped out and instruction labels are used to build register, memory and branch tables. The output of the first pass is a list of parsed tokens for each instruction of AAPL code and the program tables (registers and symbol table, memory table and label-branch table). During the second pass, the evaluation process handles the instruction list and the tables to evaluate the commands and to generate the trace output for the simulator.

To avoid a very long simulation process when an infinite loop is encountered, the termination threshold k is defined and set to $k = 10,000$ (maximum number of program instructions to be evaluated) for a simulation run. This allows us to manage the computing resource and lower the time and space overhead. All experiments were performed on a machine running an Ubuntu Linux OS, with an Intel Core 2 Duo processor, and 4 GB of RAM.

5.5.2 Evaluation

Many malware variants are created with common functionalities, but with different syntactical appearances in an attempt to avoid detection by AV tools. We evaluate our prototype malware detector against real-world malware variants. Several samples of malware and variants can be retrieved from the Internet. For the experiments, we collected two types of malware variants from the VxHeavens [Hea] website: viruses and worms. These malware samples were developed for the Windows and Linux operating systems. Originally, the total number of malware examples considered was around 62 programs (from 12 malware families); the assembly code of 24 of the samples could not be generated as the off-the-shelf tools (i.e. UPX and IDAPro) failed to handle the binaries. Of these, the UPX packer could not successfully unpack 16 of the packed malware files to recover the binary code. For the other eight excluded programs, IDAPro failed to identify the functions or produced incorrect function-start addresses in the binary code. A further 12 of the extracted binary programs produced no or short simulation traces (e.g. < 10 states) when they were simulated using the input test generated from their family and so we had to exclude them as the sizes of their semantic traces were insufficient for detection. Thus, ten malware families were used for the experiments, consisting of 26 malicious programs. Table 5.5 on page 133 lists the malware variants that were used to test our detector. These families (out of 12) were considered because (random) test input cases were successfully generated during the signature-generation step (using Algorithm 5.2 on page 124) for the malware programs (i.e. known variants) of each family. Also, a collection of benign executable programs were used to test the system for false positives. The goals of the evaluation were:

- to extract semantic signatures that can be used to *match in-the-wild variants* from the same malware family;
- to demonstrate that the detector can *detect new obfuscated malware variants* using the existing semantic signatures of their families;
- to show that the detector *produces few false positives* when running on benign programs; and
- to demonstrate that the detector is able to *classify malware samples* according to their families.

Performance Complexity. During the signature-generation phase, the method requires $O(N)$ space to store the simulation trace and $O(N^2)$ space to store the dynamic data flow information (*DDG*), where N is the length of the trace. A single trace slice from a *DDG* can be extracted in $O(N)$ time and a semantic trace can be produced (i.e. applying the abstraction functions) in $O(n)$. During the detection phase, the complexity of the method consists of the time complexity of producing a semantic trace from a simulation trace and the time required to match the trace against a single trace signature. The complexity of applying the abstraction functions can be calculated as being of $O(N)$ of the simulation trace. Matching the trace with a signature trace of length M requires $O(M.N)$. When applying the matching step using a signature with a set of traces (slices) against a semantic trace of a suspicious program, the time complexity is proportional to the number of traces and the size of each trace in the signature.

5.5.2.1 Signature Extraction

The process of producing semantic signatures from the input program consists of the following steps. First, the semantic simulator evaluates an input program (a known malware), generates a random program input and outputs the simulation trace of the program simulation. The semantic simplification step reduces the trace to eliminate any duplicate states within it. Then the trace is sliced using the trace slicer. The simulation trace is sliced into a set of traces. At the end of the trace collection process, a data dependence graph (*DDG*) is constructed using TSAlgo algorithm (step 1 in Algorithm 4.5). Then for each data manipulator *dm* defined in the trace, a call is made to TSAlgo (step 2 in Algorithm 4.5 on page 80) to compute a trace slice with respect to *dm* at the end of the simulation trace. The final step is to abstract away command syntax from each state within the trace slices. The output of the syntax simplification forms the set of semantic traces, which is paired with the input program to form a signature of the known malware program. Figure 5.10 illustrates the process of extracting the semantic traces from a sample malware program, Binom, using a program input $\mathtt{x} = (64, 20, 21\mathrm{a}, 120, 1\mathrm{f}4)$. The set of semantic traces is extracted from the simulation trace produced by the simulator. A signature is represented as a pair of a program input and a set of semantic traces. Figure 5.10 on page 131(a), Step 1, shows a fragment of the collected trace from the simulation. In Figure 5.10 on page 131(b), Step 2, the simplification function α_{sem} has been applied to the simulation trace and two

program states (states 4 and 8) have been removed from the trace. In Figure 5.10 on the following page(c), Step 3, the trace has then been sliced with respect to the defined program registers, `r17, r3` and `r7` at position (state) 13 in t. Finally, in Figure 5.10 on the next page(d), Step 4, a semantic trace has been produced by removing the trace syntax from the slices. The total number of states of the Binom simulation trace is 926. It took 460 milliseconds to produce the semantic traces (slices) at the end of the simulation. The set of semantic traces in the signature of Binom consists of 12 traces and the average number of states in each trace slice is 34. In general, the longer the collected trace, the more time it takes to extract semantic traces and construct the signatures.

However, to evaluate the effectiveness (speed and detection rates) of semantic signatures produced by the trace-slicing algorithm, we created and used another set of signatures that incorporate semantic traces, which are produced without applying the trace-slicing algorithm. We call these signatures, "*sig-wo-slice*". That is, a sig-wo-slice signature consists of a program input and a semantic trace; the semantic trace in a sig-wo-slice signature is produced after applying the semantic (α_{sem}) and context (α_e) simplifications to a simulation trace. Figure 5.11 on page 132 shows a fragment of the semantic trace that is a part of the sig-wo-slice signature of the Binom family.

5.5.2.2 Detection of In-The-Wild Variants

We used variants from ten malware families (groups of different malware). We used five variants of Bho, three variants of Binom, Mobler and Rst, and two variants of Echo, Grip, Lychan, Synk, Tefl and Xone. Each of them has many instances in the wild, ranging in size from 8 kB to 4.4 MB. For each malware family we calculated a single semantic signature from an early variant of the malware and used the signature for detecting the malware variants. For instance, Binom.a is the first instance of the virus Binom, thus, we considered it as a known malware program and used it to generate the signature for the whole family and tested other Binom variants against it.

We analysed each known malware program using our semantic simulator. We applied the same steps of the signature-generation process to generate the signatures (i.e. pairs of a program input and a set of semantic traces) of all ten malware families. We then applied the simulator with the program input, comparing the

```
-- STATE 1 --
PUSH r7, (r7=21a)
-- STATE 2 --
r7 =r1, (r7=21a,r1=64)
-- STATE 3 --
r1 = r1 - 0x8, (r7=64,r1=64)
-- STATE 4 --
CALL 80482b4, (r7=64,r1=5c)
-- STATE 5 --
PUSH r7, (r7=64,r1=5c)
-- STATE 6 --
r7= r1, (r7=64,r1=5c)
-- STATE 7 --
PUSH r3, (r7=5c,r1=5c,r3=20)
-- STATE 8 --
CALL 80482bd, (r7=5c,r1=5c,r3=20)
-- STATE 9 --
POP r3, (r7=5c,r1=5c,r3=20)
-- STATE 10 --
r3 = r3 + 0x17b3, (r7=5c,r1=5c,r3=80482b9)
-- STATE 11 --
PUSH r17, (r7=5c,r1=5c,r3=8049a6c,r17=120)
-- STATE 12 --
r17= *(r3-0x4), (r7=5c,r1=5c,r3=8049a6c,r17=120,
                                   0x1813=1f4)
-- STATE 13 --
SKIP, (r7=5c,r1=5c,r3=8049a6c,r17=1f4,0x1813=1f4)
-- STATE 14 --
...
```

(a) Step 1: A simulation trace t generated by SemSim.

```
-- STATE 1 --
PUSH r7, (r7=21a)
-- STATE 2 --
r7 =r1, (r7=21a,r1=64)
-- STATE 3 --
r1 = r1 - 0x8, (r7=64,r1=64)
-- STATE 5 --
PUSH r7, (r7=64,r1=5c)
-- STATE 6 --
r7= r1, (r7=64,r1=5c)
-- STATE 7 --
PUSH r3, (r7=5c,r1=5c,r3=20)
-- STATE 9 --
POP r3, (r7=5c,r1=5c,r3=20)
-- STATE 10 --
r3 = r3 + 0x17b3, (r7=5c,r1=5c,r3=80482b9)
-- STATE 11 --
PUSH r17, (r7=5c,r1=5c,r3=8049a6c,r17=120)
-- STATE 12 --
r17= *(r3-0x4), (r7=5c,r1=5c,r3=8049a6c,r17=120,
                                   0x1813=1f4)
-- STATE 13 --
SKIP, (r7=5c,r1=5c,r3=8049a6c,r17=1f4,0x1813=1f4)
-- STATE 14 --
...
```

(b) Step 2: A trace produced after applying α_{sem} to t.

```
-- STATE 7 --
PUSH r3, (r7=5c,r1=5c,r3=20)
-- STATE 9 --
POP r3, (r7=5c,r1=5c,r3=20)
-- STATE 10 --
r3 = r3 + 0x17b3, (r7=5c,r1=5c,r3=80482b9)
-- STATE 12 --
r17= *(r3-0x4), (r7=5c,r1=5c,r3=8049a6c,
                 r17=120,0x1813=1f4)
-- STATE 13 --
SKIP, (r7=5c,r1=5c,r3=8049a6c,r17=1f4,
                   0x1813=1f4)
```

```
-- STATE 7 --
PUSH r3, (r7=5c,r1=5c,r3=20)
-- STATE 9 --
POP r3, (r7=5c,r1=5c,r3=20)
-- STATE 10 --
r3 = r3 + 0x17b3, (r7=5c,r1=5c,r3=80482b9)
-- STATE 11 --
PUSH r17, (r7=5c,r1=5c,r3=8049a6c,r17=120)
```

```
-- STATE 3 --
r1 = r1 - 0x8, (r7=64,r1=64)
-- STATE 6 --
r7= r1, (r7=64,r1=5c)
-- STATE 7 --
PUSH r3, (r7=5c,r1=5c,r3=20)
```

(c) Step 3: Trace slices (from left to right) wrt `r17, r3` and `r7`, respectively at state 13 in t.

```
-- STATE 7 --
(r7=5c,r1=5c,r3=20)
-- STATE 9 --
(r7=5c,r1=5c,r3=20)
-- STATE 10 --
(r7=5c,r1=5c,r3=80482b9)
-- STATE 12 --
(r7=5c,r1=5c,r3=8049a6c,
     r17=120,0x1813=1f4)
-- STATE 13 --
(r7=5c,r1=5c,r3=8049a6c,r17=1f4,
                 0x1813=1f4)
```

```
-- STATE 7 --
(r7=5c,r1=5c,r3=20)
-- STATE 9 --
(r7=5c,r1=5c,r3=20)
-- STATE 10 --
(r7=5c,r1=5c,r3=80482b9)
-- STATE 11 --
(r7=5c,r1=5c,r3=8049a6c,
    r17=120)
```

```
-- STATE 3 --
(r7=64,r1=64)
-- STATE 6 --
(r7=64,r1=5c)
-- STATE 7 --
(r7=5c,r1=5c,r3=20)
```

(d) Step 4: The set of semantic traces τ of Binom after abstracting the trace syntax.

Figure 5.10: The extraction process of a fragment of the semantic traces (part of the signature) of the Binom family, where the randomly generated program input $\mathtt{x} = (64, 20, 21\text{a}, 120, 1\text{f}4)$ for data manipulators $r1, r3, r7, r17$ and 0x1813, respectively.

generated semantic trace t' of each malware variant against the signature of the known malware. In the comparison step, for each semantic trace t (slice) in the set we applied the matching algorithm (`TraceMapping` in Figure 5.1 on page 112) between t and t', calculated the similarity percentage and determined if the pair of the traces established a match. We observed that the similarity between two matchable traces (i.e. traces that are generated from malware variants) is above

```
-- STATE 1 --
(r7=21a)
-- STATE 2 --
(r7=21a,r1=64)
-- STATE 3 --
(r7=64,r1=64)
-- STATE 5 --
(r7=64,r1=5c)
-- STATE 6 --
(r7=64,r1=5c)
-- STATE 7 --
(r7=5c,r1=5c,r3=20)
-- STATE 9 --
(r7=5c,r1=5c,r3=20)
-- STATE 10 --
(r7=5c,r1=5c,r3=80482b9)
-- STATE 11 --
(r7=5c,r1=5c,r3=8049a6c,r17=120)
-- STATE 12 --
(r7=5c,r1=5c,r3=8049a6c,r17=120,0x1813=1f4)
-- STATE 13 --
(r7=5c,r1=5c,r3=8049a6c,r17=1f4,0x1813=1f4)
-- STATE 14 --
...
```

Figure 5.11: A fragment of the semantic trace in the *sig-wo-slice* signature of the Binom family from the simulation trace in Fig. 5.10(a).

70% while the similarity between two different traces (i.e. generated from two different programs) is below 45%. Therefore, we set the threshold (k) as 70% to distinguish between matched and unmatched semantic traces. Finally, we considered a malware variant as detected if the similarity between each t and t' exceeds 70%, i.e. if all semantic traces in the malware signature are matched with the semantic trace of the variant. For nine out of the ten malware families, the detector matched all of the variants to the correct family. One variant of the Bho family did not match the family signature, so its signature was stored in the semantic signature database as a different malware family. Nonetheless, we have shown that our detector is able to match malware variants from the same family using one semantic signature.

Running times for the different phases of the tool are shown in Table 5.6 on page 134 and are reasonable for a prototype and suggest that real time improvements can be achieved with an optimised implementation. We also used the *sig-wo-slice* signatures of the malware families in our detector on the same variants and measured the similarity and running time results in order to compare them with the results obtained using the semantic signature.

Table 5.7 on page 134 shows a summarised comparison of the two sets of results. The similarity results of both types of signatures have the same detection rates, i.e.

File name	Size	Type	Operating system
Binom.a	20 kB	virus	Linux
Binom.b	20 kB	virus	Linux
Binom.c	20 kB	virus	Linux
Bho.a	4.1 MB	virus	Windows
Bho.b	4.2 MB	virus	Windows
Bho.c	4.4 MB	virus	Windows
Bho.d	4.2 MB	virus	Windows
Bho.e	4.2 MB	virus	Windows
Echo.a	8.0 kB	virus	Windows
Echo.b	56.0 kB	virus	Windows
Grip.a	116 kB	virus	Windows
Grip.b	140 kB	virus	Windows
Lychan.a	12 kB	virus	Linux
Lychan.b	15 kB	virus	Linux
Mobler.h	1.1 MB	worm	Windows
Mobler.i	748 kB	worm	Windows
Mobler.g	836 kB	worm	Windows
Rst.a	314 kB	virus	Linux
Rst.b	314 kB	virus	Linux
Rst.c	314 kB	virus	Linux
Synk.a	8.0 kB	virus	Windows
Synk.b	8.0 kB	virus	Windows
Tefl.a	56.0 kB	virus	Linux
Tefl.e	48.0 kB	virus	Linux
Xone.a	12.0 kB	virus	Linux
Xone.c	12.0 kB	virus	Linux

Table 5.5: Malware variants in the wild.

the tested variants can be detected using either type of signature. Nonetheless, in terms of the precision and accuracy of a detection outcome, the similarity results of matching malware variants using semantic signatures are much more accurate than when using *sig-wo-slice* signatures. For instance, for the Grip family, the similarity rate for matching using the *sig-wo-slice* signatures of its variants is 45.7%, which is very low. Whereas, with semantic signatures, Grip variants were detected with an average similarity of over 85%. Overall, using semantic signatures, the detection rates are 30% more accurate than with *sig-wo-slice* signatures. For the average running time results, the cost of producing semantic signatures is longer due to the slice computation. In this evaluation, the running time ratio of *sig-wo-slice* to semantic signatures is approximately 5:7. However, the time of matching traces in the semantic signatures case is faster by 26% (on average) compared to the

Malware	Running time (msec)			Total time	Detection
	Simulation	Abst. and slicing	Matching		
Bho	61232.5	234.9	158.9	61.63 s	80%
Binom	7534.2	460.0	170.0	8.16 s	100%
Echo	6029.0	685.1	204.9	6.92 s	100%
Grip	34621.3	430.0	302.1	35.35 s	100%
Lychan	6322.1	900.0	92.0	7.314 s	100%
Mobler	49549.6	394.9	2098	52.04 s	100%
Rst	82371.7	398.6	268.7	83.05 s	100%
Synk	2672.0	238.8	73.5	2.98 s	100%
Tefl	3412.0	190.0	260.0	8.16 s	100%
Xone	2894.0	311.0	310.0	8.16 s	100%

Table 5.6: Results of evaluating our detector on 26 real-world malware variants.

Malware	*sig-wo-slice* signatures		semantic signatures		
	Similarity	Time[1]	Similarity	Time[1]	Time[2]
Bho	64.6%	197.3	97.4%	158.9	393.8
Binom	67.3%	534.0	86.1%	170.0	630.0
Echo	72%	473.6	100%	204.9	890.0
Grip	45.7%	630.0	85.3%	302.1	732.1
Lychan	90%	930.0	89.7%	92.0	992.0
Mobler	65%	1230.0	93.3%	2098	2492.0
Rst	75.3%	289.3	98.3%	268.7	677.2
Synk	82.0%	293.7	91.5%	73.5	312.3
Telf	85%	358.0	98%	260.0	450.0
Xone	61%	421.7	81%	310.0	621.0

[1]Time to match the signatures. [2]Time to abstract, slice and match.

Table 5.7: Comparison of the similarity and running time (in msec) for detecting malware variants using semantic signatures and *sig-wo-slice* signatures.

running time for *sig-wo-slice* signatures. We believe that the cost of the slicing computation is an up-front cost in the signature generation process and it does not relate to the detection performance (cost).

Table 5.6 shows the results of our experiments comparing malware families against their variants. From this experiment, we observed that the similarity rate of matching a malware family's semantic traces (in the signature) with its variants' traces varies between 70% and 100%. The average similarity between a malware family and its variants is 92.7%. Also, for each malware variant, we achieved 100% detection (i.e. no false negatives) using the trace semantics signatures of the malware family, with the exception of the Bho malware family where one out of the five

code variants was not detected. We believe that this (false negative) occurred due to the fact that the semantic signature contains a single program input, which was ineffective in capturing parts of the semantics of this particular variant, and, hence, the detector could not match the variant semantic trace against the semantic traces in the signature.

5.5.2.3 Detection of Obfuscated Variants

For each malware family in Table 5.6 on the previous page, we created new code variants by applying the following code obfuscation techniques (see Table 5.1 on page 108): code reordering (cr), garbage code insertion (gi), equivalent command replacement (ec), system call insertion (sc), simple command split (cs) of one instruction into two instructions and register renaming (rr). The code reordering (cr) transformation was implemented by moving some parts of the code to a different location and using jump instructions to the new code locations. The code insertion (gi) transformation inserts an irrelevant sequence of instructions into a program. We considered two types of code insertion: first, adding a sequence of `SKIP` instructions and second, adding simple sequences of operations. We considered three types of operations: arithmetic and bitwise operation commands, assignment operation commands and adding a sequence of `PUSH` and `POP` instructions, which push a value onto the stack and then pop the same value into an irrelevant register. For the equivalent command (ec) replacement transformation, we replaced `XOR` instructions with `MOV` instructions whenever an operand is `XOR`-ed with itself; a move command loads zero to the destination operand. Also, we replaced `INC` and `DEC` instructions with `ADD` by one and `SUB` by one instructions, respectively. For register renaming (rr) transformations, we replaced at most three general-purpose registers in a program with different general-purpose registers that did not already exist in the program, if possible. For the command split (cs) transformation, we replaced a `MOV` instruction with `ADD` and `SUB` instructions where a move command must load a constant value (const) into either a register (reg) or a memory location (m). That is, `mov reg/m, const` is transformed into two instructions: `xor reg/m,reg/m` and `add reg/m,const`. For system call insertion (sc), we inserted API calls of the following types: file or device open (`sys_open`) and close (`sys_close`) calls, make directory (`sys_mkdir`) calls and file truncate (`sys_ftruncate`) calls.

Malware	Transformation type							FN
	cr	gi	ec	rr	sc	cs	all	
Rst	Y	Y	Y	Y	N	Y	Y	1
Bho	Y	Y	Y	Y	N	Y	N	2
Synk	Y	Y	Y	Y	N	Y	Y	1
Mobler	Y	Y	Y	Y	Y	Y	Y	0

Table 5.8: The detection results of a set of obfuscated variants of four malware families. Y and N denote whether the detection was successful or unsuccessful, respectively. FN (False Negative) denotes the number of undetected variants.

Also, an instance of each piece of malware was generated using all of the techniques together (we labelled it with *all*). To perform the evaluation, first, we applied our semantic simulator tool to the set of new variants of each malware (seven obfuscated variants) using the program input from the malware signature. Then we used the detector to test the semantic traces of the new variants against the database of existing malware family semantic traces. Table 5.8 shows the results of the experiment for detecting obfuscated malware variants. Our detector identified all of the new variants generated using the code obfuscations with the exception of the *system call* insertion obfuscation. System call instructions, such as those used in this evaluation, alter the environment of an executing program by writing into particular special registers and, hence, altering the semantics of the malicious code variant. Thus, this shows that our detector cannot deal with new malware variants that contain different (altered) semantics from the known malware.

5.5.2.4 False Positives

We performed the same evaluation process with a set of benign executables. That is, for each malware family signature in the database, the benign programs were simulated using the program input from the signature and their semantic traces were examined by our detector to measure the false positive rate. We used ten programs, with sizes ranging from 10 kB to 2 MB, which are standard executable programs for the Linux operating system. For each benign program, we computed the matching similarity of the benign code against the whole database of malware signatures created for the malware families in Section 5.5.2.2. Our detector produced no false positives, that is, our approach successfully identified all programs as benign and none of their semantic traces matched the malware database.

Malware	Similarity			
	File1	File2	File3	File4
Bho	22.0%	17.0%	95.0%	0.0%
Binom	29.0%	16.1%	28.2%	38.3%
Echo	8.0%	100%	0.0%	15.0%
Grip	1.0%	0.0%	0.0%	79.0%
Lychan	0.0%	9.0%	4.5%	44.0%
Mobler	4.6%	7.4%	32.0%	8.2%
Rst	100%	18.3%	5.3%	3.0%
Synk	0.0%	10.4%	45.0%	3.0%
Telf	23.0%	40.0%	0.0%	0.0%
Xone	30.0%	0.0%	5.0%	0.0%

Table 5.9: The similarity rates of randomly selected variants (File1, File2, File3 and File4) compared to the semantic traces (in signatures) of the malware families. All four files were successfully classified by our tool to be variants of Rst, Echo, Bho and Grip, respectively.

5.5.2.5 Classification of Malware Variants

We examined the general classification performance of our approach. We used the signature database of malware families that were generated in Section 5.5.2.1. Also, we randomly selected four different malware variants from the real-world malware variants in Table 5.5 and labelled them as File1, File2, File3 and File4. We then applied the detector to each of the four files and recorded the similarity rates. Table 5.9 shows the percentage match for the semantic traces of the tested files against the traces in the signatures of the malware families. For File1, the highest similarity result is 100%, which means that File1 is a variant of the Rst malware family. The highest similarity rate for File2 is 100% and it is classified as a variant of the Echo malware family. File3 is classified as a variant of the Bho family and its similarity rate is 95.0%. File4 is classified as a member of the Grip family. Our tool correctly assigned all files to their malware families.

5.5.3 Prototype Limitations

We now discuss several potential approaches that may defeat and the limitations of the current prototype that may limit its detection and classification effectiveness. Because our tool simulates suspicious samples with program inputs from the malware signature, it may be susceptible to a technique where irrelevant variables

are introduced in the code. For instance, during the detection phase, inserting instructions with irrelevant input variables (e.g. registers and memory references) can consume the program input list (of the given malware signature) and hence, shift the order of the input. The simulation of a variant with this technique can be affected and the original program variables would be assigned to incorrect input values. Also, because our technique relies on execution context updates (i.e. value updates) to match semantic traces, changes to a few states in the traces, such as inserting some garbage instructions that contain the same values later in the trace are likely to influence the matching results. However, developing new variants using such a technique of inserting operations with new input variables may alter the semantics of the program. That is, producing semantically equivalent but syntactically significant different variants using this technique is a difficult task for malware authors. In terms of the limitations of the tool, one way to evade the current prototype's detection is to prevent the tool from extracting a malicious portion of the binary code by applying packers to the new malware variants' code. Our tool incorporates UPX as a tool to handle packed or hidden malware code; however, like most existing unpack tools, it is by no means complete. One way to address this problem is to employ several dynamic unpack tools, e.g. OllyBonE [Ste07] and Saffron [Val07]. Also, our tool rely on IDAPro to identify function-start locations and data structure addresses. As a result when the code is being simulated, if some computed values are used as indirect reference by a jump command to a function (a portion of the code) or a data structure in the memory, then the tool fails to determine the target instruction. In conclusion and future work chapter (Chapter 7), we suggest a more thorough approach that could mitigate this problem using a hybrid technique.

5.6 Conclusion

We have introduced a semantic trace-based approach for detecting variants of malware that increasingly evade traditional signature- and heuristics-based approaches. AV tools must cope with various obfuscation techniques deployed in generating new unknown malware variants. Effective semantic matching of malware is an important feature of today's anti-malware tools. We have developed a static-based semantic simulator that evaluates malicious low-level programs and generates semantic signatures. That is, a semantic signature describes semantics

for variants of a malware family including any new obfuscated variants. This enables a malware detector to have a compact database of semantic signatures. We have implemented the semantic matching on top of the simulator for computing the similarity of semantic traces of a malware and its variants. In order to handle obfuscation effects within the trace semantics, we used the semantic trace slicer to produce small traces as candidate semantic signatures for the matching step. Our evaluations of both malware samples and obfuscated variants of malware demonstrated that our semantic signature approach to malware variant detection not only produces very high detection rates but is also able to detect new variants of malware using the existing signature sets in the database without generating false positives.

Chapter 6

Test Data Generation for Malware Executables

As we noticed in Chapter 3, preliminary research work [PCJD07, FG09, LJL10] suggests that a semantic-based approach to malware detection has the potential to overcome various weaknesses of current and traditional approaches in detecting unknown malware variants.

The uniqueness and the quality of the semantic signatures depend on the captured simulation traces. That is, the behaviour of a program is determined from simulation runs of the program paths. However, when the semantic signature of a program is produced from a single simulation run, it is possible that much of the semantic information cannot be observed. This might cause the semantic signature matcher to fail to detect a variant of a malware family. A possible solution to this problem is to adapt the test coverage of the program under analysis. To this end we seek a method of generating tests from program control flow paths. We extend the test data generation method, proposed by Offutt et al. [OJP99], called dynamic domain reduction (DDR). Our extended method automatically generates test data for input data manipulators of AAPL programs. This approach is a promising method for computing test cases by generating a set of semantic signatures for improving the detection of obfuscated malware variants. We present two theoretical results for our method, a test data generation algorithm for low-level code and a correctness notion of the algorithm.

To summarise, the contents of this chapter are as follows:

- We develop an extended version of the DDR algorithm for AAPL that allows us to produce test data cases for executable programs. The algorithm first explores feasible program paths in the control flow graph (CFG) of an AAPL program, driven by the DDR technique and a set of values of program inputs. The algorithm then attempts to find a subset of values of program inputs under which the feasible program paths in the CFG are executed.

- We prove the correctness for our extended DDR algorithm. Our conjecture for the correctness property of the algorithm is that for a given program path in the control flow graph of an AAPL program, the DDR analysis is correct if it finds a subset of values of program inputs such that, when executing the program for test data selected from the subset of program inputs in this set, the set of output states at the end of the path is reached. The set of output states is determined by the reachability semantics of an AAPL program path language, which is based on Cousot's semantics of reachable states for transition systems [CC77].

The rest of this chapter is organised as follows. Section 6.1 provides an overview of the dynamic domain reduction approach for executable code. Preliminary definitions, syntax and semantics of program paths are presented in Section 6.2. In Section 6.3, the dynamic domain reduction method is reviewed. In Section 6.4, the DDR algorithm is extended to handle AAPL programs. The correctness proof of the new algorithm is presented in Section 6.5. Section 6.6 gives related work of software testing in the area of malware analysis and detection. Section 6.7 concludes the chapter.

6.1 Overview

The objective of test coverage or the *coverage-based testing* technique is to *cover* the code under investigation with test cases that satisfy some fixed coverage criteria [CDH+03]. In software testing, a *test case* is a set of program input values under which the testing criteria are met. There are several testing criteria used in software testing which include: statement coverage or node coverage, branch coverage, condition coverage, multiple condition coverage and path coverage. The *path coverage* or *path-oriented* approach [McM04] identifies a set of program control flow paths that cover all program commands (and branches) in the program and

then attempts to generate test data that executes every selected path. Test data generation is the process of identifying program input data that satisfy selected testing criteria.

Increasing test coverage for malicious executable programs could be done by existing test data generation methods [McM04, Har07], such as the random testing method [GKS05], where hundreds of tests are randomly performed with each malware family in different operating environments in an attempt to cover different paths in the program code. Unfortunately, performing and maintaining a large set of random test cases can be a tedious and costly task. Also, testing a malware sample in different operating environments cannot guarantee that certain feasible branches in the code are covered by test cases. That is, many inputs provided by the test cases may have no influence on the program run (e.g. operating system changes). Also, some malware programs do not invoke their malicious commands unless they receive some expected inputs (e.g. file size, date/time).

Symbolic evaluation-based methods [BEL75, VPK04, BCM04] could be used to generate test data. Symbolic execution [Kin75, Kin76] evaluates program statements along a control flow path and produces constraints (equalities or inequalities) on symbolic input values under which a selected path is traversed. However, symbolic evaluation methods require complex algebraic manipulations of intermediate algebraic expressions and have difficulty in dealing with the aliasing problem for pointer analysis [McM04]. Constraint Logic Programming (CLP) techniques [GBR00, GzAP10] have been applied to test data generation (TDG). A CLP technique dynamically builds a constraint system, which consists of program input variables, domains and constraints. Existing test data generation methods are mainly for programs in high-level languages and require modification for use with low-level machine programs. While the existing test data generation methods are assumed to be correct in computing test data for program inputs, no correctness proofs have been presented so far.

In this chapter, we propose a solution that addresses the problem of test data generation of executable code. The solution computes test data for the path coverage criterion. The basic idea is that we identify test data for program paths in a machine-level (i.e., AAPL) program by extending the automated test data generation method called the Dynamic Domain Reduction (DDR) technique presented in [OJP99]. More precisely, the DDR method computes test data using the set of possible values of program input variables. The domain of a program

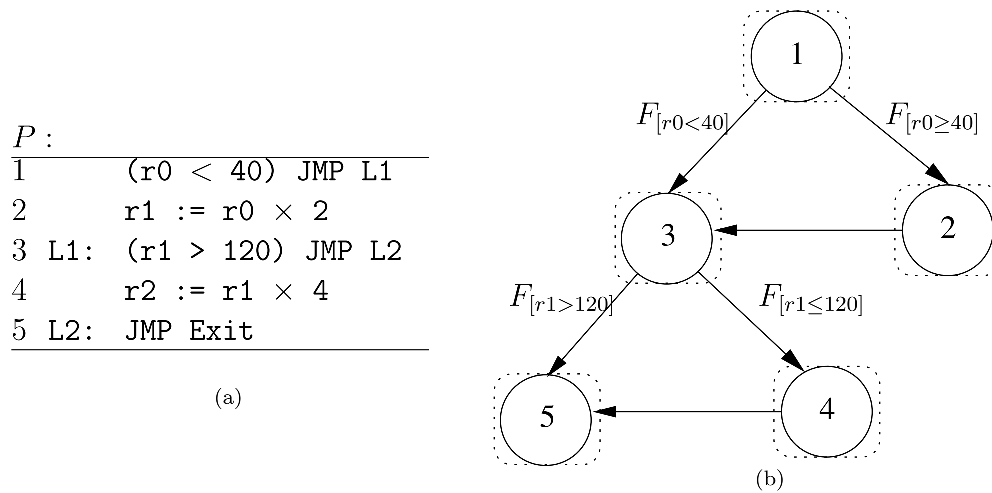


Figure 6.1: A sample of AAPL code and its control flow graph. The dotted regions represent basic blocks of the code.

input variable is the set of values that the variable can hold. This approach is based on the domain of program input variables, a set of control flow graph paths and a domain minimisation method. The domain minimisation method evaluates branch expressions (constraints) in the path by reducing the set of values of program input variables such that the reduced domain of program variables satisfies a path constraint for the path traversed. The minimisation method incorporates a search process to find suitable points for reducing the domain. Thus, the method starts with the domain of initial values of program input variables and attempts to evaluate each constraint along the selected path and reduces the domain of program input variables until the end of the path is reached. Then, the method outputs the domain of program input variables and test data can be selected from this domain that guarantees the execution of the selected path.

6.2 Preliminaries

This section introduces basic concepts used in the rest of the chapter.

The syntax and semantics of the AAPL language are given in Chapter 4. Figure 6.1 shows a sample of AAPL code and its corresponding control flow graph.

Control Flow Graph (CFG). The standard compiler techniques for converting a list of assembly program instructions to a list of basic blocks is a straightforward

Syntactic Categories:

$s \in S$ (AAPL path statements)	$\pi \in \Pi$ (finite CFG paths)
$aop \in \{+, -, *, /\}$	$bop \in \{\&\ ,\ \mid\ ,\ \oplus\ ,\ \ll\ ,\ \gg\}$
$rop \in \{==, \neq, \leq, \geq, <, >\}$	$r \in R$ (AAPL registers)
Val, $n \in \mathbb{Z}$	
$e \in E$ (Expressions)	$b \in B$ (Boolean Expressions)
$dm \in DM$ (data manipulators)	$r \in R$ (AAPL registers)

Syntax:

DM ::= r | *r | *n

Lexpr ::= DM	Rexpr ::= DM | n
BE ::= DM bop n | ¬DM	(Bitwise expressions)
AE ::= DM aop n	(Arithmetic expressions)
E ::= n | DM | AE | BE	(Expressions)
B ::= Lexpr rop Rexpr	(Branch predicate)

Path Statements:

$S ::= dm := e \mid$ `PUSH` $e \mid$ `POP` dm
$\mid$ `JMP` $e \mid$ `CALL` $e \mid$ `RTN` $e \mid$ `SKIP`
$\mid B$ (where B `JMP` E)

CFG Paths:

$\pi ::= < s_1, \ldots, s_k >$ where s_1 and s_k are the entry and an exit nodes, respectively and $\forall i,\ (1 \leq i < k)$, $\exists e \in CFG,\ e = (s_i, s_{i+1})$

Figure 6.2: Syntactic categories and Syntax of the AAPL path language.

algorithmic process [Muc97]. *A basic block* consists of one or more instructions in which there is at most one entry point and one exit point. An entry point is an instruction which accepts control from another basic block. An exit point is an instruction which transfers control to another code in another basic block. The execution of program instructions within a basic block is by definition sequential. The first instruction of each basic block is called the *leader*. The leader instruction may be the start instruction of the program, a target of a branch or an instruction immediately following a branch instruction. To identify the basic blocks that build an AAPL program, we first determine all the leaders, then we include in each leader's basic block all the instructions from it to the next leader. The *control flow graph* of an AAPL program is $CFG(P) = (V, E, s)$ where V is the set of basic blocks, and E (a subset of $V \times V$) is the set of control flow transitions between basic blocks. A CFG has one entry node s and at least one exit node.

A Program Path. A program path in a CFG is a sequence of nodes $<$

Collecting Semantics:

$V_\perp^k$ (the set of bit strings of length $k \in \mathbb{N},\ k > 0$)
σ : $DM \rightarrow V_\perp^k$ (a state)
Σ : $2^{DM \rightarrow V_\perp^k}$ (sets of states (Domains))
$\Im[\![E]\!]$: $\Sigma \rightarrow \wp(V_\perp^k)$
$\Im[\![B]\!]$: $\Sigma \rightarrow \Sigma$
$\Im[\![S]\!]$: $\Sigma \rightarrow \Sigma$

Semantics of E:

$f_e = \Im[\![\cdot\,]\!]$
$f_e[\![n]\!]\Sigma = \{n\}$
$f_e[\![dm]\!]\Sigma = \{v \mid \exists\sigma \in \Sigma,\ \sigma \vdash dm \Rightarrow v\}$
$f_e[\![\neg \mathrm{dm}]\!]\Sigma = \neg f_e[\![dm]\!]\Sigma$
$f_e[\![dm\ bop\ n]\!]\Sigma = f_e[\![dm]\!]\Sigma\ bop\ f_e[\![n]\!]\Sigma$
$f_e[\![\mathrm{AE}]\!]\Sigma = f_e[\![dm\ \mathrm{aop}\ \mathrm{n}]\!]\Sigma = f_e[\![dm]\!]\Sigma\ \mathrm{aop}\ f_e[\![\mathrm{n}]\!]\Sigma$

Semantics of B:

$f_b = \Im[\![\cdot]\!]$
$f_b[\![b]\!]\Sigma = \{\sigma \in \Sigma \mid \sigma \vdash b\ \Rightarrow True\}$

Semantics of S:

$f_s = \Im[\![S]\!]$
$f_s[\![dm := e]\!]\Sigma = \Sigma'$ where $\Sigma' = \{\sigma[dm \mapsto \{v\}] \mid \sigma \in \Sigma, \{v \in f_e[\![e]\!]\Sigma\}\}$
$f_s[\![\texttt{PUSH}\ e]\!]\Sigma = \Sigma'$ where $\Sigma' = \{\sigma[dm \mapsto \{v\}] \mid \sigma \in \Sigma, SP \mapsto SP - 1, dm = SP, \{v \in f_e[\![e]\!]\Sigma\}\}$
$f_s[\![\texttt{POP}\ dm]\!]\Sigma = \Sigma'$ where $\Sigma' = \{\sigma[dm \mapsto \{v\}] \mid \sigma \in \Sigma, \{v \in f_e[\![SP]\!]\Sigma\}, SP \mapsto SP + 1\}$
$f_s[\![\texttt{JMP}\ e]\!]\Sigma = \Sigma$
$f_s[\![\texttt{CALL}\ e]\!]\Sigma = \Sigma$
$f_s[\![\texttt{RTN}\ e]\!]\Sigma = \Sigma$
$f_s[\![B]\!]\Sigma = \Sigma'$ where $\Sigma' = f_b[\![B]\!]\Sigma \cap \Sigma$

Figure 6.3: Semantics of the AAPL path language.

$n_1, n_2, \ldots, n_k >$, such that $n_1 = s$, n_k is an exit node in a program CFG and $\forall i,\ 1 \leq i < k,\ (n_i, n_{i+1}) \in E$. A path is executable (or *feasible*) if there exists a program input for which the path is traversed during program execution, otherwise the path is unexecutable (or *infeasible*). A finite program path that begins with the entry node s and ends with an exit node is called a *complete* path. Otherwise, it is called an *incomplete* path. For instance, the path $< 1, 2, 3, 5 >$ is a complete (finite) path in the CFG of Figure 6.1 on page 143; however, the path $< 1, 2, 3, 4 >$ is an incomplete path.

Program Input Variables. Program registers and direct memory locations (i.e, addressing memory locations with an immediate offset, a register or a register

with an offset) are used to perform data manipulations during execution, such as retrieving and storing data from memory. We use the term *data manipulators* (Definition 4.3 on page 71) to denote registers and memory locations that are used to process the program inputs. Thus, a *data manipulator* is a program register or memory location used to perform data definition and manipulation operations. A program input variable of a program P is a data manipulator that appears in an instruction in P. Throughout the chapter, we use these terms interchangeably, namely: data manipulators and input variables. Also, we allow input data manipulators to be of type unsigned integer (and can be represented as a set of bits (i.e. unsigned binary integer)). Each input data manipulator is initially assigned a *domain* of as large a set of possible values as the data manipulator can hold. We let a data manipulator be a 32-bit unsigned integer (i.e. 2^{32} possible values).

The Syntax and Collecting Semantics for Program Paths. The syntax and semantics of AAPL program path constructs are shown in Figs 6.2 on page 144 and 6.3 on the previous page, respectively. The semantics are useful when constructing our correctness proof for the extended DDR algorithm. Since we are interested to compute the domains of program input variables (data manipulators) and produce a test data in which a path is exercised, we consider a subset of AAPL syntax and semantics (in Chapter 4) which affects the values of data manipulators in a particular program path. For instance, from AAPL action commands, we consider the assignment commands, `PUSH` and `POP` in the program path semantics. However, other action commands such as `CALL E`, `JMP E` and `SKIP` do not change the values of program input variables within a path execution. The program path syntax contains a statement of type *branch predicate* B, where B is part of conditional command in AAPL language, i.e. $C_B := B$ `JMP` E. The branch predicate statement may update the sets of states of a program path such that the outcome of the predicate is true.

A *bitwise* expression *BE* is created using a *bitwise* operator (*bop* $\in \{\&\ ,\ |\ ,\ \oplus\ ,\ \ll\ ,\ \gg\}$) (i.e. AND, OR, XOR, logical shift left and logical shift right, respectively) with two operands (i.e. a data manipulator and a constant). Also, a bitwise expression can have one operand (i.e. *DM*) in the case of the unary operation, i.e. the bitwise NOT that performs logical negation on each bit of a data manipulator (e.g. $\neg$DM), computing the ones' complement of the given binary value. The bitwise operators work on the binary representation of operands' values and change individual bits of a destination data manipulator. An *arithmetic* expression *AE* can contain any of the arithmetic operators (*aop* $\in \{+, -, *, /\}$) and a pair of

operands (i.e. DM and n operands). An expression E, in this path language, can represent a constant value, a data manipulator, a bitwise or an arithmetic expression. A *relational* operator forms what we refer to as a branch predicate expression B. A branch predicate involves exactly two operands (i.e. Lexpr and Rexpr operands). Note that an operand on the left is always a data manipulator and an operand on the right can either be a data manipulator or a constant value.

Figure 6.3 on page 145 shows the collecting semantics [CC76] of arithmetic and Boolean expressions and path statements. We let V^k be the set of bit strings of length $k \in \mathbb{N}$, $k > 0$. Also, we let a state to have type $DM \to V^k$, i.e. is a map from the set of data manipulators DM to the set of bit strings of length k, including $\perp$ (which is undefined). Thus, we use $\sigma_0, \sigma_1, \sigma_i, \ldots, \sigma', \sigma'' \in DM \to V^k$ to denote states which represent the information collected about the bindings of program data manipulators to their possible values. Also, we let $\Sigma, \Sigma_0, \Sigma_1, \Sigma_i, \ldots, \Sigma', \Sigma' \in 2^{DM \to V^k_\perp}$ to represent the set of states. The semantics of an expression defines the possible values that the expression can evaluate to in a given set of states (environments). The semantics of a Boolean expression B defines the subset of possible states for which the Boolean expression may evaluate to true. During the evaluation of an assignment statement in a path π, a new state can be constructed and included to the set of states $\Sigma' = \sigma \cup \Sigma$, where σ is the newly created store. We let Σ_P be the set of all possible states of a program P. Also, we let $[\![\pi]\!]$ be the path semantics for a finite program path π. Then the set of all reachable states for π according to the semantics is then $[\![\pi]\!]\Sigma_P = \{\sigma' \mid \forall\sigma \in \Sigma_P,\ \sigma' = [\![\pi]\!]\sigma\}$.

Branch Predicates. An edge corresponds to a possible transfer of control from one basic block to another basic block. A *branch* is an outgoing edge of a conditional jump instruction (i.e. $C_B :=$ `B JMP E`). A *branch predicate* is a label of a branch that describes the condition (constraint) on which that branch is traversed. We let F denote a filtering function that describes the branch predicate of a conditional jump instruction. For instance, in the program of Figure 6.1 on page 143 branch (1,2) is labelled $F_{[r0 \geq 40]}$, which means that in order to traverse the branch, the constraint $r0 \geq 40$ must be satisfied; similarly, for traversing branch (3,5), which has the label $F_{[r1 > 120]}$, the constraint $r1 > 120$ must be satisfied. Note that for the CFG in Figure 6.1 on page 143, node 1 is the entry statement of the program.

Domain of Program Inputs. Let $I = (dm_1, dm_2, dm_i \ldots, dm_n)$ $(1 \leq i \leq n)$ be a vector of input variables (i.e. data manipulators) of program P. The domain $Đ_{dm_i}$

of input variable dm_i is the set of all values which dm_i can hold. By the *domain* Đ of the program P we mean a cross product, $Đ = Đ_{dm_1} \times Đ_{dm_2} \times Đ_{dm_3} \times \ldots \times Đ_{dm_n}$, where each $Đ_{dm_i}$ is the domain for input variable dm_i. Also, a *program input* is a single point $\mathtt{x}$ in the n-dimensional input space, such that $\mathtt{x} = (d_1, d_2, d_i, \ldots, d_n)$, $d_i \in Đ_{dm_i}$ $(1 \leq i \leq n)$ and $\mathtt{x} \in Đ$. The collecting semantics that we have defined for the AAPL path language represents the behaviours of AAPL code and allows to specify the domains of program input variables. We could consider a program P as a function, $P : Đ \rightarrow Đ'$, where $Đ \subseteq \Sigma_P$ as it maps the set of all possible inputs to the set of all possible outputs $Đ'$. Also, $Đ'$ can be better described as a reachable set of program outputs of a given path as follows:

Reachable Output States $\mathcal{R}$. Let π be a finite path in the CFG of a program, P, and $Đ \subseteq \Sigma_P$ be the set of possible initial states of P. $\mathcal{R}_\pi = [\![\pi]\!]Đ$ represents all possible reachable output states of program data manipulators that could be produced when executing the path π with initial program input Đ. We use the collecting semantics of the path language to describe how the evaluation of possible reachable states $\mathcal{R}_\pi$ can be produced for an initial domain of a program and any given path π.

6.3 A Test Data Generation Problem: DDR Approach

At this point let us define the goal of the *path-oriented* automatic test data generation problem: given a path $\pi = < n_1, n_2, \ldots, n_k >$, which is a finite path in the CFG of a program, find a program input $\mathtt{x} \in Đ$ on which π will be traversed (executed). The dynamic domain reduction approach reduces this problem to a solution where a minimisation search process, referred to as the *domain reduction* method, applied to the initial domain of the program to compute a subset $Đ_\pi$ of the domain from which any program input $\mathtt{x}$ is selected, will execute π. If π is traversed (and, hence, it is a feasible path), the subset, $Đ_\pi$, that is computed by the DDR method is the solution to the test data generation problem; if not, we consider the method to be unsuccessful in finding a solution for π. Thus, we assume that there exists a path generator that takes the CFG of an AAPL program as an input and produces a set of finite paths of interest. Then, given an initial domain for the program, we apply the extended DDR approach to compute a test data solution.

Dynamic domain reduction was developed to handle most problems that exist with *constraint-based* test (CBT) data generation and the *symbolic evaluation* technique, such as those associated with handling arrays, loops and nested expressions. The DDR approach incorporates parts of previously existing testing techniques: CBT [DO91, DO93], dynamic testing [Kor90] and symbolic evaluation [Kin75]. The main development of the DDR approach is that it uses a domain reduction method for deriving a subset of program inputs, which represent conditions under which a path will be executed. Also, the DDR approach uses a new backtracking search method to discover other possible values of inputs when a condition is not met. We provide an overview of the DDR approach, and describe the original algorithm in Section 6.3.1. Section 6.3.2 provides an overview of the main procedures of DDR. Details of the algorithm can be obtained from [OJP99], which was developed for high-level programs (e.g. Java and C/C++ code) with basic arithmetic expressions and numerical data types.

6.3.1 Description of DDR Analysis

The original DDR analysis uses constraints derived from the path to progressively reduce domains of program input variables until test data that satisfy these constraints are identified. The method automatically finds values by walking through the path, using one branch predicate at a time and reducing the domain of the program step by step. The method introduces two new techniques for generating path-oriented test data. The techniques are implemented in different procedures in the DDR algorithm. Figure 6.4 on page 151 shows the flow diagram of the DDR method. In the diagram, circles represent inputs and outputs of the DDR analysis, rectangles are the DDR steps and diamonds are branches during a path evaluation process.

Domain-based Symbolic Execution. This technique evaluates the path symbolically, and as it processes the constraints along the path, the domains of program input variables are modified (i.e. the domains may be reduced for branch predicates and updated for assignments) to reflect conditions and assignments. Program instructions such as assignments are evaluated by creating special symbolic variables and expressions rather than actual values. DDR analysis maintains a symbolic state to track and update the symbolic expressions for a program path. When a new expression is encountered, the analysis creates a domain of possible

values for the expression. A new domain of an expression is computed from the domains of program input variables involved in the expression. Initially, the domain of a program input variable may be assigned minimum and maximum possible values for the host machine, or limited to a reasonable input specification range. The method represents the domain, $Đ_{dm}$ of a program input variable dm by a top and bottom limits (e.g. $[l_{dm}, u_{dm}]$, where l_{dm} and u_{dm} are the minimum and maximum values in the domain, respectively). A domain may consist of sub-domains where each sub-domain represents a set of contiguous values. For instance, the domain $\{1, 2, 3, 6, 7, 8\}$ is represented in two sub-domains as $< [1, 3], [6, 8] >$. The method uses a domain *reduction* process to select a point in the domain of a program input variable at which the domain is *split*. Also, whenever each constraint has been satisfied, the method *updates* the domains of program input variables such that their current values are consistent with the path conditions taken so far. The following example demonstrates how the method finds test data input using the domain splitting, reduction and update techniques:

Example 6.1. *To illustrate the DDR method. Consider a path in the control flow graph of the program in Figure 6.5 on page 152,* $\pi = < 1, 2, 3, 4, 5 >$*. The goal of the DDR method is to find a program input* $\boldsymbol{x}$ *(i.e. values for the input data manipulators (variables)* $I = \{r0, r1, r2\}$*) which causes* π *to be traversed. We assume that all input variables receive initial values for their domains; suppose the following values have been assigned:* $l = 1$ *and* $u = 100$*. Thus, the initial domains of the program input variables are as follows:*

$$Đ_{r0} = [1, 100],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100]$$

Since nodes 1 *and* 3 *are conditional jump instructions in the program, the constraints on edges (1,2) and (3,4) (labelled by* $F_{[r0 \geq 40]}$ *and* $F_{[r1 \leq 120]}$ *in the CFG, respectively) are included in the path. The method starts with the first constraint in the path and attempts to satisfy it, such that all possible values of* $r0$ *must be greater than or equal to the value* 40*. To find a solution for this constraint, the method performs a search process to determine a point in* $Đ_{r0}$ *such that a subdomain could be found in which all values in that subdomain agree with the constraint. The domain of* $r0$ *is split at* 50 *(the split is referred to as the* split point*), and the subdomain* $[50, 100]$ *becomes the current input domain of* $r0$*. The new domain of* $r0$ *contains all possible values that are greater than or equal to* 50*. Note that this reduction process also discards some of the valid values of* $r0$ *from its domain (i.e.* $[40, 49]$*). The method proceeds to the assignment node* 2*.*

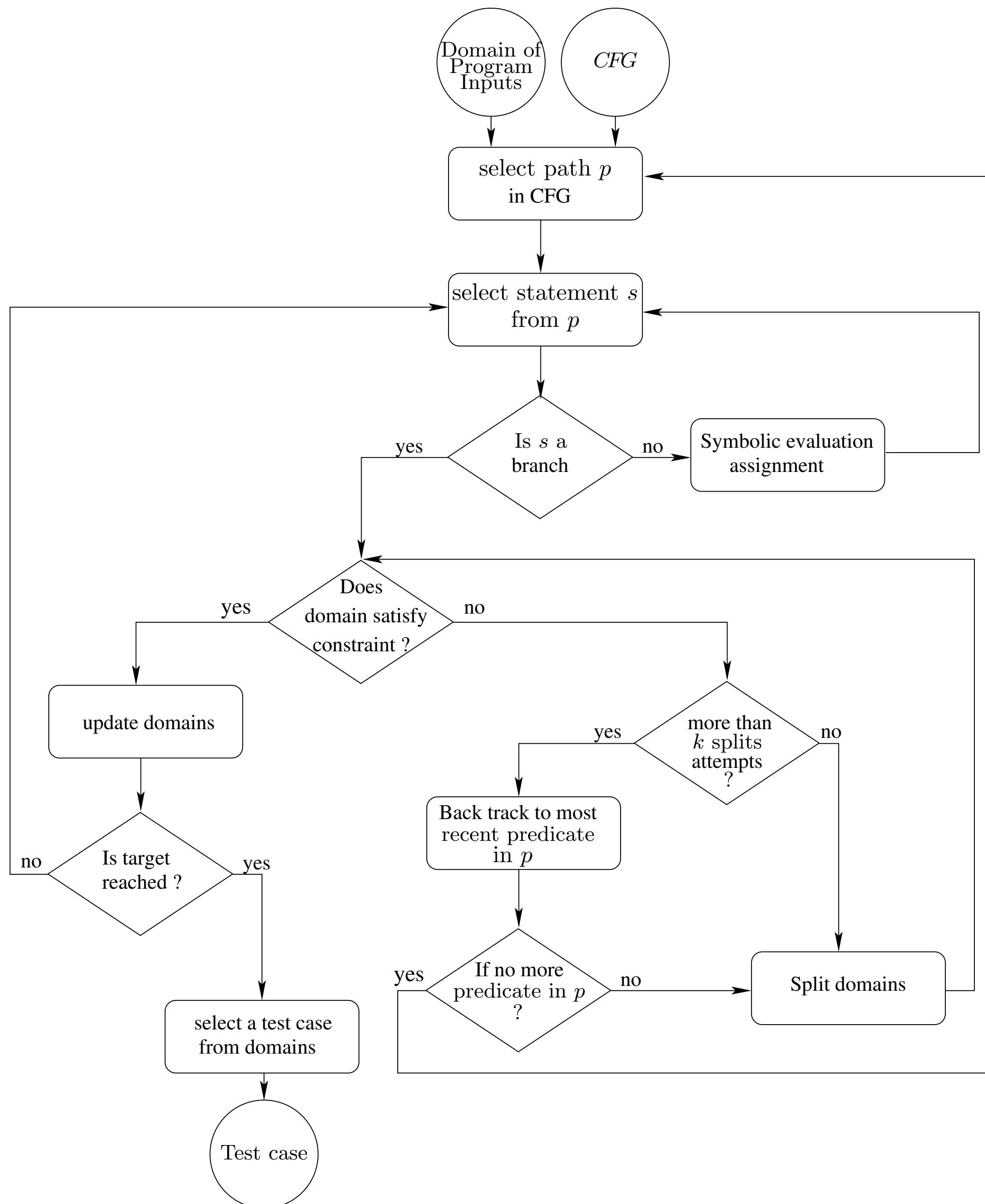


Figure 6.4: The flow diagram for the DDR algorithm.

For the expression $2 \times r0$*, the method uses the domain of* $r0$ *to symbolically evaluate the expression and create a domain for the expression. The new domain for the expression* $2 \times r0$ *is* $[100, 200] =< [100, 100],\ [102, 102], \ldots,\ [200, 200] >$*, i.e.* $Đ_{2\times r0} = \{2 \times 50, 2 \times 51, \ldots, 2 \times 100\} = \{100, 102, \ldots, 200\}$.

Next, the method evaluates the constraint $r1 \leq 120$ *on edge (3,4). To satisfy this constraint, the domain of the expression* $2 \times r0$ *must be reduced and the split point* 110 *is selected. After the reduction process, the domain of the expression*

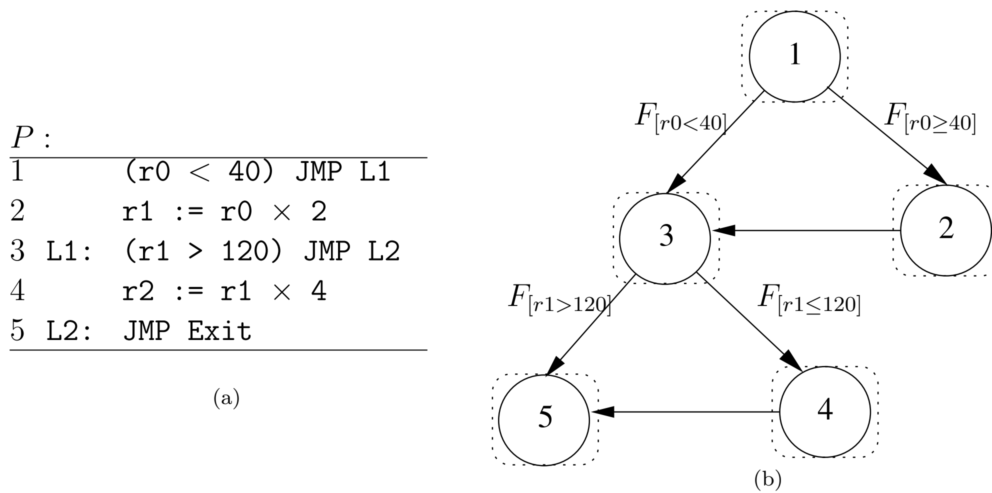


Figure 6.5: A sample of AAPL code and its control flow graph revisited in Example 6.1.

$Đ_{2\times r0}$ *becomes* $[100, 110]$*. Once the constraint is satisfied by the computed domain, the method updates the current domain of* $r0$ *(as* $r1$ *is symbolically assigned the expression* $2\times r0$ *from the previous step) to reflect the new values of the expression. Thus, the domain* $Đ_{r0}$ *becomes* $[50, 55]$*. Note that after this step,* $Đ_{r1}$ *is not changed due to the domain-based symbolic execution performed for node* 2*. The method evaluates the remaining instructions on nodes 4 and 5 where a new domain is created for the expression,* $r1 \times 4$*. At node 4, the symbolic expression* $r1 \times 4$ *is assigned to* $r2$ *in the symbolic state. The statements at Nodes 4 and 5 do not affect the domains of the program data manipulators. Thus, the output of the method at the end of the path is the set of domains of* I*. The evaluation steps of the DDR*

method on π are shown below:

$Đ_{r0} = [1, 100],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100]$

1 `r0 ≥ 40` *(split point for $r0$ selected is 50)*

$Đ_{r0} = [50, 100],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100]$

2 `r1 := r0 × 2`

$Đ_{r0} = [50, 100],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100],$

$Đ_{r0\times 2} =< [100, 100], \ldots, [200, 200] >$

3 `r1 ≤ 120` *(split point for $r0 \times 2$ selected is 110)*

$Đ_{r0} = [50, 55],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100],$

$Đ_{r0\times 2} =< [100, 100], \ldots, [110, 110] >$

4 `r2 := r1 × 4`

$Đ_{r0} = [50, 55],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100],$

$Đ_{r0\times 2} =< [100, 100], \ldots, [110, 110] >,$

$Đ_{r1\times 4} = [4, 400]$

5 `JMP Exit`

$Đ_{r0} = [50, 55],\ Đ_{r1} = [1, 100],\ Đ_{r2} = [1, 100],\ Đ_{r0\times 2} = [100, 110],$

$Đ_{r1\times 4} = [4, 400]$

A program input can be selected from the domains of program input data manipulators such as $\boldsymbol{x} = (r0 = 50,\ r1 = 1,\ r2 = 100)$, which will execute the path π.

As we have noticed in Example 6.1, when the DDR method starts processing a given path in the control flow of a program, the initial domain of the program may contain some program inputs that will execute the path. The method takes a conservative approach in producing a *safe* solution for the test data generation problem. A safe solution means that the method finds a subset of the program input domain such that the path will be taken for every program input $\mathbf{x}$ in this subset. In Section 6.5 we prove that for every solution the DDR method generates, the method is correct with respect to this property.

A Backtrack Search-based Process. The method introduces a second technique, called a backtrack search process, to handle cases when a program input

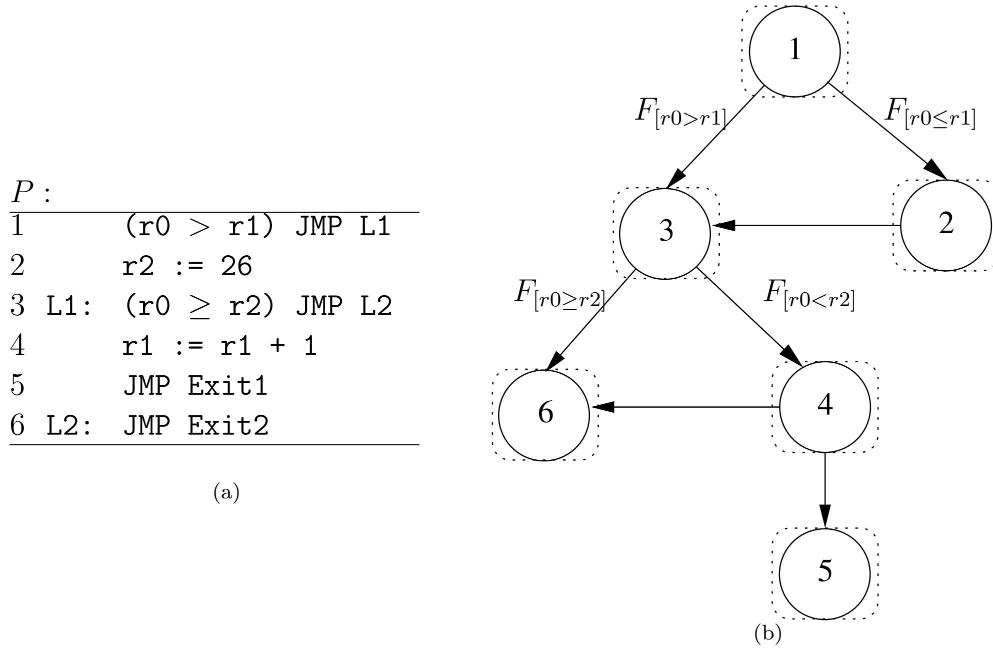


Figure 6.6: A sample of AAPL code and its control flow graph for Example 6.2.

that will traverse a path is not included in the resulting domains of the program input variables. That is, as the method reduces the domains of program input variables to satisfy current constraints along the path, there may be situations where later constraints cannot be satisfied by the current domain of the program input variables but different, previous choices of split points may produce a suitable domain. The backtrack search-based process undoes the previous steps of the domain reduction for former constraints and computes a different split point to give a new reduction decision on the domains of program input variables.

Example 6.2. *Consider a path $\pi =< 1, 2, 3, 6 >$ in the control flow graph of the program in Figure 6.6. Let us assume that the initial domains of the program input variables are:*

$$Đ_{r0} = [10, 30],\ \ Đ_{r1} = [20, 50],\ \ Đ_{r2} = [0, 100]$$

The first node in the path corresponds to a jump instruction with the constraint $r0 \leq r1$ on the edge $(1, 2)$. The method generates a split point to reduce the domains of $r0$ and $r1$ in an attempt to satisfy the constraint. The split point calculated is 25 and the domains are reduced to $Đ_{r0} = [10, 25]$ and $Đ_{r1} = [26, 50]$. At node 2, the assignment instruction $r2 := 26$ is evaluated symbolically such that a domain for the new expression (a constant) 26 is created, $Đ_{26} = [26, 26]$. Note that

the domain of $r2$ *is not modified at this step. Next, the constraint* $r0 \geq r2$ *from node* 3 *is handled in an attempt to reduce* Đ_{r0} *and* Đ_{r2}. *However, the possible values of* $r0$ *in its domain are less than all possible values of* $r2 = 26$ *(i.e. contains a single value,* $\text{Đ}_{26} = [26, 26]$*). Therefore, the method cannot find a suitable split point and it has to go back to the previous split-point step and re-compute a different split point value. The method automatically resets the domains of the program input variables to their initial values and a new split point is calculated for the constraint at node* 1. *The new split point is* 13 *and the domains are reduced as* $\text{Đ}_{r0} = [10, 13]$ *and* $\text{Đ}_{r1} = [14, 50]$. *However,* $\text{Đ}_{r0} = [10, 13]$ *is still less than* $\text{Đ}_{26} = [26, 26]$*, and the method performs another backtrack step to node* 1 *and resets the domains again. A third split point is computed to be* 28 *for the constraint* $r0 \leq r1$*, which results in reducing the domains for* $r0$ *and* $r1$ *to* $\text{Đ}_{r0} = [10, 28]$ *and* $\text{Đ}_{r1} = [29, 50]$. *The method proceeds to node* 2 *and* $\text{Đ}_{26} = [26, 26]$. *Then at node* 3*, the domains of* $r0$ *and* 26 *allow the constraint* $r0 \geq r2$ *to be satisfied. A split point is computed to be* 26 *such that* $\text{Đ}_{r0} = [27, 28]$ *and* Đ_{26} *remains as* $[26, 26]$. *The domain of* $r2$ *remains the same as the constant value* 26 *is assigned (symbolically) to* $r2$ *as an expression. The method reaches the end of the path and all constraints of the path were successfully satisfied. Below are the steps taken to reduce the domains of*

program input variables:

$Đ_{r0} = [10, 30],\ Đ_{r1} = [2, 50],\ Đ_{r2} = [0, 100]$

1 `r0` $\leq$ `r1` *(split point for* $r0$ *and* $r1$ *selected is* 25*)*

$Đ_{r0} = [10, 25],\ Đ_{r1} = [26, 50],\ Đ_{r2} = [0, 100]$

2 `r2 := 26`

$Đ_{r0} = [10, 25],\ Đ_{r1} = [26, 50],\ Đ_{r2} = [0, 100],\ Đ_{26} = [26, 26]$

3 `r0` $\geq$ `r2`

- *no split point can be computed, backtrack to Node* 1

1 `r0` $\leq$ `r1` *(split point for* $r0$ *and* $r1$ *selected is* 13*)*

$Đ_{r0} = [10, 13],\ Đ_{r1} = [14, 50],\ Đ_{r2} = [0, 100]$

2 `r2 := 26`

$Đ_{r0} = [10, 13],\ Đ_{r1} = [14, 50],\ Đ_{r2} = [0, 100],\ Đ_{26} = [26, 26]$

3 `r0` $\geq$ `r2`

- *no split point can be computed, backtrack to Node* 1

1 `r0` $\leq$ `r1` *(split point for* $r0$ *and* $r1$ *selected is* 28*)*

$Đ_{r0} = [10, 28],\ Đ_{r1} = [29, 50],\ Đ_{r2} = [0, 100]$

2 `r2 := 26`

$Đ_{r0} = [10, 28],\ Đ_{r1} = [29, 50],\ Đ_{r2} = [0, 100],\ Đ_{26} = [26, 26]$

3 `r0` $\geq$ `r2` *(split point for* $r0$ *selected is* 26*)*

$Đ_{r0} = [27, 28],\ Đ_{r1} = [29, 50],\ Đ_{r2} = [0, 100],\ Đ_{26} = [26, 26]$

6 `JMP Exit`

$Đ_{r0} = [27, 28],\ Đ_{r1} = [29, 50],\ Đ_{r2} = [0, 100],\ Đ_{26} = [26, 26]$

A program input can be arbitrarily selected from the domains of the program input data manipulators $Đ$ *(*$= Đ_{r0} \times Đ_{r1} \times Đ_{r2}$*), which was produced by the method. Thus, a program input* $\boldsymbol{x} = (r0 = 27,\ r1 = 35,\ r2 = 0)$ *will traverse* π*.*

6.3.2 DDR Procedures

From the previous examples, we observe that the DDR analysis uses a backtracking search process to find a different split point for reducing input domains whenever a later constraint in the path cannot be satisfied. Also, when an update is required,

Algorithm 6.1: Sketch of the DDR algorithm.

```
for each complete path in the CFG of a program under test:

  for each node in the path:
    if the node corresponds to a conditional instruction:
      read the constraint on the edge.
      evaluate the left and right sub-expressions,
        ExprDomain()
      search for suitable domains to satisfy the constraint,
        DomFitCnst()
        GetSplit()
      if FoundSuitableDom()
        Update()
      else
        apply the backtrack process.
    else the node corresponds to an assignment instruction:
      apply domain-based symbolic execution.

  if the path is traversed and constraints are satisfied:
    the path is feasible.
    generate test data from the domain of the program.
  else
    the path may not be feasible.
```

the analysis propagates the update values of an expression back to the domains of program input variables in the left- and right-hand sides of the expression.

Next we describe the main procedures implemented in the DDR algorithm:

- `FoundSuitableDom`
- `DomFitCnst`
- `GetSplit`
- `Update`
- `ExprDomain`

A sketch of the DDR algorithm with its main procedures is shown in Algorithm 6.1. The full detailed description of the algorithm is available in [OJP94].

FoundSuitableDom. For a given constraint (branch predicate) in a CFG path, the procedure finds suitable domains for program input variables that satisfy the constraint. The input to this procedure is a branch predicate expression B and the current domains of the Lexpr and Rexpr sub-expressions of the branch predicate. If the constraint is satisfied by the current domains then the procedure returns the value *True* and the domains of the program input variables that are used in the sub-expressions are updated. If the procedure returns the value *False*, then the constraint is not satisfied by the current domains. The procedure passes the constraint and a copy of the current domains of the program input variables and expressions to procedure *DomFitCnst* to find a suitable subset of current domains that satisfy the constraint.

DomFitCnst. This procedure takes a constraint as an input and determines if the current domains of the sub-expressions in the constraint satisfy it. The constraint is of the form *Lexpr rop Rexpr* where Lexpr and Rexpr are the left and right sub-expressions, respectively, and *rop* is a relational operator. The procedure handles two different constraint types: *DM* `rop` n (and n `rop` *DM*) and *DM* `rop` *DM*. If the domains of the two sub-expressions do not satisfy the constraint then the procedure attempts to modify and to update the domains. The procedure returns the value *True* when the update of the domains of both expressions is successful. To modify program input domains and to update changes on domains, the procedure calls the procedures *GetSplit* and *Update*, respectively.

GetSplit. This procedure accepts the domains of two expressions (e.g. expressions x, y) and returns a split point. A split point is a value in two given domains that is used to reduce the domains such that the modified domains satisfies the constraint. Each domain is represented by its bottom and top values i.e. $Đ_x = [l_x, u_x]$ and $Đ_y = [l_y, u_y]$, where l and u are the minimum and the maximum values in a domain, respectively. The procedure computes a split point sp for two given domains under one of four cases:

- if $(l_x \geq l_y) \wedge (u_x \leq u_y)$ then $sp = (u_x - l_x) * i + l_x$
- if $(l_x \leq l_y) \wedge (u_x \geq u_y)$ then $sp = (u_y - l_y) * i + l_y$
- if $(l_x \geq l_y) \wedge (u_x \geq u_y) \wedge (u_y \geq l_x)$ then $sp = (u_x - l_y) * i + l_y$
- if $(l_x \leq l_y) \wedge (u_x \leq u_y) \wedge (l_y \leq u_x)$ then $sp = (u_y - l_x) * i + l_x$

Note that in the last two cases, a third condition after the second $\wedge$ clause is added to ensure that the domains overlap, but neither is contained in the other. *GetSplit* uses the index i ($0 < i < 1$) as a search point for a split between two domains (i.e. $i \in \{1/2,\ 1/4,\ 3/4,\ 1/8,\ \ldots\}$) where i is initialised to 1/2. The split point is moved halfway in one direction then the other till a successful split point is found, allowing domains to be reduced or a predetermined maximum number of search choices have been made.

Update. This procedure takes an expression e and its current domain $[l_e, u_e]$ and updates the domains of the program input variables, which are used in the expression. The procedure propagates back the new values of the expression's domain to its sub-expressions. The domain of a program input variable is reduced if the new bottom and top values are contained within the current domain of the variable, otherwise the current domain is not modified. Also, if there are any changes that are necessitated after satisfying a constraint (a branch predicate expression), the procedure recursively updates the domains of program input variables in the expression. When all domains are reduced successfully, i.e. a domain update is feasible, the procedure returns *True* to function *DomFitCnst*.

ExprDomain. This procedure computes a possible domain for a new expression encountered in the path analysis. The procedure accepts a new expression and the current domains of the program input variables and expressions. A new domain of the expression is evaluated from its sub-expressions' domains.

6.4 Description of the Extended DDR Algorithm

Our extended version of the DDR algorithm contains additional modules, which handle the syntax and semantics of AAPL program paths shown in Figure 6.2 on page 144. In particular, the expressions in AAPL are evaluated by two different expression evaluation modules depending on the type of expression, i.e., *arithmetic* and *bitwise* expressions. In this section, we first explain the new modules (i.e. the refinements of Algorithm 6.1 on page 157) and then present pseudocode for the algorithms.

The extended version of the DDR algorithm includes the following modules (within `ExprDomain` and `Update` procedures):

1. Domain evaluation module:
   - Data manipulator evaluation.
   - Arithmetic expression evaluation.
   - Bitwise expression evaluation.
2. Domain update module for:
   - Program input data manipulators' domains.
   - Bitwise expressions' domains.

6.4.1 Expression Domain Evaluation Procedure

The procedure in Algorithm 6.2 on the next page computes a possible domain for a given expression. The set of current domains *TDom* of program inputs and the expression E are passed to the function as inputs. A new domain of the expression E is computed from the domains of operands and the operator used in the expression. When E is a data manipulator *DM* (in Algorithm 6.2 on the following page, line 11), the procedure computes the top and bottom values of the domain, $Đ_{DM}$ of *DM* by finding the upper and the lower subdomains in $Đ_{DM}$, i.e., $Đ_n$ =`USubDom(`$Đ_{DM}$`)` and $Đ_1$ =`LSubDom(`$Đ_{DM}$`)`, respectively. The top and bottom values of the domain of *DM* are then determined by taking the maximum and minimum values in these subdomains, i.e. $u_{DM} = \max(Đ_n)$ and $l_{DM} = \min(Đ_1)$, respectively. Thus, the domain of E is created with top and bottom values, $u_E = u_{DM}$ and $l_E = l_{DM}$. This procedure handles arithmetic and bitwise expressions (in Algorithm 6.2 on the next page, line 17 and line 19, respectively) using two functions `GetAExprDom` (in Algorithm 6.3 on page 163), and `GetBExprDom` (in Algorithm 6.4 on page 165), respectively. Notice that when the expression is a constant, the procedure assigns a new domain of values from n. At the end of the procedure, the new domain of a given expression is created in *TDom* of the program and it is passed to the caller procedure, which checks if the new computed domain of the expression fits a constraint in the program.

Algorithm 6.2: `ExpDomEval(`E`)` evaluates the domain of a program expression.

1: **Input:** an expression E to be evaluated, where $E ::= n \mid DM \mid AE \mid BE$ and the set of domains of program expressions and inputs in $TDom$.
2: **Output:** evaluates the domain of E and stores it in $TDom$.
3: **procedure** `GetAExprDom(`AE`,`$TDom$`)` is presented in Algorithm 6.3 on page 163
4: **procedure** `GetBExprDom(`BE`,`$TDom$`)` is presented in Algorithm 6.4 on page 165
5: $USubDom(Đ_{DM})$ finds the upper subdomain in a given domain
6: $LSubDom(Đ_{DM})$ finds the lower subdomain in a given domain
7: **begin** `ExpDomEval(`E`)`
8: **if** E is n **then**
9: $u_E = n$ and $l_E = n$
10: $Đ_E = [u_E, l_E]$
11: **else if** E is DM **then**
12: $u_{DM} = \max(Đ_n)$ where $Đ_n = USubDom(Đ_{DM})$,
13: $l_{DM} = \min(Đ_1)$ where $Đ_1 = LSubDom(Đ_{DM})$
14: $u_E = u_{DM}$
15: $l_E = l_{DM}$
16: $Đ_E = [u_E, l_E]$
17: **else if** E is AE **then**
18: $Đ_E =$ `GetAExprDom(`AE`,`$TDom$`)`
19: **else if** E is BE **then**
20: $Đ_E =$ `GetBExprDom(`BE`,`$TDom$`)`
21: **end if**
22: $TDom \rightarrow TDom \cup Đ_E$
23: **return** $Đ_E$
24: **end** `ExpDomEval(`E`)`

1. Evaluating Domains of Arithmetic Expressions

The function `GetAExprDom` uses the domain of program inputs to symbolically evaluate arithmetic expressions and produces a new domain of a given expression. This function is similar to the `ExprDomain` algorithm presented in [OJP99] except that `ExprDomain` evaluates expressions recursively by finding the domains of operands at the leaves of the expression and propagating these domains up to compute the domain of the expression. `GetAExprDom` finds the domain of the expression by applying the arithmetic operation to the domains of the operands (i.e. a data manipulator and a constant).

2. Evaluating Domains of Bitwise Expressions

The function `GetBExprDom` finds a possible domain for a bitwise expression. The inputs to this function are a bitwise expression BE and the set of current domains

of program input variables and expressions, *TDom*. Five bitwise operations are considered for AAPL statements in this function: bitwise NOT ($\neg$), AND (`&`), OR (|), XOR ($\oplus$), shift left (`<<`) and shift right (`>>`). The bitwise NOT is a unary operation that performs bitwise negation on each bit of the operand. A bitwise expression could be constructed from other bitwise operations consisting of a data manipulator and a constant as operands in the expression.

The bitwise operations, AND, OR and XOR, require a pair of operands of equal length and produce a result of the same length by performing the bitwise operation on each pair of corresponding bits. The left and right shift operators are logical shifts. Bitwise expressions of shift operations consist of a data manipulator and a constant (repeat) value, which determines the number of times the single bit shift operation is repeated. The shifts operate on the binary representation of an unsigned integer number such that when the bits are shifted, some bits will be discarded and zeros are shifted in (at the appropriate end). Since a bitwise left shift (i.e. a left shift by one) is equivalent to multiplication by 2 and a bitwise right shift (a right shift by one) is equivalent to division by 2, the function makes use of the procedure `GetAExprDom` to compute the result. The function evaluates the domain of a given bitwise expression by determining the domains of the program input (*data manipulator*) and the constant, n, and computing the bitwise operations with these domains.

6.4.2 Update Domains Procedure

1. Updating Domain Values for Bitwise Expressions

The function `UpdateBEDomVal` accepts three input parameters from the main procedure `UpdateDomVal` in Algorithm 6.5 on page 166: a bitwise expression *BE*, the top and bottom values u, l of the new domain of values of *BE* and the set of current input domains of program variables and expressions, *TDom*. Given a bitwise expression *BE*, the function finds possible domain limits (u'_{DM}, l'_{DM}) of the data manipulator *DM* involved in *BE* using the domain of *BE* and the constant value used (if any) in the expression *BE*. At the end of the function, the new possible domain limits of *DM* are passed to the main function so the function `UpdateDomVal` is called after `UpdateBEDomVal` to update the changes back to the current domain of *DM*. The function handles all types of bitwise expression defined in the syntax

Algorithm 6.3: `GetAExprDom(`AE`,`$TDom$`)` evaluates a new domain for an arithmetic expression AE.

1: **Input:** an arithmetic expression, $AE := L$ `aop` n and the set of current domains of program expressions and inputs in $TDom$.
2: **Output:** computes a new domain for the arithmetic expression AE.
3: `getOpDom(`$TDom$`,`DM`)` returns the current domain of the data manipulator operand DM from $TDom$.
4: `getaop(`AE`)` returns the arithmetic operator in AE.
5: `getop1(`AE`)` and `getop2(`AE`)` return the first and second operands in expression AE, respectively.

6: **begin** `GetAExprDom(`AE`,`$TDom$`)`
7: $a =$ `getaop(`BE`)`
8: $DM =$ `getop1(`AE`)`
9: $n =$ `getop2(`AE`)`
10: $Đ_{DM} =$ `getOpDom(`$TDom$`,`DM`)`, where
11: $Đ_{DM} =< subdomVal_1, \ldots, subdomVal_m >$
12: $Đ_n =$ `getOpDom(`$TDom$`,`n`)`, where $Đ_n = [n, n]$
13: **if** a is $+$ or $-$ **then**
14:     **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
15:         **if** a is $+$ **then**
16:             $Đ_E = Đ_E \cup [l_i + n, u_i + n]$
17:         **else**
18:             $Đ_E = Đ_E \cup [l_i - n, u_i - n]$
19:         **end if**
20:     **end for**
21: **else if** a is $*$ **then**
22:     **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
23:         $u = u_i \ * \ n$
24:         $l = l_i \ * \ n$
25:         **if** $l > u$ **then**
26:             $Đ_E = Đ_E \cup [u, l]$
27:         **else**
28:             $Đ_E = Đ_E \cup [l, u]$
29:         **end if**
30:     **end for**
31: **else if** a is $/$ **then**
32:     **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
33:         **if** $n == 0$ **then**
34:             $Đ_n = [l_n, u_n] = [1, -1]$
35:         **end if**
36:         $u = u_i \ / \ u_n$
37:         $l = l_i \ / \ l_n$
38:         **if** $l > u$ **then**
39:             $Đ_E = Đ_E \cup [u, l]$
40:         **else**
41:             $Đ_E = Đ_E \cup [l, u]$

Continued Algorithm 6.3

```
42:       end if
43:     end for
44: end if
45: return Ð_E
46: end GetAExprDom(AE,TDom)
```

of the AAPL program paths, see Figure 6.2 on page 144, in which a domain of a data manipulator is computed.

For bitwise expressions using the bitwise AND, OR and XOR operations, the function examines l_{BE} and u_{BE} of the expression and the constant value associated with the expression to determine the limits of the new domain of DM. For each of these three bitwise operations, an update evaluation technique is used to compute possible values of the operand's domain. The values of bits within the elements in the domain of the expression and the constant number are examined to compute the possible domain of the data manipulator. For instance, in the case of AND evaluation (in Algorithm 6.6 on page 168, lines 13-37), when the value of a bit for an element (e.g. l_E) in the domain of the expression, E, is equal to 1, the corresponding bit in the element l_{DM} for the domain of DM is set to 1; however, if the bit in l_E for the domain of the expression, E domain is equal to 0 then the function checks the value of the corresponding bit in the constant n and sets l_{DM} to 0 or x if n is 1 or 0, respectively, where x is a *do-not-care* value (i.e. $x = 0$ or 1).

In the case of OR evaluation (in Algorithm 6.6 on page 168, lines 38-48), the bit values of the constant n are only examined to determine the value of each bit of an element in the domain of DM. That is, when the value of a bit in n is 0 or 1, the value of the corresponding bit of the element (e.g. l_{DM} or u_{DM}) in DM is either equal to the value of that corresponding bit in the element (i.e. l_{BE} or u_{BE}) of E, or x, respectively. Note that if changes are required by decisions made when evaluating the constraints, `UpdateDMDomVal` (in Algorithm 6.7 on page 170) is called to find suitable values for bits that are set to x.

The update evaluation technique for handling XOR operations and computing the domain for a DM operand, in a bitwise expression, e.g. $BE = DM \oplus n$, is defined as follows:

$$d(i) = \begin{cases} \neg n(i) & \text{if } e(i) = 1 \\ n(i) & \text{otherwise} \end{cases}$$

Algorithm 6.4: `GetBExprDom(`*BE*`)` evaluate a new domain for a bitwise expression *BE*.

1: **Input:** a bitwise expression, $BE := DM$ `bop` $n \mid \neg DM$ and the set of domains of program expressions and inputs in *TDom*.
2: **Output:** computes a new domain for the bitwise expression *BE*.
3: **procedure** `GetAExprDom(`*AE*`,`*TDom*`)` is presented in Algorithm 6.3 on page 163
4: `getOpDom(`*TDom*`,`*DM*`)` returns the current domain of the data manipulator operand *DM* from *TDom*.
5: `getop1(`E`)` and `getop2(`E`)` return the first and second operands in expression E, respectively.
6: `getbop(`*BE*`)` returns the bitwise operator (including $\neg$) in *BE*.
7: `max` and `min` return the greater and the smaller of two values, respectively.

8: **begin** `GetBExprDom(`*BE*`)`
9: $b =$ `getbop(`*BE*`)`
10: $DM =$ `getop1(`*BE*`)`
11: $Đ_{DM} =$ `getOpDom(`*TDom*`,`*DM*`)`
12: **if** b is $\neg$ **then**
13: $u_{BE} = \texttt{max}(\neg(u_{DM}), \neg(l_{DM}))$
14: $l_{BE} = \texttt{min}(\neg(u_{DM}), \neg(l_{DM}))$
15: **else**
16: $n =$ `getop2(`*BE*`)`
17: **if** b is & **then**
18: **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
19: **for** all element k in $subdomVal_i$ **do**
20: $Đ_{BE} = Đ_{BE} \cup \{k \ \& \ n\}$
21: **end for**
22: **end for**
23: **else if** b is $\mid$ **then**
24: **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
25: **for** all element k in $subdomVal_i$ **do**
26: $Đ_{BE} = Đ_{BE} \cup \{k \mid n\}$
27: **end for**
28: **end for**
29: **else if** b is $\oplus$ **then**
30: **for** all $subdomVal_i \in Đ_{DM}$ where $1 \leq i \leq m$ and $subdomVal_i = [l_i, u_i]$ **do**
31: **for** all element k in $subdomVal_i$ **do**
32: $Đ_{BE} = Đ_{BE} \cup \{k \ \oplus \ n\}$
33: **end for**
34: **end for**
35: **else if** b is $\ll$ **then**
36: to perform a bitwise shift left, multiply *DM* by $2 * n$, then compute the domain of the expression:
37: $k = n * 2$
38: **return** `GetAExprDom(`$DM * k$`,`*TDom*`)`
39: **else if** b is $\gg$ **then**

Continued Algorithm 6.4

40: to perform a bitwise shift right, divide *DM* by $2 * n$, then compute the domain of the expression:
41: $k = n * 2$
42: **return** `GetAExprDom(`DM/k`,`$TDom$`)`
43: **end if**
44: **end if**
45: $Đ_{BE} = [l_{BE}, u_{BE}]$
46: **return** $Đ_{BE}$
47: **end** `GetBExprDom(`BE`)`

Algorithm 6.5: `UpdateDomVal(`$E, l, u, TDom$`)` updates the domains of program input variables (data manipulators).

1: **Input:** an expression, $E := n \mid DM \mid BE \mid AE$, its new domain bottom, l, and top, u, values and the set of current domains of program expressions and inputs in *TDom*.
2: **Output:** $True$, if the procedure successfully modifies the domains of program inputs or $False$, otherwise.
3: **procedure** `UpdateDMDomVal` in Algorithm 6.7 on page 170.
4: **procedure** `UpdateBEDomVal` in Algorithm 6.6 on page 168.
5: **begin** `UpdateDomVal(`$E, l, u, TDom$`)`
6: **if** E is n **then**
7: **return** $True$
8: **else if** E is *DM* **then**
9: **return** `UpdateDMDomVal(`$DM, l, u, TDom$`)`
10: **else if** E is *BE* **then**
11: **return** `UpdateBEDomVal(`$BE, l, u, TDom$`)`
12: **else if** E is *AE* **then**
13: This is part of the procedure Update in [OJP99].
14: **end if**
15: **end** `UpdateDomVal(`$E, l, u, TDom$`)`

Observe that the above evaluation technique is applied for both top, u_{DM}, and bottom, l_{DM}, limits of the domain of the operand *DM* (in Algorithm 6.6 on page 168, lines 49-67). Thus, d and e represent u_{DM} and u_{BE} in the case of computing the top limit for the domain of *DM* using the top limit for the domain of *BE*, respectively, and represent l_{DM} and l_{BE} in the case of computing the bottom limit for the domain of *DM* using the bottom limit for the domain of *BE*, respectively. The evaluation walks through each bit, $d(i)$ (in l_{DM} and u_{DM}), in the domain of the data manipulator, *DM*, and computes the bit value based on the value of the corresponding bit, $e(i)$ (in l_{BE} and u_{BE}), in the domain limit of the expression, *BE*.

The update evaluation techniques for bitwise left and right shift operations ($\ll$, $\gg$) (in Algorithm 6.6 on the next page, lines 68 and 71, are similar to the evaluation techniques for the division and multiplication operations. Thus, these two cases are handled by the update technique of the arithmetic operations presented in [OJP99], see case *AE* in Algorithm 6.5 on the preceding page.

2. Updating Domain Values for Data Manipulators

The function `UpdateDMDomVal` (in Algorithm 6.7 on page 170) updates the domain of a program data manipulator based on the new values of the domain passed to the function. The function takes as inputs a data manipulator *DM*, and the top and bottom values (`u` and `l`, respectively), of the new domain and the temporary current input domains of the program input variables and expressions.

The function consists of three main steps. The first step (Algorithm 6.7 on page 170, lines 11-16) examines the limits `u` and `l` of the new domain and assigns appropriate values to bits that are labelled as *do-not-care* (i.e. x) – meaning that these bits can hold either the value 1 or 0. Thus, the function assigns zeros to *do-not-care* bits in `l` (the bottom value) and assigns ones to *do-not-care* bits in `u` (the top value) of the new domain. This step helps to reduce the new domain values of the data manipulator so that it contains the correct values computed from the `ExpDomEval` procedure.

After all *do-not-care* bits (in the new top and bottom values) are replaced with appropriate one and zero, in the second step (lines 18-23), the function handles domains of program inputs that are involved in bitwise expressions with AND and OR operations (if any). For the `l` and `u` of new domains that contain *do-not-care* bits, the function discards a subdomain in the domain of the program input where some of its values are not consistent with the new limits `l` and `u`. Note that $lTmp$ and $uTmp$ hold the values of l and u, respectively, of the new domain, which is passed to the function. For instance, assume that the new limits of the domain are $lTmp = uTmp = 1x1x$, and that the limits of the domain of values of a data manipulator are $u = 1111$ and $l = 1010$ before step 2. Thus, the function discards subdomains between u and l when their values do not agree with $lTmp$ in ones values, e.g. 1100 and 1101 are discarded from the domain of values since the value of the second bit is not 1. Note that this step checks all individual values in the domain of $lTmp$ and $uTmp$ consistent with bit

Algorithm 6.6: `UpdateBEDomVal(`BE`,`l`,`u`,`$TDom$`)` updates the domain of a program data manipulator in a bitwise expression.

1: **Input:** a bitwise expression, $BE := DM$ `bop` $n \mid \neg DM$, its new domain bottom, l, and top, u, values and the set of current domains of program expressions and inputs in $TDom$.
2: **Output:** $True$, if the procedure successfully modifies the domains of DM or $False$, otherwise.
3: **procedure** `UpdateDMDomVal` in Algorithm 6.7 on page 170.
4: `getbop(`BE`)` returns the bitwise operator in BE.
5: `getop1(`BE`)` and `getop2(`BE`)` return the first and second operands in expression BE, respectively.

6: **begin** `UpdateBEDomVal(`BE`,`l`,`u`,`$TDom$`)`
7: `bop` = `getbop(`BE`)`, DM = `getop1(`BE`)` and n = `getop2(`BE`)`
8: **if** `bop` is $\neg$ **then**
9: $l'_{DM} = \neg l_{BE}$; $u'_{DM} = \neg u_{BE}$
10: **if** $l'_{DM} > u'_{DM}$ **then**
11: `swap(`l'_{DM},u'_{DM}`)`
12: **end if**
13: **else if** `bop` is `&` **then**
14: check each bit in l_{BE} and compute the values of the corresponding bits in l_{DM}:
15: **for** $k \to 0$; $k < 32$; k++ **do**
16: **if** $l_{BE}(k)$ is 1 **then**
17: $l'_{DM}(k) = 1$
18: **else if** $n(k)$ is 0 **then**
19: the value of bit k in l_{DM} could be set to 0 or 1:
20: $l'_{DM}(k) = x$
21: **else**
22: this is the case when $l_{BE}(k) = 0$ and $n(k) = 1$:
23: $l'_{DM}(k) = 0$
24: **end if**
25: **end for**
26: now check each bit in u_{BE} and compute the values of the corresponding bits in u_{DM}:
27: **for** $k \to 0$; $k < 32$; k++ **do**
28: **if** $u_{BE}(k)$ is 1 **then**
29: $u'_{DM}(k) = 1$
30: **else if** $n(k)$ is 0 **then**
31: the value of bit k in u_{DM} could be set to 0 or 1:
32: $u'_{DM}(k) = x$
33: **else**
34: this is the case when $u_{BE}(k) = 0$ and $n(k) = 1$:
35: $u'_{DM}(k) = 0$
36: **end if**
37: **end for**
38: **else if** `bop` is $\mid$ **then**

Continued Algorithm 6.6

39: for each element in the domain of BE: $\forall val_{\texttt{BE}} \in [l_{\texttt{BE}}, u_{\texttt{BE}}]$:
40: check the value of each bit i in n and compute the corresponding bit value in $val_{DM} \in [l', u'$ of DM:
41: **for** $k \rightarrow 0$; $k < 32$; k++ **do**
42: **if** $n(k)$ is 0 **then**
43: $val_{DM}(k) = val_{\texttt{BE}(k)}$
44: **else**
45: this is the case when the value of bit k is set to x
46: $val_{DM}(k) = x$
47: **end if**
48: **end for**
49: **else if** `bop` is $\oplus$ **then**
50: for each element in the domain of BE: $\forall val_{\texttt{BE}} \in [l_{\texttt{BE}}, u_{\texttt{BE}}]$ do:
51: check each bit in $l_{\texttt{BE}}$ and $u_{\texttt{BE}}$ of the expression domain and compute the corresponding bit value in DM, where $val_{DM} \in [l', u']$:
52: **for** $k \rightarrow 0$; $k < 32$; k++ **do**
53: **if** $val_{\texttt{BE}}(k)$ is 1 **then**
54: $val_{DM}(k) = \neg n(k)$
55: **else**
56: this is the case when the value of bit i in val_{DM} is set to $n(k)$:
57: $val_{DM}(k) = n(k)$
58: **end if**
59: **end for**
60: **for** $k \rightarrow 0$; $k < 32$; k++ **do**
61: **if** $val_{\texttt{BE}}(k)$ is 1 **then**
62: $val_{DM}(k) = \neg n(k)$
63: **else**
64: this is the case when the value of bit k in val_{DM} is set to $n(k)$:
65: $val_{DM}(k) = n(k)$
66: **end if**
67: **end for**
68: **else if** `bop` is $<<$ **then**
69: $k = 2 * n$
70: **return** `UpdateDMDomVal(DM*k,` $l, u, TDom$`)`
71: **else if** `bop` is $>>$ **then**
72: $k = 2 * n$
73: **return** `UpdateDMDomVal(DM/k,` $l, u, TDom$`)`
74: **end if**
75: **return** `UpdateDMDomVal(`$DM, l'_{DM}, u'_{DM}, TDom$`)`
76: **end** `UpdateBEDomVal(`$BE, l, u, TDom$`)`

The third step attempts to reduce the current domain of *DM* using the new domain values. This step is similar to the *Update* function presented in [OJP99] in the sense that it discards any subdomain from the current domain of a program input that is not contained within the limits of the new domain.

Algorithm 6.7: `UpdateDMDomVal(`$DM, l, u, TDom$`)` updates the domain of a program data manipulator.

1: **Input:** a data manipulator, $DM := r \mid *r \mid *n$, its new domain bottom, l, and top, u, values and the set of current domains of program expressions and inputs in $TDom$.
2: **Output:** $True$, if the procedure successfully modifies the domains of program inputs or $False$, otherwise.
3: procedure `remove`: deletes a subdomain from the domain of DM and shifts the sub domains of DM (i.e.renumber the labels of the subdomains).
4: procedure `replace`: replaces the current bottom and top values of a subdomain with new top and bottom values and shifts the subdomains in DM's domain.
5: **begin** `UpdateDMDomVal(`$DM, l, u, TDom$`)`
6: the current domain of DM is $Đ_{DM} =< subdomVal_1, \ldots, subdomVal_n >$.
7: where $1 \leq i \leq n$ and $subdomVal_i = [l_i, u_i] \in Đ_{DM}$.
8: let $lTmp \rightarrow l$ and $uTmp \rightarrow u$
9: step 1: check if any *do-not-care* bit values x exist in new l and u.
10: for all bits k in l and u, replace each bit value x with 0 and 1 in l and u, respectively.
11: **for** $k \rightarrow 0$; $k < 32$; $k{+}{+}$ **do**
12:    **if** $l(k)$ is x and $u(k)$ is x **then**
13:       $l(k) = 0$
14:       $u(k) = 1$
15:    **end if**
16: **end for**
17: step 2: check bit values of the bottom, l_i, of each subdomain in $Đ_{DM}$, and discard any subdomain which its, l_i, does not agree with the new bit values in $lTmp$, note this case applies for logic AND and OR operations only:
18: **for** $i = 1, i \leq n, i{+}{+}$ **do**
19:    where $subdomVal_i = [l_i, u_i] \in Đ_{DM}$ and $\forall k, 0 \leq k \leq 32$:
20:    **if** $lTmp(k)$ is 1 **and** $l_i(k)$ is not 1 **then**
21:       `remove(`$TDom$`,`DM`,`$subdomVal$`)`
22:    **end if**
23: **end for**
24: step 3: reduce the domain by removing any subdomains that are not contained within the new domain range.
25: **if** $l \leq l_1$ **and** $u \geq u_n$ **then**
26:    **return** $True$
27: **else if** $l > u_n$ **or** $u < l_n$ **then**
28:    **return** $False$
29: **else**
30:    handle top value of domain first:
31:    **if** $u \geq u_n$ **then**
32:       no need to update the current top value of the domain, u_n, as it is within the new top value, u.
33:    **else**
34:       now search (from u_n to u_1) for a new top value and modify the domain of DM:
35:       **for** $i = n, i \geq 1, i{-}{-}$ **do**

Continued Algorithm 6.7

36: subdomain $subdomVal_i = [l_i, u_i] \in Đ_{DM}$
37: **if** $u \geq l_i$ **and** $u \leq u_i$ **then**
38: discard subdomains $< [u+1, u_i], \ldots, [l_i, u_i] >$ from $Đ_{DM}$.
39: **for** all $subdomVal \in < [u+1, u_i], \ldots, [l_i, u_i] >$ **do**
40: `remove`($TDom$, DM, $subdomVal$)
41: **end for**
42: replace subdomain $[l_i, u_i]$ with $[l_i, u]$:
43: `replace`($TDom$, DM, $[l_i, u]$, $[l_i, u_i]$)
44: **break()**
45: **else if** $u > u_i$ **and** $u < l_i + 1$ **then**
46: discard subdomains $< [l_i+1, u_i+1], \ldots, [l_n, u_n] >$ from $Đ_{DM}$:
47: **for** all $subdomVal \in < [l_i+1, u_i+1], \ldots, [l_n, u_n] >$ **do**
48: `remove`($TDom$, DM, $subdomVal$)
49: **end for**
50: **break()**
51: **end if**
52: **end for**
53: **end if**
54: now at this point the domain of DM might be modified (i.e. reduced with a new top value m) $Đ_{DM} = < [l_1, u_1] \ldots [l_m, u_m] >$ where $m \leq n$:
55: handle bottom value of domain first:
56: **if** $l \leq l_i$ **then**
57: **return** $True$
58: **else**
59: now search (from l_1 to l_n) for a new bottom value and modify the domain of DM:
60: **for** $i = 1, i \leq n, i{+}{+}$ **do**
61: subdomain $subdomVal_i = [l_i, u_i] \in Đ_{DM}$
62: **if** $l \geq l_i$ **and** $l \leq u_i$ **then**
63: replace subdomain $< [l_i, u_i]$ with $[l, u_i] >$:
64: `replace`($TDom$, DM, $[l, u_i]$, $[l_i, u_i]$)
65: remove $< [l_1, u_1], \ldots, [l_i - 1, u_i - 1] >$ from $Đ_{DM}$ and shift the domain (i^{th} subdomain becomes the 1st subdomain):
66: **for** $subdomVal \in < [l_1, u_1], \ldots, [l_i - 1, u_i - 1] >$ **do**
67: `remove`($TDom$, DM, $subdomVal$)
68: **end for**
69: now at this point the domain of DM (both the top and bottom values) may have been reduced.
70: **return** $True$
71: **else if** $l > u_i$ **and** $l < l_i + 1$ **then**
72: discard subdomains $< [l_1, u_1], \ldots, [l_i, u_i] >$ from $Đ_{DM}$ and shift the domain such that the ith subdomain becomes the 1st subdomain:
73: **for** $subdomVal \in < [l_1, u_1], \ldots, [l_i, u_i] >$ **do**
74: `remove`($TDom$, DM, $subdomVal$)
75: **end for**

Continued Algorithm 6.7

76: **return** $True$
77: **end if**
78: **end for**
79: **end if**
80: at this point the new value $[l, u]$ is not contained within the domain of DM:
81: **return** $False$
82: **end if**
83: **end** `UpdateDMDomVal(`$DM, l, u, TDom$`)`

6.4.3 Examples

This section provides two examples to show how the extended DDR algorithm handles AAPL program inputs and generates test data.

Example 6.3. *Given a program and its control flow graph, as shown in Figure 6.7 on the following page, let the program inputs (*data manipulators*) be* ***r0****,* ***r1*** *and* ****129****. The program fetches the memory location pointed to by* ****129*** *and updates the values of data manipulators* ***r1*** *and* ****129*** *based on the values of program inputs* ***r0*** *and* ****129****. Assume that the initial domains $Đ_{r0}, Đ_{r1}, Đ_{*129}$ of the input data manipulators are as follows:*

$$Đ_{r0} = [0, 7], \; Đ_{r1} = [0, 7], \; Đ_{*129} = [0, 7]$$

To generate a test case for the program, let the path $\pi = < 1, 2, 3, 4, 5, 6, 7, 9 >$ be the input path for the algorithm. Note that the selected path consists of four assignment statements, two conditional and two unconditional jump statements. The first statement in the path is handled and the expression in the statement is symbolically evaluated by the procedure ***ExprDomEval*** *(Algorithm 6.2 on page 161) where* ***r3*** *is evaluated to the expression* ***r0*** *& 5. The expression* ***r0*** *& 5 is assigned a new domain where the bottom and the top values of the domain, $Đ_{r0\ \&\ 5}$, are $l_{r0\ \&\ 5} = 0$ and $u_{r0\ \&\ 5} = 5$, respectively. Note that Algorithm 6.4 on page 165 in lines 17-22 computes the domain of the expression as $Đ_{r0\ \&\ 5} = \{0, 1, 4, 5\}$ (i.e. there are two subdomains, $Đ_{r0\ \&\ 5} =< [0, 1], [4, 5] >$). The next statement to be handled in π is statement 2 (a conditional statement) and the constraint associated with the edge $(2, 3)$ in the control flow graph (Figure 6.7 on the next page) is* ***r3*** *≤ 4. Since* ***r3*** *is evaluated to the expression* ***r0*** *& 5, the algorithm needs to find a suitable subdomain of values for* ***r0*** *& 5 using $Đ_{r0\ \&\ 5}$ such that it satisfies the current constraint. The domain of* ***r0*** *& 5 satisfies case 2 in the* ***GetSplit***

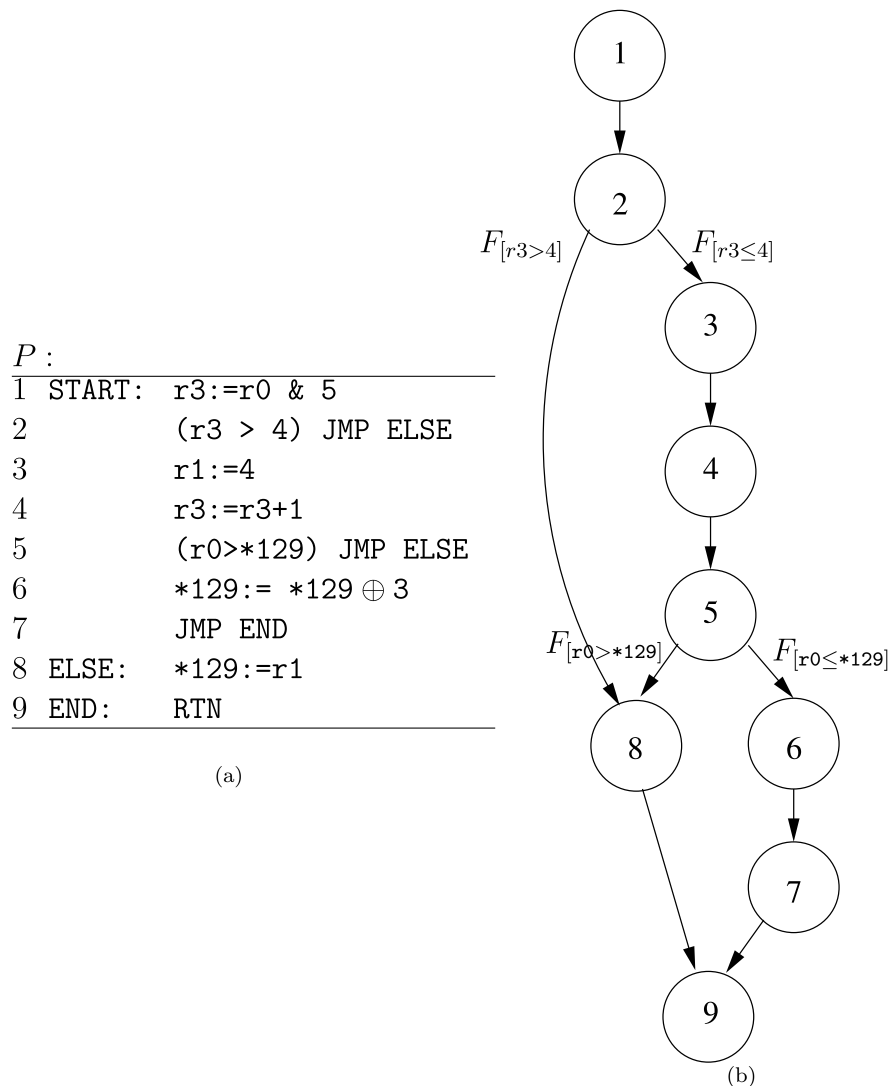


Figure 6.7: A code fragment in AAPL and its control flow graph for Example 6.3.

procedure, where $(l_{r0\ \&\ 5} \leq 4)$ *and* $(u_{r0\ \&\ 5} \geq 4)$*; thus, the split point is computed as* $sp = (u_4 - l_4) * 1/2 + l_4 = 4$ *where* 4 *is the right hand side sub-expression of the boolean expression and* $Đ_4 = [4, 4]$*. The domain of the expression* **r0** & 5 *is reduced to* $Đ_{r0\ \&\ 5} =< [0, 1], [4, 4] >$ *and procedure* **UpdateDMDomVal** *(in Algorithm 6.7 on page 170, lines 18-23) propagates the changes down to the program input* **r0***. Note that procedure* **UpdateBEDomVal** *in Algorithm 6.6 on page 168 (lines 13-37) computes the corresponding bit values of the bottom and top values of each subdomain of* **r0** *domain. For each subdomain in* $Đ_{r0\ \&\ 5}$*, the procedure computes the top and bottom of the subdomain of* r0 *domain, yielding* $Đ_{r0} =< [0, 4], [6, 6] >$ *(i.e.* $Đ_{r0} = \{0, 1, 2, 3, 4, 6\}$*). The current domain of program inputs becomes as*

follows:

$$Đ_{r0} =< [0,4],[6,6] >, \ Đ_{r1} = [0,7], \ Đ_{*129} = [0,7]$$

The next two statements in the path are statements 3 *and* 4*, which are assignment commands. These statements are symbolically evaluated where* r1 *and* r3 *are evaluated to expressions* 4 *and* (**r0** & 5) + 1*, respectively. A new domain for the expression* **r3+1** *is created by adding one to the bottom and top values of the domain of the expression* **r0** & 5*, yielding* $Đ_{r3+1} =< [1,2],[5,5] >$*. Note that at this point, the domain of the program input data manipulators* r0 *and* r1 *are not changed. Then statement* 5 *is handled and since it is a conditional jump command the constraint* (**r0** $\leq$ ***129**) *associated with the edge* (5, 6) *in the control flow graph is evaluated. Examining the domains of data manipulators involved in the constraint, case* 1 *in the* **GetSplit** *procedure is satisfied and the split point is* $sp = (u_{r0} - l_{r0}) * 1/2 + l_{r0} = 3$ *(where* $l_{r0} = 0$ *and* $u_{r0} = 6$*). The domains of* **r0** *and* ***129** *are only reduced by the* **UpdateDMDomVal** *procedure and the current domain values of the program input becomes:*

$$Đ_{r0} = [0,3], \ Đ_{r1} = [0,7], \ Đ_{*129} = [3,7]$$

Finally, the algorithm reaches statement 6 *in the selected path where the symbolic expression* ***129** $\oplus$ **3** *is assigned to* ***129** *and a new domain for the expression is created* $Đ_{*129 \oplus 3} =< [0,0],[4,7] >$*. Statement* 7 *is an unconditional jump command, which moves the control flow to statement* 9 *and exits the path. At this point, the final domains of program inputs* **r0***,* **r1** *and* ***129** *are:*

$$Đ_{r0} = [0,3], \ Đ_{r1} = [0,7], \ Đ_{*129} = [3,7]$$

Below are the steps taken to reduce the domains of program input data manipulators and expressions:

$Đ_{r0} = [0,7],\ Đ_{r1} = [0,7],\ Đ_{*129} = [0,7]$

1 `r3:=r0 & 5`

$Đ_{r0} = [0,7],\ Đ_{r1} = [0,7],\ Đ_{*129} = [0,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,5] >$

2 $r3 \leq 4$ *(split point for* `r0` & 5 *selected is* 4*)*

$Đ_{r0} =< [0,4],[6,6] >,\ Đ_{r1} = [0,7],\ Đ_{*129} = [0,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,4] >$

3 `r1:=4`

$Đ_{r0} =< [0,4],[6,6] >,\ Đ_{r1} = [0,7],\ Đ_{*129} = [0,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,4] >,$

$Đ_4 = [4,4]$

4 `r3:=r3+1`

$Đ_{r0} =< [0,4],[6,6] >,\ Đ_{r1} = [0,7],\ Đ_{*129} = [0,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,4] >,$

$Đ_{r3+1} =< [1,2],[5,5]$

2 $r0 \leq *129$ *(split point for* `r0` *and* `*129` *selected is* 3*)*

$Đ_{r0} = [0,3],\ Đ_{r1} = [0,7],\ Đ_{*129} = [3,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,4] >,$

$Đ_{r3+1} =< [1,2],[5,5]$

3 $*129 := *129 \oplus 3$

$Đ_{r0} = [0,3],\ Đ_{r1} = [0,7],\ Đ_{*129} = [3,7],\ Đ_{r0\ \&\ 5} =< [0,1],[4,4] >,$

$Đ_{r3+1} =< [1,2],[5,5],\ Đ_{*129\ \oplus\ 3} =< [0,0],[4,7] >$

Test data input can be selected randomly from within the values of the domains of the program inputs, `r0`*,* `r1` *and* `*129`*. Any selected test case should exercise the selected path and satisfy all the constraints on the path. For example, the test case* $< 3,4,3 >$ *is produced by randomly selecting one value for each data manipulator from their domains* $Đ_{r0}$*,* $Đ_{r1}$ *and* $Đ_{*129}$*, respectively. When the program is executed*

using this test case the following path statements are exercised:

test case:	$< r0 = 3, r1 = 4, *129 = 3 >$	
node	*statement*	*path constraint*
1 *START:*	`r3:=r0 & 5`	$(r3 = 1)$
2	$(r3 \leq 4)$	$(1 \leq 4)$
3	`r1:=4`	$(r1 = 4)$
4	`r3:=r3+1`	$(r3 = 2)$
5	$(r0 \leq *129)$	$(3 \leq 3)$
6	$*129 := *129 \oplus r1$	$(*129 = 7)$
7	*JMP END*	
9 *END:*	*RTN*	

Example 6.4. *Given a program and its control flow graph, as shown in Figure 6.8 on the following page, which contains a loop, assume the program inputs (*data manipulators*) are* **r0***,* **r1** *and* **r2***. The program updates the memory location* ***r1** *pointed to by the value of* **r1** *in a given state inside the loop at instructions in lines* 2*,* 5*,* 7 *and* 9*. There are three branching destinations,* **LOOP***,* **ELSE** *and* **END***, in the CFG where the program control flow may go. In this example, the algorithm will compute for the path* $\pi = < 1, 2, 3, 4, 7, 8, 1, 9, 10 >$ *a test case that exercises the loop only once. Assume that the initial domains of the input data manipulators are as follows:*

$$D_{r0} = [0, 7],\ D_{r1} = [1, 4],\ D_{r2} = [6, 15]$$

Note that the selected path consists of four assignment statements, three conditional and two unconditional jump statements. The first statement in π *is a conditional jump command and constraint associated with the edge* $(1, 2)$ *in the control flow graph (Figure 6.8 on the next page) (i.e.* $r0 < r2$*). The algorithm needs to find a suitable subdomain of values for* **r0** *and* **r2** *such that they satisfy the current constraint. The domains of* **r0** *and* **r2** *satisfy case* 4 *in the* **GetSplit** *procedure, where* $(l_{r0} \leq l_{r2})$ **and** $(u_{r0} \leq u_{r2})$ **and** $(l_{r2} \leq u_{r0})$*, thus, the split point is computed as* $sp = (u_{r2} - l_{r0}) * 1/2 + l_{r0} = 8$*. The domain of* **r2** *is reduced to* $D_{r2} = [8, 15]$*. The current domain of program inputs becomes as follows:*

$$D_{r0} = [0, 7],\ D_{r1} = [1, 4],\ D_{r2} = [8, 15]$$

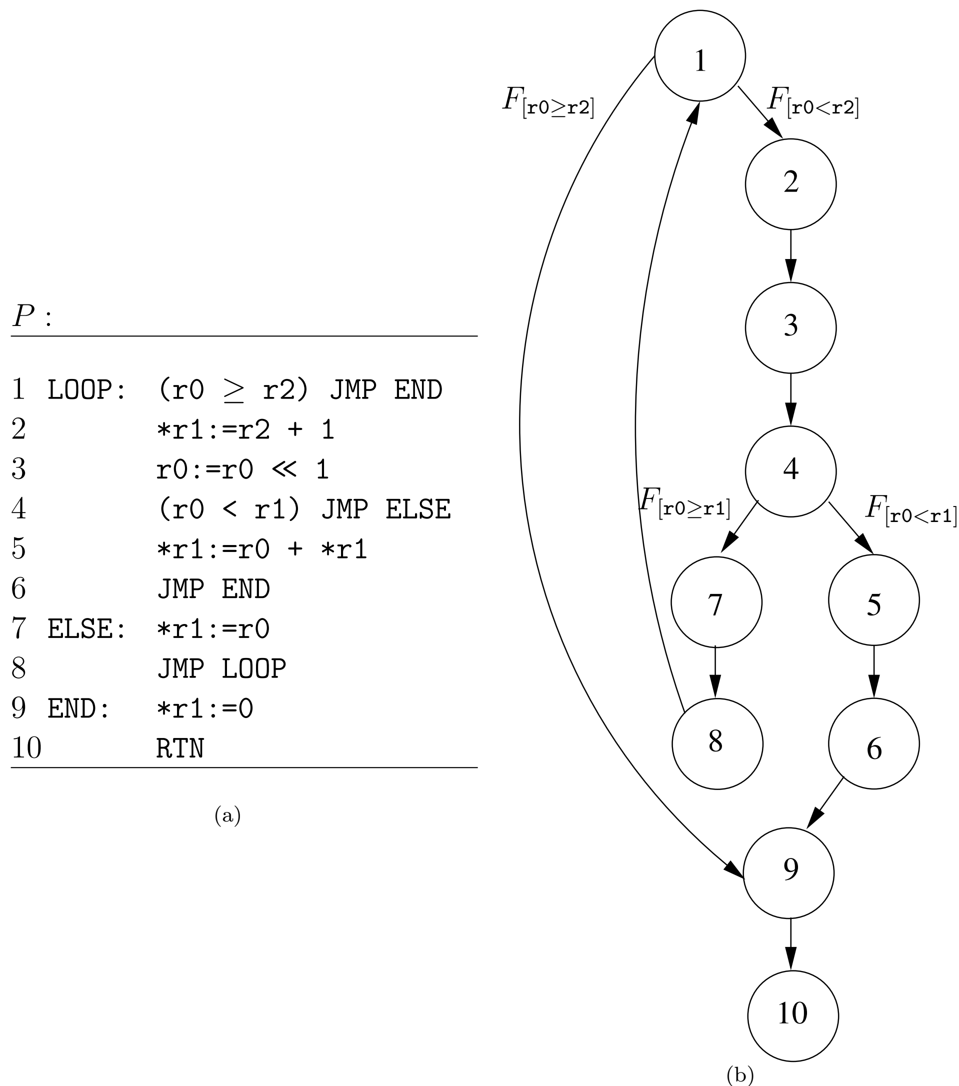


Figure 6.8: A code fragment in AAPL and its control flow graph for Example 6.4.

The next two statements in the path are statements 2 *and* 3*, which are assignment commands. These statements are symbolically evaluated and the domains of program data manipulators* ***r1** and **r0** are not changed. The method creates a new domain for the expressions* **r2+1** *and* **r0≪**1*. Then we have,* $Đ_{r2+1} = [l_{r2}+1, u_{r2}+1] = [9,16]$ *and* $Đ_{r0 \ll 1}=[l_{(r0\ll1)},u_{(r0\ll1)}]$ *where* $l_{(r0\ll1)} = l_{r0}\ll 1$ *and* $u_{(r0\ll1)} = u_{r0}\ll 1$*. Thus,* $Đ_{r0\ll1}$ *consists of even values where* $Đ_{r0\ll1} =< subdomVal_1, \ldots, subdomVal_n >$ *(where* $subdomVal_n$ *is a sub-domain of contiguous values) and* $n \geq 1$*, i.e.* $Đ_{r0\ll1} =< [0,0],[2,2],[4,4],\ldots,[14,14] >$ *(Algorithm 6.4 on page 165 lines 35-38). Then statement* 4 *is handled and since it is a conditional jump command the constraint* $($**r0** $\geq$ **r1**$)$ *associated with the edge* $(4,7)$ *in the control flow graph is evaluated and*

`r0` is expressed by the symbolic value `r0`$\ll$1. In the `GetSplit` procedure, case 2 is satisfied between $Đ_{r0\ll1}$ and $Đ_{r1}$; the split point is calculated as $sp = (u_{r1} - l_{r1}) * 1/2 + l_{r1} = 3$. Then the procedure `UpdateDMDomVal` takes sp and reduces the domains of `r0`$\ll$1 and `r1` and produces $Đ_{r0\ll1} =< [4,4],[6,6],[8,8]\ldots,[14,14] >$ and $Đ_{r1} = [1,3]$. Since the expression, `r0`$\ll$1, is symbolically assigned to `r0` at node 3, the reduced domain values of expression `r0`$\ll$1 are propagated back by procedure `UpdateBEDomVal` (Algorithm 6.6 on page 168 lines 68-70) to compute the current domain of `r0` right before statement 3. Now, the current domains of the program inputs are:

$$Đ_{r0} = [2,7],\ Đ_{r1} = [1,3],\ Đ_{r2} = [8,15]$$

Statement 7 is the next statement in the path and since it is an assignment command, it is evaluated symbolically where `*r1` is expressed by `r0` (i.e. `*r1` = `r0`). Then statement 8, which is an unconditional jump statement, transfers the control flow to the beginning of the loop. Statement 1 is re-evaluated such that the edge (1,9) has to be traversed. The constraint `r0` $\geq$ `r2` is evaluated; note that at this point `r0` is still expressed by the symbolic value `r0`$\ll$1 where $Đ_{r0\ll1} =< [4,4],[6,6],\ldots,[14,14] >$. In procedure `GetSplit`, case 4 is satisfied where $(l_{r0\ll1} \leq l_{r2})$ **and** $(u_{r0\ll1} \leq u_{r2})$ **and** $(l_{r2} \leq u_{r0\ll1})$. The split point in this case is 10. The new domains of `r0`$\ll$1 and `r2` after splitting are $Đ_{r0\ll1} =< [10,10],[12,12],[14,14] >$ and $Đ_{r2} = [8,9]$. To propagate the changes to the current input domain of `r0`, the procedure `UpdateBEDomVal` takes the computed domain $Đ_{r0\ll1}$ and updates $Đ_{r0}$. Now the current domains of the program input data manipulators after statement 1 become:

$$Đ_{r0} = [5,7],\ Đ_{r1} = [1,3],\ Đ_{r2} = [8,10]$$

Finally, statements 9 and 10 are reached, where statement 9 is an assignment command and statement 10 is an unconditional jump command. Both commands have no effect on the current domain values of the program input and thus, the algorithm terminates, and the set of domains of program inputs is computed.

Below are the steps taken to reduce the domains of program input data manipulators and expressions:

$Đ_{r0} = [0, 7],\ Đ_{r1} = [1, 4],\ Đ_{r2} = [6, 15]$

1 $r0 < r2$ (*split point for* r0 *and* r2 *selected is* 8)

$Đ_{r0} = [0, 7],\ Đ_{r1} = [1, 4],\ Đ_{r2} = [8, 15]$

2 `*r1:=r2 + 1`

$Đ_{r0} = [0, 7],\ Đ_{r1} = [1, 4],\ Đ_{r2} = [8, 15],\ Đ_{r2+1} = [9, 16]$

3 `r0:=r0 << 1`

$Đ_{r0} = [0, 7],\ Đ_{r1} = [1, 4],\ Đ_{r2} = [8, 15],\ Đ_{r2+1} = [9, 16],$

$Đ_{r0 \ll 1} =< [0, 0], [2, 2], \ldots, [14, 14] >$

4 $r0 \geq r1$ (*split point for* r0≪1 *and* r1 *selected is* 3)

$Đ_{r0} = [2, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 15],\ Đ_{r2+1} = [9, 16],$

$Đ_{r0 \ll 1} =< [4, 4], [6, 6], [8, 8] \ldots, [14, 14] >$

7 `*r1:=r0`

$Đ_{r0} = [2, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 15],\ Đ_{r2+1} = [9, 16],$

$Đ_{r0 \ll 1} =< [4, 4], [6, 6], [8, 8] \ldots, [14, 14] >$

8 `JMP LOOP`

$Đ_{r0} = [2, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 15],\ Đ_{r2+1} = [9, 16],$

$Đ_{r0 \ll 1} =< [4, 4], [6, 6], [8, 8] \ldots, [14, 14] >$

1 $r0 \geq r2$ (*split point for* r0≪1 *and* r2 *selected is* 10)

$Đ_{r0} = [5, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 10],\ Đ_{r2+1} = [9, 11],$

$Đ_{r0 \ll 1} =< [10, 10], [12, 12], [14, 14] >$

9 `*r1:=0`

$Đ_{r0} = [5, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 10],\ Đ_{r2+1} = [9, 11],$

$Đ_{r0 \ll 1} =< [10, 10], [12, 12], [14, 14] >,\ Đ_0 = [0, 0]$

10 *RTN*

$Đ_{r0} = [5, 7],\ Đ_{r1} = [1, 3],\ Đ_{r2} = [8, 10],\ Đ_{r2+1} = [9, 11],$

$Đ_{r0 \ll 1} =< [10, 10], [12, 12], [14, 14] >,\ Đ_0 = [0, 0]$

Test data input can be selected randomly from within the values of the domains of the program inputs. Any selected test case should exercise the selected path

and satisfy all the constraints on the path. For example, the test case $<5, 2, 8>$ *is produced by randomly selecting one value for each data manipulator from its domain,* $Đ_{r0} = [5, 7]$, $Đ_{r1} = [1, 3]$ *and* $Đ_{r2} = [8, 9]$*, respectively. When the program is executed using this test case the following path statements are exercised:*

test case:	$< r0 = 5,\ r1 = 2,\ r2 = 8 >$	
node	*statement*	*path constraint*
1 *LOOP:*	*(r0<r2)*	$(5 < 8)$
2	**r1:=r2+1*	$(*2 = 9)$
3	*r0:=r0≪1*	$(r0 = 10)$
4	$(r0 \geq r1)$	$(10 \geq 2)$
7 *ELSE:*	**r1:=r0*	$(*2 = 10)$
8	*JMP LOOP*	
1 *LOOP:*	$(r0 \geq r2)$ *JMP END*	$(10 \geq 8)$
9 *END:*	**r1:=0*	$(*2 = 0)$
10	*RTN*	

6.5 The Correctness Proof of Extended DDR

The **semantic evaluations** of the DDR analysis for a CFG path are phrased in terms of successive approximations of the set of all possible states, Σ_P, of a program P and the *symbolic evaluation* of expressions at every program path statement S. For the rest of this chapter we use ddrAlg to refer to the DDR algorithm. Let π be a finite path in the CFG of an AAPL program P and Σ_P be the set of all possible states of P. We let $Đ_\pi \subseteq \Sigma_P$ be the domains of program input data manipulators calculated by applying DDR to π, i.e. $Đ_\pi = \text{ddrAlg}(\pi)$. For the set of all possible initial states of a program P, Σ_P, each program data manipulator input dm is initially assigned a *domain* (i.e. a set of possible input values for dm). Then, as ddrAlg evaluates the statements in π, the domains of the program input data manipulators may be dynamically reduced to satisfy the constraints in the path. Thus, after each constraint is evaluated, the domains of the program input data manipulators are modified and may be reduced to smaller domains than the initial domains. Thus, the output of ddrAlg for a finite CFG path π of P is $Đ_\pi = \{Đ_{dm_1} \times \ldots \times Đ_{dm_k}\}$, a cross product of all k input data manipulators of P.

Our conjecture for the correctness property of the DDR analysis is that ddrAlg is correct if $\forall \sigma \in \Sigma_P$ and we have $Đ_\pi \subseteq \Sigma_P$. That is, the algorithm finds a subset of initial domains of program input data manipulators such that when executing the program using test data, `x`, selected from $Đ_\pi$, the set of output states at the end of the path is reached (Theorem 6.1). The set of output (reachable) states is represented by the reachability semantics of AAPL path in Figure 6.2 on page 144.

From Section 6.3, we observe that ddrAlg uses two main procedures, `GetSplit` and `Update`, to find a solution that is a subset of the initial domains of the program input data manipulators. Thus, it is important to show that $\forall \sigma \in \Sigma_P$ (the set of initial states) and $\forall dm$ (program input variables), `GetSplit` and `Update` modify the data manipulators' domains such that the computed domains are contained within the initial domains of the data manipulators. Lemma 6.1 covers the correctness of `GetSplit` and Lemma 6.2 presents the correctness of `Update`. In Lemma 6.3, we show that whenever ddrAlg produces a solution $Đ_\pi$ for a given path, π, it is always contained within the given set of initial states of program input data manipulators. In Theorem 6.1, we prove that ddrAlg is correct with respect to the reachability semantics of AAPL path such that the set of reachable states for π given by the domain $Đ_\pi$ is a subset of the set of all possible reachable states for π according to the semantics of AAPL.

The Correctness of Procedure `GetSplit`:

Given a pair of domain values of two program input variables, the procedure `GetSplit` attempts to find a split point whenever there is an overlap between the pair of domains. In Lemma 6.1, we show that the procedure always finds a split point value which is an element of both given domains of program input variables. We let Π denote a set of finite paths in the CFG of an AAPL program. Also, let dm be a data manipulator in a program P, and for a given state $\sigma \in \Sigma_P$, the domain of dm denoted by $DomVal^{\Sigma}_{dm}$ represents the set of values that dm can have in σ, i.e. $DomVal^{\Sigma}_{dm} = \{v \mid \exists \sigma \in \Sigma_P, \sigma[dm \mapsto v]\}$. Furthermore, we let the pair l_{dm} and u_{dm} denote the bottom and the top values of set $DomVal^{\Sigma}_{dm}$, i.e. $l_{dm} = \min(DomVal^{\Sigma}_{dm})$ and $u_{dm} = \max(DomVal^{\Sigma}_{dm})$.

Lemma 6.1. *For a program P, let dm_1, dm_2 be program input variables of P and $\exists \pi \in \Pi$ of the CFG of P, and .*

Whenever

$$DomVal^{\Sigma}_{dm_1} \cap DomVal^{\Sigma}_{dm_2} \neq \{\emptyset\},$$

the function GetSplit always produces a split point *sp such that*

$$sp \in DomVal^{\Sigma}_{dm_1} \cap \; DomVal^{\Sigma}_{dm_2}$$

Proof *Assume that there exists a non-empty set, $DomVal^{\Sigma}_{dm_1} \cap DomVal^{\Sigma}_{dm_2}$. There are four cases to consider for the* **`GetSplit`** *function when computing sp. These cases depend on the relationships between $DomVal^{\Sigma}_{dm_1}$ and $DomVal^{\Sigma}_{dm_1}$. We let $l_{dm_1} = min(DomVal^{\Sigma}_{dm_1})$, $u_{dm_1} = max(DomVal^{\Sigma}_{dm_1})$, $l_{dm_2} = min(DomVal^{\Sigma}_{dm_2})$, $u_{dm_2} = max(DomVal^{\Sigma}_{dm_2})$ and the search point index $0 \leq i \leq 1$.*

1. *When $(l_{dm_1} \geq l_{dm_2}) \wedge (u_{dm_1} \geq u_{dm_2})$. We note that $DomVal^{\Sigma}_{dm_1} \cap DomVal^{\Sigma}_{dm_2} = DomVal^{\Sigma}_{dm_1}$. Then to compute the split point for this case, we use the equation*

$$sp = (u_{dm_1} - l_{dm_1}) * i + l_{dm_1} \qquad (\text{by } \texttt{GetSplit})$$

   and $\forall i \; 0 \leq i \leq 1$ we get $l_{dm_1} \leq sp \leq u_{dm_1}$. It follows that we have $sp \in DomVal^{\Sigma}_{dm_1}$.

2. *When $(l_{dm_1} \leq l_{dm_2}) \wedge (u_{dm_1} \geq u_{dm_2})$, we have the same result as step* 1 *(by symmetry).*

3. *When $(l_{dm_1} \geq l_{dm_2}) \wedge (u_{dm_1} \geq u_{dm_2}) \wedge (u_{dm_2} \geq l_{dm_1})$. We note that $DomVal^{\Sigma}_{dm_1} \cap DomVal^{\Sigma}_{dm_2} = W = \{l_{dm_1}, \ldots, u_{dm_2}\}$. Then to compute the split point for this case, we use the equation*

$$sp = (u_{dm_2} - l_{dm_1}) * i + l_{dm_1} \qquad (\text{by } \texttt{GetSplit})$$

   and $\forall i \; 0 \leq i \leq 1$ we get $l_{dm_1} \leq sp \leq u_{dm_2}$. It follows that we have $sp \in W$.

4. *When $(l_{dm_1} \leq l_{dm_2}) \wedge (u_{dm_1} \leq u_{dm_2}) \wedge (u_{dm_1} \leq l_{dm_2})$. We note that $DomVal^{\Sigma}_{dm_1} \cap DomVal^{\Sigma}_{dm_2} = W = \{l_{dm_2}, \ldots, u_{dm_1}\}$. Then to compute the split point for this case, we use the equation*

$$sp = (u_{dm_1} - l_{dm_2}) * i + l_{dm_2} \qquad (\text{by } \texttt{GetSplit})$$

   and $\forall i \; 0 \leq i \leq 1$ we get $l_{dm_2} \leq sp \leq u_{dm_1}$. It follows that we have $sp \in W$.

This completes the proof. ∎

Lemma 6.1 shows that when the function `GetSplit` computes a split point between a pair of intersecting domains of two program input variables, the split point is chosen to be a value contained within the overlapping region of the two domains so that the dynamic domain reduction algorithm *safely* reduces the two domains. The domain of a program input variable is then modified (reduced) by procedure `Update`, using the calculated split point. Next we present a lemma that proves `Update` always reduces a given domain of a program input variable *DomVal* by including a subset of values of the program input variables that is part of the ddrAlg solution.

The Correctness of Procedure `Update`:

Let us assume that for the set of states of a program P, a domain of the program input variable, dm, is a set of sub-domains, i.e.

$$DomVal^{\Sigma}_{dm} =< [l_1, u_1], \dots, [l_n, u_n] >$$

where $l_{dm} = l_1 = \min(DomVal^{\Sigma}_{dm})$ and $u_{dm} = u_n = \max(DomVal^{\Sigma}_{dm})$. The function `Update` takes the domain of dm as an input and updates the domain such that the output is a subset of the input domain. That is $Update : DomVal^{\Sigma}_{dm} \rightarrow \overline{DomVal}^{\Sigma}_{dm}$, and $\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$.

Lemma 6.2. *Given the set of possible states Σ of a program P, $\exists \pi \in \Pi$ of the CFG of P, and for a program input variable (data manipulator) dm in π let $\overline{DomVal}^{\Sigma}_{dm} = Update(DomVal^{\Sigma}_{dm})$. Whenever the function Update is feasible (i.e. it returns true), then*

$$\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$$

Proof *(Proof by Contradiction) We let l and u be the new minimum and maximum values of the domain of dm. Assume to the contrary that the function* ***`Update`*** *is feasible and the computed new domain values $\overline{DomVal}^{\Sigma}_{dm}$ are not contained in the original domain values, i.e.*

$$\overline{DomVal}^{\Sigma}_{dm} \nsubseteq DomVal^{\Sigma}_{dm}$$

Then we have $\exists s \in \overline{DomVal}^{\Sigma}_{dm}$ *such that* $s \notin DomVal^{\Sigma}_{dm}$. *There are two cases in which the function* **`Update`** *is feasible and* $\overline{DomVal}^{\Sigma}_{dm}$ *is produced.*

- *When* $l \leq l_1 \wedge u \geq u_n$, *we note that the current* $DomVal^{\Sigma}_{dm}$ *is contained within the given* l *and* u *and the function does not need to modify* $DomVal^{\Sigma}_{dm}$. *Thus,* $\overline{DomVal}^{\Sigma}_{dm} = DomVal^{\Sigma}_{dm}$, *which contradicts our assumption.*
- *When* $DomVal^{\Sigma}_{dm}$ *is not fully contained within the given limits, the function* **`Update`** *attempts to reduce* $DomVal^{\Sigma}_{dm}$. *Let* k *be a set of domain values in* $DomVal^{\Sigma}_{dm}$ *where* $1 \leq k \leq n$, $n = |DomVal^{\Sigma}_{dm}|$ *and*

$$DomVal^{\Sigma}_{dm} =< [l_1, u_1], \ldots, [l_n, \ldots, u_n] >,$$

the function modifies the domain of dm *via the value limits* u_{dm} *and* l_{dm} *in two steps:*

1. u_{dm} *is handled, and for each set* k *in the domain, the function proceeds in three cases:*
   - *When* $u \leq u_n$ *the function does not need to change* u_{dm}.
   - *When* $u \geq l_k \wedge u \leq u_k$, *the function removes the subdomains* $D =< [l_k+1, u_k], \ldots, [l_n, u_n] >$ *and produces the modified domain* $\overline{DomVal}^{\Sigma}_{dm} = DomVal^{\Sigma}_{dm} \setminus D$. *It follows that we have* $\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$, *which is the same contradiction.*
   - *When* $u > u_k \wedge u < l_k + 1$, *the function removes the subdomains* $D =< [l_k + 1, u_k + 1], \ldots, [l_n, u_n] >$ *and produces the modified domain* $\overline{DomVal}^{\Sigma}_{dm} = DomVal^{\Sigma}_{dm} \setminus D$. *It follows that we have* $\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$ *and we get the same contradiction.*
2. *At this point, the domain of* dm *may be updated by one of the cases in the previous step, thus, we let*

$$DomVal^{\Sigma}_{dm} =< [l_1, u_1], \ldots, [l_m, \ldots, u_m] >$$

   where $m \leq n$. *Then for each set* k, $1 \leq k \leq m$, *in the domain, the function modifies* l_{dm} *in one of the following three cases:*
   - *When* $l \leq l_1$ *the function does not need to change* l_{dm}.

- *When $l \geq l_k \wedge l \leq u_k$, the function replaces the subdomain $[l_k, u_k]$ with $[l, u_k]$, and removes the subdomains $D =< [l_k, u_k], \ldots, [l_k - 1, u_k - 1] >$ from $DomVal^{\Sigma}_{dm}$. Then, the modified domain is $\overline{DomVal}^{\Sigma}_{dm} = DomVal^{\Sigma}_{dm} \setminus D$. It follows that we have $\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$, which is the same contradiction.*
- *When $l > u_k \wedge l < l_k + 1$, the function removes the subdomains $D =< [l_1, u_1], \ldots, [l_k, u_k] >$ from the variable domain, shifts the domain list where the $i + 1$th subdomain becomes the first subdomain and produces the modified domain $\overline{DomVal}^{\Sigma}_{dm} = DomVal^{\Sigma}_{dm} \setminus D$. It follows that we have the same contradiction that $\overline{DomVal}^{\Sigma}_{dm} \subseteq DomVal^{\Sigma}_{dm}$.*

3. *This step is only applied when the algorithm handles bitwise expressions of type **`AND`** and **`OR`**, and where the values of some bits of the new domain limit l and u may be set to "x" (i.e.,* do-not-care *values) meaning that the bits are allowed to be assigned values 0 or 1. These* do-not-care *bits are not considered during the update procedure; however, the bits with 1 values are used to determine which values in $DomVal^{\Sigma}_{dm}$ should be discarded and to reduce the domain. In this case, the domain of dm is modified such that it only contains values that agree with the new limits, i.e. l and u of the domain. Therefore, the domain of dm might be reduced further by removing some subdomains that contain irrelevant values. Assume that there exists some subdomain $subdomVal \in DomVal^{\Sigma}_{dm}$, and $\exists val \in subdomVal$ s.t. $val \notin [l, u]$ (the new domain of dm), then when function **`Update`** checks that the bits of val do not match with $1'$s bits in l and u (Step 2, lines 18-23 in Algorithm 6.7 on page 170), subdomVal is removed from the domain. Eventually, this step walks through all subdomains and whenever it finds incorrect values the domain of dm is reduced and computes a domain $\overline{DomVal}^{\Sigma}_{dm}$ which is a contradiction.*

This completes the proof. ∎

The Correctness of ddrAlg:

Next we show that for any given complete path π, whenever ddrAlg computes a solution $Đ_{\pi}$ for the test data, the computed domains of the program input variables in $Đ_{\pi}$ are always contained within the initial domains of the program input variables.

Lemma 6.3. $\forall\pi \in \Pi$ *of the CFG of a program* P, $\forall s \in \pi, \forall\sigma \in \Sigma$ *let* $Đ_\pi = \text{ddrAlg}(\pi)$ *where* $\pi =< s_1, \ldots, s_k >$ *and* $k \geq 1$. *Also, let* $\pi' = \pi.s$ *where* s *is a path statement in the CFG such that* π' *is produced by extending* π *with an adjacent statement,* s*, of the statement* s_k *in the program CFG. Whenever* $Đ_{\pi'} = \text{ddrAlg}(\pi')$*, it holds that*

$$Đ_{\pi'} \subseteq Đ_\pi$$

Proof *(By induction on the length of* π*) Assume that for path* π *the algorithm produces* $Đ_\pi = \text{ddrAlg}(\pi)$. *Next we add one more path statement* s_{k+1} *to* π *such that* $\pi' =< s_1, \ldots, s_k, s_{k+1} >$. *In order to compute* $Đ_{\pi'} = \text{ddrAlg}(\pi')$ *it is sufficient to apply the algorithm on the new added statement* s_{k+1} *with the current domain of input variables* $Đ_\pi$. *To this end, there are two cases to consider for* s_{k+1}*:*

- *If* s_{k+1} *is an action statement, e.g.,* $s_{k+1} = \text{dm} := \text{e}$, ***PUSH*** e, ***POP*** dm, ... *etc., then the algorithm ddrAlg symbolically evaluates the statement, creates a new domain for the expression in the statement and the domain of inputs* $Đ_\pi$ *is not affected in this case. Thus, the new domain is* $Đ_{\pi'} = Đ_\pi = \text{ddrAlg}(\pi')$.

- *If* s_{k+1} *is a Boolean statement, i.e.,* $s_{k+1} = B$*, then the algorithm checks that the current domains of the inputs satisfy the Boolean expression* B*, and it makes use of the functions* ***GetSplit*** *and* ***Update*** *in order to reduce the domains of sub-expressions involved if* B *was not satisfied. Two cases for the outcome of the Boolean expression* B *need to be considered:*

  – *When* $B = True$ *the constraint is satisfied by the current domain, i.e.,* $Đ_\pi$ *and the algorithm does not need to modify the domains. Thus,* $Đ_{\pi'} = Đ_\pi = \text{ddrAlg}(\pi')$.

  – *When* $B = False$*, the constraint is not satisfied and the algorithm attempts to modify* $Đ_\pi$ *to satisfy the constraint. Thus, the algorithm uses the functions* ***GetSplit*** *and* ***Update*** *to reduce the current domain. By Lemmas 6.1 and 6.2, we infer that whenever the function* ***GetSplit*** *finds a split point sp and the function* ***Update*** *is feasible w.r.t. to sp, the produced domain is contained within the original domain, i.e.* $Đ_{\pi'} \subseteq Đ_\pi$.

This completes the proof. ∎

Now we are in the position to establish the desired correctness result of the dynamic domain reduction algorithm.

Theorem 6.1. *Given an* AAPL program P, $\forall \pi \in \Pi$ of the CFG of P, we denote by $[\![\cdot]\!]'$ the reachability semantics function for computing the reachable output states $\mathcal{R}$ for π. If $Ð_\pi = \text{ddrAlg}(\pi)$, then $\forall \sigma \in Ð_\pi$, we have

$$[\![\pi]\!]\sigma \in [\![\pi]\!]'\Sigma$$

Proof Assume $\forall \sigma \in Ð_\pi$, $[\![\pi]\!]\sigma \in [\![\pi]\!]'\Sigma$ (IH).

Let $Ð_{\pi.s} = \text{ddrAlg}(\pi.\text{s})$, and by Lemma 6.3 and by IH, we infer that $\forall \sigma \in Ð_{\pi.s}$, and we have $\sigma \in Ð_\pi$ and $[\![\pi]\!]\sigma \in [\![\pi]\!]'\Sigma$. Therefore, our proof must show that $[\![\pi.s]\!]\sigma \in [\![\pi.s]\!]'\Sigma$. Also, we know that $[\![\pi.s]\!]\sigma = [\![s]\!]([\![\pi]\!]\sigma)$, thus, there are four cases for statement s to be considered:

- Case $s = dm := e$. Here we consider the case where the data manipulator, dm, is equal to some expression e. Assume that expression e is equal to some value v in the reachability semantics rule $f_e[\![e]\!]\Sigma = v$. We know that dm is assigned to some outcome of the evaluation of expression e. We apply the reachability semantics rule $[\![dm := e]\!]'\Sigma = \{[\![\pi]\!]\sigma[dm \mapsto \{v\}] \mid [\![\pi]\!]\sigma \in \Sigma \wedge \{v \in f_e[\![e]\!]([\![\pi]\!]\sigma)\}\}$, which produces a final state where $([\![\pi]\!]\sigma)[dm \mapsto \{v\}]$. We know that $< dm, ([\![\pi]\!]\sigma) > \rightarrow_E < v, ([\![\pi]\!]\sigma) >$ holds using the small-step semantics axiom of a data manipulator. So, we can conclude that $[\![dm := e]\!]([\![\pi]\!]\sigma) \in [\![dm := e]\!]'\Sigma$.

- Case $s = \texttt{PUSH}\ e$. Here we consider the case where a memory location (a data manipulator, dm) defined by the stack,SP, is assigned a value (the outcome of the evaluation of expression e). We apply the reachability semantics rule of the command: $[\![\texttt{PUSH}\ e]\!]'\Sigma = \Sigma' = \{\sigma[dm \mapsto \{v\}] \mid \sigma \in \Sigma, SP \mapsto SP{-}1, dm = SP, \{v \in f_e[\![e]\!]\Sigma\}\}$, and a final state is produced where $([\![\pi]\!]\sigma)[dm \mapsto \{v\}]$. We know that $< dm, ([\![\pi]\!]\sigma) > \rightarrow_E < v, ([\![\pi]\!]\sigma) >$ holds using the small-step semantics axiom of a data manipulator. So, we can conclude that $[\![\texttt{PUSH}\ e]\!]([\![\pi]\!]\sigma) \in [\![\texttt{PUSH}\ e]\!]'\Sigma$.

- Case $s = \texttt{POP}\ dm$. Here we consider the case where the data manipulator, dm is assigned to a value from the stack (i.e. pointed to by the stack register SP). Assume that the stack register is evaluated to some value v in the reachability semantics rule $f_e[\![SP]\!]\Sigma = v$. We apply the reachability semantics rule

$[\![\texttt{POP}\ dm]\!]'\Sigma = \{[\![\pi]\!]\sigma[dm \mapsto \{v\}] \mid [\![\pi]\!]\sigma \in \Sigma \wedge \{v \in f_e[\![SP]\!]([\![\pi]\!]\sigma)\}\} \cup \Sigma$, which produces a final state where $([\![\pi]\!]\sigma)[dm \mapsto \{v\}]$. We know that $< dm, ([\![\pi]\!]\sigma) > \rightarrow_E < v, ([\![\pi]\!]\sigma) >$ holds using the small-step semantics axiom of a data manipulator. So, we can conclude that $[\![\texttt{POP}\ dm]\!]([\![\pi]\!]\sigma) \in [\![\texttt{POP}\ dm]\!]'\Sigma$.

- Case $s = B$. Here we consider the case when the statement is a Boolean expression. Assume that the evaluation of B is true and $f_b[\![B]\!]\Sigma = \Sigma'$, then the one rule in the reachability semantics whose outcome looks like this is $f_b[\![B]\!]\Sigma = \{([\![\pi]\!]\sigma) \in \Sigma \mid ([\![\pi]\!]\sigma) \vdash B \Rightarrow True\} \cup \Sigma$. This rule requires $B = True$ in $([\![\pi]\!]\sigma)$. Since $B = True$ in $([\![\pi]\!]\sigma)$, we know that $< B > \rightarrow_{BExp} < True, ([\![\pi]\!]\sigma) >$ holds by using the inference rule of small-step semantics. So we conclude that $[\![B]\!]([\![\pi]\!]\sigma) \in [\![B]\!]'\Sigma$. ∎

6.6 Related Work

The goal of our work is to extend the dynamic domain reduction technique to a low-level programming language, namely AAPL. The extended technique allows us to generate input test cases for all possible feasible execution paths. A more complete picture of the trace semantics of a (malicious) code sample can be produced using test cases. This is similar to software testing where many test input generation techniques [BJS+06, DO91, GBR98, GMS98] have been developed to analyse programs in an attempt to find inputs that trigger bugs. They differ from our extension of the DDR technique because the goal of these approaches is to reach a certain node in the control flow of a program, and not to capture the complete program execution trace semantics. Other testing techniques have been proposed that cover multiple paths of a program to detect program errors. For example, [CGP+06] and [MKK07a] presented systems that can explore multiple program execution paths that depend on interesting input. Both systems use a similar symbolic execution technique to handle specific inputs. However, the main differences from the extended dynamic domain reduction approach are that, first, in [CGP+06] the goal is to search for execution paths that lead to program bugs, and second, in [MKK07a] the goal is to record comprehensive behavioural profiles of malicious code by saving snapshots of the program environment and, thus, explore alternative execution paths. While the objective of the DDR approach is to create a set of test data in which all feasible program paths can be

exercised. The system that is closest to the extended DDR approach is [ARJ07]. The system allows automatically generated input sequences to exercise a high coverage of a low-level program code and is based on static binary analysis and symbolic propagation. That is, symbolic formulas are generated, which represent input constraints at input-dependent program instructions, and these depend on the symbolically propagated program input values. Then the system can identify different input-dependent paths. Similar to our approach, the system can operate on low-level binary programs and it covers all possible program paths in the program control flow graph. The system produces an input sequence, which triggers the execution of a new path. The method of generating test data in their system is similar to that in the DDR extension for AAPL. The difference is that their approach requires the translation of input-dependent instructions into a set of Simple Theorem Prover (STP) [Gan07] conditional formulas in order to validate path conditions and generate test data. Also, none of the above testing techniques have been proved to be correct with respect to their computed test input values, while we show that the extended DDR method for low-level programs produces a subset of the value domains of the program inputs, which can be used as test cases to traverse the feasible program paths.

Constraint Logic Programming (CLP) techniques have been applied to test data generation. For solving a TDG problem, a CLP technique dynamically builds a constraint system, which consists of program input variables, domains and constraints. The process of solving a constraint system has three main components: 1) a constraint propagation method, which makes use of the constraints to reduce the search space, 2) a constraint entailment method, which tries to produce new constraints from existing ones, 3) a labelling procedure. Unlike symbolic-based approaches to test data generation, the CLP approach integrates path selection and constraint-solving steps into a single step. That is, symbolic execution of the CLP-translated program can be performed by relying on the standard evaluation mechanism of CLP, which provides backtracking and handling of symbolic expressions for free. A CLP-based method, which tackles the problem of automatic test data generation, is described in Gotlieb et al. [GBR00]. A CLP system is given to find program test data. A constraint system over CLP is generated and solved to check whether at least one feasible control flow path passing through the selected point exists, and to generate test data automatically, which correspond to one of these paths. The main idea of the approach is the use of constraint entailment techniques to reduce the search space efficiently. In the proposed CLP framework,

test data can be generated without following a path through the program. Also, the approach proposed in [GzAP10] presents a whole test data generation framework for an object-oriented (OO) imperative language using CLP. The approach has two steps: first, the imperative program is compiled into an equivalent CLP program and, second, the test data generation process is performed on the CLP program by relying only on CLP's evaluation mechanisms. This approach has one main advantage in that the whole test data generation process is formulated using CLP only. That is, their method translates the program into a constraint logic program for which symbolic execution is performed (without the need to define specific constraint operators). However, our method for finding test data was developed to consider low-level programming language features, incorporating backtracking and domain-splitting mechanisms.

6.7 Conclusion

In this chapter, we developed an extended version of the Dynamic Domain Reduction method (DDR) for AAPL programs. The goal of the DDR technique is to automatically find a domain of program inputs that can be used to produce test data for feasible program paths. To adapt this method for low-level executable malware programs, first, we developed an extension of the DDR algorithm for analysing executable malware programs (low-level code). Then we showed that the DDR method is correct in producing an under-approximation solution (a subset) from the initial domains of the program input variables. With automated test data generation using the extended DDR technique, comprehensive analysis of feasible program paths can be performed. In addition, the method automatically provides safe under-approximation program inputs, which can be used by malware detection systems to trigger and capture malicious behaviour. In particular, this method can improve our semantic signature generation by simulating more feasible paths with test cases, and, hence, extracting new semantic traces. The examples provided in this chapter demonstrate that the extended DDR algorithm for low-level programs computes a subset of values for program input variables as test data for a feasible path.

Chapter 7

Conclusion and Future Work

Malware has become a profitable business model for cybercriminals. Today, more than ever before, malware constantly poses a greater threat to individuals, businesses and government infrastructures. To overwhelm the power of existing AV tools, malware writers have increasingly used polymorphism and metamorphism techniques for increasing the production of stealthy variants of known malware.

In this dissertation, we presented new methods for the analysis and detection of (possibly obfuscated) new malware variants. The main idea of our approach is that (part of) the semantics of malware code, preserved across successive variants of the malware, can be used as a signature for detecting code variants. As a step towards tackling the malware variant detection problem using this idea, we introduced program trace semantics, or semantic signatures, as an abstraction of code semantics for identifying new instances of the malware. In Chapter 4, we described a trace-slicing method and showed that the method produces a correct trace slice with respect to given trace-slicing criteria for executables. The slicing step in the signature generation phase helps in tackling the effects of a class of obfuscating techniques in subsequent code variants and allows for improved efficiency of detection.

Our malware variant detection system in Chapter 5 is based on matching the semantics of traces of simulated malware code. A semantic simulator is developed for the system to simulate malware executable files and to capture semantic traces. That is, the semantic simulator is a static program analyser for simulating the execution of abstract machine code. We developed a matching algorithm for identifying malicious code instances. We demonstrated how the trace-slicing

method for generating semantic signatures produces an improvement in detection times and detection accuracy compared with detection without the trace-slicing method.

We proposed a testing analysis technique, and in Chapter 6 we proved it to be a correct analysis for approximating the semantics of malicious executables through identifying a set of test inputs for a set of feasible program paths. A test input is part of the semantic signature of the malicious program, which can be used to identify some of the malicious behaviour in new variants of the program.

Our proposed approach cannot solve the problem of malware detection, but it provides an important step towards incorporating program trace semantics for detection of malicious code instances and, thus, raising the bar for malware writers. We believe that current malware detection tools can be amended with our method to tackle new, unrecognised variants of an existing malware. Our method can be utilised in much the same way that signatures are currently used to detect malware, however, we can clearly enhance and decentralise the phases of distributing new semantic signatures and recognising new malware variants. Also, a possibility, as a particular use of our tool, is the use of the approach as a classification tool for analysing variants of malware samples at AV laboratories.

We envision that our approach will pave the way and inspire researchers in this area to arrive at more systematic methods for tackling some issues related to the semantics-based approach in the following areas:

1. **Discover hidden abstract traces.** This dissertation only presents semantics-based techniques for detecting malicious executables but does not explore the solution space of handling malware variants with dynamic code generation capabilities. We are interested in dealing with dynamic code obfuscation techniques via deploying a hybrid technique to malware variant detection. In this case, we may apply virtualisation techniques [VMw, Ora] to partially allow malware, possibly with dynamic code obfuscation, to generate its code such that other malicious program paths (abstract traces) can be decoded on-the-fly. Our testing method could benefit from a hybrid technique by covering more aspects of the malware code and capture more of its semantics.

2. **Reasoning about program semantics approximation.** Another possibility for future work is to investigate a framework for determining the set of

approximate semantic traces with respect to possible program paths in the control flow graph of a malicious program. As a first step in this direction, we observe that for each (unique) feasible program path in a program CFG, there may exist a set of simulation (concrete) traces that might have similar semantics. Hence, for future work, an abstraction method could be developed to characterise the relation between the abstract environment, i.e. the control flow graph, and the concrete environment, i.e. semantic traces. This method may be described as a Galois relation between two domains and may help us in reasoning about how to minimise the number of false negatives in matching trace slices of program variants.

3. **Detection performance improvements.** Parallel implementations of the detection algorithm may help to improve the efficiency of a semantics-based malware detector when extracting and matching malware signatures. Currently, our techniques, proposed in this dissertation, for analysing a known malware program and generating a signature, perform slowly because they only handle one malware sample at a time. One area we believe major improvements can be made is by using General-Purpose computing on Graphics Processing Units (GPGPU) [LHK+04, SC07]. GPGPU has drastically evolved in recent years and provides software developers with the inexpensive, massive computation power available in Graphics Processing Units (GPUs). This power can be used to code and run data-intensive algorithms that up till now were traditionally accommodated by the central processing unit (CPU). This technique may enhance the malware detection phase; in particular, it could speed up the matching process by handling a large set of suspicious programs simultaneously, quickly determine if a malicious code is not a variant of an existing malware, and, hence, a new semantic signature can be extracted automatically for the new malware.

We believe that the semantics-based approach to malware detection has the potential to strengthen AV scanners on clients' machines with semantics-enabled malware detectors. Future solutions to the above mentioned issues will improve the contributions of this dissertation in developing stronger malware detection tools.

Bibliography

[ADS91a] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 60–73, New York, NY, USA, 1991. ACM.

[ADS91b] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8:21–26, May 1991.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256. ACM, 1990.

[AHJD10] S. S. Anju, P. Harmya, Noopa Jagadeesh, and R. Darsana. Malware detection using assembly code and control flow graph optimization. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, A2CWiC '10, pages 65:1–65:4, New York, NY, USA, 2010. ACM.

[AIP04] Tankut Akgul, Vincent J. Mooney III, and Santosh Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 522–531. IEEE Computer Society, 2004.

[Alz10a] Khalid Alzarooni. `asm2aapl`: Assembly programming language translator for abstract machine code (AAPL), August 2010. Published online at `http://www.cs.ucl.ac.uk/people/K.Alzarooni/projfiles/asm2aapl`.

[Alz10b] Khalid Alzarooni. `TSAlgo`: Trace-slicing algortihm for binary executable code, August 2010. Published online at `http://www.cs.ucl.ac.uk/staff/K.Alzarooni/projfiles/TSAlgo`.

[Alz11] Khalid Alzarooni. A prototype detector for malware variant detection, May 2011. Published online at `http://www.cs.ucl.ac.uk/staff/K.Alzarooni/projfiles/semsim`.

[ARJ07] N. Aaraj, A. Raghunathan, and N.K. Jha. Virtualization-assisted framework for prevention of software vulnerability based security attacks. Technical Report CE-J07-001, Dept. of Electrical Engineering, Princeton University, December 2007.

[ARJ08] Najwa Aaraj, Anand Raghunathan, and Niraj K. Jha. Dynamic binary instrumentation-based framework for malware defense. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 64–87, Berlin, Heidelberg, 2008. Springer-Verlag.

[BCM04] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 326–335. IEEE Computer Society, 2004.

[BDEK99] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, WETICE '99, pages 184–189. IEEE Computer Society, 1999.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10:234–245, April 1975.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[BGM10] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag.

[BGS+01] Árpád Beszedes, Tamás Gergely, Zsolt Mihály Szabó, Janos Csirik, and Tibor Gyimothy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, CSMR '01. IEEE Computer Society, 2001.

[BHSP08] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*, pages 225–238, New York, NY, USA, 2008. ACM.

[BJS+06] David Brumley, Newso James, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[BKK06] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference*, EICAR'06, 2006.

[BM09] Philippe Beaucamps and Jean-Yves Marion. On behavioral detection. In *Proceedings of the 18th European Institute for Computer Antivirus Research Annual Conference*, EICAR'09, 2009.

[BMM07] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007.

[BOA+07] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection*,

RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CC02] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, Jan. 2002. ACM Press, New York, NY.

[CDH+03] J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings Software*, 150(3):161 – 175, June 2003.

[CF97] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, page 188. IEEE Computer Society, 1997.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM.

[CHY11] Patrick Chan, Lucas Hui, and S. Yiu. Dynamic software birthmark for java based on heap memory analysis. In *Communications and Multimedia Security*, volume 7025 of *Lecture Notes in*

Computer Science, pages 94–107. Springer Berlin / Heidelberg, 2011.

[CJ04] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2004 (ISSTA'04)*, pages 34–44, Boston, MA, USA, July 2004. ACM.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[CL06] Mohamed R. Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In *Proceedings of the 4th ACM workshop on Recurring malcode*, WORM '06, pages 73–78, New York, NY, USA, 2006. ACM.

[CMJ+10] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. SplitScreen: enabling efficient, distributed malware detection. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 25–25, Berkeley, CA, USA, 2010. USENIX Association.

[Coh87] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6:22–35, February 1987.

[Col08] Michael Collins. *A Protocol Graph Based Anomaly Detection System*. PhD thesis, Carnegie Mellon University, 2008.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, The University of Auckland, July 1997.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM.

[CW00] David M. Chess and Steve R. White. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference*, VB2000, 2000.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[DEMDS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22:378–415, March 2000.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17:900–910, September 1991.

[DO93] Richard A. DeMillo and A. Jefferson Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2:109–127, April 1993.

[Far02] Csaba Faragó. Union slices for program maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM'02, Washington, DC, USA, 2002. IEEE Computer Society.

[FDJ09] Zhang FuYong, Qi DeYu, and Hu JingLin. MBMAS: A system for malware behavior monitor and analysis. In *International Symposium on Computer Network and Multimedia Technology*, CNMT 2009, pages 1–4, January 2009.

[FG09] Min Feng and R. Gupta. Detecting virus mutations via dynamic matching. In *Proceedings of the 25th IEEE International Conference on Software Maintenance*, ICSM '09, pages 105 –114, September 2009.

[fsg] Fast Small Good (FSG) - a malware packer. Published online at `http://in4k.untergrund.net/index.php?title=Exe_Tweakers_and_Linkers#EXE_Packers_.28and_Linkers.29`. Last accessed on 10 May 2010.

[FSR09] M. Feily, A. Shahrestani, and S. Ramadass. A survey of botnet and botnet detection. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09. Third International Conference on*, pages 268 –273, June 2009.

[Gan07] Vijay Ganesh. *Decision procedures for bit-vectors, arrays and integers*. PhD thesis, Stanford, CA, USA, 2007.

[GBMKJYM07] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow to detect malware. In *Proceedings of the Inter-Regional Workshop on Rigorous System Development and Analysis*, Nancy France, 2007.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Software Engineering Notes*, 23(2):53–62, 1998.

[GBR00] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. A CLP framework for computing structural test data. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 399–413, London, UK, 2000. Springer-Verlag.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. *SIGSOFT Software Engineering Notes*, 23(6):231–244, 1998.

[GSHC09] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 101–120, Berlin, Heidelberg, 2009. Springer-Verlag.

[GzAP10] Miguel GÓmez-zamalloa, Elvira Albert, and GermÁn Puebla. Test case generation for object-oriented imperative languages in CLP. *Theory and Practice of Logic Programming*, 10:659–674, July 2010.

[Har07] Mark Harman. Automated test data generation using search based software engineering. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.

[Hea] VX Heavens. Published online at `http://vx.netlux.org/`. Last accessed on 17 January 2011.

[HKV07] Andreas Holzer, Johannes Kinder, and Helmut Veith. Using verification technology to specify and detect malware. In *Proceedings of the 11th International Conference on Computer Aided Systems Theory*, EUROCAST'07, pages 497–504, Berlin, Heidelberg, 2007. Springer-Verlag.

[HMK06] Christian Hammer, Grim Martin, and Jens Krinke. Dynamic path conditions in dependence graphs. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Mrogram Manipulation*, PEPM '06, pages 58–67, New York, NY, USA, 2006. ACM.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. *ACM SIGPLAN Notices*, 24(7):28–40, 1989.

[IM07] N. Idika and A. P. Mathur. A survey of malware detection techniques. *SERC Technical Reports*, (SERC-TR-286), March 2007.

[Int] Intel. IA-32 architectures software developer's manuals. Published online at `http://developer.intel.com/products/processor/manuals/index.htm`. Last accessed on 11 December 2010.

[Int90] CORPORATE Intel Corp. *i486 microprocessor programmer's reference manual*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1990.

[JDF08] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy.

Journal in Computer Virology, 4:251–266, 2008. 10.1007/s11416-008-0086-0.

[JM08] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2nd edition, 2008.

[KGG07] Abhishek Karnik, Suchandra Goswami, and Ratan Guha. Detecting obfuscated viruses using cosine similarity analysis. In *Proceedings of the First Asia International Conference on Modelling and Simulation*, AMS '07, pages 165–170, Washington, DC, USA, 2007. IEEE Computer Society.

[Kin75] James C. King. A new approach to program testing. *ACM SIGPLAN Notes*, 10:228–233, April 1975.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, July 1976.

[KJLG03] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '03, pages 118–127, Amsterdam, The Netherlands, September 2003. IEEE Computer Society.

[KKB+06] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th conference on USENIX Security Symposium*, volume 15, Berkeley, CA, USA, 2006. USENIX Association.

[KKSV05] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, July 2005.

[KKSV10] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Proactive detection of computer worms using model checking. *IEEE Transactions on Dependable and Secure Computing*, 7:424–438, October 2010.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[KL90] Bogdan Korel and Janusz Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13(3):187–195, 1990.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[KRT10] P. K. Kerr, John Rollins, and C. A. Theohary. The stuxnet computer worm: Harbinger of an emerging warfare capability. Technical Report R41524, Congressional Research Service, December 2010.

[Kuz07] *On the Concept of Software Obfuscation in Computer Security*, volume 4779 of *Lecture Notes in Computer Science*. Springer, 2007.

[KY94] Bogdan Korel and Satish Yalamanchili. Forward computation of dynamic program slices. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 66–79, New York, NY, USA, 1994. ACM.

[LB97] James R. Lyle and David Binkley. Program slicing in the presence of pointers (extended abstract), 1997. Published online at `http://hissa.nist.gov/unravel/papers/serf.ps`. Last accessed on 12 July 2008.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[LHK+04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[LJL10] Jusuk Lee, Kyoochang Jeong, and Heejo Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1970–1977, New York, NY, USA, 2010. ACM.

[LRHK10] Mu-Woong Lee, Jong-Won Roh, Seung-won Hwang, and Sunghun Kim. Instant code clone search. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 167–176, New York, NY, USA, 2010. ACM.

[MC04] Ginger Myles and Christian S. Collberg. Detecting software theft via whole program path birthmarks. In Kan Zhang and Yuliang Zheng, editors, *Proceedings of the 7th International Conference on Information Security (ISC '04)*, volume 3225 of *Lecture Notes in Computer Science*, pages 404–415, Palo Alto, CA, USA, 2004. Springer.

[McG] Paul McGuire. The pyparsing module. Published online at `http://pyparsing.wikispaces.com/`. Last accessed on 21 January 2011.

[McM04] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Software Testing, Verification and Reliability*, 14:105–156, June 2004.

[MDT07] Anirban Majumdar, Stephen J. Drape, and Clark D. Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *Proceedings of the 2007 ACM workshop on Digital Rights Management*, DRM '07, pages 70–81. ACM, 2007.

[MFT+08] Robert Moskovitch, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. Unknown malcode detection using opcode representation. In *Proceedings*

of the 1st European Conference on Intelligence and Security Informatics, EuroISI '08, pages 204–215, Berlin, Heidelberg, 2008. Springer-Verlag.

[MKK07a] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[MKK07b] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference*, ACSAC '07, pages 421–430, 2007.

[MKM+06] Durga P. Mohapatra, Rajeev Kumar, Rajib Mall, D. S. Kumar, and Mayank Bhasin. Distributed dynamic slicing of Java programs. *Journal of Systems and Software*, 79:1661–1678, December 2006.

[MM06] G. B. Mund and Rajib Mall. An efficient interprocedural dynamic slicing method. *Journal of Systems and Software*, 79:791–806, June 2006.

[MME+07] Shinsuke Miwa, Toshiyuki Miyachi, Masashi Eto, Masashi Yoshizumi, and Yoichi Shinoda. Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, Berkeley, CA, USA, 2007. USENIX Association.

[MMS02] G. B. Mund, Rajib Mall, and S. Sarkar. An efficient dynamic program slicing technique. *Information and Software Technology*, 44(2):123–132, 2002.

[MMS03] G. B. Mund, R. Mall, and S. Sarkar. Computation of intraprocedural dynamic program slices. *Information and Software Technology*, 45(8):499–512, 2003.

[Mor01] P. Morley. Processing virus collections. In *Proceedings of the 2001 Virus Bulletin Conference*, VB '01, 2001.

[MRD02] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques 2002*, PACT 02, Charlottesville, Virginia, September 2002.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1st edition, August 1997.

[Nac97] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40:46–51, January 1997.

[Nac10] Carey Nachenberg. All about computer viruses with software installing, February 2010. Published online at `http://www.articlesbase.com/software-articles/all-about-computer-viruses-with-software-installing-/1830222.html`. Last accessed on 17 July 2010.

[Net04] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, United Kingdom, 2004.

[OJP94] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation: design and algorithms. Technical Report ISSE-TR-94-110, ISSE Department, George Mason University, August 1994.

[OJP99] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. *Software: Practice and Experience*, 29(2):167–193, 1999.

[Ora] Oracle. VM Virtualbox 4.0. Published online at `http://www.oracle.com/us/technologies/virtualization/oraclevm/061976.html`. Last accessed on 20 March 2011.

[OSH01] A. Orso, S. Sinha, and M.J. Harrold. Effects of pointers on data dependences. In *Proceedings of the 9th International Workshop on Program Comprehension*, 2001.

[Pan08] PandaLab. The amount of new malware that appeared in 2007 increased tenfold with respect to the previous year, January 2008. Press Releases. Published online at `http://`

www.pandasecurity.com/homeusers/media/press-releases/viewnews?noticia=9077&sitepanda=particulares. Last accessed on 17 November 2008.

[Pan10] PandaLabs. Pandalabs annual malware report 2009, January 2010. Published online at http://www.pandasecurity.com/img/enc/Annual_Report_PandaLabs_2009.pdf. Last accessed on 23 April 2010.

[PCJD07] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 377–388. ACM Press, 2007.

[PCJD08] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems*, 30:25:1–25:54, September 2008.

[PP07] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12):3448–3470, 2007.

[PTK11] Jeremy Pate, Robert Tairas, and Nicholas Kraft. Clone Evolution: A Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, September 2011.

[QL09] Daniel Quist and Lorie Liebrock. Visualizing compiled executables for malware analysis. In *Proceedings of the 6th International Workshop on Visualization for Cyber Security*, VizSec '09, pages 27–32, Atlantic City, NJ, October 2009. IEEE.

[RC07] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. TR 2007-541, School of Computing, Queen's University, 2007.

[Res] Data Rescure. IDA Pro disassembler: Multi-processor, windows hosted disassembler and debugger. Published online at http://www.datarescue.com/idabase/. Last accessed on 5 March 2008.

[Res08] Panda Research. Packer (r)evolution. March 2008. Published online at `http://research.pandasecurity.com/packer-revolution/`. Last accessed on 22 May 2008.

[RHW+08] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[RKLC03] Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski, and Robert K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, WORM '03, pages 76–82, New York, NY, USA, 2003. ACM.

[RLG02] Juergen Rilling, Hon F. Li, and Dhrubajyoti Goswami. Predicate-based dynamic slicing of message passing programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, Washington, DC, USA, 2002. IEEE Computer Society.

[RM10] Kevin A. Roundy and Barton P. Miller. Hybrid analysis and control of malware. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 317–338, Berlin, Heidelberg, 2010. Springer-Verlag.

[RRZ+09] Y. Robiah, S. Siti Rahayu, M. Mohd Zaki, S. Shahrin, M. A. Faizal, and R. Marliza. A new generic taxonomy on hybrid malware detection technique. *International Journal of Computer Science and Information Security*, 5, 2009.

[SC07] Jay Steele and Robert Cochran. Introduction to GPGPU programming. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACM-SE 45, pages 508–508, New York, NY, USA, 2007. ACM.

[SF01] Péter Ször and Peter Ferrie. Hunting for metamorphic. In *Proceedings of the 2001 Virus Bulletin Conference*, VB '01, pages

123–144, Prague, Czech Republic, September 2001. Virus Bulletin.

[SL03] P.K. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *Proceedings of the 2003 IEEE Systems, Man and Cybernetics Society Information Assurance Workshop*, IAW '03, pages 298–300, 2003.

[SMEG09] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1):16–29, 2009.

[Sop09] Sophos. MMX gives FakeAVs a new trick, April 2009. Published online at `http://nakedsecurity.sophos.com/2009/04/12/mmx-fakeav-clothes/`. Last accessed on 21 June 2009.

[Sop10] Sophos. Security threat report 2010, January 2010. Published online at `www.sophos.com/security/topic/security-report-2010.html`. Last accessed on 17 July 2010.

[Sop11] Sophos. Security threat report 2011, January 2011. Published online at `http://www.sophos.com/security/topic/security-threat-report-2011.html`. Last accessed on 23 April 2011.

[Ste07] Joe Stewart. Ollybone, January 2007. Published online at `http://www.joestewart.org/ollybone/`. Last accessed on 10 September 2010.

[STMF09] M. Zubair Shafiq, S. Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 121–141, Berlin, Heidelberg, 2009. Springer-Verlag.

[SVBY10] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. Pattern recognition techniques for the classification of malware packers. In *Information Security and Privacy*, volume 6168 of *Lecture Notes in Computer Science*, pages 370–390. Springer Berlin / Heidelberg, 2010.

[SWL08] Muazzam Siddiqui, Morgan C. Wang, and Joohan Lee. A survey of data mining techniques for malware detection using file features. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 509–510, New York, NY, USA, 2008. ACM.

[Sym03] Symantec. Internet security threat report, 2003. Published online at `http://www.securitystats.com/reports/Symantec-Internet_Security_Threat_Report_vIII.20030201.pdf`. Last accessed on 12 October 2007.

[Sym10] Symantec. Global internet security threat report trends for 2009, April 2010. Published online at `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xv_04-2010.en-us.pdf`. Last accessed on 17 July 2010.

[Szö05] Péter Ször. *The Art of Computer Virus Research and Defense.* Addison-Wesley, Reading, MA, USA, 2005.

[Tip94] Frank Tip. A survey of program slicing techniques. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1994.

[TONM04] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic software birthmarks to detect the theft of windows applications. In *Proceedings of the 8th International Symposium on Future Software Technology*, ISFST '04, Xian, China, 2004. SEA.

[upx10] Ultimate packer for executables (upx) disassembler, 2010. Published online at `http://upx.sourceforge.net/`. Last accessed on 10 May 2010.

[Val07] Danny Quist Valsmith. Covert debugging: Circumventing software armoring techniques. In *Black Hat Briefings*, USA, 2007.

[VMw] VMware. Virtualization via hypervisor, virtual machine and server consolidation. Published online at `http://www.vmware.com/virtualization/virtual-machine.html`. Last accessed on 15 August 2010.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29:97–107, July 2004.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[WJZL09a] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 280–290, New York, NY, USA, 2009. ACM.

[WJZL09b] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 149–158, Honolulu, Hawaii, December 2009. IEEE Computer Society.

[WR07] Tao Wang and Abhik Roychoudhury. Hierarchical dynamic slicing. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 228–238, New York, NY, USA, 2007. ACM.

[XQZ+05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[YHR89] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical Report 840, Department of Computer Sciences, University of Wisconsin, Madison, WI, USA, April 1989.

[Yus] Oleh Yuschuk. OllyDbg debugger. Published online at `http://www.ollydbg.de/`. Last accessed on 14 December 2010.

[YY10] I. Yim and K. You. Malware obfuscation techniques: A brief survey. In *Proceedings of the Fifth International Conference on Broadband, Wireless Computing Communication and Applications*, BWCCA '10, pages 297–300, 2010.

[ZGZ03] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329. IEEE Computer Society, 2003.

[ZGZ05] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Cost and precision tradeoffs of dynamic data slicing algorithms. *ACM Transactions on Programming Languages and Systems*, 27:631–661, July 2005.

[ZSSY08] Xiaoming Zhou, Xingming Sun, Guang Sun, and Ying Yang. A combined static and dynamic software birthmark based on component dependence graph. In *Proceedings of the International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, IIHMSP '08, pages 1416 –1421, August 2008.

[ZYH+07] Boyun Zhang, Jianping Yin, Jingbo Hao, Dingxing Zhang, and Shulin Wang. Malicious codes detection based on ensemble learning. In *Autonomic and Trusted Computing*, volume 4610 of *Lecture Notes in Computer Science*, pages 468–477. Springer Berlin / Heidelberg, 2007.