AUTOMATIC DEOBFUSCATION AND REVERSE ENGINEERING OF OBFUSCATED CODE

by

Babak Yadegari

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2016

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Babak Yadegari, titled Automatic Deobfuscation and Reverse Engineering of Obfuscated Code and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Saumya Debray — Date: 14 April 2016

Christian Collberg — Date: 14 April 2016

John Hartman — Date: 14 April 2016

David K. Lowenthal — Date: 14 April 2016

Date: 14 April 2016

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director: Saumya Debray — Date: 14 April 2016

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of the source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Babak Yadegari

DEDICATION

Dedicated to my parents and my beloved wife Mina, who supported and encouraged me to accomplish this work.

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

List of Algorithms

ABSTRACT

Automatic malware analysis is an essential part of today's computer security practices. Nearly one million malware samples were delivered to the analysts on a daily basis on year 2014 alone while the number of samples submitted for analysis increases almost exponentially each year [131]. Given the size of the threat we are facing today and the amount of malicious codes emerging every day, the ability to automatically analyze unknown and unwanted software is critically important more than ever. On the other hand, malware writers adapt their malicious codes to new security measurements to protect them from being exposed and detected. This is usually achieved by employing *obfuscation* techniques that complicate the reverse engineering and analysis of the code by adding lots of unnecessary and irrelevant computations. Most of the malicious samples found in the wild are obfuscated and equipped with complicated anti-analysis defenses intended to hide the malicious intent of the malware by defeating the analysis and/or increasing the analysis time.

Deobfuscation (reversing the obfuscation) requires automatic techniques to extract the original logic embedded in the obfuscated code for further analysis. Presumably the deobfuscated code requires less analysis time and is easier to analyze compared to the obfuscated one. Previous approaches in this regard target specific types of obfuscations by making strong assumptions about the underlying protection scheme leaving opportunities for the adversaries to attack. This work addresses this limitation by proposing new program analysis techniques that are effective against code obfuscations while being generic by minimizing the assumptions about the underlying code. We found that standard program analysis techniques, including well-known data and control flow analyses and/or symbolic execution, suffer from imprecision due to the obfuscation and show how to mitigate this loss of precision. Using more precise program analysis techniques, we propose a deobfuscation technique that is successful in reversing the complex obfuscation techniques such as virtualization-obfuscation and/or Return-Oriented Programming (ROP).

CHAPTER 1

Introduction

Automatic malware analysis is an essential part of today's computer security practices. Over one million malware samples are delivered to the analysts on a daily basis, nearly 317 million samples in year 2014 alone [131]. Data breaches and infections are so common nowadays that businesses no longer think about if an attack will take place but prepare themselves for when there is going to be an actual attack. With the size of the threat we are facing today and the number of new malicious attacks emerging every day, the ability to automatically analyze unknown software is needed more than ever. On the other hand, malware writers are aware of state-of-the-art emerging security systems proposed and designed by security experts and adapt their malicious codes to protect them from being exposed and detected by these automated malware analysis systems. Therefore most of the malicious samples that are found in the wild are obfuscated and equipped with complicated anti-analysis defenses intended to hide the malicious intent of the malware to help it live longer and infect more victims.

Code obfuscation techniques are used to confuse the analyst and complicate the reverse engineering of the code. This is usually done by adding lots of unnecessary computations, such as run-time code packing/unpacking and virtualization-obfuscation, without modifying the semantics of the original code. Multiple layers of obfuscation can be stacked on top of each other, targeting both static and dynamic analysis techniques, to further complicate the analysis and reverse engineering process. This raises an important question: *is it possible to automatically discover the program logic embedded and hidden under layers of obfuscation?* An ultimate solution to the above question is to implement techniques that can automatically distinguish obfuscated code from the original and extract the original code which requires less analysis time and resources. Researchers have tried to address this problem by different approaches

but their approach targets specific types of obfuscation techniques and so make strong assumptions about the underlying code limiting the generality of their approach [117]. Obfuscation-specific approaches have the significant limitation that they can only be effective against previously-seen obfuscations; they are, unfortunately, of limited utility when confronted by new kinds of obfuscations or new combinations of obfuscations that violate their assumptions. In particular, this dissertation tries to address this question by proposing new program analysis techniques that are:

- Effective against code obfuscation techniques
- Generic in such a way that do not make strong assumptions about the underlying obfuscation technique so can be used against unseen obfuscations
- Built upon (and improve) previously known standard program analysis techniques.

1.1 Overview of the Dissertation

Any automatic approach to deobfuscation requires low-level understanding and analysis of a program's behaviors. This, in general, involves understanding fundamental program behavior and low-level characteristics of the code such as control and data flows. Obfuscation techniques usually target these low-level characteristics to obscure and/or complicate the data/control flow of the code by transforming normal compiler-generated code to more complex computations. For instance, in emulation-based obfuscation, the computation being obfuscated is implemented using an emulator for a custom-generated virtual machine together with a byte-code-like representation of the program's logic [101, 129, 138, 102]. Examination of the obfuscated code reveals only the emulator's logic, not that of the emulated code.

Figure 1.1 gives an overview of the deobfuscation process and the different components required to capture the logic and reason about different characteristics of a particular program. We use a dynamic analysis approach to get around obfuscations intended to thwart static analysis such as runtime code unpacking. Hence we record

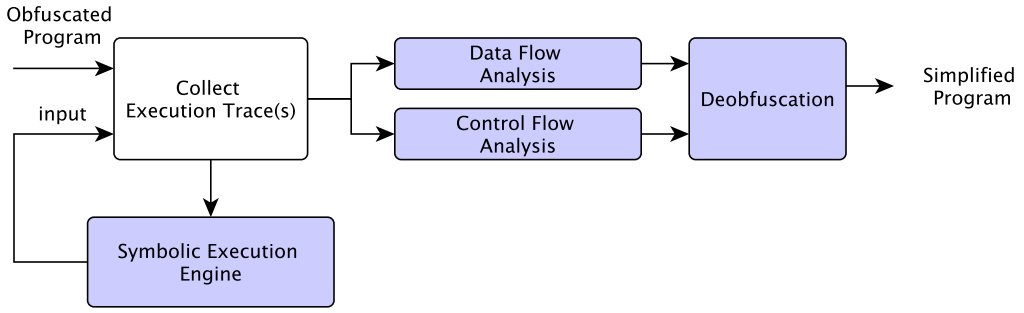


Figure 1.1: Dissertation overview

an execution trace of the program being analyzed. This dissertation makes the following contributions:

- It describes important shortcomings in current data flow analysis caused by obfuscations and proposes a more precise algorithm that is more resilient against various obfuscations (Chapter 2)
- It discusses a twist in standard control dependence analysis of interpretive systems as well as obfuscations with interpretive-like behaviors and proposes a solution to mitigate this limitation (Chapter 3)
- It proposes an algorithm for deobfuscation of obfuscated code that uses the proposed data and control flow analyses to capture and simplify the logic of the code that computes the input-to-output computation (Chapter 4)
- It describes several challenges facing symbolic execution when applied to obfuscated code and discusses solutions to increase the efficacy and performance of the algorithm against obfuscations (Chapters 5 and 6)

1.1.1 Automated Generic Deobfuscation

This work on generic deobfuscation [154] is motivated by our need to understand and analyze obfuscated code in a generic way such that it can be effective even when encountered with new and unseen obfuscations. The underlying intuition is that the

semantics of a program can be understood as a mapping, or transformation, from input values to output values that are observable through the program's interactions with the operating system. Deobfuscation thus becomes a problem of identifying and simplifying the code that effects this input-to-output transformation while keeping the semantics unchanged. We use taint propagation, including both explicit and implicit (through control dependencies) data flows, to track the flow of values from the program's inputs to its outputs. Next we use semantics-preserving code transformations to simplify the logic of the instructions that operate on and transform values through this flow.

We try to minimize the assumptions we make about the underlying obfuscations, whether that be emulation, or ROP, or anything else. A list of general assumptions that we make in out approach is described in Section 4.2.2 on page 75. Since runtime code unpacking and/or encryption/decryption is common among obfuscated codes especially malware, this study focuses on developing techniques around a dynamic analysis framework. Experiments using several emulation-obfuscation tools, including Themida, Code Virtualizer, VMProtect, and EXECryptor, as well as a number of return-oriented implementations of programs, suggest that the approach is effective in reconstructing the logic of the original program. Chapter 4 describes the deobfuscation approach in detail and the rest of the work discusses some of the difficulties in identifying the computation that computes the input-to-output mapping.

1.1.2 Bit-level Dynamic Taint Analysis

For the deobfuscation approach to be successful, it is crucially important that the input-to-output mapping is correctly identified: inaccuracy in computing control/data dependencies results in over/under approximation of the input-to-output computation, and degrades the accuracy of the deobfuscation process significantly. Unfortunately, standard program analysis techniques for computing data flow are designed for normal compiler-generated code and suffer from significant imprecision when applied to obfuscated code; thus they need to be improved to be useful and effective against

obfuscations. Chapter 2 discusses some of the limitations of current taint analysis techniques against obfuscated code [151]. It also discusses an improved approach to taint analysis which is effective against some kinds of code obfuscation. It addresses problems that arise in traditional taint analysis via a precise dynamic taint analysis algorithm that extends the basic taint analysis algorithm in three ways: (1) it uses a fine-grained bit-level analysis, (2) it uses precise mapping functions to model the taint effects of different operations, and (3) it distinguishes between, and keeps track of, different taint sources. The net effect of all these refinements is a more precise taint analysis that prevents *taint explosion*.

Utilizing the proposed enhanced taint analysis helps us prevent over-tainting which improves the deobfuscation results significantly (Section 4.3.3). Other experimental results show the proposed analysis is more accurate and is subject to less false positives and hence can be used in other areas where taint analysis is used such as symbolic execution (Chapter 5).

1.1.3 Control Dependence in Interpretive Systems

Control flow obfuscation is a widely used technique among malicious codes to hide the control flow of the code. These obfuscations include virtualization-obfuscation or ROP where the code is transformed to an interpreter-like computation. Furthermore, Interpretive systems—interpreters and associated software components such as just-in-time (JIT) compilers—are ubiquitous in modern computing systems. Control dependence is an important analysis that finds numerous applications in many fields including debugging, as well as deobfuscation. In deobfuscation, control dependence information is required to reason about the implicit flows, i.e. the data flows that are due to control dependencies in the computation. Existing algorithms for (dynamic) control dependence analysis do not take into account some important runtime characteristics of interpretive computations, and as a result produce results that may be imprecise and/or unsound. In an interpretive system (and similarly in a virtualized-obfuscated code) the logic of the program is influenced by multiple

components, namely interpreter and byte-code program and considering only one of them does not reveal enough information about the computation.

In this work on control dependence analysis [153] we first describe an important shortcoming of existing control dependence analyses in the context of interpretive systems. Second, we show how the problem can be addressed using a novel notion of control dependence that extends the classical notion of control dependence to interpretive systems. Finally, we propose an algorithm that computes an over-approximation to our notion of control dependence. The approximation algorithm is useful in cases where the necessary information to compute the exact control dependencies are not available or not trivial to get such as in virtualized-obfuscated code. This approximation algorithm can be used in the analysis of obfuscated code and in identifying the inputs-to-outputs computation in our deobfuscation approach discussed in Chapter 4.

Using the introduced notion of control dependency, we were able to improve client analyses such as dynamic program slicing significantly. Moreover, deobfuscation results using the approximation algorithm to control dependencies show the approach is successful in recovering the original logic embedded in the obfuscated code and suggests the applicability and accuracy of the approach as shown in Chapter 4.

1.1.4 Symbolic Execution of Obfuscated Code

Dynamic analysis has both advantages and disadvantages over static analysis. Obfuscation techniques such as runtime self-modifying and runtime code unpacking are problematic for static analysis because they are only observed at runtime making dynamic analysis more suitable in such cases. However, dynamic analysis suffers from limited code coverage because the analysis only observes one execution path of the program. One solution to overcome this limitation is to run program symbolically rather than with concrete values. The general idea behind symbolic execution is to use symbolic variables in the program to represent computations along a particular execution path using logical formulas. Constraint solving techniques can then be

applied to identify inputs that would cause the program to take alternative execution paths.

Since our deobfuscation approach is based on dynamic analysis to overcome obfuscations that are problematic for static analysis, a natural solution to improve the code coverage issue of our approach is to use symbolic execution. When applied symbolic execution to obfuscated code, we realized that several existing code obfuscation techniques (or simple variations on them) used by malware can significantly affect the precision of current symbolic analyses. Some obfuscations used in software protection tools such as EXECryptor and/or VMProtect lead to path explosion, i.e. produce too many states for the analysis to discover, or conceal branch points in various ways. This imposes a big limitation on using symbolic execution against malware. This makes it important to devise solutions in order to mitigate such loss of precision when performing symbolic analysis on obfuscated code. For example, some obfuscations, such as those used in the software protection tool EXECryptor [129], can cause large amounts of over-tainting and lead to a path explosion in the symbolic analysis; others, such as those used by the obfuscation tool VMProtect [138], transform conditional branch instructions into indirect jumps that symbolic analyses find difficult to analyze.

This situation is problematic because a significant motivation behind using symbolic/concolic execution in malware analysis is to get around code obfuscations. This makes it especially important to devise ways to mitigate such loss of precision when performing symbolic analysis of obfuscated code. Chapter 5 of this dissertation discusses our work [152] on improving symbolic execution against obfuscated code. It discusses major improvements to symbolic execution by first identifying shortcomings in existing symbolic execution algorithms caused by different obfuscation techniques. Second, it describes a general approach, based on a combination of fine-grained taint analysis and architecture-aware constraint generation, that can be used to mitigate the effects of these obfuscations.

1.1.5 Control Flow Obfuscation Using Signals

Chapter 6 discusses another issue that current symbolic execution analyses suffer from [155]. Exception-based control transfers—which are well-known and have been widely used in malicious code—can be used to realize implicit information flows in ways that are not detected by any of the existing state-of-the-art dynamic analysis tools. This is because currently symbolic execution techniques rely on information about the execution path and control flow of the program to reason about alternative execution paths. The simplest manifestation of the idea of using exceptions to obfuscate control transfers is to implement unconditional jumps using exceptions and redirect the control flow to an exception handler. However, it is straightforward to extend this idea such that the exception is raised only under specific conditions, thereby realizing exception-based conditional control transfers.

Current techniques for dynamic taint analysis and symbolic execution only take into account regular control flow transfers that are captured in a CFG of a program. This will give an attacker opportunities to exploit the exception mechanism to implement arbitrary control transfers and thereby realize implicit information flows that are not detected using existing analysis techniques; or hide conditional branches from symbolic execution. This represents a significant shortcoming of such analyses. Addressing the problem is not straightforward because identifying the potential control transfers—i.e., the locations where exceptions can be raised and those where they may be handled—is not always straightforward and requires detailed runtime information.

Chapter 6 proposes and discusses a solution to this problem taking into account the possible control transfers that arise from exceptions where the application code contains user-defined handlers. In Chapter 6 we demonstrate the problems that arise in current dynamic taint analysis and symbolic execution due to exception-based control transfers Then we discuss a generic architecture-agnostic solution for both dynamic taint analysis and symbolic execution that addresses the shortcomings in the analyses and hence improves their accuracy and robustness. Finally our experimental

results show the effectiveness of the proposed technique when confronted with programs that utilize exception-based control flow obfuscation.

This thesis is based on the following publications:

- Bit-level Taint Analysis [151]
- A Generic Approach to Automatic Deobfuscation of Obfuscated Code [154]
- Symbolic Exection of Obfuscated Code [152]
- Control Dependence Analysis of Interpretive Computations [153]
- Analysis of Exception-based Control Transfers [155]

CHAPTER 2

Bit-Level Taint Analysis

2.1 Introduction

Dynamic taint analysis tracks the flow of data through a program's execution and marks all data derived from certain sources of interest (e.g., user input). There is a wide body of literature on taint analysis (e.g., see Schwartz *et al.* [114]). The key idea is to maintain and propagate meta-data (the "taint") alongside the program's computation such that values derived from a source of interest are flagged as tainted. Most taint analyses maintain one taint bit per byte or word of program data, though some researchers have discussed finer-grained analyses [51, 144]. Taint analysis is used in a variety of applications, including application security [96, 78, 98, 103], software testing and debugging [89], malware analysis [158, 92], and deobfuscation of obfuscated code [125, 154].

The main application of taint analysis in our deobfuscation approach is in identifying the flow of data from inputs to the outputs of a program (see Section 4.2.4). As discussed in this chapter and in Chapter 4, imprecise taint analysis results in either over- or under-simplification of obfuscated code. Furthermore, a precise taint analysis is needed in order to reason about different execution paths of an obfuscated code using symbolic execution. Chapter 5 describes some obfuscation techniques that are problematic to symbolic execution. It is also shown how precise taint analysis is used to mitigate some of these problems.

The extensive use of taint-based security analyses has led to attacks aimed at defeating such analyses [27]. Broadly speaking, there are two such kinds of attacks. In the first kind of attack, the attacker wants to control the contents of some locations (e.g., a return address or function pointer) and the defender uses taint analysis to detect whether this happens. Such attacks typically rely on implicit information

flows to induce *under-tainting* or false negatives, i.e., cause taint analyses to infer that certain values are not tainted even though they are influenced by tainted data [27]. In the second kind of attack, the attacker wants to conceal the functionality of some code (e.g., a malware attack payload) by obfuscating the code, and the defender uses taint analysis to track the flow of values through the code and infer its functionality. Such attacks typically rely on *over-tainting* or false positives, i.e., cause taint to flow to so much of the program that various unrelated computations cannot be teased apart. The work in this chapter is concerned with the second kind of attack/imprecision, namely, over-tainting. Under-tainting that is usually caused by data flows through control dependencies is discussed in Chapter 3.

This study makes the following contributions: first it discusses some sources of over-tainting in traditional taint analyses; second it describes an enhanced bit-level taint analysis that addresses these problems and thereby significantly improves the precision of taint analysis. We discuss the application of our ideas in the context of a tool for automatic deobfuscation (Chapter 4) of obfuscated binaries, focusing in particular on code obfuscations that target the flow of values through a computation [38, 50].

The remainder of this chapter is organized as follows. Section 2.2 gives some background on taint analysis and briefly discusses some evasion techniques used to make the analysis imprecise. Section 2.3 gives the details of our method for dynamic taint analysis. Section 2.4 evaluates different taint analysis algorithms including our method with a few emulated test programs followed by discussing related works in the area in Section 2.5.

2.2 Background and Motivation

Taint analysis was originally designed as a way to track the flow of data through a computer system, potentially including both application program state (e.g., program variables and memory) and system state (e.g. operating system state, files). This mechanism can be used to detect runtime violations of security policies, e.g., where an

externally-supplied or user-controlled value is written to a security-sensitive memory location [96, 78, 98, 103, 158, 92]. More recently, taint analysis has been used to reason about and simplify obfuscated code [125, 154]. This work uses taint analysis for the purpose of identifying data flow throughout the program which then helps simplifying the input-to-output map of the code. The important role played by taint analysis in these applications means that it is important that the implementation is precise and resilient against attacks and evasion techniques.

2.2.1 Imprecision in Taint Analysis

There are two kinds of imprecision in taint analysis: *over-tainting* and *under-tainting*. Over-tainting happens when the code/data identified by the analysis is not actually tainted versus under-tainting which happens when the code/data that is influenced by a tainted source is not identified by the analysis as tainted. Such imprecision can be problematic, especially in systems where the result of the taint analysis is critically important. For example, if taint analysis is used to detect leakage of sensitive information, under-tainting means that the system has failed to ensure the privacy of the sensitive data; in a policy enforcement system using taint analysis, over-tainting can lead to disruptive false alarms. For our deobfuscation approach, under-tainting will cause an under-approximation of the input-to-output mapping being achieved (see Section 4.2.4) that causes the analysis to incorrectly simplify parts of the program that have not been identified as part of the input-to-output mapping, while over-tainting prevents the deobfuscation process from simplifying all the obfuscations leaving noise in the final result. There have not been enough studies on taint analysis limitations, and in particular, how to improve the precision of the current existing taint analysis techniques. Researchers (e.g. [113, 27, 156, 122]) have raised concerns about some specific shortcomings of taint analysis and how these limitations could be used to attack the analysis. Neither the mentioned studies nor any previous work have studied the effects of code obfuscation on the precision of the taint and in general data-flow analysis or how to address them.

```
1   int a, b, c
2   a = read()
3   b = ~a
4   c = (a & b)
5   if (c == 0){
6       // True
7   } else {
8       // False
9   }
```

Figure 2.1: An example of over-tainting when standard taint analysis applied: propagating taint from `read()` causes `c` to be tainted.

```
1     char a, b, w1, w2
2     char odd_bits = 01010101b
3     char even_bits = 10101010b
4     a = read()
5     b = 10
6     w1 = (a ∧ even_bits)
          ∨ (b ∧ odd_bits)
7     w2 = (a ∧ odd_bits)
          ∨ (b ∧ even_bits)
          ...
i     a = (w1 ∧ even_bits)
          ∨ (w2 ∧ odd_bits)
i+1   b = (w1 ∧ odd_bits)
          ∨ (w2 ∧ even_bits)
i+2   print(b)
```

(a) Variable splitting

```
1     edx = flags
2     eax = 0x6b30f626
3     edx = edx ∨ 0xfffff73e
4     eax = edx ∧ eax
5     edx = eax
6     eax = rotate_right(eax, 0x10)
7     dx = rotate_left(dx, 0x3)
8     edx = edx + 0x6c7caf05
9     edx = edx ⊕ 0xe0395c49
10    ax = ax ⊕ dx
11    edx = eax
          ...
i     if((eax ∧ 0xe0000000) == 1){
i+1       True branch
i+2   } else {
i+3       False branch
i+4   }
```

(b) An example code of EXECryptor

Figure 2.2: Code examples where standard taint analysis over-taints

Standard taint analyses, performed on ordinary compiler-generated code, usually yield an acceptable level of precision since the code has normal data and control flow. However, this may not hold true for all binaries. For example, code obfuscation can transform normal code/data flows of a program to make analysis difficult. The resulting obfuscated control/data flow can lead to a loss of precision in taint analysis.

2.2.2 A Motivating Example

Figure 2.2 shows a simple example where the standard taint analysis over-taints. First, three variables are defined at line 1 and variable `a` is assigned a value read from input at line 2. We start tainting by *introducing* taint to variable `a` and propagate it through the code snippet. Since `a` is tainted, variables `b` and `c` become tainted at line 3 and 4 respectively, leaving both `a` and `b` tainted. Variable `c` being tainted makes the `if` statement tainted at line 5 and to any dynamic or static analysis, the behavior of the program at line 5 is not predictable. Looking at the example more carefully, variable `b` contains the negation of `a` so the result of `AND`-ing these two variables will always be zero regardless of the value of `a`. Since the value of `c` does not depend on data from any taint source, it should not be considered as tainted. However, standard taint analysis will identify `c` as tainted because the operands of the operation defining `c` (line 4) are tainted. The problem here is that the standard algorithm does not propagate enough information to determine the taint status of `c` precisely.

Figure 2.2 shows other examples where conventional assumptions about flow of data may not hold. Figure 2.2(a) shows an example of *split variables* obfuscation [38] and illustrates the over-tainting effects that arise when the taint analysis is not sufficiently fine-grained. If we start tainting by marking variable `a` at line 4 and propagate taint at byte or at higher levels of granularity, variables $\texttt{w}_1$ and $\texttt{w}_2$ get tainted at lines 6 and 7 respectively, since `a` is tainted. Variable `b` also gets tainted at line `i+1` because $\texttt{w}_1$ and $\texttt{w}_2$ were previously tainted. However, notice that only the even bits of $\texttt{w}_1$ are tainted, since its odd bits come from `b`; similarly, only the odd bits of $\texttt{w}_2$ are tainted. Thus, when `b` is being retrieved at line `i+1`, only even

bits of $\texttt{w}_1$ and odd bits of $\texttt{w}_2$ are used, so `b` should not be tainted. Figure 2.2(b) shows a snippet of obfuscated code generated by a commercial obfuscation tool called *EXECryptor* [129]. This code snippet, rephrased from assembly language for ease of understanding, shows an initial segment of a long sequence that involves thousands of instructions; the full sequence is omitted due to space constraints. This piece of code actually shows the conditional control transfer mechanism that can be implemented in a virtual machine by examining the appropriate bit of the actual `PSW` flags of the `CPU`. Line 1 of the code retrieves the `EFLAGS` register of the cpu.[1] The rest of the code involves a long sequence of arithmetic operations on the flags and finally at line `i` the flag is being tested to whether carry out the control transfer or not. This kind of obfuscation causes the taint analysis to over-taint and leads to *taint explosion*.

2.2.3 Emulation-based Obfuscation

In emulation-based obfuscation, a program P is represented using the instruction set of virtual machine V_P and interpreted using a custom emulator I_P for V_P. A common representation choice for P is as a sequence of byte code instructions B_P for V_P, where the emulator I_P uses the familiar fetch-decode-execute loop of byte-code interpreters; however, other interpreter implementations, such as direct or indirect threading, are also possible. The instruction set for V_P can be perturbed randomly such that different instances of V_P look very different even if the program P does not change. Further, emulation can be combined with other obfuscations, such as run-time code unpacking, to further complicate analysis.

2.3 Bit-level Taint Analysis

2.3.1 Overview

In general, taint analyses consist of three major components that determine the accuracy and precision of the taint analysis: 1) *taint markings* or *taint labels* which

[1] The corresponding instruction sequence in `x86` assembly is `pushfd` followed by a `pop` *reg*; in this case *reg* is `edx`.

mark the taint sources with appropriate labels, 2) *mapping functions* that determine the propagation policy for propagating taint from source to destination and, 3) *granularity* which determines how much data is needed to keep track of taint for each bit of data, such as a bit of taint for each tainted bit, or a bit of taint for each byte and so on. This discussion focuses on how different choices for these components affect the precision of taint analysis. We discuss different factors that impact the precision of the analysis as well as some of the techniques used to defeat taint analysis based on the characteristics of these components. We then propose a combination of these factors that can effectively identify data flows in obfuscated code while keeping the performance of the analysis reasonable.

2.3.2 Taint Labels

Taint labels refer to the information needed to be kept for annotating and marking the relevant code that either affects the data or involves propagating it. Typically ***T*** is used to denote if the data/code is tainted and ***F*** for the data/code that is not tainted. Clearly this binary representation is only able to determine if something is tainted or not and so only one bit is sufficient to keep track of a tainted entity where the size of this entity can be a bit, byte, word or a double-word. While simple and efficient, this imposes a big limitation on taint analysis. By the amount of information that can be inferred from one bit, it is not possible to determine from which taint source a tainted data unit (bit, byte, word etc.) is derived. This limits the ability of the analysis to reason about the effects of various kinds of arithmetic and logic operations in the language on tainted data. For example the result of `xor`-ing two bits will be marked as tainted if either input is tainted, however if both bits are from the same taint source the result will always be zero regardless of the actual values of the tainted bits, i.e., unaffected by the tainted input.

Collberg and Thomborson introduce an obfuscation transformation called *split variables* [38] that can intermix bits from different variables. The obfuscation shuffles the bits of a number of (not necessarily related) variables such that each variable contains bits from other variables and each variable's bits are distributed among

other variables. In such complicated scenarios it is not enough to only mark a bit as tainted or untainted: we have to also keep track of each source of the taint individually at the bit-level. Moreover, using individually distinct taint labels for each bit imposes a relatively large memory consumption increase on the analysis. The experiments show that this technique gives us enough accuracy to deal with obfuscations but it is relatively simple to extend the idea to any kind of taint source.

An even more challenging code obfuscation technique that is problematic for taint analysis to deal with is opaque predicates or opaque variables [8, 94]. An opaque variable is a variable that holds some property at some specific point in the program. This property is known to the obfuscator but it is difficult to infer by the analysis. For example, constants can be represented by opaque variables to obscure the data-flow and incur over-tainting in the analysis. Similarly, an opaque predicate is a predicate in which the outcome of its evaluation is known to the obfuscator but it is generally hard for the analysis to reason about the evaluation results of the predicate making it obscure. These kinds of obfuscations can degrade the analysis precision significantly by obscuring the data-flow. One can imagine scenarios where an opaque variable seems to be participating in carrying taint but in fact they carry a constant value (e.g., see Figure 2.1). We have seen these kinds of obfuscations (e.g., see Figure 2.2) employed in commercial obfuscation tools such as EXECryptor [129] and VMProtect [138] as well as malicious codes.

Going back to the code snippet in Figure 2.1, it can be shown that using different taint labels can prevent over-tainting at line 4 in the code snippet. Taint analysis starts tainting by introducing taint to the `input` variable at line 2 by marking it with taint label $\mathtt{T}_1$. `input` then is copied to `a`, so `a` gets the taint label $\mathtt{T}_1$ as well. At line 3, `b` is equal to the negation of `a` and we need to use a different taint label to annotate `b`. To mark `b` as tainted, we use taint label $\mathtt{T}_2$ that is equal to $\overline{\mathtt{T}_1}$ in value. Taint label $\mathtt{T}_2$ is achieved by applying the negation operator to the taint label of variable `a`. The details of how taint labels are assigned to the destination operand(s) of an operation is discussed in *Mapping Functions* (see Section 2.3.3 on page 31). At line 4, `c` is computed by `and`-ing variables `a` and `b`. According to the taint labels,

taint label of `b` is the negation of taint label `a`, hence the taint label for variable `c` that is computed by `and`int taint labels of `a` and `b` is always zero, leading us to conclusion that `c` is not tainted. Having only a binary representation to track the taint does not give us enough clues about the outcome of the operations on tainted sources, so for a precise result we need to keep distinct labels for different sources.

2.3.3 Mapping Functions

Mapping functions or *propagation policies* define how the taint labels of the source operands of a statement are propagated to its destination(s). Standard taint analysis typically marks the destination operand as tainted if any of the source operands is tainted regardless of how the specific semantics of a statement affects its destination operands. This conservative approach can be imprecise and lead to over-tainting. Note that changing the granularity of the analysis does not help: even at bit-level granularity, a conservative mapping mechanism causes the destination to become tainted if any source operand is tainted, thereby giving the same result as performing the analysis at coarser levels of granularity.

Moreover, sometimes only a part of the destination gets tainted. For example, performing logical shift on a tainted operand will leave a portion of the result untainted while the simple union based mapping function will mark the whole destination tainted. It is important to have mapping functions that are somehow equivalent to the functional semantics of the corresponding statement in the language, because the way taint propagates in a statement from its sources to its destinations correlates with the functional semantics of the statement.

Returning to the example in Figure 2.1, the mapping function for assignment at line 1 simply copies the taint labels of the source, which is `input`, to the destination variable `a`, so if the `input` is marked with $\mathtt{T}_1$, $\mathtt{T}_1$ is copied to `a`'s taint label. This is actually consistent with the functional semantics of the assignment that copies the source to destination without any modifications. For the negation operation at line 3, label $\mathtt{T}_1$ can be any arbitrary unique label, but it makes sense to use $\overline{\mathtt{T}_1}$ which is the result of applying the negation operation on $\mathtt{T}_1$. This choice of label $\mathtt{T}_1$ simplifies the

taint propagation process since for most of the operations, the propagation policy is to apply the operation to the taint labels of the source operands. For instance, at line 4 we need to compute the taint label for `c` which is the result of a logical and on the `a` and `b` variables. The logical `and` operation can be applied on the source taint labels as the mapping function for this operation. Taking logical and of $\mathtt{T}_1$ and $\mathtt{T}_2$, since $\mathtt{T}_2=\overline{\mathtt{T}_1}$, leaves variable `c` untainted, which is what matches the expectations. Using semantics of statements as propagation policies along with distinct taint labels allows the taint analysis to model the meta-data (taint data) according the transformation on the actual data which results in a more precise taint analysis.

2.3.4 Granularity

Another important factor that affects the accuracy of the taint analysis is the granularity that the analysis is performed on. This could vary anywhere from word-level to bit-level depending on the application and the domain where the analysis is used. This property determines how much of the data can be represented by a bit of taint. For example with word-level granularity, one bit of data is enough to mark any word in the program to determine whether it is tainted or not. Clearly the more fine-grained the analysis is, the more accurate the result will be, i.e., doing the analysis at bit-level will give the best result. Sometimes it is necessary to perform the analysis at bit-level where the sources of taint are single bits, e.g., the `EFLAGS` register where every single bit carries important information individually when the flags are affected by an operation with tainted sources or some of the bits may not even be tainted depending on the operation causing flags to be tainted, so marking the whole `EFLAGS` register with one marking will introduce unnecessary imprecision to the analysis. The bit-level granularity is also important to track taints that are caused by `EFLAGS` and are propagated through the program; bit-level propagation is needed to accurately track each bit of `EFLAGS` tainted bit.

It turns out that traditional byte- or word-level taint analysis is too imprecise for our needs and can result in significant over-tainting. To address this problem, we turn to a finer-grained bit-level taint analysis: instead of having a taint mark for

each byte b, we associate each bit with a taint mark. The final algorithm actually might use a mixture of different sized units for different variables. For example, if a variable is only accessed via byte-sized reads or writes, it suffices to associate each byte of the variable with a (possibly distinct) taint mark. However for single-bit variables like control flags of the CPU, it is better to work at bit-level granularity and use distinct taint marks for each control bit.

All the above factors have significant impact on the overall precision of the analysis and changing one of the factors has significant effects on the other factors. For instance, changing the granularity level from byte-level to bit-level requires new mapping functions that reflect the semantics of the operations at bit-level instead of byte-level without modifying the functional semantics of the mapping functions. Moreover, taint labels that previously were able to track each byte of tainted data using one bit of taint should be changed to track each individual bit of tainted data using a tainted bit.

2.3.5 Algorithm

Algorithm 1 gives a high level overview of the proposed taint analysis algorithm. The provided algorithm conceptually extends the standard taint analysis algorithm by addressing the shortcomings discussed in this chapter that existed in previous analyses. The algorithm is given an execution trace, that includes information about the executed instructions such as opcode and operands for each instruction. The algorithm then annotates the trace by marking the tainted instructions and operands, if any.

2.3.5.1 Identifying Taint Sources

As shown in Algorithm 1, identifying taint sources or *introducing taint* is the first step of the analysis. Conceptually, the analysis can start propagating taint from any variable or value, but since our application of taint analysis is to identify the flow of data from inputs of the program to its outputs, the algorithm marks every input

Algorithm 1: Taint analysis algorithm

Input: An execution trace T
Result: Annotated trace T'

```
1 T' ← IdentifyTaintSources(T)
2 TaintedVars ← InitializeTaintedVars(T')
3 Label ← CreateLabels(T')
4 s = T(0)
5 while s ← T'.NextInstruction()! = ∅ do
6 |   T' ← Annotate(Label, s)
7 |   TaintedVars, Label ← Map(Label, s)
8 end
```

variable to the code as tainted. System calls are the standard way of communicating inputs and outputs for a program[2] so the algorithm starts identifying taint from system calls. Defining the taint sources to be the output of system calls gives the analyst flexibility to filter the inputs to a program that are interesting to her. For example if the analyst is only interested in the flow of data that a program sends or receives over the network, she can restrict the input functions to be of those communicating with the network and skip the others. Once the sources of taint were identified, we need to propagate this taint through the program.

2.3.5.2 Taint Labels

The algorithm continues by defining the labels at the beginning. By our definition, every bit of a tainted source can be a distinct taint label. There might be a situation where it is not feasible to assign a distinct taint label to every single bit of tainted sources, for instance when a program processes an infinite incoming stream of data, so there is a trade off between precision and performance of the analysis. To achieve the best result with acceptable performance, one might choose a mixed model of taint labels. For those variables of bigger size–like byte- or word-sized variables–a taint label is chosen as large as the size of the variable and for those of finer-grained sizes, taint labels could be assigned to each bit. For example, for a byte in memory,

[2]Some instructions can also be used to get inputs such as `RDTSC` in `x86` instruction set.

one taint label assigned to the whole byte would be enough but for control flag bits, one taint label assigned to each flag bit is needed. The initial set of taint labels are defined after identifying the sources and, as the algorithm proceeds, it keeps updating this set of labels by either adding new labels or removing ones that no longer exist. Moreover, some of the sources can have the same labels. For instance, if a program reads a file and writes it to another file without doing any interesting computations on the data, there is no significance in distinguishing between the taint labels of the data.

2.3.5.3 Mapping Functions

In algorithm 1, mapping functions are closely related to functional semantics of the underlying statements. The `Map` function in Algorithm 1 takes the current taint labels and then it applies the appropriate mappings from the marked sources (if any) to the affected data or destination operand(s). The behavior of this mapping is determined based on the statement `s` and its operands, whether they are constant or not. The `Map` function can be thought of as a function that emulates the effects of statement `s` on the taint labels of the operands. By emulate we mean that this is not always as simple as executing the statement `s` on taint labels of the operands and some operations need modifications to be precise (e.g, Figure 2.3 shows mapping functions for `add` and `xor` instructions).

Generally `x86` instructions can be divided into three major categories: *Data handling and memory operations*, *Arithmetic and logic operations* and *Control flow operations*. We define three major categories of mapping functions, corresponding to these three categories, as follows:

- *Data handling and memory operations*: Most of the instructions in this category are involved in copying data between sources like registers and/or memory. The behavior of the operations on this category is not complicated semantically compared to arithmetic and operations provided by the machine. Therefore it

suffices for a mapping function of a data movement operation to map the taint labels of the source operand(s) to the destination operand(s).

- *Arithmetic and logic operations*: As shown in Figure 2.2, the operations on this category are the main source of precision loss in standard taint analysis. Failing to propagate the taint precisely for an arithmetic operation will soon cause over-tainting. A mapping function for an arithmetic operation can be thought of emulating the arithmetic operation on the taint labels of the statement based on the semantic evaluation of the statement and its operands.

- *Control flow operations*: These operations are not explicitly involved in data-flow. Since this study focuses on explicit data flow, we do not discuss the effects of control transfers on the taint analysis. The analysis of implicit information flow through control transfers has been studied in detail elsewhere (e.g. [35, 74]) and the solutions proposed there can be adopted in our work.

Algorithm 2: Taint propagation mapping function

```
1  Procedure Map(Label, s)
3      src ← IdentifySources(s)
5      dst ← IdentifyDestinations(s)
7      switch s do
8          case s ∈ Data handling and memory operations
9              dst.label ← src.label
10         end
11         case s ∈ Arithmetic and logic operations
12             dst.label = ArithmeticMap(s, src, src.label)
13         end
14         otherwise
15             Continue
16         end
17     endsw
19     return
```

Algorithm 2 describes the mapping functions in pseudo code. Mapping functions are the main component in propagating the taint through the program. The behavior

of the mapping function determines how the taint labels are defined in the algorithm. Defining mapping functions based on the functional semantics of operations is only significant when we keep distinctive taint labels for distinct taint sources, otherwise applying functional semantics on the same taint labels introduces even more imprecision to the analysis. For instance, if the analysis only uses a binary representation as taint labels, then every `xor` operation would mark the results untainted if both of the source operands are tainted. So for the sake of discussion we assume that the analysis assigns distinct taint labels to different taint sources.

The algorithm starts by identifying source and destination operand(s) of the statement at lines 2 and 3. Every instruction or statement is a transformation of the source operand(s) to its destination operand(s), hence by identifying source and destination operands, we want to apply the appropriate transformation from the source to the destination. The transformation for data handling statements or memory operations is simply making a copy of the source to the destination operand, which is what line 9 of Algorithm 2 does. An example of a data handling operation is '`pop eax`' instruction in the `x86` assembly language. This instruction copies the memory location on top of the stack, pointed to by `esp` register, to the `eax` register. For this instruction, the source operand is the memory location on top of the stack and the destination is the `eax` register. To propagate taint for this instruction, taint labels of the source memory location should be copied to the taint labels of the destination register.

The transformation however is different and more complicated for arithmetic and logical operations. The transformation depends specifically on the functional semantics of the underlying operation as well as the taintedness of the operands. For many of the arithmetic operations it usually suffices to apply the same arithmetic or logical operation on taint labels of the source operand(s) and use the resulting taint labels for the destination operand(s). Those condition code flags that were modified by the operation will also get unique taint labels. Examples of these operations include logical shift and rotation where the amount of shift or rotate is determined by an immediate value. However, some operations need to be processed

more carefully. Figure 2.3 lists pseudo code of the mapping procedure for the two arithmetic operations `add` and `xor`. Each operation takes two source operands, $\texttt{src}_1$ and $\texttt{src}_2$, and stores the result in the `dst` operand.

```
foreach(bit i of srcs):
 if(src1[i] is constant):
  if(src1[i] is 1):
   dst[i].t = src2[i].t
  else:
   dst[i].t = src2[i].t
 elif(src2[i] is constant):
  if(src2[i] is 1):
   dst[i].t = src1[i].t
  else:
   dst[i].t = src1[i].t
 else:
  dst[i].t =
   scr1[i].t ⊕ src2[i].t
```

(a) `xor`

```
foreach(bit i of srcs):
 if(src1[i] is constant):
  dst[i].t = src2[i].t
  if(src1[i] is 1):
   dst[i+1].t = src2[i].t
 elif(src2[i] is constant):
  dst[i].t = src1[i].t
  if(src2[i] is 1):
   dst[i+1].t = src1[i].t
 else:
  dst[i].t =
   scr1[i].t + src2[i].t
```

(b) `add`

Figure 2.3: Mapping functions for `add` and `xor` operations, each operation stores the result in `dst` with source operands `src1` and `src2`

In the mapping functions of Figure 2.3, we assume at least one of the operands is tainted, otherwise the instruction is not involved in taint propagation[3]. Depending on taintedness of the operands, we consider different cases as follows:

1. If one operand is tainted and the other is constant, the taint labels for the result of the operation are obtained by applying the operation to the constant operand and the taint labels. For example, as shown in Figure 2.3(a), for `xor` binary operation where one of the source operands is tainted with taint label $\texttt{T}_1$ and the corresponding bit of the second operand is constant, the resulting bit will be marked with label $\texttt{T}_1$ if the constant bit is `0`. This is because `xor`-ing a bit `b` with `0` produces the same bit `b`. Similarly, if the constant bit is `1`,

[3]Taint sources, such as in the `x86` instruction `RDTSC`, may produce tainted destination operands even if they have no tainted source operands. Such instructions are handled in Step 1 of the algorithm.

then the resulting bit gets label $\overline{\texttt{T}_1}$ (the complement of $\texttt{T}_1$) because `xor`-ing a (tainted) bit `b` with `1` flips the bit `b`.

2. The second case is when both bits are tainted. Depending on the operation, the result is either a new taint label or not tainted at all. There are three possibilities depending on the first or the second operand being a constant (outer `if-else` clause in Figure 2.3(a)):

   - both operands have taint label $\texttt{T}_1$, meaning they are from the same taint source making the result of xor `0`
   - One bit is $\texttt{T}_1$ and the other is $\overline{\texttt{T}_1}$ where the xor result is 1 and hence not tainted
   - Taint labels are $\texttt{T}_1$ and $\texttt{T}_2$ where the result of xor is unknown so the corresponding bit can be marked with a new taint label.

Another interesting example of mapping functions is the add operation which stores the sum of its two operands in the destination operand. The pseudo-code for this operation is also given in Figure 2.3(b). Similar to the `xor` operation, there are two cases to consider:

1. The first where one of the source operands is tainted and the other is a constant. Intuitively, adding `0` to a bit does not have any effect on the result. Similarly adding `1`, depending on the summand bit, produces a carry if the other bit is one and not otherwise. In case of adding a tainted bit with taint label $\texttt{T}_1$ and a constant bit, since we do not know what the tainted bit is, there are different cases as follows:

   - If the constant bit is `0` then the result is the same as the tainted bit so the resulting bit gets the taint label $\texttt{T}_1$
   - If the constant bit is `1`, the addition may produce a carry so the next bit that receives the carry should also be tainted hence both bits get taint label $\texttt{T}_1$ and so on.

2. Second, where both operands are tainted, then labels of operands are added. For example the taint label of adding two tainted bits with labels $\mathtt{T}_1$ and $\mathtt{T}_2$ is $\mathtt{T}_1$ + $\mathtt{T}_2$. The advantage of adding taint labels is that if $\mathtt{T}_2=\overline{\mathtt{T}_1}$, then we know that a bit is being summed with its complement and hence the result should be `0` no matter what the tainted bit is.

For those arithmetic operations with only one operand such as `not` instruction where the source and destination operands are the same, we apply the semantics of the operation on the taint labels of the source operand and store the result in the destination operand.

2.3.5.4 Limitations

It is not always possible to infer the taint labels of the destination operand(s) by simply applying the operation on the source operand(s). For instance if the arithmetic operation is shift right/left, then the functional semantics of this operation would be to shift right/left the operand while the amount of shift is determined by the second operand. If the second operand is constant, we can shift the taint labels by the amount determined by the constant value, but if the second operand is also tainted, then the functional semantics of the operation depends on some tainted value and can not be determined. In this situation the mapping function will simply apply the union function on the labels of the source operands and the destination gets the new taint labels. This applies to more complicated arithmetic operations such as multiply and/or division where identifying taint labels of the destination operand(s) requires more complicated logic. An adversary can utilize this and cause over-tainting for the taint analysis by using more complicated instructions.

2.3.6 Implementation

The focus of this research study is not to produce a framework to carry out dynamic taint analysis in general, but we rather tried to provide enough details for a precise algorithm that can be implemented by researchers based on their specific needs and

their domain of study. For instance in security, researchers probably need to be able to perform taint analysis in an online manner where they can detect and respond to any suspicious activity, while for a binary analyst, an offline system that analyzes an execution trace is more desirable.

We have implemented a prototype tool to evaluate our proposed ideas for `x86` assembly language. Our implementation is an offline system: we collect an execution trace of the program we want to analyze and then perform further analysis on this trace. As shown in Algorithm 1, we start by identifying taint sources. This involves finding registers and memory sets that are passed to the program by system calls of interest. We used the `Udis86` disassembly library [132] for disassembly of `x86` instructions, but all the semantic evaluations of the instructions were done by our tool.

Our implementation uses bit-level granularity to propagate taint for code/data, i.e. tracks each bit of tainted data individually. Our implementation uses two sets of mapping functions and a mixed set of taint labels. For the computations involving the control flags register, we use taint labels that distinguish between taint sources at bit-level, i.e., distinct taint labels for each control flag bit. In order to propagate these symbolic taint marks we assigned a unique integer to each flag bit and so for every variable we keep a vector of integers to store the taint marks, one integer for each bit. Likewise a mapping function to handle flags data applies the semantics of the underlying operation to the integer vector of the variable. For any other computation in the program our implementation uses the standard notion of $\boldsymbol{T}$ and $\boldsymbol{F}$: $\boldsymbol{T}$ to denote tainted data and $\boldsymbol{F}$ otherwise. One bit of taint for every bit in the system is needed to do this so we used a bit-vector to propagate the taint for registers and memory locations.

This approach allows us to track every single control bit individually. This is particularly effective against emulation-based obfuscation where the virtual machine that interprets the protected program usually uses its own implementation of conditional control transfers. In `x86` assembly, conditional transfers are usually done immediately after an instruction that affects the `PSW flags` register like `cmp` or `test`.

In a virtual machine however, the machine first saves the CPU flags register, picks the single bit that is going to be used for control transfer, e.g. `zero flag` or `overflow flag` and does the control transfer based on the actual value of that particular flag. Sometimes the bit is even used to index a jump table. In either case we need to be able to track every bit to determine whether a control transfer depends on some tainted input or not. This can be done using two sets of mapping functions: one to propagate taint among registers and memory addresses that use binary labels with bit-level granularity, and the other to propagate taint labels used for control flags with different labels and again at bit-level granularity. As we will see in section 2.4, this approach gives good performance while achieving precise results in the analysis of heavily obfuscated code.

2.4 Evaluation

One challenge in evaluating a taint analysis algorithm is to show the precision of the analysis, i.e., the algorithm does not over- or under-taint. This requires a ground truth for what the algorithm computes that is not available for taint analysis. One way to address this issue empirically is to execute the code exhaustively on an emulator, with different inputs, and monitor the execution. If a statement is marked tainted by the algorithm, then it should be input dependent and therefore produce at least two different behaviors for distinct inputs. However this approach has its own limitations. For instance we do not know how many different inputs should be examined for a statement to be affected in a program, and in general it may not be possible to execute a program on all possible inputs.

For ordinary compiler generated code (i.e., without obfuscation), our algorithm gives the same result as the standard dynamic taint analysis. Using the operational semantics of the language, which turns to different mapping functions, and distinct taint labels, we are able to *sanitize* the taint and avoid unnecessary taint spread and so prevent any taint explosion.

In order to show the effectiveness of our analysis against obfuscations, we will present the results of two experiments. In the first experiment we measure the amount of code marked tainted using different taint approaches and the results are compared. In the second experiment we show how precise each taint is when it is used with our deobfuscation approach to recover the original logic of the code. Detailed analysis of the second experiment is presented in Section 4.3.3 of Chapter 4 where the deobfuscation approach is described.

We used six programs for our evaluation: four synthetic benchmarks, *binary-search*, *bubble-sort*, *huffman*, and *matrix-multiply*; and two malicious programs: *hunatcha*, a filedropper[4] whose C source code was obtained from the VX Heaven web site [139], and *stuxnet*, the encryption routine taken from the decompiled code for the Stuxnet worm [84]. Each program was obfuscated using four different commercial software protection tools: *Code Virtualizer* [101] , *EXECryptor* [129], *Themida* [102] and *VMProtect* [138] (the first experiment uses obfuscated programs using only EXECryptor and VMProtect). These programs read some data from input and do simple (*bin-search*) to moderately complex (*stuxnet*) computations on the input data. While the original (unobfuscated) programs are relatively simple, the obfuscated versions are significantly larger and have much more complex data and control flow characteristics (e.g., see Figure 4.5 on page 98). We collected execution traces of the original and obfuscated binaries using a modified version of Ether [49]. The execution trace records each executed instruction and register values.

For our experiments we used three different taint analysis approaches: *Byte-level*, *Bit-level* and *Enhanced*. The first two are standard taint analyses at byte-level and bit-level granularity respectively; the third, *Enhanced taint analysis*, is the approach discussed in Section 2.3. The propagation policy for both standard and bit-level analysis is the union function where the destination operand is marked tainted if any of the source operands are tainted. Since we are interested in the flow of input values in each program through the trace, the initial taint labels are the same for

[4]A filedropper is a malware that is designed to install an unwanted program (possibly a malicious code) on the victim's machine.

	Program	% Tainted Instructions		
		standard	*Bit-level*	*Enhanced*
EC	*binary-search*	30	23	19
	bubble-sort	30	26	18
	huffman	32	24	21
	hunatcha	30	30	27
	matrix-multiply	30	26	19
	stuxnet	31	28	17
VM	*binary-search*	48	48	7
	bubble-sort	69	67	7
	huffman	58	57	25
	hunatcha	45	45	17
	matrix-multiply	60	57	6
	stuxnet	64	61	7

Table 2.1: Percent of instructions marked tainted with different taint analysis approaches on different programs

all three algorithms, namely, the outputs of all system calls made by the program. Experiments showing the accuracy of the analysis is presented in Section 4.3.3 where different taint analysis approaches are used in our deobfuscation approach. The accuracy of taint analysis determines how accurately the deobfuscation process identifies and then simplifies the input-to-output mapping.

Table 2.1 shows the percent of the tainted instructions for each program/method. The rows *EC* and *VM* stand for programs protected using EXECryptor and VMProtect respectively. It can be seen from the table that in all cases our algorithm marks a much smaller portion of the trace as tainted. As expected, running the standard taint analysis with bit-level granularity will also mark fewer instructions as tainted so it will produce a more accurate result compared to the byte- or word-level analysis. But, as the numbers show, it does not significantly improve the standard byte-level algorithm. This shows that doing dynamic taint analysis even at bit-level is not effective when the code is obfuscated. Intuitively, the percentage of the trace being tainted largely depends on the program logic but it also depends on the obfuscation techniques and tools.

	Program	*Time (seconds)*		
		standard	*Bit-level*	*Enhanced*
EC	*stuxnet*	27.71	53.81	36.87
EC	*huffman*	21.21	21.24	26.68
VM	*stuxnet*	28.19	109.33	54.96
VM	*huffman*	196.03	518.11	449.73
	Average	68.28	175.6	142.6

Table 2.2: Analyses' speed on four largest traces

The analysis was carried out on a machine with 2× quad-core 2.66 GHz Intel Xeon processors with 96 GB of RAM running Ubuntu Linux 12.04. The analysis speed for our four largest traces, *stuxnet* and *huffman* programs protected with EXECryptor and VMProtect is given in Table 2.2. Table 2.2 shows the amount of time needed to run each of the three taint analysis algorithms on selected traces. The rows *EC* and *VM* stand for EXECryptor and VMProtect. The trace size for stuxnet and huffman protected by EXECryptor was nearly 5.4 and 6.8 million instructions long and for those protected by VMProtect was about 12.5 and 32.3 million instructions long. For our largest trace the speed of the analysis translates to nearly 164k instructions/second for byte-level, 61k instructions/seconds for bit-level and 72k instructions/seconds for enhanced taint analysis.

Increasing the number of distinct taint labels affects the amount of memory needed by the algorithm but it does not significantly slow down the analysis since the speed of the algorithm mostly depends on the granularity of the analysis.

2.5 Related Work

Several researchers have discussed dynamic taint analysis techniques in recent years. Most of these works are applications of taint analysis [97, 66, 67, 74, 150]; we are not aware of much work focusing on improving the accuracy of the analysis itself. The work conceptually closest to ours is that of Clause *et al.* [35], which proposes a generic framework for dynamic taint analysis. However, this paper does not discuss the precision of the algorithm nor its effectiveness against various obfuscation techniques.

Schwartz *et al.* define dynamic taint analysis based on the operational semantics of the language [114]. However, they do not consider notions of distinct taint labels and different mapping functions. Cavallarro *et al.* [27] and Sarwar *et al.* [113] discuss approaches for defeating taint analyses. Blazy *et al.* [17] describe a protection scheme to confuse the attacker by inserting small pieces of code in obfuscated code intended to render the taint results false while they appear to be correct. The wrong dynamic taint results waste the attacker time.

Drewry et al. describe a bit-precise taint analysis system named *flayer* [51]. *Taintgrind* [144] is another taint analysis tool which is based on *flayer*. Despite carrying out the analysis at bit-level, these tools use the same notion of taint propagation techniques as standard taint analysis in terms of taint labels and propagation policies, so neither is precise enough to deal with obfuscations or abnormal data-flow.[5]

[5]We were not able to compile the code for *flayer*, but have verified our hypothesis with *taintgrind* which is based on *flayer* and is as precise as *flayer* in terms of granularity.

CHAPTER 3

Control Dependence Analysis of Interpreted Computations

3.1 Introduction

Interpretive systems—interpreters together with input interpreted programs, often accompanied by dynamic code transformation via just-in-time (JIT) compilers—are ubiquitous in modern computer systems. They are used for a wide variety of purposes, ranging from general-purpose software development to web and multimedia applications and even in operating system kernels. They are also common in software security contexts: certain code obfuscation techniques rely on randomized custom interpreters to make code that is difficult to reverse engineer [117, 154]; and return-oriented programs (ROPs) [112], widely used in malicious attack code, can be understood in terms of direct-threaded interpreters dispersed among the code bytes of a program.

The ubiquity of interpretive systems, and the complexity of their runtime behavior, makes it important to devise analysis algorithms that can be used to reason about and understand such systems. An especially important analysis in this context is that of *control dependence*, which specifies whether or not one statement in a program controls the execution of some other statement. Control dependence analysis plays a fundamental role in a number of important program analyses and applications, including code optimization [55], program parallelization [82, 90], program slicing [145, 79, 134], program understanding [106, 58], partial evaluation [6], information flow analysis [47, 48, 89], and data lineage analysis [36, 160].

Moreover, the deobfuscation approach discussed in Chapter 4 uses control dependence information to identify and reason about the input-to-output mapping of the computation. On the other hand, control flow obfuscation is a widely used technique in malicious code trying to hide the control flow of the protected code. These ob-

fuscation techniques, such as control flow flattening [39], emulation-obfuscation and Return-Oriented Programming (ROP), transform the original code to an interpreter-like computation where identifying the original control flow is hard. We need to compute control dependencies in order to accurately identify the original logic that is now obfuscated.

This work makes the following technical contributions. First, it identifies an important shortcoming of existing control dependence analyses in the context of modern interpretive systems. Second, it shows how the problem can be addressed using a novel notion of control dependence that extends the classical notion of control dependence to interpretive systems; this allows a wide class of dynamic optimizations to be handled cleanly and uniformly. Experiments using a prototype implementation of our ideas, built on top of a Python system equipped with an LLVM-based back-end JIT compiler[147], show that our notion of control dependence significantly improves the results for client analyses such as dynamic program slicing. To the best of our knowledge, this is the first proposal to reason about low-level dynamic control dependencies in interpretive systems in the presence of dynamic code generation and optimization. Third, it discusses a method that computes an approximation of the control dependence notion introduced in the chapter. The approximation can be used in situations where information needed to accurately compute the control dependencies is not available. This arises in cases where the purpose of the obfuscation is to obscure the normal control/data flows and dependencies such as in virtualized-obfuscated or ROP binaries. Experiments using the approximation algorithm in recovering the control flow graph and control dependencies of the original program from the obfuscated code suggests the applicability and accuracy of the approach.

The rest of the chapter is organized as follows: Section 3.2 discusses the standard notion of the control dependency and it application in program analyses. Section 3.3 describes the ideas and algorithms to compute control dependencies in interpreters followed by implementation details in Section 3.4. Experimental evaluations are presented in Section 3.5 and Section 3.6 discusses related work.

3.2 Background

3.2.1 Interpreters

An interpreter implements a virtual machine (VM) in software. Programs are expressed in the instruction set of the VM, and encoded in a manner determined by the interpreter's architecture (e.g., byte-code, direct-threaded). Each operation A in the VM's instruction set is processed within the interpreter using a fragment of code called the handler for A (written $handler(A)$). The interpreter uses a *virtual instruction pointer* (`vip`) to access successive VM instructions in the input program and a *dispatch* routine to transfer control to appropriate handler code. Figure 3.1 shows the structure of a typical interpreter.

An interpreter must accommodate all control flow behaviors possible for all of its input programs despite having a fixed control flow graph itself. This is done by having the handler for each VM instruction update vip to the appropriate value for the next VM instruction that should be executed. Control then goes from the handler to the dispatch code, which uses the updated vip to fetch the appropriate next VM instruction. When the VM instruction involves straight-line control flow in the input program, the handler simply increments vip ; when the VM instruction is a control transfer, the handler for the control-transfer VM instruction sets vip to the appropriate target. As a result, control dependencies in the input program are transformed into data dependencies through vip in the interpreter's execution. This also means that the handlers are no longer control dependent on each other, but rather are all control dependent on the dispatch code, as shown in Figure 3.1.

3.2.2 Return-Oriented Programming

Return-oriented programming (ROP) was introduced as a way to bypass Data Execution Prevention and other defenses against code injection attacks [112, 116]. It uses a multitude of "gadgets," which are small snippets of code ending in a *return* instruction, that are present in the existing code in a computer system, whether in the kernel, libraries, or running applications. Each gadget achieves a small piece

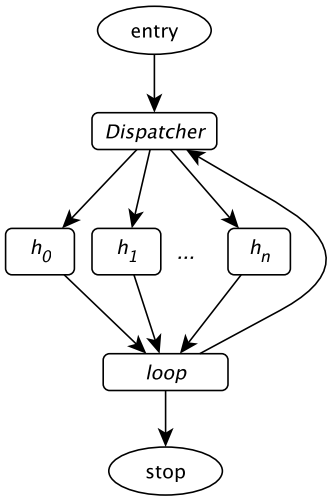


Figure 3.1: CFG of an interpreter with a dispatcher

of computational functionality. The gadgets are strung together by writing their addresses as a contiguous sequence into a buffer that is then used to effect a chain of *return* actions: each *return* then causes the invocation of the next gadget in the buffer. This basic idea has been generalized in various ways to obviate the need for explicit *return* instructions [18, 32].

There are a number of characteristics of ROPs that can make reverse engineering challenging. The first is that the code for a ROP can be scattered across many different functions and/or libraries, making it difficult to discern the logical structure of the code. If these libraries employ Address Space Layout Randomization, or are loaded into dynamically allocated memory, they may occur at different addresses. ROP sequences can take advantage of this fact by being generated just-in-time for the attack, making it difficult to examine what the ROP sequence will do without knowledge of the memory space of the target machine. Secondly, since ROP gadgets are constructed opportunistically from whatever code is already available on a system, they may contain "useless" instructions (from the gadget's perspective) that can be tolerated as long as they do not interfere with the desired functionality of the gadget. However, this opens up the possibility that the same gadget can be invoked in different ways at different times, where a given instruction within the gadget may

serve a useful purpose in some invocations and be useless in others. Finally, gadgets can overlap in memory in ways not usually encountered in ordinary programs. This is illustrated by the following two gadgets, from `ntdll.dll` in Microsoft Windows XP, with GADGET A consisting of three instructions $\{A_1, A_2, A_3\}$ and GADGET B consisting of four intructions $\{B_1, B_2, B_3, B_4\}$. These gadgets overlap in memory (the four bytes `5e 0c 0f c3`) but do not share any instructions:

GADGET A			A_1	A_2		A_3				
BYTES	08	8b	5e	0c	0f	c3	47	08	0f	c3
GADGET B	B_1						B_2	B_3		B_4

GADGET A	A_1 `pop esi` A_2 `or al, 0x0f` A_3 `ret`
GADGET B	B_1 `or [ebx + 0xc30f0c5e], cl` B_2 `inc edi` B_3 `or [edi], cl` B_4 `ret`

3.2.3 JIT Compilation

Just-in-time (JIT) compilers are widely used in conjunction with interpreters to improve performance by compiling selected portions of the interpreted program into (optimized) native code at runtime.

While the specifics vary from one system to another, the general idea is to use runtime profiling to identify frequently-executed fragments of code (which may be entire functions or portions of a function), then—once one or more such fragments are considered to be "hot enough"—to apply optimizations to improve the code. The very first such optimization is to transform the code from an interpreted representation, such as byte code, into native code. Some JIT compilers support multiple levels of runtime optimization, where the dynamically created code can be subjected to additional optimization if this is deemed profitable [123]. We refer to such dynamically-generated native code as "JITted code."

For our purposes, JIT compilation can be seen as a sequence of alternations between two phases: execution, where code from the input program—which may or may not include previously JITted code—is executed; and optimization, where code transformations are applied to hot code to improve its quality. We refer to each such optimization phase as a *round* of JIT compilation.

3.2.4 Program Slicing

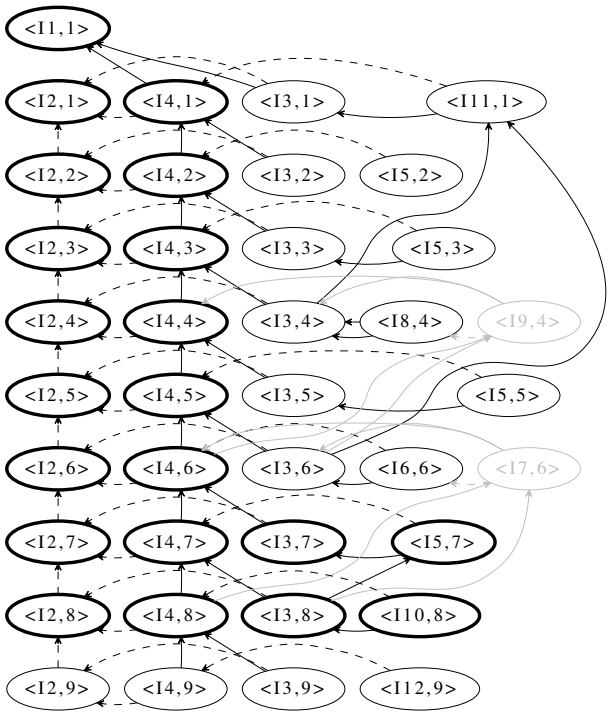
Program slicing [145] is a widely used technique in software debugging. It produces a subset of program or set of slices w.r.t a *slice criterion* which typically consists of a statement in the program and a set of variables. The computed slices are the ones that affect a value of the slicing criterion. Given that the computed slice is a subset of the program, locating errors is easier since only a subset of the program needs to be analyzed. Almost every algorithm that computes program slices w.r.t a slice criterion tend to use Program Dependence Graph (PDG) [55, 149, 134] that is a directed graph in which nodes represent statements in the program and edges represent either a data or a control dependency between nodes. Agrawal *et al* [2] introduced *Dynamic Dependence Graph (DDG)* for dynamic program slicing to distinguish between different occurrences of statements in an execution trace. Agrawal observed that different occurrences of a statement might incur different dependencies that can not be taken into account statically and this leads the computed slices to be unnecessary large. Their approach is to use separate nodes for different occurrences of a statement to make these dynamic dependencies explicit. But dynamic slicing suffers from limited code coverage since only one execution path of the program is observed. *Relevant Slicing* [3] improves the accuracy of dynamic slicing by taking into account those statements that did not affect the criterion but could have affected it had they evaluated otherwise.

```
1: n = input()
2: y = 1
3: x = 1
4: if n != 5
5:   y = 6
6: if n == 5
7:   x = 5
8: print x
9: halt
```

(a) An input program

```
I1: vip = 0;
I2: while(true)
I3:   op0, op1, dest =
          GetOperands(vip)
I4:   switch(InputPgm[vip++])
        case ASSIGNMENT:
I5:       *op1 = op0
          break
        case IF_EQ:
        /* == should be != */
I6:       if (op0 == op1)
I7:         vip = dest
          break
        case IF_NOT_EQ:
I8:       if (op0 == op1)
I9:         vip = dest
          break
        case PRINT:
I10:       print(op0)
           break
        case INPUT:
I11:       *op0 = input()
           break
        case HALT:
I12:       exit();
```

(b) A simple interpreter program



(c) DDG of the the interpreter in (b) interpreting input program (a)

Figure 3.2: A motivating example: dynamic program slicing of an interpreter

3.2.5 A Motivating Example

Consider the backward program slicing problem [79, 2] posed for a simple interpreter, shown in Figure 3.2(b), executing the input program shown in Figure 3.2(a). Figure 3.2(c) shows the dynamic dependence graph (DDG) of the interpreter in Figure 3.2(b) with the input program in Figure 3.2(a) executed on an input other than 5. The nodes in the DDG graph represent execution instances of the statements in the interpreter program of Figure 3.2(b). Each node in the DDG graph contains a pair `<first,second>` where `first` corresponds to the statement in the interpreter code and `second` corresponds to the line number in the input program of Figure 3.2(a) that caused the `first` to be executed in the interpreter. Dashed edges in the graph show the control dependencies between statements in the code and solid edges show the data dependency. Nodes represent statements in the interpreter code and bold nodes in the graph represent the program statements that were included in the computed program slice. Furthermore, gray nodes in the graph represent those that did not execute in this particular execution. We want to compute a slice with the criterion `(8, x)` of the input program in Figure 3.2.

There is a bug in the interpreter at $I7$ where the predicate for the conditional operator is wrong. This results in line 7 of the input program to be executed on inputs other than 5. We use both dynamic program slicing and relevant slicing algorithms and show both algorithms suffer from imprecision when traditional notion of control dependence is used–the former does not include the buggy statement in the slice and the latter includes the bug but also includes irrelevant statements in the final slice.

Applying the slicing algorithm on the input program will correctly include lines 7, 6 and 1 in the slice because indeed the value of `x` at line 8 depends on all the statements in those lines. Now if we want to do the slicing on machine level instructions executed by the interpreter, using dynamic program slicing algorithm [2] with the slicing criterion `(I10, op0)`, the slicing algorithm starts from the node `<I10,8>` of the DDG which is the interpreter's code that implements the `print`

call in the input program. Nodes that are included in the program slice are made bold. According to the input program, node <I6,6> should be included because of a control dependency between lines 6 and 7 of the input program. However, this control dependence edge is missing in the DDG of the interpreter because I5 is not control dependent on I6 in the interpreter and so <I6,6> is not included in the computed slice.

Relevant slicing [3] can be used along with dynamic slicing to improve the slicing results. Relevant slicing also includes the predicates that did not affect the output but could have affected if they were evaluated differently. In our example, relevant slicing considers the gray nodes in the DDG that were not considered by dynamic program slicing and also the data dependencies shown by gray arrows. Using relevant slicing, node <I6,6> will be included in the slice because <I7,6> could have affected the output. However, for the same reason, node <I8,4> will also be included in the slice (the predicate at line 4 of the input program) because of node <I9,4>. So although relevant slicing helps including the missing statements in the slice computed by dynamic program slicing, it also includes irrelevant statements because of the data dependencies carried over the whole program by vip and this increases the size of the slice significantly for programs with larger sizes.

3.2.6 Terminology and Notation

Interpreters on modern computer systems very often work in close concert with just-in-time (JIT) compilers to execute input programs.[1] To emphasize this, we refer to the combination of an interpreter and an associated JIT compiler as an *interpretive system*: an interpretive system with interpreter $\mathbb{I}$ and JIT compiler $\mathbb{J}$ is denoted by $\mathcal{I}(\mathbb{I}, \mathbb{J})$; where the interpreter and JIT compiler are clear from the context, or do not need to be referred to explicitly, we will sometimes write the interpretive

[1] There may be additional software components in the runtime system, e.g., a profiler to identify hot code that should be JIT-compiled, a garbage collector, etc. For the purposes of this study we focus on the interpreter and the JIT compiler.

system as $\mathcal{I}$. The set of all possible execution traces of such an interpretive system $\mathcal{I}$ on an input program P is denoted by $\mathcal{I}(\mathcal{P})$.

We assume a sequential model of execution. An *execution trace* for a program P is the sequence of (machine-level) instructions encountered when P is executed with some given input. A *dynamic instance* of an instruction I in an execution refers to I together with the runtime values of its operands at some specific point in the execution when I is executed. An instruction I in the static code for P may correspond to many different dynamic instances in an execution trace for P (e.g., if I occurs in a loop); where necessary to avoid confusion, we use positional subscripts such as I_k, where k refers to a position in the execution trace, to refer to a particular dynamic instance of an instruction I. We use $\prec$ to denote the sequential ordering on the instructions in a trace: thus, $I \prec J$ denotes that I is executed before J.

3.2.7 Control Dependence

The idea of control dependence characterizes when one instruction controls whether or not another instruction is executed. This notion is defined formally as follows [55]:

Definition 3.2.1 Static Control Dependence: J is *statically control dependent* on I in a given *control flow graph* G (written $J \xrightarrow{\text{static(G)}} I$) if and only if:

1. there is a path P in G from I to J such that for every K on P ($K \neq I, J$), J post-dominates K; and

2. J does not post-dominate I in G.

where node a post-dominates node b in G if every path from b to exit node in G goes through a. ∎

In addition to static control dependency that reasons about the program statically, Dynamic Control Dependence [148] reasons about control dependencies between program statements over a particular execution of the program.

Definition 3.2.2 Dynamic Control Dependence: J is *dynamically control dependent* on I in a given execution trace and control flow graph G (written as $J \xrightarrow{\text{dynamic(G)}} I$) if and only if: (i) $I \prec J$; (ii) $J \xrightarrow{\text{static(G)}} I$; and (iii) for all K such that $I \prec K \prec J$ it is the case that $K \xrightarrow{\text{static(G)}} I$. ∎

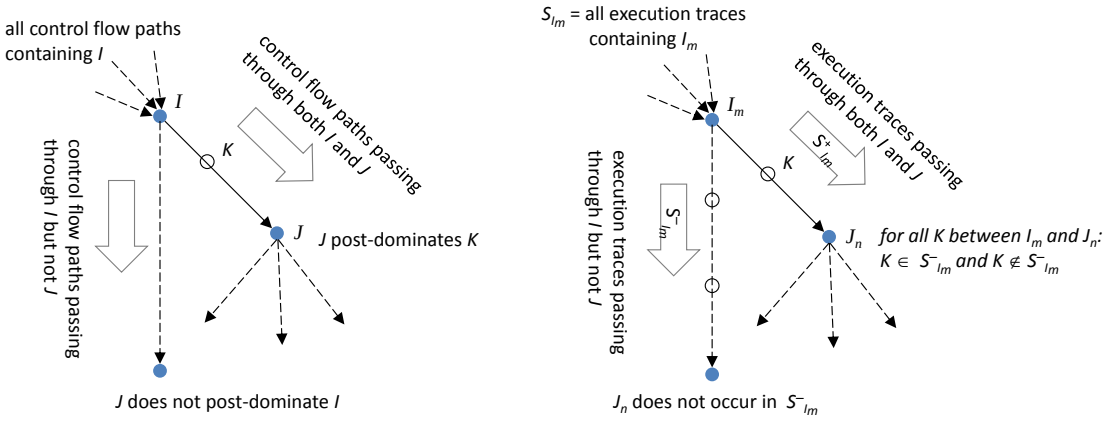
The intuition here is that if J is control dependent on I, statically or dynamically, then one control flow successor of I always leads to J while the other may or may not lead to J.

These definitions rest on two implicit assumptions. The first, which we refer to as the *realizable paths assumption*, assumes that all "realizable" paths in a program—i.e., all paths subject to the constraint that procedure calls are matched up correctly with returns—are executable; or, equivalently, that either branch of any conditional can always be executed. This assumption, which Barth refers to as "precision up to symbolic execution" [11], is standard in the program analysis literature and is fundamental to sidestepping the undecidability problems arising from Rice's Theorem [68]. The second assumption is that the static control flow graph of the program contains all of the control flow logic of the computation. This assumption does not hold in interpretive systems for two reasons: first, the logic of the input program necessarily influences the control flow behavior of an interpreter's computation; and second, JIT compilation can introduce new code at runtime whose control flow behavior is not accounted for in such definitions.

3.3 Control Dependence in Interpretive Systems

3.3.1 Semantic Control Dependency

To account for these aspects of interpretive computations mentioned in Section 3.2.7, we adapt the notion of control dependence in two ways. First, instead of considering all possible executions of the interpreter on all possible input programs—which is what we get from the traditional notion of control dependence applied to the control flow graph of the interpreter—we focus on all possible executions of the interpreter for a given input program being interpreted. This makes sense in our



(a) Static control dependence (b) Semantic control dependence

Figure 3.3: Parallels between Static and Semantic Control Dependence

context because our goal is to improve the results of dynamic analyses of interpretive systems. Second, instead of tying our notion of control dependence to a fixed static control flow graph—an approach that does not work in the presence of dynamically generated code—we give a semantic definition in terms of execution traces.

Definition 3.3.1 Semantic Control Dependence: Given an interpretive system $\mathcal{I}$ and an input program P, let I_m and J_n be dynamic instances of instructions on some execution of $\mathcal{I}$(P) such that $I_m \prec J_n$. Let $S_{I_m} \subseteq \mathcal{I}(\mathcal{P})$ be the set of all execution traces in $\mathcal{I}$(P) that contain the instruction instance I_m. Then J_n is *semantically control dependent* on I_m (written $J_n \overset{\text{semantic}}{- - \rightarrow} I_m$) if and only if S_{I_m} can be partitioned into two nonempty sets $S^+_{I_m}$ and $S^-_{I_n}$ satisfying the following:

1. $J_n \in S^+_{I_m}$ and $J_n \notin S^-_{I_m}$
2. $\forall K$ such that $I_m \prec K \prec J_n$, $K \in S^+_{I_m}$ and $K \notin S^-_{I_m}$

∎

This definition parallels the traditional definition of control dependence, as shown in Figure 3.3. The notion of static control dependence (Definition 3.2.1), shown in Figure 3.3(a), uses the structure of the static control flow graph of the program

to express the idea that, in the static program code, when we consider the control flow paths that contain I, some control paths from I lead to J while others do not. Figure 3.3(b) uses the notion of execution traces—the dynamic analogue of control flow paths—to express the idea, in the dynamic program code, when we consider the set of execution traces that contain I_m, some traces from I_m lead to J_n while others do not. Here $S^+_{I_m}$ denotes the set of execution traces that lead from I_m to J_n while $S^-_{I_m}$ denotes those that do not.

This notion of semantic control dependence differs from the traditional notion of static control dependence in two crucial respects. First, it takes into account both the interpretive system $\mathcal{I}$ and the input program P. Second, it is not tied to a fixed static CFG and therefore can be used for situations, such as with JIT compilers or obfuscated code with runtime code unpacking, where the code changes at runtime.

Definition 3.3.1 is not computable as stated and so cannot be embedded directly into an algorithm for computing control dependencies. Instead, we use this definition to reason about the soundness of our approach by showing that, considered in conjunction with the realizable paths assumption, it specifies exactly the set of control dependencies that is computed by our algorithm, given in Algorithm 3.

3.3.2 Instruction Origin Functions

We use a mechanism called instruction origin functions to incorporate control dependence information from the input program into the determination of control dependencies in the (native-code) execution trace. The instruction origin functions is defined below:

Definition 3.3.2 Given a native-code instruction I in the execution of an interpreter, an *instruction origin function* Γ is defined as:

$$\Gamma(I) = \begin{cases} A & \text{if there exists a VM instruction } A \text{ such that } (i)\ I \in \mathit{handler}(A);\ \text{or } (ii)\ I \text{ is in code obtained by JIT-compilation of } A \\ \bot & \text{otherwise} \end{cases}$$

The essential intuition here is as follows. Suppose that, for each VM instruction A, we can determine the set of native instructions implementing $handler(A)$, then the instruction origin function maps I to the VM instruction A that I "originated from" if one exists; and $\perp$ (denoting "undefined") otherwise. Conceptually, the mapping Γ can be thought of in terms of labels associated with instructions in the input program that will be propagated into the interpreter code. Each part of the interpreter that emulates one particular instruction will be labeled with the original instruction label. The propagation does not need to be limited to the interpreter; if other parts are involved in the execution process, e.g. JIT compiler, they also inherit the labels from the original byte-code instruction. In the case of JIT compiler, the native code that is produced from some byte-code instruction will be labeled according to the labels in the byte-code program.

3.3.3 Interpretive Control Dependencies

Given an execution trace of an interpretive system $\mathcal{I}$ on an input program P, in order to reason about the control flow due to the computational logic of both $\mathcal{I}$ and P, we define a notion of *interpretive control dependence* ("i-control-dependence" for short) that combines control dependence information from $\mathcal{I}$ and P and the code that is produced by the JIT compiler if there is any JIT compilation involved in the execution:

Definition 3.3.3 **i-control-Dependence:** Given an execution trace of an interpretive system $\mathcal{I}$ and an input program P, let $\mathbf{J}_i$ be the JITted code added at round i of JIT compilation. We define $J_n \xrightarrow{\text{interpretive}} I_m$ if either of the following hold:

1. both I_m and J_n are in JITted code, i.e., $\exists i : I_m, J_n \in \mathbf{J}_i$, and $J_n \xrightarrow{\text{dynamic}(\mathbf{J}_i)} I_m$; or

2. at least one of I_m and J_n is not in JITted code, and $J_n \xrightarrow{\text{dynamic}(\mathcal{I})} I_m$ or $\Gamma(J_n) \xrightarrow{\text{static(P)}} \Gamma(I_m)$.

Te notion of i-control-dependency uses the idea of dynamic control dependence to capture the control dependencies between interpreter instructions and JITted code. However, previously defined notions of control dependencies cannot be applied directly to capture the logic of the input program. To capture the control dependencies that are enforced by the input program on both the interpreter and the JITted instructions, i-control-dependency uses the origin function from Section 3.3.2. The intuition is that for VM instructions A and B in the input program P where $B \overset{\text{static(P)}}{- - \rightarrow} A$, if there are instructions I_m and J_n in the execution trace such that $\Gamma(I_m) = A$ and $\Gamma(J_n) = B$, then the execution of J_n depends on the execution of I_m. Hence I_m and J_n are *semantically* control dependent even if they are not directly control dependent according to the control dependencies of the interpreter or the JITted code.

Since the JITted code executes natively without any sort of interpretation, in the case where $I_m, J_n \in \mathbf{J}_i$, i-control-dependency is not required to lookup the control dependencies in the input program. This makes the i-control-dependency flexible in situations where the JIT compiler uses optimization transformations. Some of these transformations may not preserve the control dependence relationship among the VM instructions in the input program and may even conflict with the control dependencies in the input program (transformations such as loop unrolling or loop permutation). This is because the transformed control dependencies are reflected in the JITted code and it suffices to compute the control dependencies over the JITted code. However, if one of instructions belongs to $\mathbb{I}$ and the other belongs to $\mathbf{J}_i$, it is still required to lookup the control dependencies in the input program.

Notice the use of static control dependence definition for the input program P; the reason is that the origin function Γ maps instructions in the execution trace to the static VM instructions in the input program not their execution instances. Requiring static control dependence information for the input program eliminates the need of execution trace for the input program. Static control dependencies can be computed from the static CFG of the input program. Given an execution trace, dynamic control dependencies for the instructions in the interpreter can be

computed using the CFG of the interpreter and Definition 3.2.2. Likewise for JITted instructions, dynamic control dependencies can be computed by the CFG of the created code at runtime.

Algorithm 3: Finding control dependencies in an interpreter

Input : Execution trace $\mathcal{T}$ of the interpreter $\mathcal{I}(\mathcal{P})$; CFG of the interpreter $\mathbb{I}$; CFG of the input program; CFG of the k rounds of JITted code

Output : Annotated trace $\mathcal{T}$' with the control dependencies based on i-control-dependency

```
1  for I_m, J_n ∈ T such that J_n ≺ I_m, and ∀K : I_m ≺ K ≺ J_n; K ≠ I do
2      if I_m ∈ 𝕀 or J_n ∈ 𝕀 then
3          if J_n --static(𝕀)--> I_m and ∀K : I_m ≺ K ≺ J_n, K --static(𝕀)--> I_m then
4              Mark J_n control dependent on I_m ;
5          else if Γ(J_n) --static(P)--> Γ(I_m) then
6              Mark J_n control dependent on I_m ;
7          end
8      else I_m and J_n belong to the JITted code
9          for i ∈ [1, k] do
10             if J_n --static(J_i)--> I_m and ∀I_m ≺ K ≺ J_n, K --static(J_i)--> I_m then
11                 Mark J_n control dependent on I_m ;
12         end
13 end
```

Algorithm 3, in a high-level, shows how the i-control-dependencies can be computed over an execution trace of an interpreter and an input program. The algorithm takes as input an execution trace $\mathcal{T}$ of the interpretive system $\mathcal{I}$ executing an input program P, and the CFGs of $\mathbb{I}$ and the input program P in addition to CFG of any JITted code that was produced in the execution trace. The Γ function can be implemented differently and some of them are discussed in the Section 3.4. Different execution instances of a particular instruction is handled according to the conditions outlined in Definition 3.2.2, i.e., by assuming there is not another executed instance of I, between two particular instructions I and J.

We next give a soundness result for the notion of i-control-dependence: namely, that under the realizable-paths assumption standard in program analysis, the notion

of semantic control dependence is identical to that of i-control-dependence. Here we only sketch the outline of the proof; the details are in Appendix A.

Lemma 3.3.1 Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let I_m and J_n be instructions in an execution trace in $\mathcal{I}(\mathcal{P})$. Then, $J_n \overset{\text{semantic}}{---\rightarrow} I_m$ implies $J_n \overset{\text{interpretive}}{---\rightarrow} I_m$.

Proof. By induction on the number of rounds $k \geq 0$ of JIT compilation. □

Lemma 3.3.2 Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let I_m and J_n be dynamic instances of instructions in an execution trace in $\mathcal{I}(\mathcal{P})$. Then, $J_n \overset{\text{interpretive}}{---\rightarrow} I_m$ implies $J_n \overset{\text{semantic}}{---\rightarrow} I_m$.

Proof. From Definition 3.3.3, $J_n \overset{\text{interpretive}}{---\rightarrow} I_m$ implies that either (1) $J_n \overset{\text{dynamic}(\mathcal{I})}{---\rightarrow} I_m$; (2) $J_n \overset{\text{dynamic}(\mathbf{J})}{---\rightarrow} I_m$; or (3) $\Gamma(J_n) \overset{\text{static(P)}}{---\rightarrow} \Gamma(I_m)$. In the first two cases, the lemma follows from the definition of dynamic control dependence (Definition 3.2.2) applied to the code of the interpreter or the JITted code; In the second case, we use the definition of Γ to apply the definition of static control dependence to the input program. □

The following result is now immediate.

Theorem 3.3.1 Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption, let I_m and J_n be instructions in an execution trace in $\mathcal{I}(\mathcal{P})$. Then, $J_n \overset{\text{interpretive}}{---\rightarrow} I_m$ if and only if $J_n \overset{\text{semantic}}{---\rightarrow} I_m$.

3.3.4 An Approximation to i-control-Dependence

In Chapter 4, we will use control dependence information to capture the computation that transforms inputs of a program to its outputs and simplify the identified computation. In obfuscation techniques such as virtualized-obfuscation and/or ROP where the obfuscated code resembles the behavior of an interpreter, traditional control dependency analysis is not accurate enough to capture the entire logic. Moreover,

the CFG for the byte-code program is not readily available and thus in addition to identifying the byte-code program, which is not trivial in some cases, it is necessary to understand and reverse engineer the byte-code language for the particular runtime virtual machine that is hard to achieve as well [117]. For ROP programs since gadgets are typically identified opportunistically and instruction addresses in gadgets may not correspond to instruction addresses in the "real" program (i.e., execution of a gadget may involve jumping into the middle of the instruction bytes of an instruction in the original program), it is not clear how the CFG of the ROP interpreter should be defined. The situation is made even more complicated by the fact that the vip in a ROP program is implicit through the stack pointer. For these cases, we discuss an approximation that computes an over-estimation of i-control-dependencies which is used in our deobfuscation technique.

The only information available to the analyzer is the static CFG of the binary that is the CFG of the runtime interpreter[2], i.e., the control dependencies due to the byte-code program is missing. Since computing the control dependence information is not feasible in these situations, we briefly sketch an algorithm that over-approximates the control dependence relationship between instructions given the CFG of the interpreter which is the virtualized binary or the ROP program. The algorithm takes the following steps:

1. Compute *explicit* control dependencies that are computed directly from the CFG of the binary using post-dominance relationship

2. Compute *implicit* control dependencies that are due to the byte-code program. This step computes a (relevant) dynamic forward slice on the instructions which are found control dependent in the previous step

3. If there is any control transfer instruction in the dynamic forward slice computed in the previous step, its target basic block is marked control dependent on the original control transfer found in step 1.

[2]For the sake of discussion, let's assume that the static CFG is available, or is built using multiple execution traces of the binary.

The control dependence computed by the above algorithm is denoted by $\stackrel{approx}{\longrightarrow}$ relationship. We now prove that what is computed by the above algorithm is an over-approximation of $\stackrel{\text{semantic}}{\longrightarrow}$ control dependence relationship.

Theorem 3.3.2 For two instructions I_m and J_n in an execution trace of an interpreted binary, $J_n \stackrel{\text{semantic}}{\longrightarrow} I_m$ implies $J_n \stackrel{approx}{\longrightarrow} I_m$.

Proof. The proof is by contradiction. Suppose that $J_n \stackrel{\text{semantic}}{\longrightarrow} I_m$ holds but $J_n \stackrel{approx}{\not\longrightarrow} I_m$.

Case 1. J_n is not explicitly control dependent on I_m based on the CFG of the binary meaning that J_n is post-dominated by I_m so $J_n \stackrel{\text{semantic}}{\not\longrightarrow} I_m$.

Case 2. J_n is not implicitly control dependent on I_m meaning that execution of I_m has no effect on whether J_n will be executed or not since J_n is not in the dynamic forward slice computed from I_m. This suggests that J_n may or may not appear in all the execution traces containing I_m which satisfies the first condition of Definition 3.3.1 but does not satisfy the second condition so $J_n \stackrel{\text{semantic}}{\not\longrightarrow} I_m$.

This implies that $J_n \stackrel{\text{semantic}}{\not\longrightarrow} I_m$ holds true for both cases which is contradictory to the hypothesis and thus $J_n \stackrel{\text{semantic}}{\longrightarrow} I_m$ implies $J_n \stackrel{approx}{\longrightarrow} I_m$. □

3.3.5 Data Dependencies

As discussed in Section 3.2.1, the process of interpretation transforms control flow in the input program into updates of vip; this results in a chain of data dependencies through vip: since vip is updated after the processing of each VM instruction in the interpreter, there is a chain of data dependencies through vip that includes each VM instruction interpreted during execution. This causes the vip update code for each such VM instruction to be included in the dependence chain, resulting in a significant loss of precision. For example, when computing dynamic slices, all of the vip updates will be included in the slice.

In the context of control dependence analysis, this problem can be addressed by considering data dependencies through vip only when the interpreter is handling a jump or related control-flow-affecting instruction in the input program. We implement this idea using instruction origin functions: given a native-code instruction I_m that updates vip , we check whether $\Gamma(I_m)$ is a control-transfer instruction in the input program. This is especially important for analyses, such as slicing and information flow analysis, that use both data and control dependencies: avoiding spurious dependencies through vip can improve precision significantly.

3.4 Implementation

To evaluate our ideas we have implemented a prototype of our control dependence analysis algorithm in the context of the *Unladen-swallow* implementation of Python [147], an open-source integration of a Python interpreter with a JIT compiler. *Unladen-swallow* uses the LLVM compiler framework [83] to dynamically map frequently executed code to LLVM-IR, which is then optimized and written out as JIT-compiled native code. *Unladen-swallow* was chosen because it is built up on two popular components, Python interpreter and LLVM compiler that are widely used both by researchers and in industry. Moreover, *Unladen-swallow* provide mechanisms to control the JIT compiler behavior from the input program, e.g. forcing the JIT compilation of a piece of code, that simplifies the evaluation.

For the interpreter, we disassemble[3] the interpreter (CPython) and reconstruct its control flow graph, from which we compute (intra-procedural) control dependencies in the interpreter and the JIT compiler. To simplify the implementation effort of our prototype, our current implementation treats the LLVM code generator as an external library that is not included in this control flow graph; however, this is simply a matter of convenience and there is nothing in our approach that precludes the inclusion of the LLVM libraries if so desired.

[3]We currently use the `objdump` utility for disassembly, invoking it as '`objdump --disassemble --source`'; however, any other disassembler would work. The '`--source`' option allows us to identify control flow targets for indirect jumps corresponding to `switch` statements.

To identify control dependencies among the native code instructions, we use the CFG of the byte-code produced by the CPython compiler and mark the instructions in the execution trace. Moreover, we have instrumented the LLVM back-end JIT compiler to produce CFG of the compiled code that can be used to identify control dependencies among the JITted instructions.

3.4.1 Implementation of Origin Functions

Usually debug information is enough to compute the Γ function to map the executed instructions to the instructions in the input program. Interpreters, as well as JIT compilers, need to keep some debug information that relates executed instructions to the source program statements to make step-by-step debugging possible. In order to do this, an interpreter needs to keep information about the source program to individual byte-codes, and so for each handler code, the information about the source is accessible through these debug information. For the JIT compiler, however, since the code is first being compiled, the debug information is usually stored in a standard format, such as `DWARF`, and stored along the executable code to be accessed by a debugger. Similar to how a debugger accesses this `DWARF` data to display the source code, the mapping information for each individual instruction can be generated.

We have instrumented the CPython compiler used in *Unladen-swallow* to produce and emit control flow graph of the byte-coded input program. The instrumented compiler also inserts additional byte-codes to mark the beginning of each basic block. These special byte-codes were added to the language to assist the implementation of the origin functions. This byte-code will tell the interpreter to emit some information about the basic blocks as they are getting executed such that they can be identified in the execution trace and mapped back to the basic blocks in the input program. Moreover, to identify control flow instructions in the interpreter that implements the control transfer edges in the input program, we mark specific parts in the interpreter handlers which are implementing control flow transfers in the input program. This helps identifying the actual predicates and control transfer instructions in the execution trace when a basic block in the input program is identified control

dependent on another one. Again, this was a matter of simplicity and convenience, and is not in any way fundamental to the ideas presented here: other approaches, e.g., reverse engineering the byte code obtained from the front-end compiler, would have produced exactly the same results. Figure 3.4 illustrates the this idea of mapping instructions in the execution trace to the basic blocks of the byte-code program.

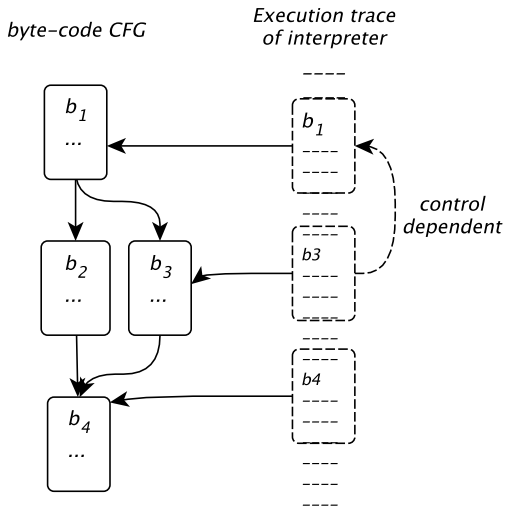


Figure 3.4: Mapping of instructions in trace to basic blocks in byte-code

For the JIT compiler on the other hand, *Unladen-swallow* uses debugging information that was originally kept by the CPython compiler in the compiled byte-code, and uses it to produce `DWARF`-like debug information for the generated native code. Using the debug information produced by LLVM compiler, it is possible to construct a mapping from executed instructions to the statements in the original input program, and then to Python byte-codes. More accurately, our instrumented runtime system creates a mapping from the address ranges of the compiled code to the line numbers of the source program using the debugging information.

3.5 Experimental Results

We have evaluated our idea of control dependency presented in this chapter for two client analyses: *dynamic program slicing* and *CFG recovery*, for the former we applied the i-control-dependency notion discussed in 3.3.3 and the latter uses the notion discussed in 3.3.4. We use a tracing tool build on top of Pin [88] to collect execution traces. The tracing tool records each instruction with some runtime information such as instruction addresses.

For program slicing, we use a prototype implementation described in Section 3.4 and ran experiments on and analyzed two bugs that were found in the Python interpreter code and reported in the Python bug tracking system at `http://bugs.python.org/`. We found two issues that have special characteristics: the wrong behavior is because of some bugs in the interpreter code that are only triggered on some particular input programs[4]. We constructed examples of the bugs reported and executed them with the Python interpreter used in *Unladen-swallow* . These two samples are representative of a class of issues that may arise due to the imprecision of the analysis discussed here and share similar characteristics. The goal of the experiment is to use dynamic slicing from the point in the program where the wrong behavior is observed and determine weather the buggy code is included in the computed slice or not using both traditional notion of control dependence and the one discussed in this chapter.

For CFG recovery analysis, the goal is to recover the CFG of the original code that is obfuscated and the control dependencies are transformed to data dependencies. We applied the i-control-dependence approximation to obfuscated samples to measure the accuracy of the analysis in recovering the original program. The detailed analysis of control flow recovery is presented in Section 4.3. The control dependence information identified by the approximation algorithm is used to identify input-dependent implicit data flows that are essential for simplification process.

[4]Issues 4296 and 3720 can be found at `http://bugs.python.org/issue4296` and `http://bugs.python.org/issue3720` respectively.

Notion of control dep.	*Execution Mode*	*Issue No. (bug id.)*	*Trace size (instrs)*	*Raw slice size*	*Slice size (%)*	*Correctness*
Traditional	Interpreter	3720	16045	2247	14.00	×
		4296	9555	589	6.16	×
	Interp+JIT	3720	350903	70330	19.90	×
		4296	571144	130763	22.79	×
i-Control	Interpreter	3720	16045	82	0.51	✓
		4296	9555	152	1.59	✓
	Interp+JIT	3720	350903	206	0.05	✓
		4296	571144	738	0.12	✓

Table 3.1: Program slicing results

3.5.1 Program Slicing

An important application of program slicing is in debugging: slicing helps identify buggy statements by producing a subset of the program starting from where the wrong behavior was observed, i.e., slicing criterion. Hence, two crucial properties of a desired program slice is that it needs to include the buggy parts of the code in the slices, and the size of slice should be minimal, specially when the size of the program being analyzed is large. For evaluation of program slicing, we compare the correctness of the dynamic program slicing using the proposed control dependence information and traditional control dependence. We also compare the size of computed slices using both control dependence approaches to measure the usability of the analysis.

We used backward dynamic program slicing (method three discussed in [2]) with 1) traditional notion of control dependencies in the interpreter code; and 2) our notion of control dependency in the interpreter and the input program; both analyzing the execution trace of the interpreter code. The criterion for both algorithms was the statement in the interpreter code or the JITted code where the wrong behavior was observed. We then checked whether the code that produced the bug was included in the slices or not manually.

Pure Interpreter: We first evaluated our work for the case where there is only the interpreter code involved. The results are presented in Table 3.1. As it can be

seen from the table, for the two issues we examined, the slicing algorithm based on control dependencies computed with traditional definitions failed to include the sources of the bugs in the program slice because of missing control dependencies in the input program that was necessary to accurately pin down the bug. In the contrary, using our notion of control dependency for interpreters, slicing algorithm was able to realize precise dependencies which resulted in the sources of the bug being included in the computed slice.

The size of the computed slices are also included in Table 3.1 for both slicing algorithms. The sizes are normalized to the size of the code observed in the execution trace. Since we are computing dynamic program slice, the size of the static code that was observed in the execution trace of the program was considered as the program size. It can be seen that the slice sizes using our notion of control dependencies are smaller because our definition prevents spurious and unnecessary data and/or control dependencies to pollute the results.

Interpreter Plus JIT compiler: As noted before, *Unladen-swallow* has a JIT compiler component that in parallel to the Python interpreter, compiles and optimizes frequently executed code into native code. It is also possible to manually enforce JIT-compilation of a piece of code in the input program. The way the JIT compiler is designed, it only optimizes away the dispatcher part of the interpreter so the compiled code is a specialized version of the interpreter with respect to the code being compiled. This means that to handle programming constructs, the JIT compiler still uses different parts of the interpreter in the compiled code resulting in including any existing bug in the interpreter in the compiled code as well. This allows us to use the same bugs we used for the pure interpreter case with the JIT compiler too. In the second experiment, we modified the input programs in such a way that the code triggering the bug was compiled to native code and executed subsequently.

The results of the experiment when the JIT compiler is activated is presented in Table 3.1. The results are similar to pure interpreter case: dynamic program slicing using traditional notion of control dependencies was not able to include the

buggy statements in the computed slice because the code is generated at runtime and is not part of the static code; while using our notion of control dependence, those statements are identified and included as part of the slice. Similar to the interpreter-only case, the size of the slices using our notion of control dependence is smaller suggesting the result is more accurate. Moreover, the results for the case where the code is JITted tends to be better compared to Interpreter only mode. That is because the JIT compiler optimizes away the interpretation and thus reduces the noise in the analysis. The larger trace sizes in presence of JIT compiler is due to the runtime compilation.

The analyses were run on a machine with 2 Intel(R) Xeon(R) 2.64GH CPUs and 64GB of RAM running Ubuntu operating system. The running time of the analysis mostly depends on the execution trace of the program. For the samples we used for our evaluation, the running times of the analysis in the Interpreter only execution mode is 4.37 and 0.58 seconds for issues 3720 and 4296 respectively; and in the Interpreter+JIT case 12.04 and 21.74 minutes. The trace sizes for the Interpreter+JIT case are significantly larger than the Interpreter only execution mode which is the reason for the increase in the analysis running time.

3.6 Related Work

There is an extensive body of research on control and data dependence analysis, including: program representations for control and data dependencies [55, 69]; frameworks for control dependence analysis [15, 127]; handling control-flow features of modern languages [110]; and efficient algorithms and representations for control dependence analysis [43, 148]. None of these works or systems consider situations where the executing code can change dynamically due to run-time JIT compiler optimizations.

The issue of imprecision of analysis resulting from "overestimation" of control dependences, which in the case of interpretive systems arises from the transformation of control dependences in the original program being transformed into data depen-

dences through `vip` in the interpreter, has conceptual similarities with problems due to over-tainting that arise when dealing with implicit flows in the context of dynamic information flow analysis. The latter problem is discussed by Bao *et al.* [10] and Kang *et al.* [74], who propose algorithms for considering control dependencies selectively, i.e., disregarding dependencies that do not satisfy certain properties of interest. This is reminiscent of the idea of selective consideration of data dependencies discussed in Section 3.3.5. High-level conceptual parallels notwithstanding, however, the details of the problems are very different from those considered here, as are the proposed solutions.

Information about control dependencies finds a large number of applications, including code optimization [55], program parallelization [5, 82, 90], program slicing [145, 79, 134, 126], program understanding [106, 58], partial evaluation [6], information flow analysis [47, 48, 89], and data lineage analysis [36, 160]. These works, however, consider contol dependence information only insofar as it can be used for the particular analysis problem of interest; none of them consider issues specific to interpretive systems.

There is a considerable body of work on analysis and optimization of interpreters and interpretive systems, but much of this focuses on individual components of interpretive systems—e.g., the input program [137, 59, 37], the interpreter [108, 105, 26, 41, 52, 130, 60], or the JIT compiler [14, 9, 61, 1]. Research on partial evaluation has considered the effect of specializing interpreters with respect to their input programs and shown that this is essentially equivalent to compiling the input program [73, 99, 133].

CHAPTER 4

A Generic Approach to Deobfuscation

4.1 Introduction

Malicious software are usually deployed in heavily obfuscated form, both to avoid detection and also to hinder reverse engineering by security analysts. Much of the research to date on automatic deobfuscation of code has focused on obfuscation-specific approaches. While important and useful, such approaches are of limited utility against obfuscations that are different from the specific ones they target, and therefore against new obfuscations not previously encountered. We aim to address this problem via a generic semantics-based approach to deobfuscation; in particular, this chapter focuses on two very different kinds of programming/obfuscation techniques that can be challenging to reverse engineer: *emulation-based obfuscation* and *return-oriented programming*.

Obfuscation-specific approaches have the significant limitation that they can only be effective against previously-seen obfuscations; they are, unfortunately, of limited utility when confronted by new kinds of obfuscations or new combinations of obfuscations that violate their assumptions. Our work on generic deobfuscation is motivated by the need for deobfuscation techniques that can be effective even when applied to previously unseen obfuscations. The underlying intuition is that the semantics of a program can be understood as a mapping, or transformation, from input values to output values. Deobfuscation thus becomes a problem of identifying and simplifying the code that effects this input-to-output transformation. We use taint propagation to track the flow of values from the program's inputs to its outputs, and semantics-preserving code transformations to simplify the logic of the instructions that operate on and transform values through this flow. We make few assumptions about the nature of the obfuscation being used, whether that be emulation, or ROP,

or anything else. Experiments using several emulation-obfuscation tools, including Themida, Code Virtualizer, VMProtect, and EXECryptor, as well as a number of return-oriented implementations of programs, suggest that the approach is helpful in reconstructing the logic of the original program.

In this chapter, we describe our deobfuscation approach that uses analyses from previous two chapters. Section 4.2 describes our approach to deobfuscation followed by experimental results in Section 4.3 and related works discussed in Section 4.4.

4.2 Our Approach

The term *deobfuscation* refers to the process of removing the effects of obfuscation from a program—i.e., given a program P_{orig} and P_{obf} be the obfuscated version of P_{orig}, analyzing and transforming the code for P_{obf} to obtain a program P_{deobf} that is functionally equivalent to P_{obf} (and hence to P_{orig}) but is simpler and easier to understand. Ideally, we want P_{deobf} to resemble the complexity of P_{orig} according to some software complexity metric. We chose program's control flow graph to measure relative complexity thus comparing two program's control flow graph determines their similarity or dissimilarity.

4.2.1 Threat Model

Our threat model assumes that the adversary knows our semantics-based approach to deobfuscation as described in this chapter, as well as some—but not necessarily all—of the transformation rules used for trace simplification. The latter assumption is justified by the fact that our approach is parameterized by the set of transformation rules used, and these rules do not form a static set but can be augmented with new rules as needed or desired.

4.2.2 Overview

Any approach to deobfuscation needs to start out by identifying something in the code (or its computation) as "semantically significant;" which is then used as the

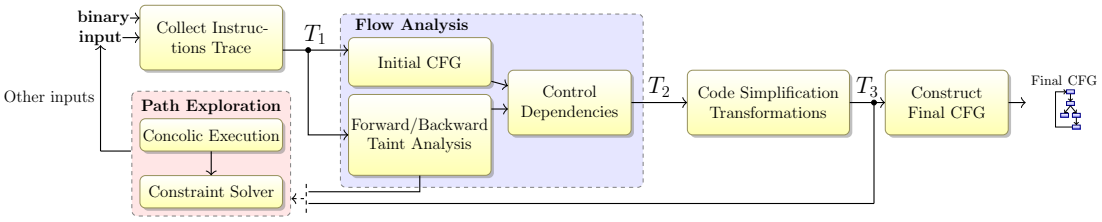


Figure 4.1: Overview of the deobfuscation. T_1 is the original trace, consisting of instructions and register values. T_2 is a trace with taint analysis and control dependence information and T_3 is a simplified trace from which a final control flow graph is constructed.

basis for subsequent analysis. For example, when disassembling obfuscated binaries, Kruegel *et al.* begin by identifying control transfer instructions [80]. Automatic unpacking tools such as Renovo [75] look for memory locations that are written to and then executed. More directly relevant to this work, Sharif *et al.* use memory access characteristics of emulation-obfuscated code to identify the emulator's virtual program counter, which they then use to reverse-engineer the emulator. The emulator then is used to identify the byte-code program to reveal the control flow graph of the original program [117]. Typically, such notions of semantic significance are based on specific aspects of the code that are either preserved by the obfuscation (e.g., control transfers in the work of Kruegel *et al.* [80]) or else are introduced by the obfuscation (e.g., write-then-execute memory locations for unpacked code [75], emulator components in the work of Sharif *et al.* [117]). In each case, the notion of what constitutes semantically significant code, and the process of identifying such code, is intimately tied to the particular obfuscation(s) being considered.

While such obfuscation-specific assumptions may simplify the process of deobfuscation, they have two drawbacks. First, such assumptions limit the future applicability of the deobfuscation technique to new and as-yet-unseen types of obfuscation. Second, they impose a weakness point to the approach which adversaries can exploit by violating the assumption. (as an example see Figure 4.11 for an instance of a case where a malware can thwart the analysis of [117]).

In order to come up with a generic approach therefore we have to minimize assumptions about the underlying obfuscation technique being used. In particular, we do not want to presuppose that any specific kind of obfuscation is being used. Since the identification of semantically significant code is typically closely tied to the obfuscations under consideration, this poses a quandary: what can be considered significant without making assumptions about what obfuscations are being used? To address this, we take an approach inspired by a notion of program semantics where programs are seen as mappings, or transformations, from inputs to outputs [128]. This definition of program semantics however, forces some implicit assumptions about the kinds of transformations that obfuscations can do. We briefly discuss these assumptions below:

- The notion of program semantics based on inputs-to-outputs computation assumes that the obfuscation does not change the inputs and outputs of the program. Changing the inputs and outputs, however, changes the program's semantics (i.e., its observable interactions with its environment). Since deobfuscation must preserve program semantics, a deobfuscation tool cannot reasonably be expected to automatically disregard some of the semantically significant operations of the program. Thus, our approach will not be able to automatically recover the logic of the original program in this case. However, our ideas can be easily extended to deal with such obfuscations interactively: the tool user can (optionally) specify some set of input and/or output operations to be disregarded, and the deobfuscation tool can simply not perform taint propagation for the disregarded operations.

- We assume that the obfuscation added to the original code is not deeply involved in the input-to-output computation, such that it can be simplified away. Entwining the obfuscation code with the input-to-output flow of values of the original program in a way that is semantically significant, but which at the same time can be guaranteed to preserve the behavior of the program being obfuscated, is a challenging problem in general. Example of such transforma-

tion is making the vip of the interpreter and/or the byte-code program input dependent in virtualization-obfuscation. These transformations can affect the observable behavior of the program, e.g., by changing use/definition relationships, introducing arithmetic overflow/underflow, or perturbing condition code settings. This means that any such entanglement of the obfuscation code will, at the very least, require sophisticated program analyses.

- To recover the original logic accurately, our approach needs a full coverage of the code. Because of this, our system uses symbolic execution in order to reason about multiple execution paths in the program. However, we realize it might not be possible to achieve full coverage mainly due to the obfuscation (e.g. see Section 5.3 on page 121) and the size of the code, so our approach computes an over-approximation of the original logic when a partial coverage is available.

Since malicious code often involves self-modifying and/or dynamically unpacked code, which is difficult to analyze statically, we use dynamic analysis: we collect one or more execution traces of the program, then analyze and simplify these traces. Figure 4.1 gives a high level overview of our approach and it consists of the steps below:

4.2.2.1 Identifying Input and Output Values

We consider the notion of "input" broadly so as to comprise values obtained from the command line and execution environment of the process (e.g., the Process Environment Block, which is sometimes used by malware to check whether it is being debugged or otherwise monitored, e.g., see [57]) as well as those obtained via explicit input operations; similarly, the notion of "output" is considered to be any externally observable side effect (e.g., creation or deletion of files or processes) as well as the results of explicit output operations and computations.

In our current prototype implementation, input and output values are determined as follows. Any value that is obtained from the command line, or which is defined

(written) by a library routine and subsequently read by an instruction in the program, is treated as an input value; any value that is defined (written) by an instruction in the program and subsequently read by a library routine is treated as an output value.[1] This captures all the inputs received to the program through system calls, such as network traffic. The program can receive some certain kinds of inputs from other sources (e.g. reading current time using `RDTSC` instruction in `x86` architecture) that are not captured by only looking at function calls. However, the number of these input sources are limited and it is easy to extend input/output identification module to capture these kinds of inputs as well.

Our dynamic analysis environment, which uses a modified version of Ether [49], collects execution traces for library routines as well as the main program, and the flow of values written within the program and subsequently read within a library routine, or vice versa, can be determined by examining the trace. We use a combination of taint propagation and control-dependence analysis to identify instructions in the execution trace that are influenced by input values and/or influence output values.

4.2.2.2 Forward taint propagation

After identifying Input sources, we should propagate the input taint through the trace to find all the instructions which are influenced by input values. In order to do this, we use a taint propagation technique which is a well-known and useful analysis tool in the fields of static and dynamic analysis. As described in Chapter 2 in detail, conventional byte-level taint analysis is not precise enough to track the data flow in obfuscated code, so we use the enhanced bit-level taint analysis discussed in Section 2.3. This initial computation captures explicit information flow from input to output, i.e. those computations that are directly influenced by input values, but does not capture implicit flows, i.e., associations between data values that arise due to control dependencies rather than data dependencies. To this end, we use

[1] This is an over-approximation, since not all library routines interact with the program's execution environment, and so may sometimes lead to a loss in precision of analysis. However, it is conservative.

dependence analysis to identify control dependencies, which we then combine with the explicit data dependencies identified earlier to capture implicit as well as explicit flow of information from inputs to outputs.

The following example illustrates the idea of implicit flow:

```
1  int factorial(int n) {
2    int i, p;
3    p = i = 1;
4    while (n > 0) {
5      p = p*i
6      i = i+1
7      n = n-1
8    }
9    return p;
10 }
```

Suppose that the goal is to identify the statements that are influenced by the input to the function, `n`. The factorial function computes the value of `p` in a loop that is controlled by variable `n` and is initialized by a constant at line 3. Considering only those statements that are explicitly influenced by `n` only captures statements at lines 4 and 7 of the function. However, the final values of variables `p` and `i` depend on how many times the loop executes and therefore on `n`. This is called implicit flow [28] which makes statements that are control dependent on a tainted variable also tainted. Hence all the statements that are inside the while loop should also be marked as tainted.

4.2.2.3 Code Simplification

Once we have identified the input-to-output value flows, i.e. those instructions that are directly or indirectly dependent on input values, either through control dependencies or data dependencies, as well as instructions that directly or indirectly influence output values, we iteratively apply semantics-preserving code transformations to simplify the execution trace. The simplification process simplifies away irrelevant code that was added by the obfuscation and "optimizes'' the execution trace for the identified input-to-output mapping. For example with virtualization-obfuscation, the

simplification simplifies away the part of the code that implements the logic of the interpreter, i.e., fetch-decode-execute loop. The resulting simplified trace represents the behavior of a program that is functionally equivalent to the original program (at least for the particular execution that was observed) but which is simpler.

4.2.2.4 Control Flow Graph Construction

The simplified trace is used to construct a control flow graph (CFG) that makes explicit some of the higher-level control flow structures such as conditionals and loops. The final step of our deobfuscation process is to apply semantics-preserving transformations to the CFG to eliminate some spurious execution paths and produce a more precise CFG. The resulting simplified CFG is then produced as the output of our deobfuscation system.

Dynamic analysis is more powerful than static analysis when dealing with self-modifying code or runtime code unpacking. However, dynamic analysis suffers from low code coverage because each execution instance of the binary only examines one execution path in the code. To achieve full coverage one needs to determine different inputs that trigger all execution paths in the code and that is a non-trivial problem [93, 20]. In order to address this problem we have implemented a symbolic execution system (Chapter 5) that is built on top of the taint analysis engine. Taint analysis identifies the computations that are influenced by the inputs and a symbolic execution engine builds the path constraint along the execution trace from the tainted instructions. The path constraints are then solved by a Satisfiability Modulo Theory (SMT) [62, 45] solver to identify other input values that cause alternative execution paths in the code. An SMT solver can decide weather a given symbolic formula is satisfiable or not. By generating appropriate formulas and sending them to an SMT solver, symbolic execution is able to generate inputs that can satisfy different execution paths to be taken in branch points in the program. The symbolic execution part is discussed in details in Chapter 5 so the details are omitted here.

Algorithm 4 gives a high-level overview of the deobfuscation process, below each step is discussed in more detail.

Algorithm 4: Deobfuscation process overview

Input: an execution trace T
Result: a simplified trace T' and control flow graph G'

```
1 Identify the input-to-output flow of values in T
2 while there are simplifiable instructions in T do
3 |   Apply simplifying transformations
4 end
5 Construct a control flow graph G' from the simplified trace T'
6 Simplify G'
7 Output simplified trace T' and simplified control flow graph G'
```

4.2.3 Identifying Input and Output Values

This step identifies the inputs and outputs to the program. In general, programs use operating system APIs to interact with the outside world and hence use system calls for any sort of input or output. Therefore, in our approach, we monitor the system calls that are made by the program and mark those variables that are overwritten by the system calls as inputs to the code. Similarly, every variable (register or memory location) that is defined by the program and is used in a system call is considered output.

4.2.4 Identifying Input-to-Output Flow

The next step of our algorithm is to identify the flow of values from input operations to output operations, and thereby the instructions that transform input values to output values. To this end, we first use forward taint propagation to identify the explicit flow of values from inputs to the program and backward taint propagation to identify the flow of data to the outputs of the program. Then we use control dependence analysis to identify implicit flows, i.e., those instructions that affect the outputs of the program and are control dependent on some input-dependent predicate.

This captures the implicit flow from inputs to outputs that are not captured by only tracking explicit data flows.

4.2.4.1 Taint Analysis

We use taint analysis (see Section 2) to identify the runtime flow of values from a program's inputs to its outputs; this information is then used for control dependency analysis. The essential idea is to associate each value computed by the program with a bit indicating whether or not it is "tainted," i.e., derived directly or indirectly from an input value. Initially, only values that are obtained directly from inputs are marked as tainted. Taint is then propagated iteratively to other values by marking any value that is computed from a tainted value as tainted.

Our approach uses two kinds of taint analysis:

1. *Forward taint analysis.* This is used to identify the flow of input values through the program. It is especially important for finding code that is control dependent on input values. We perform taint analysis for registers, memory, and condition-code flags.

2. *Backward taint analysis.* This starts from output values and works backwards by identifying variables and values that influence the program's outputs. In some ways this resembles dynamic program slicing where the slicing criterion is the program's observable output but we are only interested in the data flow so we do not consider the control dependencies when computing the backward slice. The logic of this analysis is conceptually analogous to that of forward taint analysis except that it works backwards against the flow of control.

The precision of the forward taint analysis is particularly important because the rest of the deobfuscation depends significantly on how well the taint analysis identifies the decision points in the program being examined. As discussed in more detail later, when simplifying the code it is important to identify static computations whose iteration counts are influenced by dynamic input, e.g. loops where the iteration is

determined by input values, and imprecision in taint propagation adversely affects the deobfuscation of such loops, e.g., under-tainting leads to too much of the code getting simplified away, and over-tainting leads to too little simplification.

Algorithm 5: Finding control dependencies in obfuscated code

Input: An initial input/output tainted trace T
Result: The trace T with control dependencies between instructions identified

```
1 Construct an initial control flow graph G
2 Compute post-dominator relations in G [4]
3 Use post-dominator relationships to compute explicit control
  dependencies [4]:
4 (a) C = the set of input-tainted conditional control transfers; and
5 (b) DepVars ={x | ∃C ∈ C: x control dependent on C}
6 while ∃ an indirect control transfer Ins dependent on some x ∈ DepVars
  do
7 |   BBl ← basic block of Ins in G
8 |   Mark BBl as dependent on the direct control transfer in C that x is
  |   dependent on
9 end
```

4.2.4.2 Control Dependency Analysis

Given two instructions (statements) I and J in a program, J is said to be *control-dependent* on I if the outcome of I determines whether or not J is executed. More formally, J is control dependent on I if and only if there is a non-empty path π from I to J such that J post-dominates each instruction in π except I [4]. The identification of control dependencies has been well-studied in the compiler literature [4]. However, the situation is a little different in our case. In the analysis of emulation-obfuscated code, some of the control transfers encountered are due to the logic of the program being emulated while others are simply an artifact of the emulation process and therefore not interesting from the perspective of identifying dependencies. In other words, some control transfers are because of the original program and some of them are because of the runtime interpreter. In the deobfuscation process, only the control

dependencies of the original program is important. As discussed in detail in Chapter 3, identifying only control dependencies that are due to the original code might not always be possible (or is not trivial to compute) where the logic of the byte-code is not available for analysis (which is the case for obfuscated code). Therefore we compute an over-approximation of the control dependencies discussed in Section 3.3.4.

The approach we take is shown in Algorithm 5. We consider two types of control flows: *explicit* and *implicit*. Explicit control flows are those control transfers where the predicate is explicitly reflected in the transfer of control, e.g., as in conditional jump instructions. Finding explicit control dependencies is straightforward using post-dominators [4]. Implicit control flows are those indirect control transfers of the form '`jmp [`ℓ`]`' where the location ℓ is data-dependent on the set DepVars of dependent variables identified in Algorithm 5. Implicit control dependencies are those that are transformed to data dependencies carried over with the vip (or a variable that resembles vip such as `esp` in ROP) in the obfuscated code[2]. Intuitively, implicit control dependencies account for the fact that a control dependence between two instructions I and J may arise indirectly through an assignment D of the value of a variable x if D is control dependent on I and where x determines the target of an indirect control transfer to J.

Figure 4.2 shows an example of an implicit control dependency. The value of register `eax` at line 6 depends on the evaluation of the conditional jump of line 2, so the target of the `jmp` instruction at line 6 also depends on which path is taken on line 2 by the conditional jump. This makes the basic block targeted by the `jmp` instruction at line 6 control dependent on the conditional transfer on line 2 as well. In fact, the data dependency from line 6 to lines 3 and 5, through the value of `eax`, is really a control dependency in disguise.

[2]Note that this only exists in virtualization-like obfuscation where the control flow is mandated by a virtual instruction pointer, e.g., ROP.

```
1        test  ecx, eax
2        jnz   L1
3        mov   eax, 0
4        jmp   L2
5   L1:  mov   eax, 1
6   L2:  jmp   [edx+4*eax]
```

Figure 4.2: An example of implicit control dependency

4.2.5 Trace Simplification

Once we have identified the instructions in the trace that participate in computing output values from input values, the next step is to map these instructions to an equivalent but simpler instruction sequence. To minimize the assumptions as much as possible about the obfuscations we may be dealing with, we use a set of simple and general semantics-preserving transformations.

An important concept in the simplification step is the notion of a quasi-invariant location. We define a location ℓ to be *quasi-invariant* for an execution if ℓ contains the same value ℓ_c at every use of ℓ in that execution. For constant propagation purposes, we consider a value to be a constant during an execution if either it is an immediate operand of an instruction or if it comes from a memory location that is quasi-invariant for that execution.

Quasi-invariant locations allow us to handle transient modifications to the contents of memory locations, e.g., due to unpacking, as long as we see the same value each time a location is used. Moreover, for virtualization-obfuscation, since the byte-code program (original code) is fixed and stored in the binary, quasi-invariant concept allows us to simplify the interpreter's logic and spurious control transfers. Quasi-invariants can be identified in a single forward pass over a trace keeping track of memory locations that are modified and, for each such modification, the value that is written. The notion of quasi-invariance can be extended in various ways, e.g., we may consider whether a memory word contains the same value every time it is used for an indirect branch (this is useful, for example for dealing with jump tables whose

elements are kept in encrypted or encoded form, decrypted prior to use, and then re-encrypted).

The transformations we use include the following (this is a non-exhaustive list):

1. *Arithmetic simplifications*: In essence this is a straightforward adaptation of the classic compiler optimization of constant folding to work with dynamic traces and quasi-invariant locations. However, as described below, it has to be controlled to avoid over-simplification. For example, in the code sequence shown above, the constant value `0xa4` loaded into the register `bh` can be propagated through the bit-manipulation instructions following it, and the entire sequence of instructions manipulating `bh` can be replaced by a single instruction '`mov bh, 0x8b`'.

2. *Indirect memory reference simplification*: An indirect memory reference through a quasi-invariant location ℓ that holds a value A is simplified to refer directly to A. This transformation is applied to both control transfers and data references.

3. *Data movement simplification*: We use pattern-driven rules to identify and simplify data movement. For example, one of our rules states that the following simplification can be performed provided that the sequence of instructions *Instr* does not access the stack and does not change the value of A:

   ```
   push A
   Instrs     ⟶   Instrs
   pop B              mov B, A    /* B := A */
   ```

4. *Dead code elimination*: Instructions whose destinations are dead, i.e., not used subsequently in the computation, are deleted. This transformation must consider all destinations of an instructions, including destination operands that are implicit and which may not be mentioned in the instruction. Such implicit destinations may include condition flags as well.

5. *Control transfer simplification*: Control transfer instructions whose targets are constant are replaced by direct jumps. Candidates for this transformation

include *return* instructions to constant targets in ROP code as well as indirect jumps to fixed targets in emulation-based obfuscation. In general, those control transfers that are not influenced by some input value are simplified in the final trace. Using control flags implicitly to control the transfer flow of the program is common among interpreters and is also used in ROPs. For example the following code shows how loops can be implemented using a gadget in a ROP program:

```
mov eax,0
sub counter,1
adc eax,eax       /* eax := 1 if counter=0 */
push [L+eax*4]
ret
```

where L is the address of the memory location which points to the beginning of the loop and subsequent location points to where loop should exit to. In this example, the target of the return instruction is affected by the outcome of the carry flag so the `ret` instruction can be replaced by a conditional jump which directly uses the carry flag.

Example 4.2.1 Figure 4.3 gives an example of indirect memory reference simplification. Figure 4.3(a) shows a small program that sits in a loop making indirect jumps through successive elements of a read-only array `T`. Figure 4.3(b) shows the unsimplified trace for this code. Since `T` is read-only, its elements are constant, making indirect calls through this table amenable to indirect memory reference simplification; the resulting trace is shown in Figure 4.3(c). Since `T` is no longer being used for indirect jumps, instructions that load from `T` then become dead and are removed via dead code elimination. Similarly, constant propagation converts the `add` instructions into `mov` instructions that load constants into register `ebx`. This then determines the outcome of each of the `cmp` instructions, and allows the `cmp` and `jne` instructions to be simplified away; once this happens the instructions that load into `ebx` also become dead and are removed.

(read-only)

T:	0x500000
	0x520000
	0x550000

```
     mov ebx, 0
L:   mov eax, T[ebx]
     jmp [eax]
     add ebx, 4
     cmp ebx, 12
     jne L
```

(*a*) Static code

```
mov ebx, 0
mov eax, T[ebx]
jmp [eax]
Trace of code at 0x500000
add ebx, 4
cmp ebx, 12
jne L

mov eax, T[ebx]
jmp [eax]
Trace of code at 0x520000
add ebx, 4
cmp ebx, 12
jne L

mov eax, T[ebx]
jmp [eax]
Trace of code at 0x550000
add ebx, 4
cmp ebx, 12
jne L
```

(*b*) Unsimplified trace

```
mov ebx, 0
mov eax, 0x500000
jmp 0x500000
Trace of code at 0x500000
mov ebx, 4
cmp ebx, 12
jne L

mov eax, 0x520000
jmp 0x520000
Trace of code at 0x520000
mov ebx, 8
cmp ebx, 12
jne L

mov eax, 0x550000
jmp 0x550000
Trace of code at 0x550000
mov ebx, 12
cmp ebx, 12
jne L
```

(*c*) Trace after constant propagation and indirect memory reference simplification

```
~~mov ebx, 0~~
~~mov eax, 0x500000~~
jmp 0x500000
Trace of code at 0x500000
~~mov ebx, 4~~
~~cmp ebx, 12~~
~~jne L~~

~~mov eax, 0x520000~~
jmp 0x520000
Trace of code at 0x520000
~~mov ebx, 8~~
~~cmp ebx, 12~~
~~jne L~~

~~mov eax, 0x550000~~
jmp 0x550000
Trace of code at 0x550000
~~mov ebx, 12~~
~~cmp ebx, 12~~
~~jne~~ L
```

(*d*) Trace after dead code elimination

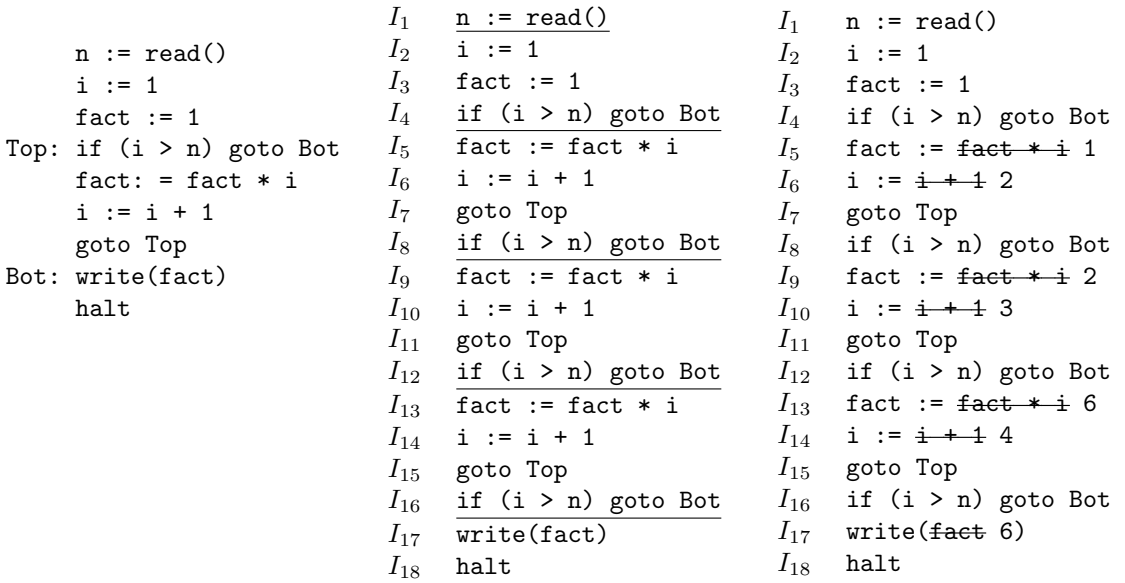
Figure 4.3: An example of indirect memory reference simplification

The final simplified trace is shown in Figure 4.3(d). What is left is pretty much just the code executed at the addresses that, in the original program, had been reached via a sequence of indirect jumps through the jump table `T`. In the simplified trace, almost everything other than the code eventually executed has been simplified away.

The indirect jump behavior illustrated in this example is very similar to the dispatch code of an emulator. Indirect memory reference simplification allows us to replace the dispatch jumps of an emulator with direct jumps that can then be candidates for further optimization. ∎

While the trace simplification process described above is crucial for removing obfuscated code, it has to be carefully controlled so that it does not remove too much of the logic of the computation. The problem is illustrated by Figure 4.4. Figure 4.4(a) shows the static code for a simple iterative factorial computation, written in a C-like notation. Figure 4.4(b) shows the execution trace for this program for an input value of 3, with input-tainted instructions shown underlined. Figure 4.4(c) shows the result of trace simplification: it can be seen that constant propagation has been applied to all of the updates to the variables `fact` and `i`, and as a result the output operation at the end has been reduced to '`write(6)`'. This is not helpful for understanding the logic of the computation, i.e., the mapping from input values to output values.

To understand the problem, consider the instruction $I_5 \equiv$ '`fact := fact * i`'. The variables `i` and `fact` have both been initialized to the value 1 at this point, so the value of the expression '`fact * i`' is inferred to be a constant. Constant propagation then simplifies this instruction to the assignment '`fact := 1`'. Arguably, this simplification does not preserve the logic of this computation because it suggests that this assignment computes a fixed constant value when, in reality, the value that is computed by this instruction depends on the number of iterations of the loop, which in turn depends on the input value. The same observation applies to the other arithmetic simplifications carried out on this trace. The problem arises because the simplification fails to take into account the fact that the instruction being simplified



(*a*) Static code

(*b*) Unsimplified trace (input = 3). Input-tainted instructions are shown underlined.

(*c*) Result of oversimplification.

Figure 4.4: An example illustrating over-simplification

is control-dependent on the input-tainted instruction $I_4 \equiv$ '`if (i > n) goto Bot`', which induces an implicit information flow from the input to I_5.

We address this problem by restricting the propagation of constants across input-tainted conditional jumps. This is done as follows. We first identify control dependencies as described in Algorithm 5. Given an instruction X, let $ControlDeps(X)$ denote the set of input-tainted instructions in the execution trace that X is control-dependent on. Then, a backward-tainted arithmetic operation I is *simplifiable* only if every source operand of I is either an immediate operand, or else is defined by an instruction J such that $ControlDeps(J) = ControlDeps(I)$. Applying this condition to the trace of Figure 4.4(b), we find that instruction I_5 is control-dependent on the input-tainted instruction $I_4 \equiv$ '`if (i > n) goto Bot`', but its operands `fact` and `i`, which are defined by instructions I_3 and I_2 respectively, which are not control dependent on any instruction and therefore in particular are not control dependent on I_4. Thus, $ControlDeps(I_5) \neq ControlDeps(I_3)$ and so I_5 is not simplifiable. The

constant value of `fact` defined by I_3 is therefore not propagated to I_5, which is what we want.

Algorithm 6: Final control flow graph construction

Input: Simplified execution trace T
Result: Control flow graph G for T

Let B_0 be the first basic block in T
$t_{curr} := v_{curr} := B_0$
$G := (V, E)$ where $V = \{v_{curr}\}$ and $E = \emptyset$
EdgeStack:= NULL
while *there are unprocessed blocks in* T **do**
    let t_{next} be the next block after t_{curr} in T
    if t_{next} *is already a successor of* v_{curr} **then**
        $v_{next} := t_{next}$
    else if *a successor can be added to* v_{curr} **then**
        /* *add* t_{next} *as a successor to* v_{curr} */
        Let v_{next} be a basic block in G that its entry point has the same address as t_{next} in T
        if v_{next} = *NULL* **then**
            $v_{next} := t_{next}$
            add v_{next} to V
        add $e \equiv$ '$v_{curr} \rightarrow v_{next}$' to E
        push e on *EdgeStack*
    else
        /* *backtrack using EdgeStack* */
        pop $e \equiv$ '$a \rightarrow b$' from *EdgeStack*
        t_{curr} := block in T corresponding to a
        t_{next} := block in T corresponding to b
        v_{curr} := block in G corresponding to t_{curr}
        v_{next} := Duplicate(t_{next})
        add $e \equiv$ '$v_{curr} \rightarrow v_{next}$' to E
        push e on *EdgeStack*
    $v_{curr} := v_{next}$
    $t_{curr} := t_{next}$
Output G

4.2.6 Control Flow Graph Construction

The final step in our deobfuscation process is to construct a CFG over the simplified trace obtained from the trace simplification step. For deobfuscation purposes, one issue that arises in this context is that of reuse of code in a way that complicates the program's control flow structure. In obfuscated code, we very often find that a given functionality I—e.g., an emulator operation such as addition or subtraction (in emulation-obfuscation), or a gadget for an operation such as copying one register to another (in return-oriented programming)—is implemented using a single code fragment C_I; control is then directed to C_I whenever the functionality I is needed in the program. This means that if there are k different occurrences of I in the original program, they will end up executing the same piece of code C_I in the emulated program k times, with k corresponding repetitions of C_I in the execution trace. A CFG constructed in a straightforward way will then have k pairs of control flow edges coming into and out of the code region C_I, which will cause the control flow behavior of the program to appear very tangled.

During deobfuscation, therefore, we try to construct the CFG in a way that attempts to untangle some of the paths by judiciously duplicating basic blocks. Intuitively, we want to minimize the amount of such code duplication, while at the same time reducing the number of "spurious" control flow paths (paths that are possible in the constructed CFG but which are not observed in the trace(s) used to construct the CFG). Solving this problem optimally seems combinatorially challenging, and similar to the problems in computational learning theory that are known to be computationally hard: the problem of identifying a CFG that is consistent with a given trace (i.e., which admits that trace but may also admit other execution paths) can be modeled as that of constructing a DFA consistent with a given set of strings (i.e., which accepts those strings but may also accept other strings). Unfortunately the problem of finding the smallest DFA (or the smallest regular expression) that is consistent with a given regular language is NP-hard [7, 65] and is not even efficiently approximable [104].

Given these results, we augment the usual CFG construction algorithm [4] with heuristics aimed at balancing the number of vertices and the complexity of the constructed CFG, using a depth-first backtracking search to explore the search space as is shown in the Algorithm 6.

The simplified trace, from which we construct the deobfuscated control flow graph, is a sequence of instructions that can also be considered as a sequence of basic blocks such that if a block B is followed by a block B' in the (simplified) trace it corresponds to an edge $B \to B'$ in the corresponding control flow graph. Our algorithm traverses the sequence of basic blocks in the trace, constructing a control flow graph G using the usual CFG construction algorithm, by adding basic blocks and/or edges to G, as long as this does not violate any structural constraints of any vertex in G. Currently, the primary structural constraint that is enforced is the *out-degree constraint*: namely, that a basic block ending with a conditional jump can have at most two successors or if it is ending with an indirect jump or a return instruction, there is no restriction on the number of its successors. This requirement is checked at the line 9 of the Algorithm 6. If the algorithm encounters a situation where adding a block and/or edge to G would violate this structural constraint, it backtracks to the most recently added vertex that can be duplicated without violating the out-degree constraint (Algorithm 6 lines 18-25). This vertex is then duplicated, and the algorithm continues constructing the CFG by adding basic blocks and edges normally.

Another problem that the simplification might cause is removing dynamically dead instructions and causes the CFG construction algorithm to duplicate a basic block for the code in which dynamically dead instructions are missing. This situation happens where different executed instances of a basic block end up being simplified differently after the simplification process, which results in creation of distinct basic blocks for the same piece of code in the obfuscated code. The final step of deobfuscation is to apply semantics-preserving transformations to simplify the control flow graph. In particular, we identify and merge basic blocks that differ solely due to dynamically

dead instructions. The following snippet of code, to compute the factorial function, illustrates the problem:[3]

```
int factorial(int n) {
  int i, p;
  p = i = 1;
  while (n > 0) {
    p = p*i
    i = i+1
    n = n-1
  }
  return p;
}
```

Suppose this function is called with the argument `n` = 2. The resulting execution trace for this function is:

```
/* 1 */   i = 1
/* 2 */   p = 1
/* 3 */   n > 0?        /* n == 2 */
/* 4 */   p = p*i
/* 5 */   i = i+1
/* 6 */   n = n-1
/* 7 */   n > 0?        /* n == 1 */
/* 8 */   p = p*i
/* 9 */   i = i+1
/* 10 */  n = n-1
/* 11 */  n > 0?        /* n == 0 */
/* 12 */  return p
```

The statement at position 9 in this trace, '`i = i+1`', is dynamically dead, since the value it computes at that point in the execution is not used later, and so it is removed during trace simplification. When a control flow graph is constructed from the simplified trace, however, we get two different versions of the loop body:

```
p = p*i
i = i+1
n = n-1
n > 0?
```

and

```
p = p*i

n = n-1
n > 0?
```

[3]In reality we work with assembly instructions. This example uses C code for the program, and a quasi C syntax for the trace, for simplicity and ease of understanding.

The first of these corresponds to the iterations up to the last iteration, while the second corresponds to the last iteration. More generally, depending on the dependence structure/distance of the loop(s) we may get multiple such loop body fragments with some code simplified away. Such blocks are treated as distinct by the control flow graph construction algorithm, resulting in a graph that has more vertices, and is more cluttered, than necessary. A similar situation arises with function calls if some call sites use the return value but others do not.

We deal with this situation by identifying and merging basic blocks that are identical modulo dynamically dead instructions. Define two blocks B_1 and B_2 to be *mergeable* if the following conditions hold:

1. B_1 and B_2 span the same range of addresses (except possibly for any dynamically dead instructions at the beginning and/or end of either block).

2. [*Non-dynamically dead instructions*] If an instruction I occurs in both B_1 and B_2, then it is the identical instruction in both B_1 and B_2. I.e., the operands should not have changed (e.g. due to constant propagation).

3. [*Dynamically dead instructions*] For each instruction $I \in B_1$ that does not occur in B_2, I is dead if it is added into B_2 at the appropriate position; and analogously with instructions that are in B_2 but not in B_1.

To simplify the control flow graph, we repeatedly find mergeable basic blocks and merge them to obtain the final control flow graph.[4]

4.3 Experimental Evaluation

We have evaluated our ideas using a prototype implementation of our approach. Execution traces of the original and obfuscated binaries were collected using a modified version of Ether [49]. Trace simplification was carried out on a machine

[4]From an implementation perspective, it turns out to be simpler to modify the simplified trace to reintroduce, where necessary, dynamically dead instructions that had been simplified away, and then rebuild the control flow graph.

with 2× quad-core 2.66 GHz Intel Xeon processors with 96 GB of RAM running Ubuntu Linux 12.04. The results of our experiments are discussed below. To quantify the similarity between the original and the deobfuscated programs (and, for completeness, the obfuscated programs as well), we use an algorithm of Hu, Chiueh, and Shin for computing the edit distance between two control flow graphs [70]. Given two control flow graphs G_1 and G_2, this algorithm computes a correspondence between the vertices of G_1 and G_2 using maximum bipartite matching, then uses this correspondence to determine the number of edits, i.e., the number of vertex and edge insertion/deletion operations necessary to transform one graph to the other. To facilitate comparisons between CFGs of different sizes, we normalize the edit distance to the total size of the graphs being compared using the following formula. Let $\delta(G_1, G_2)$ be the edit distance between two control flow graphs G_1 and G_2, then their similarity is computed as

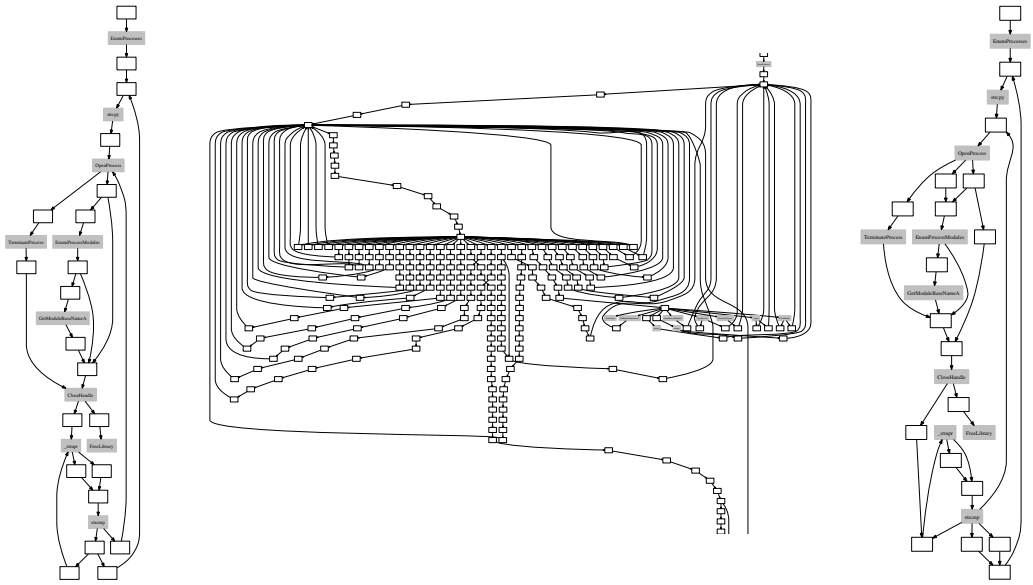
$$sim(G_1, G_2) = 1 - \frac{\delta(G_1, G_2)}{|G_1| + |G_2|}$$

where $|G|$ is the size of the graph G and is given by the total number of vertices and edges in G. A similarity score of 0 means that the graphs are completely dissimilar while a similarity score of 1 suggests that the graphs are identical.

Our experimental samples, including source code for the test programs and executables for the original and obfuscated programs, are available at `www.cs.arizona.edu/projects/lynx/Samples/Obfuscated/`.

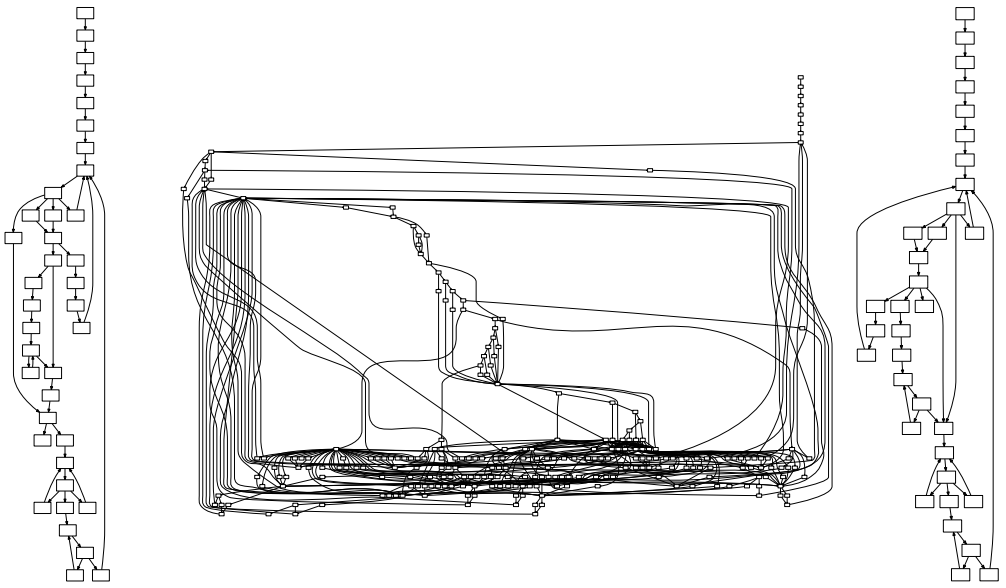
4.3.1 Emulation-based Obfuscation

We evaluated our deobfuscator using four commercial emulation-obfuscation tools: Code Virtualizer [101], EXECryptor [129], Themida [102], and VMProtect [138]. Code Virtualizer and VMProtect are representative of obfuscation tools that have been considered in previous work [40, 117]; these authors do not discuss EXECryptor so we do not know whether they are able to handle software obfuscated using this tool. As far as we know, none of the existing approaches on deobfuscation of



Original Obfuscated (cropped) Deobfuscated

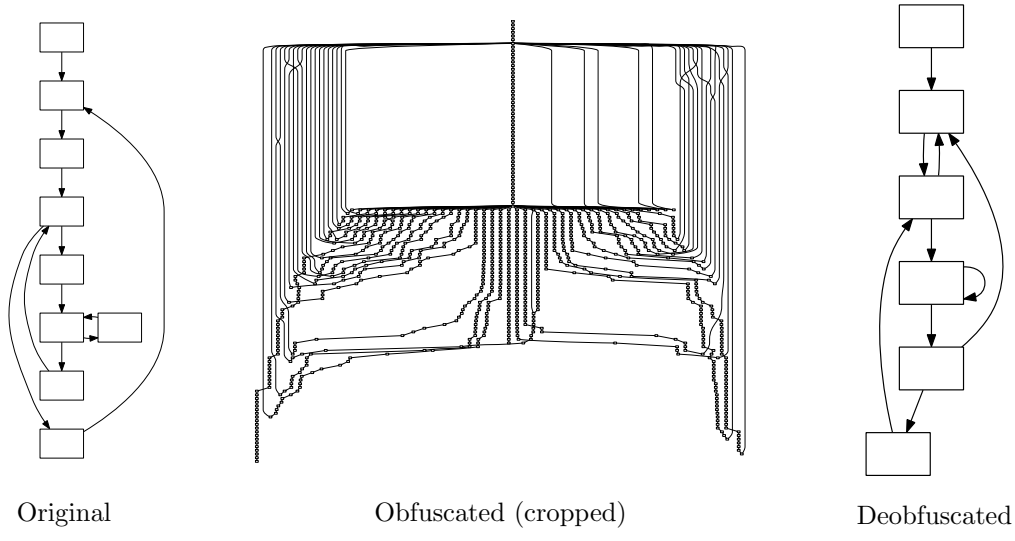
(*a*) Netsky_ae1: Code Virtualizer



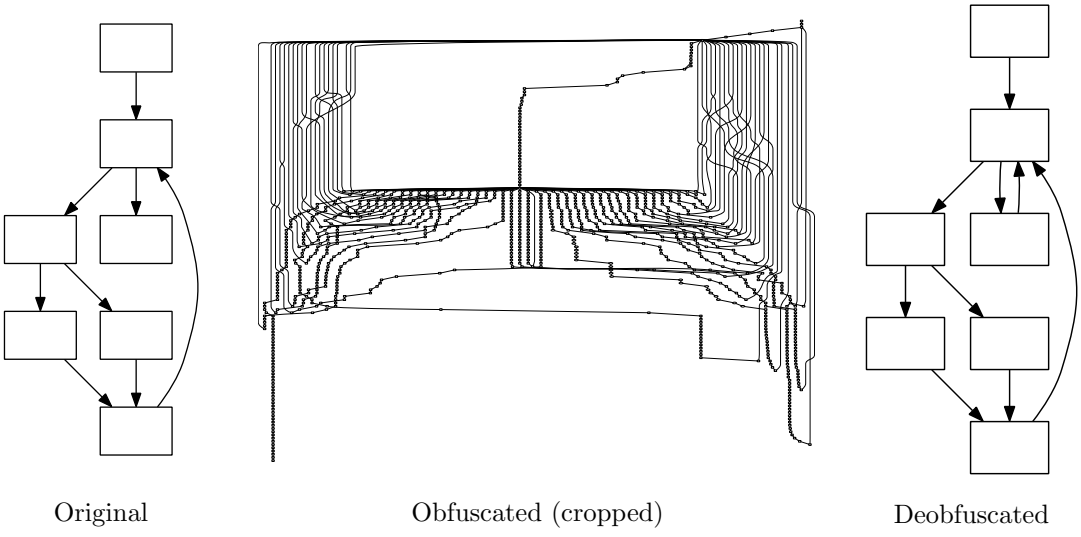
Original Obfuscated (cropped) Deobfuscated

(*b*) Hunatcha: ExeCryptor

Figure 4.5: Effects of obfuscation and deobfuscation on the control flow graphs of some malware samples

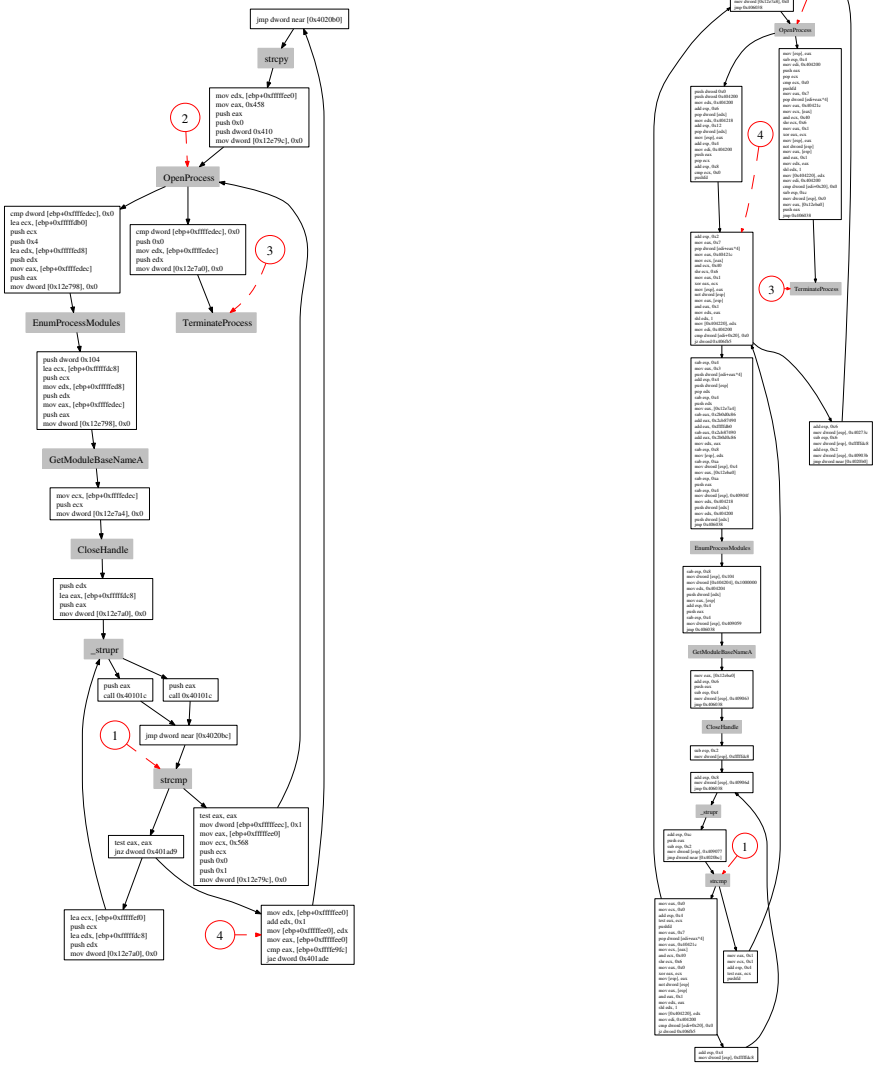


(a) Matrix-multiply: Themida



(b) binary-search: Themida

Figure 4.6: Effects of obfuscation and deobfuscation on the control flow graphs of some sample program



(*a*) Original program

(*b*) Deobfuscated program

Figure 4.7: CFGs containing instructions for original and deobfuscated Netsky_ae obfuscated with Code Virtualizer

emulation-obfuscated software are able to handle binaries obfuscated using Themida. When obfuscating programs using Themida, users can select various parameters, including the complexity of the VM instructions: for our experiments used the setting '*mutable CISC processor*' with one VM whose opcode type is '*metamorphic level-2*'.[5] Moreover, while virtualization is the main obfuscation technique used by these tools, all of the obfuscation tools use a wide variety of other obfuscation techniques, such as runtime code unpacking, data encoding/decoding, opaque predicate insertion, anti-analysis and anti-tracing techniques etc. where many of these techniques are by default applied to the protected binary. Please refer to the vendor's website for a full list of obfuscation features embodied in these obfuscation tools.

4.3.1.1 Single-Level Emulation

Single-level emulation refers to obfuscation where there is just a single level of emulation, namely, that of the emulator introduced by the obfuscation process.

To evaluate the quality of deobfuscation results using our approach on single-level emulation, we applied the commercial obfuscators named above to several malware programs, whose source code we obtained from VX Heavens [31], together with two synthetic benchmarks we wrote ourselves. The malware programs we used were: *Blaster* [146], *Cairuh*, *epo*, *hunacha*, *newstar*, and *netsky_ae* [140]. Of these programs, *Blaster* is a network worm; *Cairuh* is a P2P worm; *hunatcha* is a file dropper; *newstar* and *epo* are file infectors that implement different file infection mechanisms to drop payloads into other files; and *netsky_ae* is a worm whose functionality we divided into different pieces: *netsky_ae1* searches and eliminates antivirus and monitoring software running on the system, *netsky_ae2* installs the malware for surviving the system boots, *netsky_ae3* infects the system with encrypted variations of the malware and *netsky_ae4* recursively copies the malware into shared folders.

[5] "Mutable CISC processor" and "metamorphic level-2" are settings in the Themida tool; the available documentation does not specify, in any further detail, exactly how these settings affect the low-level characteristics of the obfuscated code.

In addition to these malware programs, we also used four synthetic benchmarks, *binary-search*, *bubble-sort*, *huffman*, and *matrix-multiply*, to explore how our techniques handled various combinations of conditionals and nested loops.

Figures 4.5 and 4.5 give a high-level visual impression of the effect of emulation-based obfuscation, together with the deobfuscated programs obtained using our approach, for two different malware samples: *Netsky_ae1* obfuscated with Code Virtualizer, *Hunatcha* obfuscated with EXECryptor, and *matrix multiply* and *binary-search* programs obfuscated using Themida. These samples have have reasonably interesting control flow structure, consisting of nested loops and conditionals. In order to focus the discussion on the core portion of the computation, the graphs shown omit the program setup/takedown and I/O code. It can be seen, from visual inspection, that the control flow graph resulting from deobfuscation is in each case very similar to that of the original program while the obfuscated code has a very complex CFG which requires significant resources to be analyzed.

netsky_ae1 as mentioned, searches the processes list to find antivirus software and terminates those that their name matches to one of the names in a preinitialized list. It can be seen, from Figure 4.5, that the control flow graph for the obfuscated program—while not identical to the input program—is generally quite close to it. For *hunatcha*, the deobfuscator is able to recover the loop structure of the computation which copies a file from a specified system directory to every disk drive on the infected system and For *Hunatcha* it is able to capture the logic that the malware uses to infect other executable files. In the deobfuscated results The most common issue that decreases the similarity of the deobfuscated CFGs to the original ones seems to be that the deobfuscated graphs have a small number of "extra" control flow edges: e.g., in *binary-search* there is an extra edge from the exit block back to the loop header. This is mostly due the code transformations that are caused by obfuscation process that must be untangled in the final CFG construction algorithm (see Section 4.2.6).

The results of the similarity comparisons are shown in Table 4.1. Columns labeled '*Obf.*' give the similarity of the obfuscated programs with the original programs;

	Control flow graph similarity (%)							
	CV		*EC*		*TH*		*VM*	
Program	*Obf.*	*Deobf.*	*Obf.*	*Deobf.*	*Obf.*	*Deobf.*	*Obf.*	*Deobf.*
binary-search	03.85	85.29	06.72	100.0	01.26	95.83	07.15	90.32
bubble-sort	05.47	80.64	01.38	93.15	01.64	95.08	07.94	79.24
huffman	20.75	72.24	06.08	83.50	06.03	83.91	16.45	46.40
hunatcha	22.43	90.30	04.82	90.04	05.60	84.84	15.57	73.65
matrix-mult	06.50	81.63	01.31	83.95	01.56	81.63	07.22	75.55
Cairuh	39.37	89.02	26.46	94.04	NA	NA	28.68	82.39
blaster	13.25	84.54	02.40	84.87	NA	NA	14.07	89.24
newstar	09.09	94.38	02.15	92.56	02.21	96.70	08.49	75.20
epo	29.26	92.51	07.86	80.92	09.28	81.23	20.03	96.28
netsky_ae1	19.78	88.03	08.19	87.27	06.15	84.14	19.00	82.81
netsky_ae2	50.90	80.85	13.12	93.40	19.75	88.17	24.50	89.95
netsky_ae3	11.52	92.85	02.43	85.49	03.84	82.81	09.35	94.36
netsky_ae4	30.30	86.60	20.71	75.04	14.04	82.66	22.65	87.85
AVERAGE	23.01	84.20	08.68	93.19	07.60	92.01	16.91	83.60

Table 4.1: Similarity of original and deobfuscated control flow graphs: Emulation-obfuscation

those labeled '*Deobf.*' give the similarity between the deobfuscated programs and the original programs. Not surprisingly, the obfuscated programs are usually very different from the original code structurally: by and large these similarity numbers are in the 6%–8% range, with several programs showing similarities of less that 10%, and a few (e.g. *huffman*, *hunatcha* and *epo* for Code Virtualizer, and *huffman* for VMProtect) with similarity values over 15%. The exceptions here are *Cairuh*, *netsky_ae2* and *netsky_ae4* which because of having switch statements in their code, they are structurally similar to the virtualized binaries so they are in fact more similar to the obfuscated binaries than the other programs.

By contrast, the control flow graphs resulting from our deobfuscation algorithm have significantly higher similarities. While nearly similar on average, they are highest for Code Virtualizer and EXECryptor, ranging from 72% to 95% for Code Virtualizer and in the range of 75% to 100% for EXECryptor. On average the similarity values for Code Virtualizer and EXECryptor are 84.2% and 93.1%. The

deobfuscation results are comparable for Themida and VMProtect, ranging from 82% to 96% for Themida and from 46% to 96% for VMProtect. However, it should be noted that our approach still achieves significant improvements in similarity relative to the obfuscated code.

Our Ether-based tracing infrastructure crashed on the *Cairuh* and *blaster* programs obfuscated with Themida so we were unable to collect an execution trace for these programs.

The inaccuracies in the simplified CFGs are mainly caused by our final CFG construction algorithm where we use heuristics to untangle pieces of simplified code to recover the original logic. Ultimately, one can compare the semantics of the simplified code to that of the original code. But comparing the semantics of two pieces of code automatically is a hard problem and is not in the scope of this study, hence we used CFGs to show how similar the simplified codes are to the original ones. For most of our evaluations, we manually tested and confirmed that the deobfuscation preserves the original logic in the simplified trace.

In Figure 4.7 we have included the CFGs of a sub-trace of the *netsky1_ae* program with instructions included in the graph: Figure 4.7(a) corresponds to the original program and (b) corresponds to the deobfuscated program obfuscated using Code Virtualizer. This shows that with the high level information that can be recovered by the CFGs, program semantic information is also included at the instructions level. For example in Figure 4.7, it can be seen that in both graphs, there is a test on the output of the `strcmp` function call marked with label 1. The program is trying to kill all the unwanted processes currently running in the system and by comparing process names with ones in a list, it determines whether to terminate the process or not. If the comparison satisfies, it calls `OpenProcess` (labeled with 2) and then terminates the process using a call to `TerminateProcess` (labeled with 3). There is correspondence between two graphs and the semantics are equivalent in both the original and deobfuscated programs. Getting this level of information from the obfuscated program, where the graph is shown on Figure 4.5(b), is very unlikely, if not impossible, and requires significant amount of time and effort.

However, there is one difference between two graphs that should be noted here. As it was discussed in Section 4.2.6, the CFG construction algorithm tries to balance between the code duplications and the number of paths in the final graph. Doing so, the CFG constructed for the deobfuscated program uses an existing block (pointed by label 4) rather than duplicating it for the corresponding block in original program (also pointed by label 4). This is mostly because in the original program, only one target branch is observed (for the basic block pointed by label 4) and so the CFG construction algorithm is not aware of the other branch existing in the original program. It should also be noted that this does not however affect the semantics of the program and the constructed graph still represents the original logic correctly and this is a general limitation for dynamic analysis where the code coverage is an issue rather a specific limitation of our approach.

Analysis speed depends partly on the input trace size but mostly on the number of iterations of code simplification needed, which in turn depends on how entangled the obfuscations are; there seems to be a non-linear component to the execution time that we are currently looking into. Execution times for the three largest trace files, *Cairuh*-VMProtect (6.4M instructions), *hunatcha*-Themida (7.7M instructions), and *huffman*-Themida (56.6 M instructions) are 188 sec, 244 sec, and 4,726 sec respectively, which translate to speeds of 34,042 instrs/sec, 31,557 instrs/sec, and 11,976 instrs/sec respectively. However the process can be parallelized. For taint analysis there are studies looking into how to parallelize dynamic taint analysis [91] that can be utilized in our approach as well. Similarly, the simplification step can be parallelized by dividing up the execution trace into different chunks. The chunks then can be simplified in parallel iteratively while neighbor chunks in the trace can communicate information about what variables can be simplified.

4.3.1.2 Multi-level Emulation

We have also applied our approach to programs obfuscated using multiple levels of emulation [54, 63], i.e., where one emulator interprets another emulator which in

turn interprets byte code for the program to be executed: the results are similar to those presented here, in that we are able to remove most of the obfuscation and recover deobfuscated control flow graphs that are very similar to those shown here. We selected a subset of our test programs which we used for single-level emulation, including *binary-search*, *bubble-sort* and *matrix-multiply* and obfuscated them using Code Virtualizer, and then applied another round of emulation using EXECryptor. Each of these programs therefore had two levels of emulation. We also wrote an emulator, modeled on DLXsim and SPIM, for a small RISC-like processor that we call *tinyRISC* (see Appendix B for source code), and ran it on hand-compiled byte-code for a binary-search program. This sample was used because it has another layer of emulation in addition to those added by the obfuscation tools. This program was also obfuscated using Code Virtualizer and EXECryptor and is included as *tinyRISC:bin-search*; this program uses three levels of emulation (the tinyRISC emulator, Code Virtualizer, and EXECryptor). Table 4.2 shows the similarity numbers for the obfuscated and deobfuscated CFGs of our test programs. It can be seen that the similarity of the deobfuscated CFGs and the original CFGs ranges from 80.6% to 87.9%. This shows that our approach is effective in cutting through multiple levels of emulation.

The similarity between the numbers for the multi-level emulated binaries and the ones obfuscated using only Code Virtualizer in Table 4.1 suggests that applying additional levels of emulation does not change the structure of the underlying interpreted program, although the obfuscated programs are quite different (see CFG similarity numbers for the obfuscated programs in the two cases), and the execution traces differ significantly with those for multi-level emulation being significantly larger.

4.3.2 Return-Oriented Programs

We evaluated our prototype implementation with two different sets of ROP test cases. The first set of binaries were simple synthetic programs including *factorial*, *fibonacci*, *matrix-multiply* and *bubble-sort*. These programs were implemented by

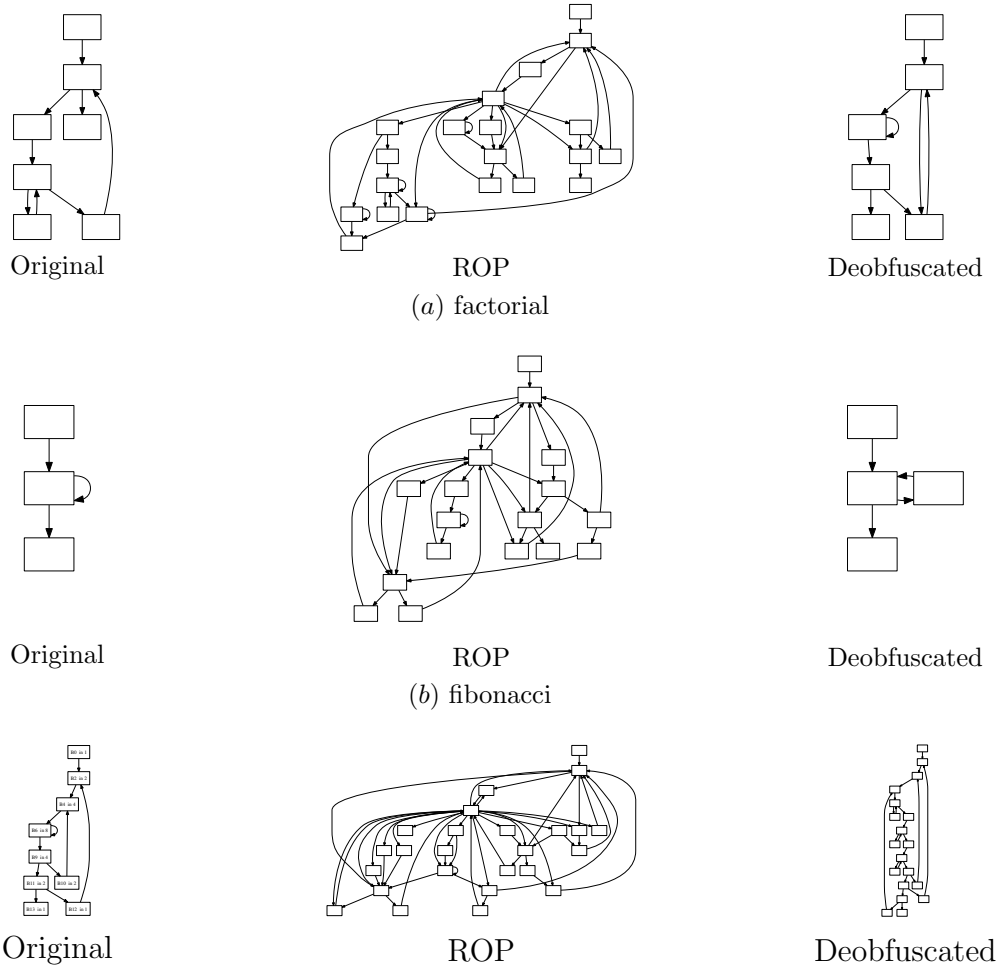
Program	No. of Levels	CFG similarity (%)	
		Obf.	*Deobf.*
binary-search	2	4.45	85.29
bubble-sort	2	6.41	80.64
matrix-multiply	2	5.26	81.63
tinyRISC:bin-search	3	4.45	87.87
Average		5.14	83.85

Table 4.2: Similarity of original and deobfuscated control flow graphs: multi-level emulation. *No. of Levels* gives the number of emulation levels in the obfuscated code.

chaining relevant ROP gadgets from Windows system libraries such as `ntdll.dll` and `msvcrt.dll` rather than a high level programming language to carry out the intended computation so they can simulate the behavior of ROP attacks. We chose these programs because they have enough complex structures such as loops and conditional statements to measure the ability of a reverse engineering system which tries to recover the logic of the underlying computation. For comparison purposes we also created the non-ROP version of the programs which are written in C. We also applied our approach to several ROP malware samples, but found that our ROP malware samples had a relatively simple control flow structure since all they were trying to do was to change the access permissions on some memory pages to make them executable. As a result, our hand-crafted ROP benchmarks presented a greater challenge for deobfuscation than the malware samples we tested.

The similarity numbers for our synthetic programs are presented in Table 4.3. The column labeled *Obf.* shows the CFG similarity of the ROP version of the program to its non-ROP version and column labeled *Deobf.* shows the similarity of the deobfuscated ROP program to its non-ROP version. The table shows that our method is also able to reverse engineer the ROP gadgets and produce a very similar control flow graph to the non-ROP version by simplifying the ROP version execution trace.

We have included the set of control flow graphs of two ROP programs, *factorial* and *fibonacci* and *matrix-multiply* in Figure 4.8 very similar to Figures 4.5 and 4.6.



(*a*) factorial

(*b*) fibonacci

(*c*) matrix-multiply

Figure 4.8: Some examples of ROP deobfuscation results

Note that the factorial program has a nested loop; the reason is that we did not find a multiplication gadget in `ntdll.dll` or `msvcrt.dll`, so we simulated this using a loop of additions.[6]

[6]This problem with unavailability of multiplication gadgets in Windows system libraries, and a solution using iterated addition, is also discussed by Roemer *et al.* [112].

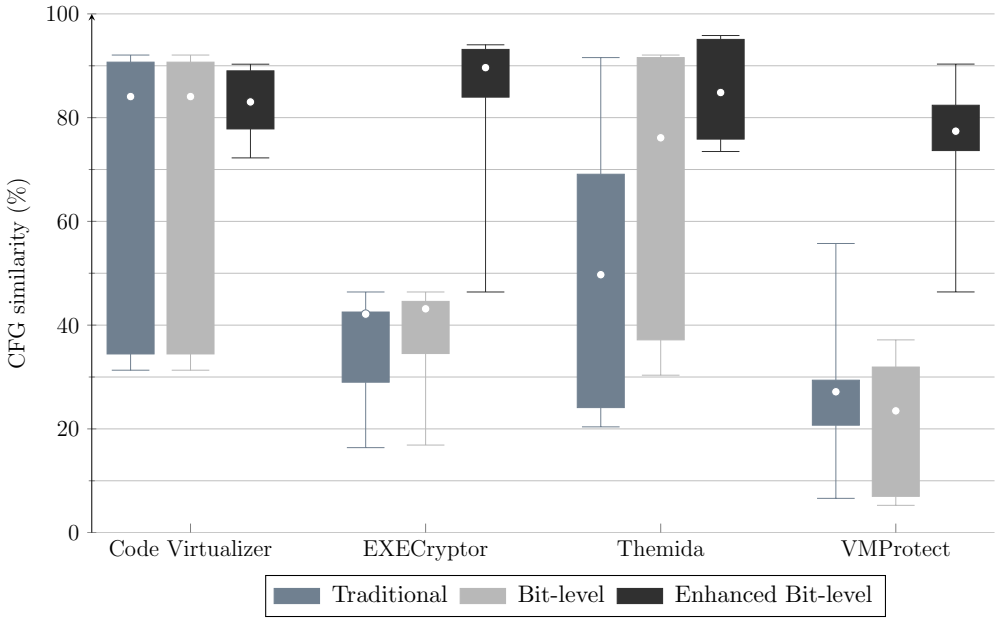


Figure 4.9: Comparing different taint analysis approaches in deobfuscation

4.3.3 Effects of Different Taint Analysis Approaches

To show the difference between various taint analysis approaches discussed in Section 4.2.4.1, we repeated the above experiment each time while only changing the forward taint analysis component. The results are shown in Figure 4.9. To help the reader better understand the performance of each algorithm, the data is reported in the form of Boxplots. Each box shows CFG similarity numbers for the set of test programs for each obfuscator computed using one of the taint analysis algorithms. Blue boxes show the numbers computed using traditional taint analysis algorithm, gray boxes are for numbers generated with bit-level taint analysis algorithm and dark boxes show the result when enhanced bit-level taint analysis algorithm was used.

Generally it can be understood from the picture that bit-level taint analysis is better than doing it in byte-level and enhanced bit-level taint analysis outperforms the previous two algorithms. This is particularly true in cases when the obfuscator is EXECryptor or VMProtect since these two obfuscators introduce more complicated obfuscations to the control flow structure of the underlying program. These obfus-

Program	*CFG similarity (%)*	
	Obf.	*Deobf.*
factorial	47.61	88.88
fibonacci	30.61	85.71
matrix-multiply	64.51	79.22
bubble-sort	48.22	82.85
Average	47.73	84.16

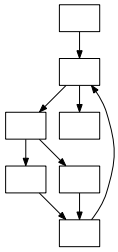
Table 4.3: Similarity of original and deobfuscated control flow graphs: ROPs

cations confuse the taint analysis and results in taint explosion and degrades the deobfuscation results significantly. For Themida and Code Virtualizer, the results of the first two algorithms are comparable since doing the taint analysis in bit-level or byte-level, will mostly discover the structure of the interpreted program. However, the enhanced bit-level analysis helps the simplification process to produce a more precise deobfuscation.

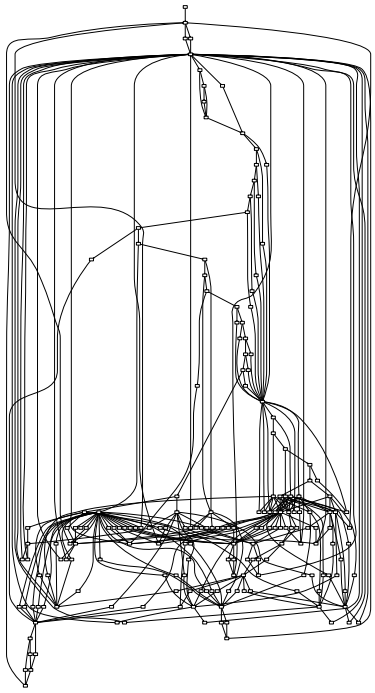
Figure 4.10 shows the control flow graphs for the binary-search program obfuscated using EXECryptor. Figure 4.10(a) shows the original CFG while (b) is the obfuscated program. Figures 4.10(c), (d) and (e) show the simplified graphs with standard, bit-level and enhanced taint analysis algorithms respectively. It can be seen from the simplified CFGs that neither of the graphs in Figure 4.10(c) and (d) are similar to the original CFG. The reason is that both standard byte-level and bit-level taint analysis algorithms, while the bit-level analysis doing slightly better than the standard byte-level algorithm, are too imprecise in propagating the taint and over-taint irrelevant code making the analysis not being able to resemble the original logic. Nevertheless, Figure 4.10(e) shows that with enhanced taint analysis we are able to recover the original logic of the obfuscated program suggesting that the enhanced algorithm is able to propagate the taint precisely.

4.3.3.1 Malware Analysis Results

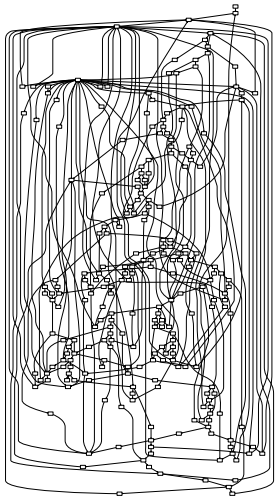
We have also applied our deobfuscator to a number of emulation-obfuscated malicious binaries that we obtained from *virusshare.com*, including *Win32/Kryptik*, *Trojan-*



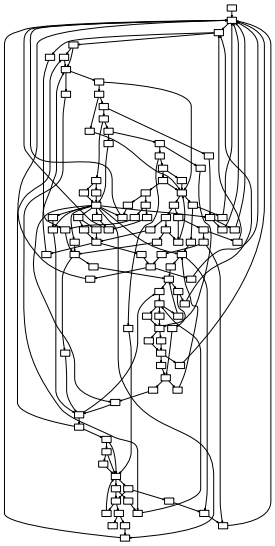
(a) Original CFG



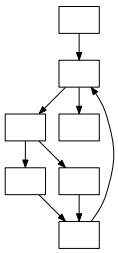
(b) Obfuscated CFG



(c) Simplified CFG using standard byte-level taint analysis



(d) Simplified CFG using standard bit-level taint analysis



(e) Simplified CFG using enhanced taint analysis

Figure 4.10: Deobfuscation result of binary-search program protected by EXECryptor using different taint analysis methods

Downloader.Banload, *Win32.Dubai*, *W32/Dialer*, and *Backdoor.Vanbot* where the CFG results for *Win32/Kryptik* and *Backdoor.Vanbot* are shown in 4.11 and 4.12 respectively. We found that in the samples we tested, emulation was typically applied selectively to selected sensitive code regions, with multiple layers of unpacking added subsequently to further obfuscate the malicious payload. Our deobfuscator was able to remove all of the emulation and unpacking code, leaving only the logic of the malicious payload with a much simpler CFG. The time taken to perform this simplification for the malware samples we tested was around 10 minutes per sample.

Overall, these results show that while our prototype implementation is not yet perfect, it is nevertheless able to extract control flow graphs that closely resemble those of original unobfuscated programs. Notably, it is able to do this for both "ordinary" emulation-obfuscated programs and also Themida-obfuscated programs, which combine runtime unpacking with emulation and, as far as we know, are not handled by any previously proposed techniques for automatic deobfuscation. Considering that we make very few assumptions about the nature of the obfuscations applied, we consider this encouraging. We are currently working on improving our analyses to improve the deobfuscation results further.

We show below the results of deobfuscating two emulation-obfuscated malware samples. Figure 4.11 shows the obfuscated and simplified control flow graphs of a malware sample obfuscated using Code Virtualizer known as `Win32/Kryptik` according to Virustotal. Our analysis indicates that this sample has 3 layers of code unpacking; this is confirmed by the Ether unpacking service (`http://ether.gtisc.gatech.edu/web_unpack/`). Basic blocks that perform code unpacking, i.e., write to memory locations that are subsequently executed as code, were made slightly bigger and are shown in red. The result of deobfuscation gets rid of the unpacking code since, from the perspective of the semantics of this program, it is the unpacked code that matters. The resulting CFG can be seen to be considerably simpler than the original CFG. With the malware sample *Win32/Kryptik*, the emulation-obfuscation applied to just the top-level unpacker routine rather than the application logic (see Figure 4.11). We conjecture that this selective application of emulation-based obfuscation

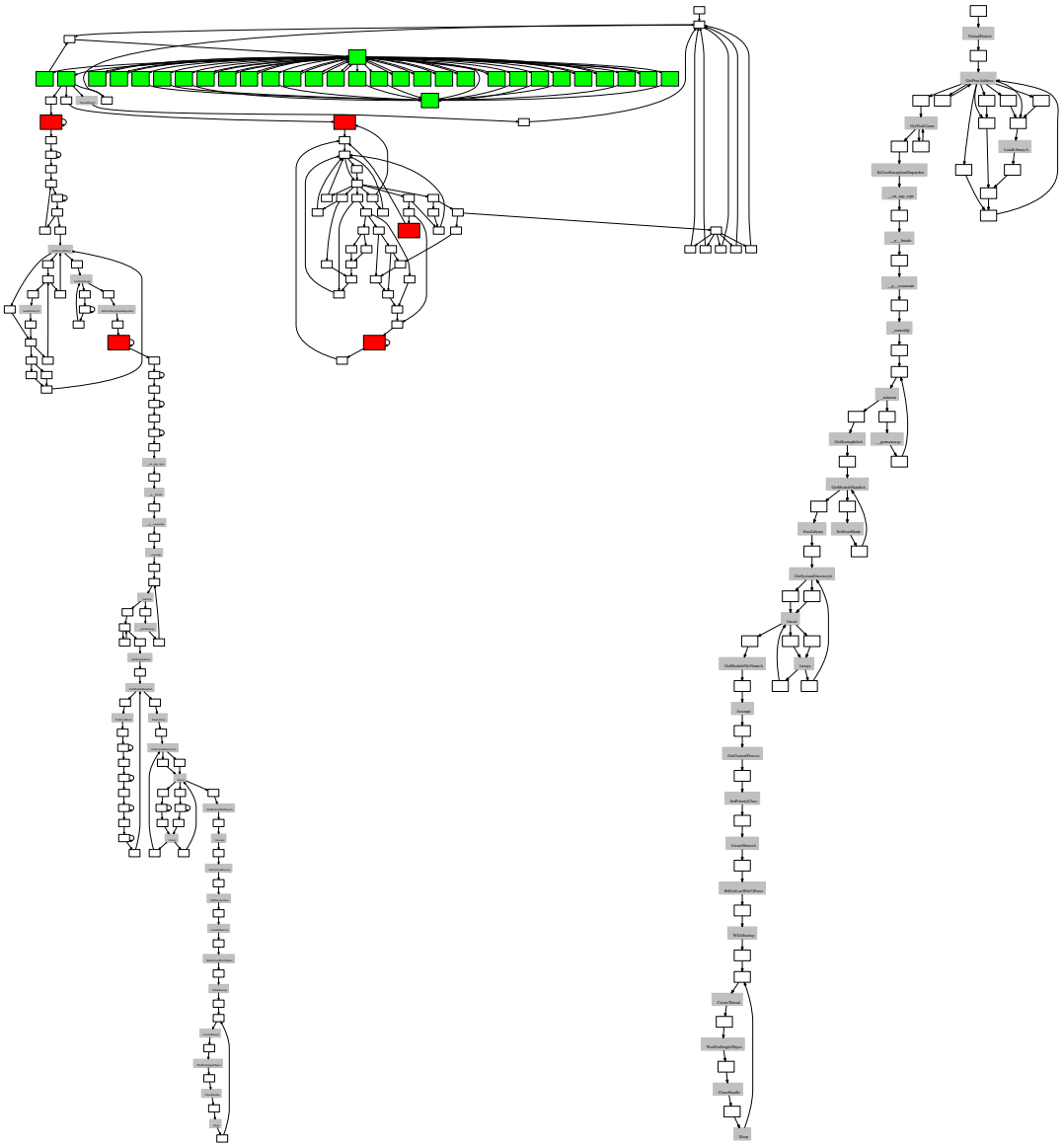
may have been motivated by a desire to avoid the space and time overheads that would result from applying this obfuscation to the entirety of the code.

Figure 4.12 shows the analysis result of a sample known as `Backdoor.Vanbot` and it is protected using EXECryptor. This sample has 9 layers of unpacking; these are again marked with red and bigger basic blocks. Again, the CFG resulting from deobfuscation is significantly simpler than that of the original obfuscated code.

4.3.4 Comparison With Coogan et al.

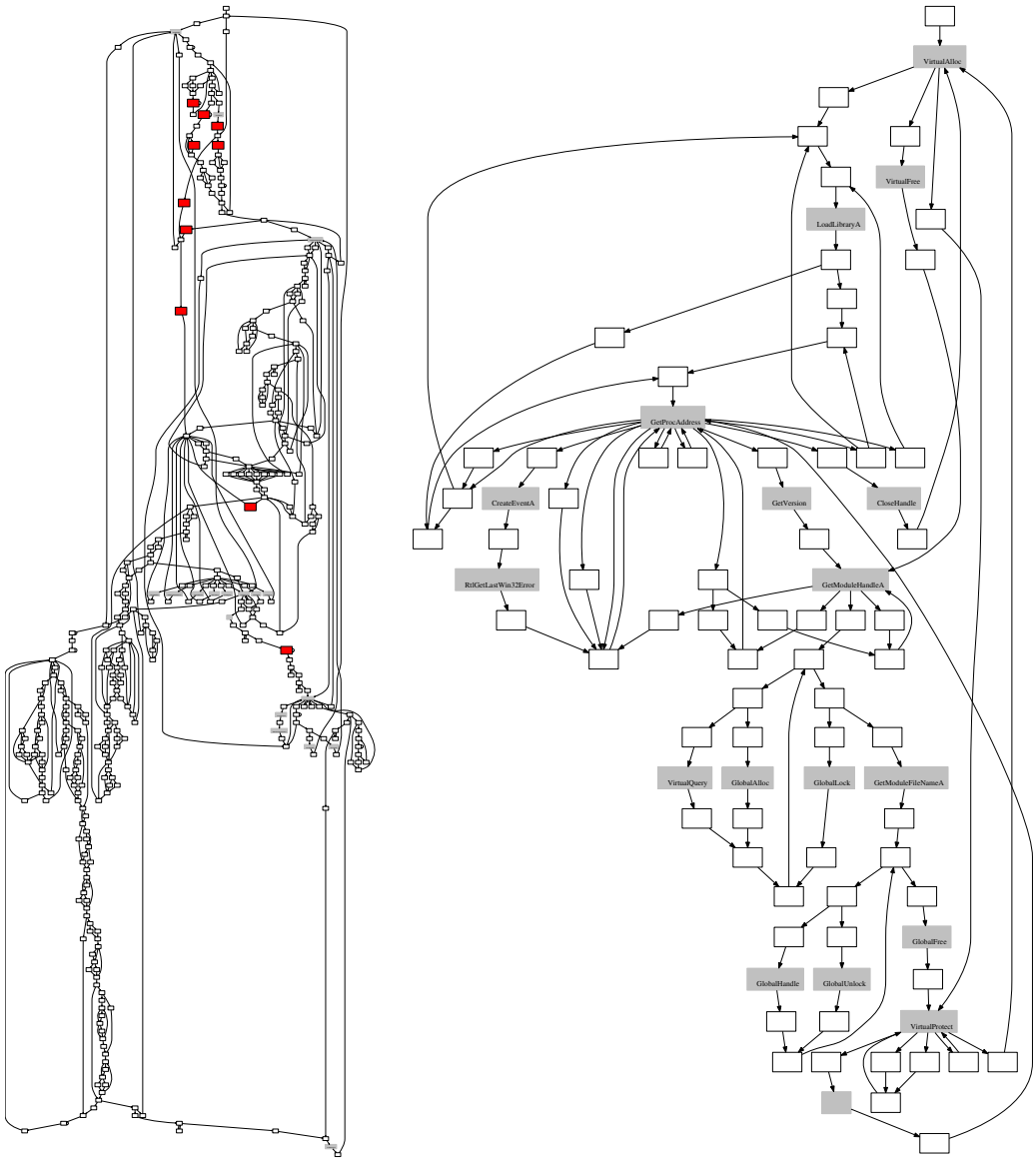
We tested our approach against that of Coogan *et al.* [40]; the results are shown in Figure 4.13. Coogan's approach results in complex equations that are difficult to map to CFGs, especially for nontrivial programs. Our approach, by contrast, produces CFGs that can be meaningfully compared to the original program's CFGs. So we think that our approach produces more understandable results than Coogan's. We ran Coogan's tool on their set of test programs and mapped the resulting *relevant sub-traces* (which is equivalent to the *deobfuscated program* in our terminology) to CFGs. We first applied our tool on the traces used by Coogan *et al.* in their experiments [40] and compared the similarity of the resulting deobfuscated traces with the original ones. To compare the result of the two tools, we also generated CFGs of the relevant sub-traces produced by their tool and compared the CFGs to the original programs. It can be seen, from Figure 4.13, that our system outperforms Coogan's tool with a 30% to 60% higher similarity numbers in all the programs. We were not able to get a result of their tool on the *md5* program obfuscated using Code Virtualizer because the computation did not finish on time so we do not have any data for that. The small difference between similarity numbers of the programs that are common in our set of input programs and the set they used for evaluation, e.g., *hunatcha*, is that the programs used by Coogan *et al.* and represented in Figure 4.13 are slightly different from those used for Table 4.1.

Coogan *et al.* do not apply their technique to obfuscations other than emulation, nor do they provide results for multi-level emulation.



Original Deobfuscated

Figure 4.11: Deobfuscation of *Win32/Kryptik.OHY* (protected by Code Virtualizer)



Original

Deobfuscated

Figure 4.12: Deobfuscation of *Backdoor.Vanbot* (protected by EXECryptor)

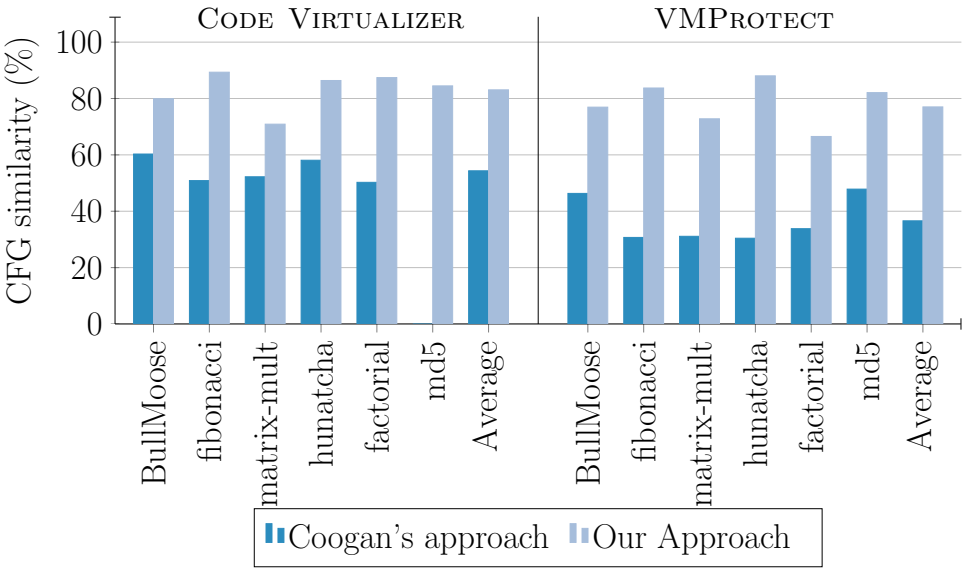


Figure 4.13: Comparison with Coogan *et al.*

4.4 Related Work

The work that is philosophically closest to ours is that of Coogan *et al.* [40], who use equational reasoning about assembly-level instruction semantics to simplify away obfuscation code from execution traces of emulation-obfuscated programs. While their goals are similar to ours, the technical details are very different. The biggest difference between the two is in the processing and simplification of execution traces. The equational reasoning approach of Coogan *et al.* has some significant drawbacks, the most important being that it is difficult to control the equational simplification, making it hard to separate out the different components of nested loops or complex control flow. This makes it difficult for their approach to extract the logic of the underlying computation into higher-level structures such as control flow graphs or syntax trees. By contrast, our approach offers a lot more control over the deobfuscation process and allows us to recover higher-level representations, such as control flow graphs, with a high degree of precision, as illustrated by the data in Figure 4.13. Importantly, Coogan *et al.* limit themselves to emulation-based obfuscation, and provide data only for one level of emulation; by contrast, we are

able to handle multiple levels of emulation with good results, and applies to other kinds of programs, e.g., ROPs.

Sharif *et al.* describe an approach [117] that works from the outside in: it first reverse engineers the VM emulator; uses this information to work out individual byte code instructions; and finally, recovers the logic embedded in the byte code program. This outside-in approach can be very effective when the structure of the emulator meets the assumptions of the analyzer. However, when the emulator uses techniques that do not fit these assumptions the deobfuscator may not work well. For example, this approach does not fully deobfuscate code that has been obfuscated using Themida, which virtualizes the unpacker routine for emulator instructions; for such programs, it is able to automatically recover only the unpacker logic (rather than that of the application), with further analysis then done manually. We have recently seen similar characteristics in code obfuscated with other emulation-based obfuscators as well: e.g., `Win32/Kryptik`. This development suggests that obfuscation-specific approaches that focus on identifying and reverse-engineering the emulator may become less effective in the face of selective application of obfuscation. This approach may also not generalize easily to code that uses multiple layers of emulation, since it may be difficult to distinguish between instruction fetches for various emulators.

Some researchers have proposed static approaches for simplifying (quasi-)interpretive code. Udupa *et al.* [136] discuss techniques for deobfuscating code that has been obfuscated using *control flow flattening* [141], which in some ways resembles emulation-based obfuscation. Jones *et al.* [73] describe a technique called *partial evaluation* for specializing away interpretive code. The analyses and transformations described in these works are static, which suggests that it may not be straightforward to apply them to highly obfuscated malware binaries, e.g., due to dynamic unpacking and self-modifying code.

There is a significant and growing body of literature on ROP, but most of it deals with attacks [112, 116, 18, 32] or defenses [44, 87, 100, 33]. Lu *et al.* discuss the conversion of ROP shellcode to semantically equivalent shellcode that does not use ROP [87], but this work is specific to ROP.

CHAPTER 5

Symbolic Execution of Obfuscated Code

5.1 Introduction

Symbolic and concolic execution play important roles in a variety of security and software testing applications, e.g., test case and exploit generation [24, 23, 64, 115, 30], vulnerability detection [24, 25, 34], and code coverage improvement in dynamic analysis of malware code [21, 13, 93]. The general idea behind symbolic/concolic execution is to represent computations along a particular execution path using logical formulas and apply constraint solving techniques to identify inputs that would cause the program to take alternative execution paths. Analyses based on symbolic execution are especially important for dealing with programs that are difficult to analyze using conventional techniques. This makes the precision of such analyses an important consideration in security applications: on the one hand, identifying too many candidate execution paths, with corresponding inputs, can overwhelm the analysis and slow down processing; on the other hand, missing some execution paths can cause the analysis to fail to explore important parts of the input program.

Given the importance of symbolic analysis for code coverage improvement in dynamic analysis of potentially malicious code, it is important to identify and understand any potential weaknesses of this approach. Previous studies have discussed attacks on symbolic execution systems using cryptographic hash functions [118] or unsolved mathematical conjectures [142] to construct computations that are difficult to invert. These are sophisticated attacks and help define theoretical boundaries for symbolic analyses, however they do not speak to potential problems in symbolic analysis arising out of code obfuscation techniques used by existing malware.

It turns out that several existing code obfuscation techniques used by malware (or simple variations on them) can significantly affect the precision of current concolic

analyses. For example, some obfuscations, such as those used in the software protection tool EXECryptor [129], can cause large amounts of over-tainting and lead to a path explosion in the symbolic analysis; others, such as those used by the obfuscation tool VMProtect [138], transform conditional branch instructions into indirect jumps that symbolic analyses find difficult to analyze; and finally, a form of runtime code self-modification, variations of which we have seen in existing malware, can conceal conditional jumps on symbolic values such that they are not detected by concolic analysis. This situation is problematic because a significant motivation behind using symbolic/concolic execution in malware analysis is to get around code obfuscations. This makes it especially important to devise ways to mitigate such loss of precision when performing symbolic analysis of obfuscated code.

This work on symbolic execution makes the following contributions: First, it identifies and describes the shortcomings in existing concolic analysis algorithms by describing different anti-analysis obfuscations that cause problems for symbolic execution. These obfuscations were selected because (1) they, or simple variants of them, are currently already used in malware, e.g., through tools like VMProtect and EXECryptor; and (2) the problems they cause for symbolic execution are not discussed in the research literature. Second, it describe a general approach, based on a combination of fine-grained taint analysis described in Section 2.3 and architecture-aware constraint generation, that can be used to mitigate the effects of these obfuscations. For the sake of concreteness, the discussion is in many places formulated in terms of the widely used *x86* architecture; however, the concepts are general and apply to other architectures as well. Our experiments indicate that the approach we describe can significantly improve the results of symbolic execution on obfuscated programs.

The rest of this chapter is organized as follows: Section 5.2 discusses background on concolic execution and introduces problems that arise in concolic analysis of obfuscated code. Section 5.3 discusses these challenges in greater detail. Section 5.4 describes our approach for dealing with these challenges. Section 5.5 presents experimental results and Section 5.6 discusses related work.

5.2 Background

5.2.1 Concolic Execution and Input Generation

Concolic (concrete+symbolic) execution uses a combination of concrete and symbolic execution to analyze how input values flow through a program as it executes, and uses this analysis to identify other inputs that can result in alternative execution behaviors [64, 115]. The process begins with certain variables/locations—typically, those associated with (possibly a subset of) the program's inputs—being marked as "symbolic." The instructions of the program are then processed as follows: if any of the operands of the instruction are marked symbolic, then the instruction is "executed" symbolically: the output operands of the instruction are marked as symbolic, and the relationship between the input and output operands of the instruction is represented as a constraint between the corresponding symbolic variables; otherwise, the instruction is executed normally and the program's state is updated. If a location or variable x becomes marked as symbolic, we say that x "becomes symbolic." The constraints collected along an execution path characterize the computation along that path in terms of the original symbolic variables, and can be used to reason about what inputs to the program can cause which branches in the program to be taken or not. Symbolic analysis can identify input classes to the program if there are control transfers in the program affected by the input values [76] by which program takes different execution paths.

Concolic execution is especially useful for exploring different execution paths through programs that are too complex for conventional analyses to produce accurate results. A subset of concrete values are used to substitute the symbolic variables with which in return simplifies the constraints and allows the analysis to move forward. It therefore finds important applications in automatic test-case generation for software testing and increasing code coverage for malware analysis [93, 21]. Many tools have been implemented to incorporate this technique for security and software testing purposes [24, 22, 34, 115]. Schwartz *et al.* [114] give a more thorough discussion of this topic.

5.2.2 Concolic Execution of Obfuscated Code

Figure 5.1 shows the problem with concolic analysis of obfuscated code. Our test program, shown in Figure 5.1(a), consists essentially of a single symbolic variable and two `if` statements, nested one inside the other, that give rise to a total of three distinct execution paths. Our goal is to use concolic execution to identify different inputs that will, between them, cover all three execution paths. Symbolic execution of this simple program is almost trivial: the concolic execution engine S2E [34] finds just two states and makes just seven queries, and the analysis takes less than 20 seconds overall. If we run this simple program through the obfuscation tool VMProtect [138], however, the results are dramatically different: a depth-first search strategy times out after more than 12 hours, having encountered close to 15,000 states and generated over 14,000 queries, but failing to generate any alternative inputs. A random search strategy does somewhat better in that it does not time out, but it takes nearly 800 times as long to generate alternative inputs compared to the unobfuscated version. This strategy encounters 25,800 states and generates more than 25,000 queries—an increase of four orders of magnitude. That such a trivial program should pose such a formidable challenge to symbolic execution when it has been obfuscated is sobering in its implications for more complex code: VMProtect and other similar obfuscators have been used for protecting malware against analysis for a decade or more (e.g., the Ilomo/Clampi botnet, which used VMProtect to protect its executables, was encountered in 2005 [46]). The problem is not specific to S2E: for example, when invoked on the obfuscated version of the program shown above, Vine [124] exits with an error message. The remainder of this chapter examines the reasons underlying the problems described above and some possible ways by which the problems may be mitigated or remedied.

5.3 Anti-Concolic Obfuscations

While there has been a great deal of work on constructing and defeating different kinds of obfuscations, for the purposes of this study we are concerned primarily

```
int main(int argc, char **argv){
    int n = atoi(argv[1]);  /* n is symbolic */
    int retVal;
    int r = n+6;

    if(r < 10){
        retVal = 10;
        if (r == 6){
            retVal = 4;
        }
    } else {
        retVal = 12;
    }
    printf("%d\n", retVal);
    return retVal;
}
```

(*a*) Program source code (unobfuscated)

Search strategy	*Version*	*No. of states*	*No. of queries*	*Analysis time (sec)*
DFS	original	2	7	18
	obfuscated	14,928	14,015	timeout ($>$ 12 hrs)
Random	original	2	7	17
	obfuscated	25,800	25,094	14,160

(*b*) Analysis statistics (S2E)

Figure 5.1: Effects of code obfuscation on concolic analysis performance (Obfuscator: VMProtect [138]; concolic engine: S2E [34])

					Overflow				Sign	Zero				Parity		Carry
Flag Bit		NT	IOPL		OF	DF	IF	TF	SF	ZF		AF		PF		CF
position	15	14	13-12		11	10	9	8	7	6	5	4	3	2	1	0

Figure 5.2: x86 `FLAGS` register [71]

with obfuscations that affect concolic analysis, focusing in particular on concolic analysis to improve code coverage in obfuscated and malicious code.[1] Such analyses use constraints on execution paths leading up to conditional jumps to determine alternative inputs that can cause a different execution path to be taken. There are basically two broad ways in which this approach can be attacked:

1. The conditional jump can be manipulated in ways that make it difficult to identify a relationship with the original inputs:
   (a) The conditional jump can be transformed into an indirect jump whose target depends on the predicate of the original conditional jump.
   (b) The conditional jump can be transformed into a different conditional jump whose predicate depends on, but is different from, that of the original conditional jump.
2. The conditional jump, or its relationship with the input, can be concealed:
   (a) The conditional jump can be injected into the instruction stream at runtime, in the form of a direct unconditional jump, using code self-modification which we call it "symbolic instruction" (See Definition 5.3.1 on page 129).
   (b) Implicit information flows can be used to conceal a conditional jump's dependence on inputs.

Of the possibilities listed above, here we focus on approaches 1(a), 1(b), and 2(a). Since the goal of symbolic execution is to examine all possible execution paths,

[1]The obfuscation tools we used to evaluate our techniques, discussed in Section 5.5, incorporate many additional obfuscations, but in our experience these other obfuscations did not have much of an effect on symbolic execution.

implicit flow does not impose a threat to symbolic execution. Sharif *et al.* discuss using cryptographic hash functions to realize an extreme form of approach 2(b) [118] but they also mention that using cryptographic functions raises suspicions so it is less likely to be heavily used by malicious codes. The discussion here considers simpler (and stealthier) forms of this approach that can nevertheless pose problems for concolic analysis. We have observed these obfuscations, or simple variants of them, in existing malware.

5.3.1 Conditional Jump to Indirect Jump Transformation

In the x86 architecture, conditional logic of the form

if e **then** S

is usually realized as follows: first, the expression e is evaluated and the condition code flags set; then, depending on the predicate involved in e, the appropriate combination of flags is used in a conditional branch instruction:

```
FLAGS := evaluate e
jcc  A_S
```

where *cc* represents the particular combination of flags corresponding to the predicate in e, and A_S is the address of the code for S. The architecture of `FLAGS` register on x86 processors is shown in Figure 5.2. However, this same effect can be realized by using the condition code flags resulting from the evaluation of e to compute the target address:

```
FLAGS := evaluate e
r := f(FLAGS)       /* compute target address */
jmp r
```

In this case, the function f uses the condition code flag values to compute the target address; in particular, when the flag values indicate that the predicate e is true, the address computed by f(`FLAGS`) is A_S. A key difference between these two approaches

is that in the first case, the use of a conditional branch instruction makes explicit the two possible control flow targets that are possible. This is not the case, however, for the indirect jump in the second case. As a result, the indirect jump is harder to analyze symbolically than the first: Schwartz *et al.* refer to this as the *symbolic jump problem* [114].

Obfuscators sometimes exploit this situation by transforming conditional branches to indirect jumps. This is illustrated by the following example.

Example 5.3.1 Consider the following code fragment:

```
1   r0 := input();
2   FLAGS := test(r0)   /* x86:  test */
3   push(FLAGS)         /* x86:  pushf */
4   r1 := pop()
5   r2 := and r1, 0x40
6   r3 := 0x500000
7   r4 := or r3, r2
8   jmp r4
```

Instructions 2–4 above check the input value and move the condition code flags into register r_1. After some bit manipulation (instruction 5), it is bitwise or'd with the value in register r_3 (instruction 7). The resulting value is then used as the target of an indirect jump (instruction 8).

What is actually going on here is that instruction 5 extracts the bit corresponding to the Zero Flag (`ZF`), in bit position 6, from r_1 into r_2. The result of the bitwise or operation (instruction 7) is therefore either `0x500040` (if `ZF` had the value 1 after instruction 2) or `500000` (if `ZF` was 0). The indirect jump at instruction 8 is therefore really a conditional jump to one of these two addresses depending on the value of `ZF` from instruction 2. ∎

While this example is couched in terms of the widely-used x86 architecture, the ideas are not x86-specific: e.g., the ARM architecture allows similar direct manipulation of condition code bits with its `MSR`/`MRS` instructions. Such obfuscations are particularly an issue with emulation-based obfuscation (Section 2.2.3). Several

commercial software protection tools are based on this approach [101, 129, 102, 138]; these tools are also used sometimes to protect malware code [53, 135].

There are several different ways in which such an interpreter can implement conditional statements in the input program, which all amount to setting the vip to one of two alternatives depending on the value of some predicate. VMProtect [138] uses arithmetic on the condition code flags to determine the address of the appropriate vip value. Since the flags are in general symbolic, this causes the interpreter's vip to become symbolic as well.

Symbolic execution of virtualized programs becomes challenging if the interpreter's vip becomes symbolic. The problem is that the constraint solving process used to identify such inputs has no way of distinguishing between alternative execution paths arising due to the interpreter running on a different byte code program, and those arising from a different input to the original byte code program. In effect, symbolic execution turns the interpreter into a generator of inputs it can interpret or accept [24], except that in this case the byte-code is not dependent on the input and so is not itself symbolic—it suffices to make the vip symbolic. If the vip becomes symbolic, the number of possible alternatives for the symbolic execution engine to consider at the VM's dispatch point is equal to the number of opcodes in the VM's instruction set. The resulting search space contains all the programs the interpreter is capable of running and exploring it exhaustively is impractical even for small interpreters. Furthermore, even if we hypothesize a successful exploration of the search space, i.e., discovering all of the interpreted programs executable by the interpreter, it is only the interpreter whose execution paths are fully explored, not the interpreted byte-code. Note that this is a more general situation than handling of symbolic memory addresses although the issue with symbolic addresses is still a problem with symbolic execution engines [114, 30].

It is not difficult to cause the vip to become symbolic: all that is needed is to make the vip input dependent at some point, e.g., by transforming control dependencies into direct data dependencies. Moreover, this attack against symbolic execution can

be used with arbitrary predicates, which makes it more flexible than that of Sharif *et al.* [118], which is restricted to equality predicates.

5.3.2 Conditional Jump to Conditional Jump Transformation

The previous section discussed how an obfuscator could use explicit arithmetic on the condition code flags to turn conditional jumps into indirect jumps that are harder to analyze symbolically. Here we discuss how similar arithmetic operations can be used to transform the predicate associated with a conditional jump to a completely different predicate, as illustrated by the following example.

Example 5.3.2 Consider the following code fragment:

```
1   r0 := input();
2   FLAGS := test(r0)   /* x86:  test */
3   push(FLAGS)         /* x86:  pushf */
4   r1 := pop()
5   r2 := r1 >> 4       /* x86:  shr */
6   push(r2)
7   FLAGS := pop()      /* x86:  popf */
8   jpe L    /* jpe:  jump if parity even */
```

Instructions 2–4 above check the input value and move the condition code flags into register r_1. This register is then right-shifted by four bits (instruction 5) and the resulting value is moved back into the condition code flags (instructions 6, 7), which is used to perform a conditional jump (instruction 8). The conditional branch instruction, `jpe`, is not a very common one: it stands for "*jump if parity is even*" and is taken if the parity flag is set. In reality, however, the bit that is actually being tested is not the parity flag, but rather the bit that was shifted into the parity flag's position by instruction 5—namely, the zero flag. In other words, the condition that

is really being tested is whether the input value read into r_0 by instruction 1 is zero or not; however this is being done using a very different predicate.[2] ∎

The approach illustrated above can also be used to construct opaque predicates, i.e., conditional jumps that are either always taken or always not taken.

The issue described here is orthogonal to that of transforming an input value to a different value and applying a different predicate to the transformed value [118], since it involves using architecture-specific knowledge to transform meta-information. The commercial obfuscation tool EXECryptor [129] uses this approach to produce long sequences of this kind of bit-shuffling operations to hamper analysis.

Note that while concolic analyses have to map conditional jump instructions to predicates on values, reasoning about such bit-level manipulations of condition code flags additionally requires fine-grained taint-tracking. Conventional byte- or word-level taint tracking can lead to significant over-tainting in the presence of the sorts of bit manipulation illustrated above. Over-tainting occurs when imprecision in taint propagation causes the taint analysis to determine values to be tainted, and deemed to be symbolic, when that are in fact independent of the inputs appear to be dependent on them. Conditional branches on expressions involving such spurious symbolic variables are then treated as candidates for generating inputs that can lead to alternative execution paths, resulting in additional computational load on the constraint solver and degrading the overall performance of the system. In the worst case, a very large number of such spurious symbolic variables and associated conditional branches can use up so much resources that the system crashes or is unable to make progress on identifying inputs that would in fact cause the program to take alternative paths.

[2] `jpe` instruction in this example is used to highlight how different the predicate of the jump instruction can be from the actual condition on the input value. In practice one would expect the obfuscated code to use more common instructions.

```
    call  get_input()
    cmp   eax, TRIGGER
    jz    L
    call  abort()
L:  call  payload()
```

(*a*) Original code

```
    call  get_input()
    sub   eax, TRIGGER
    add   al, 0xEB
    lea   ebx, L1
    lea   ecx, L2
    sub   ecx, ebx
    mov   ah, cl
    mov   word [L1], ax
L1: nop
    nop
    call   abort()
L2: call   payload()
```

(*b*) Obfuscated code executed with non-trigger input

```
    call  get_input()
    sub   eax, trigger
    add   al, 0xEB
    lea   ebx, L1
    lea   ecx, L2
    sub   ecx, ebx
    mov   ah, cl
    mov   word [L1], ax
pc→ L1: jmp    L2
    call  abort()
L2: call  payload()
```

(*c*) Obfuscated code executed with trigger input

Figure 5.3: An example of symbolic instruction

5.3.3 Symbolic Instruction

symbolic instruction can be seen as an extension of a code obfuscation technique commonly used in malware, where the program modifies the code region ahead of the program counter, such that execution then falls into the modified code. symbolic instruction extends this idea to carry out the code modification using an input-derived value. The idea is that, if the input meets some appropriate condition, the modified bytes encode a jump instruction to some desired address; otherwise, the modified bytes encode some non-jump instructions. The effect is that execution branches to the target of the jump if and only if the input satisfies that condition. We define symbolic instruction below:

Definition 5.3.1 Symbolic Instruction: Given a program P with an input n, instruction I is a *symbolic instruction* if any part of I (opcode or operand) is modified at runtime using input n, such that for some value of n the modified instruction I' changes the control flow of P at instruction I. ∎

The key characteristic of symbolic instruction is that this is done without executing an explicit comparison or conditional jump on an input-derived symbolic value, which means that if the input condition is not satisfied, standard concolic analysis does

not see a conditional jump in the instruction stream and therefore does not consider the possibility of an alternate execution path.

Figure 5.3 shows an example of this approach. Figure 5.3(a) shows the original code where the behavior of the code is based on an input value. The code in 5.3(b) shows the obfuscated code statically where the obfuscation tries to hide the control transfer based on some trigger value. The code uses the input value to overwrite an instruction in the code in such a way that the execution results in the control being transferred to a code when the value of the input is the desired one. For other inputs either the instruction constructed is an illegal instruction or the control does not reach the hidden code. 5.3(c) shows the code where the input triggers the execution of the hidden code. With the input value being the desired value, the computed instruction is a jump which transfers the control to the label `L2`.

Symbolic instruction is a straightforward variation on an obfuscation technique that has long been used in malware: namely, to modify a few bytes ahead of the execution and have execution fall into the modified bytes. This is illustrated in Figure 5.4, which shows instructions from the *NetSky.aa* worm (first encountered in 2004). Figure 5.4(a) shows the first few instructions from a static disassembly of the code. When this code is executed, the `add` instructions at addresses `0x403e64` and `0x403e68` modify five bytes at address `0x403e6e`; execution then falls into the newly created instructions, thereby installing an exception handler at address `0x5cbc32`, which is then used to field the exception raised via a (deliberate) null-pointer dereference by the `mov` instruction at address `00403e84`. The main difference that the symbolic instruction technique brings to bear is that the bytes used to create the modified code are input-dependent.

Symbolic instruction can be used to conceal trigger-based behaviors, i.e., behaviors that are exhibited only under specific external or environmental triggers [21]. Existing proposals for detecting such latent behaviors using symbolic execution assume that the control transfers associated with these triggers rely on conditional branches [21, 42]. Symbolic instruction can evade such approaches by conditionally creating an unconditional jump instruction, e.g., by using input values to create the modified

```
00403e5f  mov eax, 0x403e6e
00403e64  add byte [eax], 0x28
00403e67  inc eax
00403e68  add dword [eax], 0x1234567
00403e6e  nop
00403e6f  retf
00403e70  jbe 0x4c
00403e72  call dword near [eax+0x64]
00403e74  push dword [0x0]
00403e7b  mov [fs:0x0], esp
00403e82  xor eax, eax
00403e84  mov [eax], ecx
```

(*a*) Static disassembly

```
00403e5f  mov eax, 0x403e6e
00403e64  add byte [eax], 0x28
00403e67  inc eax
00403e68  add dword [eax], 0x1234567
00403e6e  mov eax, 0x5cbc32
00403e73  push eax
00403e74  push dword [fs:0x0]
00403e7b  mov [fs:0x0], esp
00403e82  xor eax, eax
00403e84  mov [eax], ecx
```

(*b*) Runtime code sequence

Figure 5.4: Self-modifying code in *NetSky.aa* worm

instruction(s) in such a way that only the desired input (trigger) will result in the desired (malicious) execution, but for the rest of values the malicious part does not get exposed to the analysis. Since the resulting control transfer does not use a conditional branch instruction, existing approaches will not consider it as a candidate for symbolic analysis to identify inputs that can trigger alternative execution paths.

5.4 Handling Obfuscations for Symbolic Execution

Since the primary focus of this work is to improve concolic analysis of obfuscated code, we do not intend to address other potential problems with concolic analyses, e.g., path selection algorithms or dealing with system calls with symbolic arguments. For these problems, we adapt current state-of-the-art solutions that are already used in symbolic execution engines [34, 30]. The key idea behind our approach is to use a combination of bit-level architecture-aware taint analysis discussed in Section 2.3, bit-level constraints on symbolic values derived from condition-code flags, and architecture-aware constraint generation to reason about and identify the path constraints. Solving the computed path constraints reveals inputs to the code which triggers alternative execution paths to improve the code coverage.

5.4.1 Bit-Level Dynamic Taint Analysis

We use the dynamic bit-level taint analysis approach described in Chapter 2. Our experiences with obfuscations, e.g., those that use bit manipulations to obfuscate conditional jumps, as discussed in Sections 5.3.1 and 5.3.2, indicate that the ability to track taint at the level of individual bits can be crucial for dealing with obfuscated code. We therefore carry out taint propagation at bit level granularity. Additionally, since concolic analysis involves reasoning about the conditions under which different execution paths may be taken, we keep track of taint sources arising from condition code flags. This is done using taint tags or *labels*. For our concolic analysis, taint labels can be of two kinds:

1. A '*generic taint*' label that indicates that the taint originated from an input value rather than a condition code flag.

2. A triple $\langle ins, flag, polarity \rangle$ where *ins* refers to (a particular dynamic instance of) an instruction in an execution trace; *flag* encodes a condition code flag; and *polarity* indicates whether the bit that the taint label refers to has the same value as that of the original flag value it was derived from or whether it has been inverted.

Taint analysis is performed similar to what described in Chapter 2. Values obtained as inputs (e.g., set by system calls) are considered to have all of their bits tainted labeled with generic taint. For instructions that set condition code flags (which include most arithmetic and logical operations as well as the `test` and `cmp` instructions), if any input operands are tainted then taint is propagated to the flags along with the appropriate taint labels. Let $\ell[i]$ denote the i^{th} bit position of an operand (i.e., location or value) ℓ. Taint propagation for an instruction I in the trace is done as follows:

- If none of the source operands of I are tainted, or if the value of a destination bit $dst[i]$ is fixed and independent of the values of the source operands, then $dst[i]$ is marked 'not tainted'. (In general, it is necessary to take implicit flows

into account in order to avoid under-tainting [29]. Existing approaches to incorporating implicit information flows into taint analyses [35, 74] can be adapted to our purposes. Since this is not the focus of our work, we do not discuss it further here.)

- Otherwise, if all of the source operands of I have the marking '*generic taint*' then:
  - each non-condition-code destination operand of I gets the taint marking '*generic taint*';
  - each condition code flag f affected by I gets the taint marking $\langle I, f, 1\rangle$.
- Otherwise, for each destination bit $dst[j]$ of I (including condition code flags):
  - if the value of $dst[j]$ can be determined from some particular source operand bit $src[k]$, then:
    - if $dst[j]$ has the same value as $src[k]$ then $dst[j]$ gets the same taint marking as $src[k]$;
    - otherwise $dst[j]$ gets the same taint marking as $src[k]$ but with the polarity reversed.
  - Otherwise: $dst[j]$ is marked *tainted* and its taint mark is determined as follows:
    - Each condition code flag f gets a new tag marking $\langle I, f, 1\rangle$.
    - Each non-condition-code bit gets the mark '*generic taint*'.

We keep track of taint labels in terms of bit values—namely, a condition code flag along with its polarity—to simplify reasoning about code obfuscations that manipulate these bits. However, a taint label $\langle I, \mathit{flag}, \mathit{polarity}\rangle$ also corresponds to a predicate on one or more values in the computation. Since a particular flag may be set differently by different instruction operations, the specifics of the predicate will depend on the instruction I that set the flag. For example, the `cmp` (compare) and `sub` (subtract) instructions set `CF` if there is a borrow in the result; some forms of

the integer multiply instruction `imul` set `CF` if the result of multiplication has been truncated; and some bit-rotate instructions (e.g., `rcl`, `rcr`) include `CF` in the rotation and so set it depending on the bit that is moved into it due to the rotation. Given a taint label $t \equiv \langle I, f, p \rangle$, we can use the semantics of the instruction I, together with the flag f and the polarity p, to determine the predicate associated with the taint label t. We refer to this predicate as the *flag condition* for t, written $\mathsf{FlagCond}(t)$.

Taint labels allow us to improve the precision of the taint analysis by identifying operations on bits that originate from the same value and build the appropriate predicate for symbolic execution. As an example, consider the following instruction sequence:

```
1   r0 := input();
2   FLAGS := test(r0)   /* x86:  test */
3   push(FLAGS)         /* x86:  pushf */
4   r1 := pop()
5   r2 := !r1           /* x86:  neg */
6   r3 := r1 ^ r2       /* x86:  xor */
```

In this example, instructions 2–4 check the input value and move the condition code flags into register r_1 (in a real-life example the input might be the result of timing the execution of a fragment of code, and the check might determine whether the value falls within a range indicating that the program is not running within an emulator). Instructions 5–7 then carry out a variety of bit manipulations on the flag bits, e.g., as performed in obfuscation tools such as VMProtect and EXECryptor. In this example, our taint analysis will determine that the bitwise negation operation in instruction 5 flips the bits of r_1 into r_2, which means that, after instruction 5, the low bit of r_2 is different from that of r_1, and therefore that the low bit of r_3 after the `xor` operation in instruction 6 is necessarily 1. Since the value of the low bit of r_3 is constant and thus independent of the input, it will be marked as untainted.

5.4.2 Handling Obfuscated Jumps

Given a conditional or indirect jump instruction I that is controlled by a tainted (i.e., symbolic) value, we compute the predicate corresponding to it as follows.

1. Identify the condition code flags that control I:

   - For a conditional jump this is obtained from the jump condition of the instruction.
   - For an unconditional jump this is obtained from the tainted bits in the target address whose taint marking is not '*generic taint*'.

   Denote this set of flags by $C(I)$.

   If $C(I) = \emptyset$ then I is an input-dependent indirect jump that is not dependent on any conditional jump in the code. We currently do not handle this case since handling symbolic indirect jump is challenging in general [114] and is not in the scope of this study.

2. The predicate corresponding to I is then given by

$$\mathsf{InstrPred}(I) \;=\; \bigwedge_{t \in C(I)} \mathsf{FlagCond}(t)$$

   where $\mathsf{FlagCond}(t)$ is the condition associated with the instruction and condition code flag referred to by t (see the previous section).

Let the path constraint up to the instruction prior to I be π, then the path constraint up to and including I is given by $\pi \wedge \mathsf{InstrPred}(I)$.

Example 5.4.1 The instruction sequence below is semantically identical to that of Example 6.3.1. but expressed in x86 syntax to illustrate how the analysis works.

```
1  call get_input
2  test eax, eax
3  pushfd
```

```
4  pop ebx
5  and ebx, $0x40
6  mov ecx, $0x500000
7  or ebx, ecx
8  jmp ebx
```

The taint propagation goes as follows.

After instruction 1, each bit in `eax` has the taint marking *generic taint*.

After instruction 2, the condition code flags in the `EFLAGS` register are tainted as follows. Bit positions 0, 2, 6, 7, and 11, corresponding to the flags Carry (`CF`), Parity (`PF`), Zero (`ZF`), Sign (`SF`), and Overflow (`OF`), gets the taint markings $\langle 2, \texttt{CF}, 1\rangle$, $\langle 2, \texttt{PF}, 1\rangle$, $\langle 2, \texttt{ZF}, 1\rangle$, $\langle 2, \texttt{SF}, 1\rangle$, and $\langle 2, \texttt{OF}, 1\rangle$ respectively (here, the instruction value '2' refers to the position of the instruction that set the flag, and the polarity value 1 indicates that the bit has not been inverted).

The data movement instructions 3 and 4 simply copy the taint marks of their source to their destination. Thus, after instruction 3, the corresponding bits of the top word on the stack get these taint markings, and similarly for the register `ebx` after instruction 4. The resulting taint markings of `ebx` are: $\texttt{ebx}[0] \mapsto \langle 2, \texttt{CF}, 1\rangle$; $\texttt{ebx}[2] \mapsto \langle 2, \texttt{PF}, 1\rangle$; $\texttt{ebx}[6] \mapsto \langle 2, \texttt{ZF}, 1\rangle$; $\texttt{ebx}[7] \mapsto \langle 2, \texttt{SF}, 1\rangle$; and $\texttt{ebx}[11] \mapsto \langle 2, \texttt{OF}, 1\rangle$.

After instruction 5, the only bit of `ebx` that is tainted is `ebx`[6], which has the marking $\texttt{ebx}[6] \mapsto \langle 2, \texttt{ZF}, 1\rangle$. After instruction 7, this bit position remains the only tainted bit in `ebx`, with the same taint marking, $\langle 2, \texttt{ZF}, 1\rangle$. From the semantics of instruction 2, namely, `test eax, eax`, the flag condition for this taint marking is that register `eax` is 0. Thus, the instruction predicate for the indirect jump at instruction 8 is that `eax` has the value 0 at instruction 2. ∎

In this case, it is possible to reason about the possible values of the tainted bits flowing into the indirect jump, and thereby identify the set of possible targets of the jump. From the perspective of concolic analysis to generate alternative inputs and improve code coverage, this is not really necessary since it is enough to identify the instruction predicate InstrPred() for the indirect jump. The ability to explicitly identify the other possible targets of such obfuscated jumps can be useful, however,

for other related analyses of obfuscated code, such as incremental disassembly [95] and deobfuscation (Chapter 4).

The following example shows how this approach can deal with obfuscated conditional jumps.

Example 5.4.2 The code fragment below rephrases Example 5.3.2 in x86 syntax.

```
1  call get_input
2  test eax, eax
3  pushfd
4  pop ebx
5  shr ebx, $4
6  push ebx
7  popfd
8  jpe L
```

Instructions 1–4 of this example are the same as in Example 5.4.1 and their analysis is similar to that shown above. After instruction 4, the taint markings of `ebx` are: `ebx`$[0] \mapsto \langle 2, \texttt{CF}, 1\rangle$; `ebx`$[2] \mapsto \langle 2, \texttt{PF}, 1\rangle$; `ebx`$[6] \mapsto \langle 2, \texttt{ZF}, 1\rangle$; `ebx`$[7] \mapsto \langle 2, \texttt{SF}, 1\rangle$; and `ebx`$[11] \mapsto \langle 2, \texttt{OF}, 1\rangle$.

After instruction 5 (`shr`, shift right), the taint markings for register `ebx` are updated to account for the shift. Thus, `ebx`[2] (i.e., bit position 2) gets the taint marking $\langle 2, \texttt{ZF}, 1\rangle$; `ebx`[3] gets $\langle 2, \texttt{SF}, 1\rangle$; and `ebx`[7] gets $\langle 2, \texttt{OF}, 1\rangle$.

The data movement instructions 6 and 7 then copy the resulting bits from `ebx` to `EFLAGS`, and their taint is propagated correspondingly. In particular, after instruction 7, the condition code flag at `EFLAGS`[2], namely, `PF`, gets the taint marking for the corresponding position of `ebx`, i.e., $\langle 2, \texttt{ZF}, 1\rangle$.

When the conditional jump in instruction 8 is encountered, the semantics of the `jpe` instruction specify that it is taken if the `PF` flag is 1. The taint mark for this flag is $\langle 2, \texttt{ZF}, 1\rangle$, i.e., (since the polarity on the taint mark is 1) that `ZF` $= 1$ from instruction 2. From the semantics of instruction 2, namely, `test eax, eax`, the flag condition for this taint marking is that register `eax` is 0.

Thus, the instruction predicate for the conditional jump at instruction 8 is that `eax` has the value 0 at instruction 2. ∎

5.4.3 Handling Symbolic Instruction

We detect symbolic instruction when an instruction writes a tainted value to a memory location that forms part of a subsequently executed instruction I. The way in which such a write is handled depends on which portions of the instruction I become tainted as a result:

- If the opcode byte is tainted, then a different input can cause a different instruction to be written into I's location and subsequently executed. While the total number of other possible opcodes is quite large, for the purposes of reasoning about input-dependent conditional jumps we focus on control transfer instructions. To this end, we construct an instruction predicate that gives, as alternatives for the opcode byte(s) of I, all of the binary opcodes for control transfer instructions (direct and indirect unconditional jumps, conditional jumps, and procedure calls and returns).

  In this case it is also possible for part of all of the operand bytes of I (and possibly the instruction following I) to be overwritten. However, our current implementation focuses on identifying alternative inputs that can cause a control transfer instruction to be created in place of I since this is fundamental to identifying and exploring alternative execution paths.

- If the opcode byte is not tainted but one or more other bytes of the instruction are overwritten with tainted bits, then the corresponding operands are flagged as tainted as the taint analysis proceeds from that point.

  For example, suppose that an operand is overwritten to become the immediate value `1`. If any of the bits involved in this are tainted, then in effect the computation uses the instruction overwriting to conditionally incorporate an input-dependent value into the computation, so the input-dependent value should be considered tainted.

5.5 Evaluations

We evaluated the ideas presented in this chapter using a prototype system we have implemented. Our system (called ConcoLynx) uses Pin [88] to collect execution traces;[3] these traces are then post-processed to propagate taint from symbolic inputs. We ran our experiments on a Linux machine running Ubuntu operating system with an Intel Core i7 (2.6 GHz) CPU with 8 cores and 6 gigabytes of memory.

We used two sets of programs for our evaluations:

- The first set consists of three small programs: *simple-if* (shown in Figure 5.1(a)); *bin-search*, a binary search program; and *bubble-sort*. The *simple-if* program takes a single input which is marked symbolic in our analysis. We ran *bin-search* on an array of size eight; and only the number to be searched for within this array is marked as symbolic. The *bubble-sort* program was run on an array of size three where all the elements were marked symbolic.

- The second set consists of four malicious programs whose source code we obtained from VX Heavens [139]. Each of these programs demonstrates trigger-based behaviors based on some system calls: *mydoom* and *netsky_ae* both check system time to execute their payload if the current time meets the trigger condition; *assiral* checks whether it is being debugged by calling the API function `IsDebuggerPresent()` and then takes different execution paths based on the result; and *clibo* checks Windows registry keys to check whether it is running for the first time in the system.

The toy programs in the first set were deliberately chosen to have a small amount of simple but nontrivial control flow, so as to make it easier to separate out the performance and precision effects of code obfuscation on concolic analysis. The small size and simple logical structure of these programs were intended to provide a sort of lower bound on expectations for concolic analyses. The programs in the second

[3]This choice of tracing tool is not fundamental to our approach so any other tracing tool can be adapted by our technique.

set were chosen as representative samples of trigger-based behavior in malware. We used source code because of the requirements of the obfuscation tools listed below. For S2E, the inputs to the programs were annotated with S2E's APIs to introduce symbolic inputs to the programs. For Vine, the desired function calls were hooked to introduce taint to programs. The goal was not so much to examine the latest in trigger-based evasion behaviors in malware, but rather to study the impact of code obfuscation on symbolic execution under carefully controlled experimental conditions.

For each of the programs listed above, we examined the behavior of five executables: the original program together with four obfuscated versions obtained using four commercial obfuscation tools: Code Virtualizer [101], EXECryptor [129], VMProtect [138], and Themida [102]. These obfuscation tools create obfuscated binaries for Windows operating system, so for collecting an execution trace, we ran the obfuscated binaries along with Pin tool on a Windows XP service pack 3 operating system running on VMware workstation. Additionally, we built versions of the first set of test programs to incorporate symbolic instruction into the program's execution.

We compared ConcoLynx with two symbolic execution systems, S2E [34] and Vine [124]. S2E is based on KLEE and is built on top of the LLVM compiler and can discover program states using symbolic execution and virtualization. Vine is a static analysis tool based on Bitblaze and can analyze traces collected with TEMU [124] where the traces are taint annotated.

We performed two types of analysis. The first experiment looked at the effect of obfuscation on the accuracy and the efficacy of the tools while the second examined the practicality of the tools by looking at the cost that obfuscation imposes on the symbolic analysis in terms of the time of analysis and the number of queries which were submitted to the constraint solver. In our experiments with S2E, we used the concolic execution configuration with two path selection strategies available in S2E: depth-first and random state search.

System	Program	Obfuscation Tool			
		CV	*EC*	*VM*	*TH*
ConcoLynx	*simple-if*	✓	✓	✓	✓
	bin-search	✓	✓	✓	✓
	bubble-sort	✓	✓	✓	✓
	assiral	✓	✓	✓	–
	clibo	✓	✓	✓	✓
	mydoom	✓	✓	✓	✓
	netsky_ae	✓	✓	✓	✓
Vine	*simple-if*	Err	StpErr	Err	Err
	bin-search	Err	StpErr	Err	Err
	bubble-sort	Err	StpErr	Err	Err
	assiral	Err	StpErr	Err	Err
	clibo	Err	StpErr	Err	Err
	mydoom	Err	StpErr	Err	Err
	netsky_ae	Err	StpErr	Err	Err
S2E (DFS)	*simple-if*	✓	✓	×	✓
	bin-search	✓	✓	×	×
	bubble-sort	×	✓	×	×
	assiral	✓	✓	×	×
	clibo	✓	✓	×	×
	mydoom	✓	✓	×	×
	netsky_ae	✓	✓	×	✓
S2E (random)	*simple-if*	✓	✓	×	×
	bin-search	×	✓	×	×
	bubble-sort	×	✓	×	×
	assiral	✓	✓	×	×
	clibo	✓	✓	×	×
	mydoom	✓	✓	×	×
	netsky_ae	✓	✓	×	✓

Key: CV: Code Virtualizer; EC: EXECryptor; VM: VMProtect; TH: Themida

✓: tool produced at least one input

×: timeout or fail to produce any results

StpErr: produced constraints crashed STP

Err: runtime error

Table 5.1: Efficacy of analysis: code coverage

5.5.1 Efficacy

5.5.1.1 Code Coverage

These experiments evaluate the extent to which symbolic execution makes it possible to identify and explore different execution paths in obfuscated programs. Table 5.1 shows, for each program, whether the symbolic analysis was able to generate any alternative input that would cause the program to take a different execution path than it did in the analysis.

For the programs analyzed with our tool we have manually verified that for each branch point in the program the alternative counter example would lead us to another execution path in the program.

In our experiments, Vine generally failed to produce path constraints on programs obfuscated using Code Virtualizer, VMProtect and Themida: for these programs, it gave error messages and exited and the ERR in Table 5.1 corresponds to this behavior. Vine was able to produce path constraints for programs obfuscated using EXECryptor but even then the constraints created by Vine, crashed the STP that is shipped with the Vine tool.

As can be seen from the table, S2E was not able to produce any test case for programs obfuscated with VMProtect and for most of the programs obfuscated with Themida. Moreover, while S2E was able to generate some test cases for many of the programs protected with Code Virtualizer and EXECryptor, the test cases generated for programs obfuscated using EXECryptor cases were redundant, meaning that S2E generated multiple test cases that all resulted in the same execution path being taken. The generation of such redundant inputs can be a problem because it can use up time and computational resources and thereby slow down the overall progress of analysis of a potentially malicious code sample.

Overall, the results indicate that anti-symbolic obfuscations can significantly hinder multi-path exploration using symbolic analysis.

5.5.1.2 Symbolic Instruction

We built versions of our synthetic programs to incorporate symbolic instruction into their logic. S2E did not detect the symbolic instruction and so did not generate any inputs that would cause different symbolic instruction to be generated while our tool was able to detect the symbolic instructions and generate appropriate constraints thus generated inputs that would trigger other execution paths in the synthetic programs.

5.5.2 Cost

Tables 5.2 and 5.3 presents normalized data of the analysis time and the number of path constraint queries submitted to the underlying SMT solver by S2E and ConcoLynx (our tool) for test-cases obfuscated using obfuscation tools Code Virtualizer and EXECryptor in Table 5.2 and Themida and VMProtect in 5.3. Since we were not able to get any useful results out of Vine, it is omitted from the costs table and only the performance data for S2E is given. Our system post-processes an execution trace of the program to generate path constraints, while S2E saves program states whenever it reaches a possible branch point in the program's execution. In order to be able to provide a fair comparison of the systems, the numbers presented here are normalized with respect to unobfuscated programs for each tool. For the *assiral* program obfuscated with Themida, we were not able to execute the binary on our tracing facility: the program crashed while generating the trace so we were unable to apply our tool to this program. We chose 12 hours timeout for our toy programs and 6 hours timeout for malicious codes.

The data in Tables 5.2 and 5.3 lead to the following conclusions:

1. ConcoLynx is able to identify the branch points of the obfuscated programs.

   The total number of queries submitted to the constraint solver by our system is seen to go up for programs obfuscated using EXECryptor: for these programs, we have manually verified that the obfuscation tool inserts additional conditional code that uses tainted values, for example the obfuscation tool inserts additional

System	Program	*Orig.*	*No. queries (normalized)*		*Analysis time (normalized)*	
			CV	*EC*	*CV*	*EC*
ConcoLynx	*simple-if*	1.0	1.0	35	5.6	1.9
	bin-search	1.0	1.0	13	9.7	6.6
	bubble-sort	1.0	1.0	14.5	3.9	12.9
	assiral	1.0	1.0	11	8.8	3.5
	clibo	1.0	1.0	17.6	24.8	3.3
	mydoom	1.0	1.0	8.6	5.7	24.6
	netsky_ae	1.0	1.0	8.6	7.7	3.6
S2E (DFS)	*simple-if*	1.0	19	52.8	4.7	8.5
	bin-search	1.0	7.9	26.7	TIMEOUT	12.6
	bubble-sort	1.0	0.1	155.6	154.6	30.7
	assiral	1.0	15.1	12.2	2.1	2.3
	clibo	1.0	10.8	14.7	2.7	3.5
	mydoom	1.0	17.2	23.2	4.1	5
	netsky_ae	1.0	24.5	22.1	3.6	4.2
S2E (random)	*simple-if*	1.0	19	53.2	4.2	8.5
	bin-search	1.0	24.3	26.6	TIMEOUT	14.8
	bubble-sort	1.0	57.7	155.5	67.5	74.4
	assiral	1.0	15.5	12.25	1.3	1.3
	clibo	1.0	12.1	14.7	3.2	3.9
	mydoom	1.0	17.2	23.7	2.7	3.1
	netsky_ae	1.0	24.5	22.2	2.5	2.5

Key: CV: Code Virtualizer; EC: EXECryptor;

Table 5.2: Cost analysis of ConcoLynx compared to S2E for obfuscated programs, normalized to the cost of unobfuscated programs: Code Vritualizer and EXECryptor

code that checks the sign of the result of an arithmetic operation that is not checked in the original program, making the symbolic engine produce and send more queries to SMT solver.

The increase in analysis time for the obfuscated code ranges from a low of about 2× for Code Virtualizer to a maximum of about 340× for one Themida-obfuscated program. This increase is due primarily to the larger number of instructions executed by the obfuscated programs.

System	Program	*Orig.*	*No. queries (normalized)*		*Analysis time (normalized)*	
			VM	*TH*	*VM*	*TH*
ConcoLynx	*simple-if*	1.0	1.0	1.0	14.6	112.7
	bin-search	1.0	1.0	1.0	19.1	339.3
	bubble-sort	1.0	1.0	1.0	38.5	152.7
	assiral	1.0	1.0	–	63.7	–
	clibo	1.0	1.0	1.0	123.9	10.8
	mydoom	1.0	1.0	1.0	63.9	55.9
	netsky_ae	1.0	1.0	1.0	20.3	27.5
S2E (DFS)	*simple-if*	1.0	2002.1	24.4	TIMEOUT	TIMEOUT
	bin-search	1.0	1.2	0	2	TIMEOUT
	bubble-sort	1.0	0	0	TIMEOUT	TIMEOUT
	assiral	1.0	123	0	8.2	TIMEOUT
	clibo	1.0	7	0	TIMEOUT	TIMEOUT
	mydoom	1.0	0	0	TIMEOUT	TIMEOUT
	netsky_ae	1.0	181.5	31	TIMEOUT	41.7
S2E (random)	*simple-if*	1.0	3584	49.2	847	TIMEOUT
	bin-search	1.0	17	40.4	16.5	TIMEOUT
	bubble-sort	1.0	62.5	14.8	TIMEOUT	TIMEOUT
	assiral	1.0	52	239	TIMEOUT	TIMEOUT
	clibo	1.0	37.2	96.5	TIMEOUT	TIMEOUT
	mydoom	1.0	35.6	3.9	TIMEOUT	TIMEOUT
	netsky_ae	1.0	31	74.6	TIMEOUT	40.6

Key: VM: VMProtect; TH: Themida

Table 5.3: Cost analysis of ConcoLynx compared to S2E for obfuscated programs, normalized to the cost of unobfuscated programs: VMProtect and Themida

2. S2E is able to successfully analyze most of the programs obfuscated using Code Virtualizer (except for a timeout on *bin-search*) and EXECryptor (except for a failure on *bin-search* with the Random search strategy). This result is encouraging. For VMProtect and Themida, however, S2E failed or timed out on most of the test programs.

Table 5.4 gives the actual amount of time of the analysis. In Table 5.4, T_0 is the time for each program to execute without tracing, T_1 shows the time that is needed to collect an execution trace for each program and T_2 and T_3 show the analysis

	Program	Time (milliseconds) Native Exec. Time (T_0)	Tracing (T_1)	Tainting Byte-level (T_2)	Tainting Bit-level (T_3)
CV	*simple-if*	16	66,764	12.15	180.88
	binary-search	16	71,573	72.03	465.50
	bubble-sort	16	70,686	64.53	996.72
	assiral	15	55,764	52.16	466.34
	clibo	15	35,804	83.98	855.32
	mydoom	16	93,472	23.43	160.42
	netsky_ae	15	24,288	41.77	411.36
EC	*simple-if*	15	116,187	03.74	35.48
	bin-search	16	120,967	76.36	635.38
	bubble-sort	16	122,284	202.04	1584.00
	assiral	16	104,752	50.31	789.77
	clibo	16	88,723	53.78	405.24
	mydoom	16	136,400	39.09	451.01
	netsky_ae	16	124,735	38.06	306.04
TH	*simple-if*	453	679,110	920.47	9632.42
	bin-search	469	904,851	956.01	6576.10
	bubble-sort	406	728,041	1381.13	11957.30
	clibo	125	957,996	187.09	1478.65
	mydoom	422	1,000,077	307.47	1820.35
	netsky_ae	423	768,828	176.60	1847.81
VM	*simple-if*	15	229,510	88.24	728.36
	binary-search	16	236,892	207.97	1587.92
	bubble-sort	16	236,729	329.33	3056.27
	assiral	32	430,615	610.79	1484.02
	clibo	47	492,316	587.11	1830.58
	mydoom	32	582,506	173.09	935.98
	netsky_ae	31	334,940	146.03	975.00

Key: CV: Code Virtualizer; EC: EXECryptor; TH: Themida; VM: VMProtect;

Table 5.4: Cost of analysis with comparison of byte-level and bit-level taint analyses

time of conducting standard byte-level taint analysis and bit-level taint analysis (see Section 5.4.1) respectively on each execution trace. Table 5.5 shows the overhead ratio for each of the above analyses using the absolute time values in Table 5.4: T_1/T_0 is the overhead of recording an execution trace of a program compared to the

	Program	Overhead Ratio T_1/T_0	T_2/T_0	T_3/T_0	T_3/T_2
CV	*simple-if*	4172.50	0.75	11.30	14.88
	binary-search	4473.12	4.50	29.09	6.46
	bubble-sort	4417.50	4.03	62.29	15.44
	assiral	3717.33	3.47	31.08	8.94
	clibo	2386.67	5.59	57.02	10.18
	mydoom	5841.88	1.46	10.02	6.84
	netsky_ae	1618.67	2.78	27.42	9.84
EC	*simple-if*	7745.33	0.24	2.36	9.48
	bin-search	7560.00	4.77	39.71	8.32
	bubble-sort	7642.50	12.62	99.00	7.84
	assiral	6546.88	3.14	49.36	15.69
	clibo	5545.00	3.36	25.32	7.53
	mydoom	8525.00	2.44	28.18	11.57
	netsky_ae	7795.93	2.37	19.12	8.04
TH	*simple-if*	1499.14	2.03	21.26	10.46
	bin-search	1929.32	2.03	14.02	6.87
	bubble-sort	1793.20	3.40	29.45	8.65
	clibo	7663.92	1.49	11.82	7.90
	mydoom	2369.67	0.72	4.31	5.92
	netsky_ae	1817.54	0.41	4.36	10.46
VM	*simple-if*	15300.70	5.88	48.55	8.25
	binary-search	14805.60	12.99	99.24	7.63
	bubble-sort	14795.00	20.58	191.01	9.28
	assiral	13456.60	19.08	46.37	2.42
	clibo	10474.70	12.49	38.94	3.11
	mydoom	18203.10	5.40	29.24	5.40
	netsky_ae	10804.50	4.71	31.45	6.67
Geometric Mean		5540.01	3.22	26.14	8.06

Key: CV: Code Virtualizer; EC: EXECryptor; TH: Themida; VM: VMProtect;
T_0: Native exec. time; T_1: Tracing time; T_2: Byte-level taint; T_3: Bit-level taint;

Table 5.5: Cost of analysis with comparison of byte-level and bit-level taint analyses: overhead ratios

native execution time. This overhead ranges from 1499× to 18203× with a geometric mean of 5540× slowdown. T_2/T_0 and T_3/T_0 show how much overhead different taint analyses algorithms impose compared to the runtime of the program. This overhead for byte-level taint analysis ranges between 0.2× and 21× with geometric mean

of 3.2× slowdown and for bit-level taint analysis ranges from 2.6× to 191× with geometric mean of 26.1× slowdown. Finally, T_3/T_2 refers to the overhead of bit-level taint over byte-level taint analysis which ranges between 3× to 15.7× with geometric mean of 8.06× slowdown.

The numbers shown in Table 5.4 suggest that although the taint approach used in ConcoLynx is more expensive than the standard byte-level taint approach, the overhead of tracing a program is significantly higher than the rest of the analyses and so the increased overhead imposed by our taint analysis is dominated by that of trace recording. Moreover, the runtime of our approach, including the overhead of bit-level taint analysis, is much better than the running-time of other tools (S2E and Vine) when dealing with obfuscated code which makes our approach more practical.

5.6 Related Work

There is a considerable body of research on symbolic and concolic execution: Schwartz *et al.* [114] give a survey. The analysis of malicious and/or obfuscated code forms a significant application of this technology [12, 13, 21, 92, 117, 42, 157]. While such techniques can be effective when obfuscation is not an issue (e.g., in environmentally triggered programs where the trigger code uses unobfuscated conditional branches), they fail in the face of obfuscations such as those discussed in this study. The general problems associated with such analyses, e.g., path explosion or symbolic jumps, are known [114], however most of the research literature do not address them explicitly. This is especially problematic for applications of symbolic and concolic analysis to malware code since these programs are often heavily obfuscated to avoid detection and/or hamper analysis. Furthermore, code obfuscation can raise its own challenges for symbolic and concolic analysis, e.g., the symbolic instruction problem, discussed earlier, which we have not seen discussed elsewhere in the research literature. Sharif *et al.* [118] and Wang *et al.* [142] discuss ways to hamper symbolic execution via computations that are difficult to invert.

Concolic analyses typically use taint analysis to distinguish between concrete and symbolic values. Related works on dynamic taint analysis is discussed in Section 2.5. There is a large body of literature on detection and analysis of obfuscated and malicious code (see, e.g., [80, 81, 136, 40]). None of these works consider ways in which code obfuscations can hamper symbolic analysis.

CHAPTER 6

Analysis of Exception-Based Control Transfers

6.1 Introduction

Dynamic taint analysis and symbolic execution find numerous applications in security-related program analyses. Given the numerous security-related applications of dynamic taint analysis and symbolic execution, it is important to understand any potential shortcomings of current techniques and devise solutions to mitigate their weaknesses; a failure to do so can potentially lead to flawed conclusions about the software we analyze, e.g., leave application vulnerabilities unidentified or malware execution paths unexplored. Previous studies have discussed some of the limitations of dynamic taint analysis [28, 113] and symbolic execution [118, 152]. We discussed some of the problems that arise for symbolic execution due to the code obfuscation in Chapter 5. These works all involve "normal" executions where the execution path is (at least in principle) available for inspection. This work focuses on a different kind of control flow construct, namely, exception-based control transfers, where the execution path is not explicit and which can be significantly harder to analyze than conventional control transfers.

It turns out that exception-based control transfers — which are well-known and have been widely used in malicious code —can be used to realize implicit information flows in ways that are not detected by any of the existing state-of-the-art dynamic analysis tools we tested including KLEE [22], S2E [34], FuzzBall [16] and angr [119]. The problem arises both for dynamic taint analysis and for symbolic execution. This is because currently, implicit information flow detection and symbolic execution techniques rely on information about the execution path and control flow of the program to reason about alternative execution paths. For instance, dynamic taint analysis approaches (e.g. [35, 74]) use a program's control flow graph (CFG) to

determine whether a dataflow is implicit, i.e., occurs through a control transfer. Similarly in symbolic execution, the analysis relies on syntactic characteristics such as instruction opcodes to identify jump instructions and thereby determine branch conditions and path constraints.

An attacker can exploit the exception mechanism to implement arbitrary control transfers and thereby realize implicit information flows that are not detected using existing analysis techniques. This represents a significant shortcoming of such analyses. Furthermore, fixing the problem is not straightforward because identifying the potential control transfers—i.e., the locations where exceptions can be raised and those where they may be handled—is not always straightforward. We have observed such information flows in existing exploit code.

In this chapter we propose a solution to this problem that takes into account possible control transfers arising from exceptions where the application code contains user-defined handlers. In particular, this study makes the following contributions: 1) it demonstrates the problems that arise in current dynamic taint analysis and symbolic execution when dealing with exception-based control transfers; 2) it proposes a generic architecture-agnostic solution for both dynamic taint analysis and symbolic execution that addresses these shortcomings and hence improves their accuracy and robustness; and 3) it describes a prototype implementation that outperforms the state-of-the-art systems for symbolic execution and dynamic taint analysis when applied against real malicious codes as well as sample test programs that use exception-based control transfer obfuscations.

The rest of the chapter is organized as follows: section 6.2 discusses motivations and background on signals and control transfers due to exceptions. Section 6.3 demonstrates the problem in more detail and discusses our ideas to solve this problem followed by our prototype implementation details in Section 6.4. Our evaluation results are presented in Section 6.5 and previous and related works discussed in Section 6.6.

6.2 Background and Motivating Example

6.2.1 Signals

Signals (exceptions) serve to notify processes of an event or a software or hardware fault. There are two types of signals: synchronous and asynchronous. A synchronous signal typically results from an error in executing an instruction, e.g., "illegal instruction" (`SIGILL`) or "illegal memory access" (`SIGSEGV`). Asynchronous signals are delivered asynchronously and do not depend on the current execution context, e.g., `SIGSTOP`, which stops execution of the receiving process, or `SIGKILL`, which terminates it. Modern operating systems provide APIs to user applications to overwrite default exception handler routines with their own code where the control is transferred to in case the signal is delivered to the process. This gives the processes flexibility to provide their own routines to recover or clean up after unrecoverable faults. Windows uses a mechanism called *Structured Exception Handling (SEH)* and Linux uses *Signals*. Despite different implementation details, the high-level concepts are similar between different operating systems. As mentioned earlier, in general, there are two types of control transfers caused by unexpected events: *synchronous* and *asynchronous*.

Synchronous signals are triggered in the course of execution of instructions if the hardware or the operating system is not able to successfully execute the instruction. Such errors are usually severe enough that the execution of the process that caused the fault can not be continued. These types of exceptions are common among different operating systems. Asynchronous signals, on the other hand, are specific to the Unix/Linux operating systems and can interrupt the execution of a process at any instruction. These signals are primarily used for synchronization purposes between different or parent/child processes. Since each category is fundamentally different from the other, we propose different solutions for each of these types of events in the programs that are discussed in the rest of this section.

```
1:   int public, secret;
2:   void fpe_handler(){
3:       /* public is equal
                to secret here! */
4:   }
5:   void segv_handler(){
6:       public++;
7:       int c = secret-public;
8:       c = c/c;
9:   }
10:  int main(){
11:      signal(SIGFPE,  fpe_handler);
12:      signal(SIGSEGV, segv_handler);
13:      secret  = get_secret();
14:      char *p = NULL
15:      (*p)++;      // dereference p
16:  }
```

(*a*) Using synchronous signals for implicit flow

```
1:   void alarm_handler(){
2:       abort();
3:   }
4:   int main(){
5:       int i;
6:       signal(SIGALRM, alarm_handler);
7:       alarm(1);
8:       for (i=0; i < 1000; i++){
9:       /* this should finish in
                  less than 1 second*/
10:     }
11:     alarm(0);
12:     /* continue */
13:  }
```

(*b*) Using asynchronous signals to obfuscate control transfers

Figure 6.1: Branch obfuscation using signals

6.2.2 A Motivating Example

The mechanism in which users are able to define their own exception handlers can be exploited by attackers to obfuscate branch points in the code to confuse analysis. Figure 6.1 shows two sample programs using this technique to implicitly propagate sensitive data (Figure 6.1a), and to redirect the flow of execution on certain environments (Figure 6.1b).

6.2.2.1 Synchronous Signals

The code in 6.1(a) sets handlers for both `SIGSEGV` and `SIGFPE` signals where the former is raised by OS and sent to the process on memory access violations and the latter is raised in case of arithmetic errors such as divide by zero. After copying the sensitive data to `secret` value at line 13, the code intentionally dereferences a `NULL` pointer which raises a `SIGSEGV` and the control is transferred to `segv_handler`. After the `segv_handler` function returns, the control transfers to the instruction that originally generated the fault and since the problem has not been fixed, the operating system raises another `SIGSEGV` signal and so on. This continues until variable `c` equals zero meaning that variable `public` is equal to `secret` at line 7. At this point, the division operation line 8 generates an arithmetic fault (divide by zero) which then transfers the control to the `fpe_handler`. Doing so, the attacker is able to copy the `secret` value into another variable without any explicit data flow. Furthermore, the data flow is not implicit because the control transfer edges are not present in the static CFG of the program.

6.2.2.2 Asynchronous Signals

Figure 6.1(b) shows a timing technique that is quite frequently used by malicious codes simply as a timeout mechanism. The process sets a handler for `SIGALRM` signal at line 6 which aborts the process whenever the process receives an alarm signal. Line 7 registers a wake up call to the operating system telling it to send a signal to the calling process in one second. There is a second call to `alarm` system

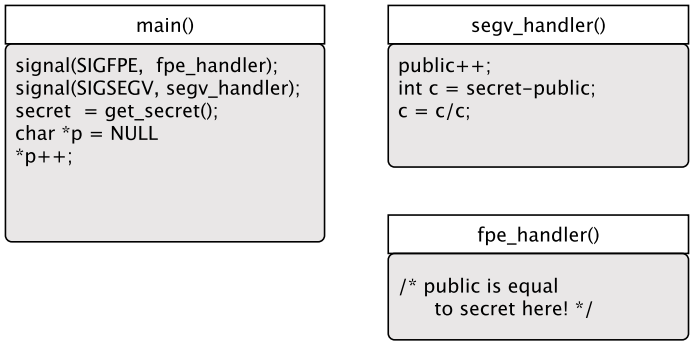
call which clears any pending alarm signals meaning that if the codes at lines 8-10 were executed before the signal arrives (in less than a second) the process continues execution; otherwise the delivered signal causes the process to abort. This behavior is similar to timing attacks used in malicious codes to detect analysis environments and evade detection [85, 109]. Similar to the previous example, there is no obvious control flow edge in the code to guide the symbolic execution to different execution paths. In fact, being able to automatically handle this case is even more subtle than the previous example since the `SIGALRM` is raised asynchronously by the operating system meaning that the *possible* control transfer edge to `alarm_handler` function could be anywhere between two calls to the `alarm` system call. The time in which the process receives the signal depends on many variables (with `CPU` cycles being the most important one) making the problem non-trivially hard.

6.3 Analyzing Exception-based Control Transfer Behavior

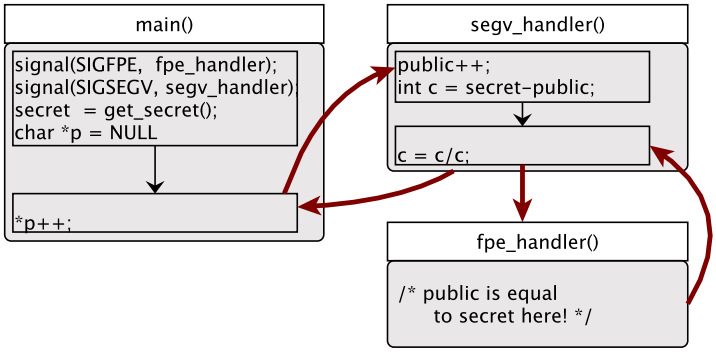
This section discusses our proposed solution to address the analysis problems arising from exception-based control transfers.

6.3.1 Synchronous Events

As discussed previously, to be able to detect implicit information flow propagation, dynamic taint analysis relies on the static CFG of the program to propagate tainted data along control transfer edges in the CFG. If a location (variable) x is defined by an instruction J that is control dependent on a control transfer instruction I, then x inherits the taint labels of the variable(s) used by the control transfer instruction I; Schwartz *et al.* refer to this as *control-flow taint* [114]. Correct handling of control-flow taint requires a static CFG that reflects the possible control transfers of the program. A key problem in dealing with exception-based control transfers is that CFGs constructed using conventional techniques typically do not account for exception-based control transfers, potentially leading to under-tainting.



(*a*) Static CFG of the code



(*b*) Static CFG augmented with control flow edges to signal handlers

Figure 6.2: Static and augmented static CFGs for the code in Figure 6.1(a)

6.3.1.1 Control Flow Graph Augmentation

A natural approach to fixing the problem of missing control flow edges in the static CFG of the program would be to insert additional edges corresponding to exception-based control transfers. However, a naive solution that statically includes every possible exception-based control transfer edge, from any instruction or statement that can possibly raise an exception, can become so large and cluttered as to be unusable.

Our solution charts a middle ground where we augment a conventional static CFG with additional control transfer edges that are added at runtime as the program executes and more information is available. We add control transfers from instructions that potentially can raise exceptions to the appropriate exception handler. Given a program P and input $\bar{x}$, we do this as follows. Here, S is a mapping that maps different exceptions to (the address of) the handlers that have been registered for them.

1. Static analysis phase:

Construct the static CFG for P.

2. Dynamic analysis phase:

1. Initialize the exception-handler mapping S to $\emptyset$.
2. Execute P on the given input $\bar{x}$. For each instruction I of this execution, do:
   (a) If I is a call to register an exception handler H at address A_H for an exception e, update the exception-handler mapping S to map the exception e to A_H.
   (b) Otherwise: for each exception e for which a handler is known to exist in S, use symbolic execution to determine whether there exists an input that can cause instruction I to raise exception e. If there exists such an input, add a control flow edge from I to the handler $S[e]$.

Identifying whether an instruction raises an exception or not requires runtime information, such as virtual memory pages associated to the process, that are not

available statically. The static CFG needs to be modified or augmented at runtime to contain necessary control flow information that are not available otherwise.

Figure 6.2 shows this transformation on the CFG of the sample code in Figure 6.1(a). Figure 6.2(a) is the original CFG that is built using standard definitions of control flow graph construction. It can be seen from the CFG, all three functions are isolated and there is no explicit control transfer edge between them because there is not a control flow statement that targets another function from anyone else. However, as shown in Figure 6.2(b), there are control transfers between different functions at runtime. The reason is that functions `segv_handler` and `fpe_handler` are registered as signal handlers for `SIGSEGV` and `SIGFPE` signals respectively that cause the control to be transferred to these functions on memory and arithmetic errors. The control flow edge from `main` to `segv_handler` is because a `NULL` pointer dereference in the `*p++` statement in the `main` function and the control returns back to the same statement after the called handler returns. Similarly, there is a control flow edge from the statement `c = c/c` in the segv_handler function to `fpe_handler` that is because of a divide by zero if variable `secret` is equal to `public` in `segv_handler` function. In fact, this logic implements a loop without any explicit control transfer statement where the code in function `segv_handler` is executed until the variable `secret` is copied to `public` and then exits to `fpe_handler` function.

After creating the augmented CFG, it needs to be analyzed to produce post-dominator information necessary for dynamic taint analysis. Basically, those non-control flow instructions that have an outgoing edge to an exception handler can be treated as a conditional jump for the purpose of control dependence and post-dominator analyses. The following section discusses how we identify the instructions that have a control flow edge to exception handlers.

6.3.1.2 Determining Control Flow Edges

As noted earlier, a challenge in identifying the possible control transfer edges from instructions to their intended exception or signal handlers is that this requires specific

runtime information that is not available statically. For synchronous signals, we use symbolic execution to determine the possibility of an exception during the course of an instruction's execution. This is done by constructing the path constraint along the execution path and sending appropriate queries to the SMT solver on instructions that can potentially raise exceptions. The idea is similar to those using symbolic execution for automatic fault detection [24] and exploit generation [22, 30]. The following example describes the idea more clearly.

Example 6.3.1 Consider the following code fragment:

```
1    r0 := input()
2    r1 := r0 − 10
3    r2 := r0 − 20
4    r3 := r1/r1
5    if r0 > 20:
6        r3 := r2/r2
```

Instruction 1 gets an input value and copies it to the register r_0 and instructions 2 and 3 calculate $r_0 - 10$ and $r_0 - 20$ in r_1 and r_2 respectively. Instruction 4 performs a division operation which can raise an exception if the divisor (r_1) is zero. Knowing the symbolic expression for r_1, expression $r_1 == 0$ is satisfiable for inputs equal to 10, instruction 4 can raise an exception and, if there is a handler registered for divide-by-zero exception, a control flow edge should be added from instruction 4 to the the handler function. Similarly for instruction 6, inputs of 20 can cause an exception to be raised but since the path constraint mandates $r_0 > 20$ and so $r_2 > 0$, the formula $r_2 == 0$ is not satisfiable and the instruction is immune to divide-by-zero exceptions. ∎

In the symbolic execution engine we have implemented three major synchronous exceptions that are commonly used to obfuscate normal control flow in malicious codes and/or in binary packer tools [56]. Our symbolic execution engine constructs appropriate constraints for synchronous exceptions that have a user exception handler

registered for it. As mentioned earlier, we are interested in finding exceptions that are triggered based on some tainted or symbolic condition or value. For this purpose, the symbolic execution first checks whether a particular instruction has the desired characteristics, and then builds a formula to represent the exception and sends it to the SMT solver to check its satisfiability. A few of the most important exceptions and how they are handled by the symbolic execution engine are as follows:

- `SIGSEGV` : This exception is caused by memory access violations such as accessing a memory location that is not available to the process. For memory pointers that are tainted, the symbolic execution engine constructs a constraint that checks whether the symbolic pointer can point anywhere that is not a legal memory address at that point in the program. If the constraint is satisfiable, the constraint solver finds an input that will cause the pointer to point to an illegal address that if fed into the program, transfers control to the signal handler for `SIGSEGV`

- `SIGFPE` : This exception is raised in case of arithmetic errors such as divide by zero. In case of a divide instruction and if the dividend is symbolic, the symbolic execution engine constructs a constraint that checks whether the dividend used in the instruction can be zero based on some inputs. If such an input exists, it will report it to the user.

- `SIGILL` : This exception is raised when an instruction opcode is not recognizable by the `CPU`. If in the execution of the program, an instruction is being overwritten with a symbolic value, similar to the symbolic instruction described in Section 5.3.3, the symbolic execution engine constructs a constraint to check whether the opcode that is being written can be illegal based on some input.

Using symbolic execution to identify exception-based control transfers has two advantages. First, the analysis can find alternative inputs to trigger alternative execution paths in the code other than those paths that existed because of normal control flow structure such as condition jumps. This is actually beneficial when

analyzing malicious or obfuscated code that want to hide their control flow through exception-based control transfers. Secondly, the alternative paths that are found can be used towards implicit information flow detection that otherwise was not possible to detect because of missing control transfer edges in the static CFG. It helps identifying and extracting execution paths that are hidden, and also identifies control flow edges that are missing in the static CFG which improves dynamic taint analysis. As discussed in Section 4.2.4, implicit flows are important in determining the input-to-output logic of the computation and so identifying exception-based control transfers improve the deobfuscation results.

6.3.2 Asynchronous Events

Asynchronous signals are those that are not the direct result of the execution of an instruction, but rather are delivered to the process from some external source. Asynchronous signals are more challenging than synchronous ones. This might not be an issue for symbolic execution since a control flow transfer caused by an asynchronous signal at point A in the code may result in the same path constraints as if the control transfer was to happen at point B. The same path constraints being computed results in the same alternative inputs and thus alternative execution paths being determined no matter at which point in the program signal arrives. The situation is however different for dynamic taint analysis since the analysis is sensitive to the data flows that might change by changing in the execution path. In other words, data flow equations for dynamic taint analysis may differ depending on what execution path was taken. To address this challenge, we need to take into account different possibilities of the program state when a signal is to be delivered to the program.

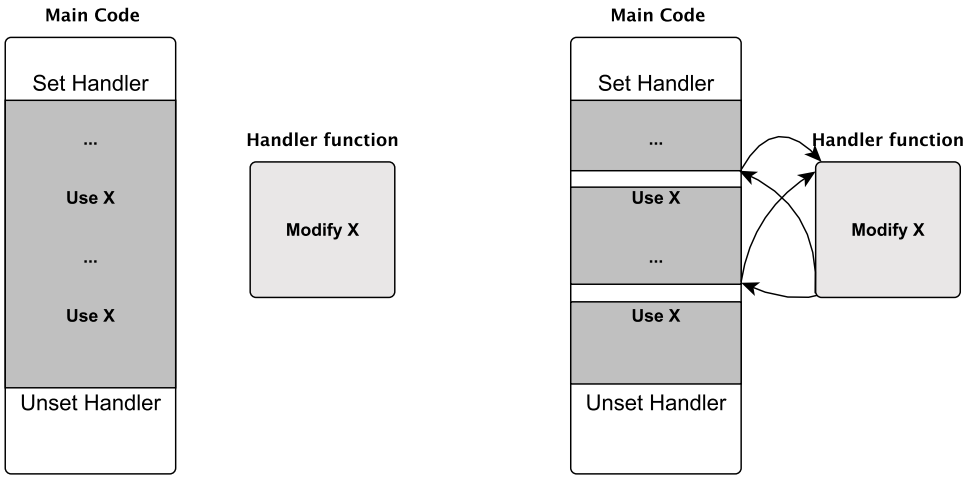
Similarly, for the unexpected control transfers caused by asynchronous signals we use the augmentation technique that was used for synchronous signals. The only difference here is that since there is no fixed point in the code where the control diverges from the main execution thread, we need to analyze different possibilities and account for different situations. A trivial solution is to add a new control flow

```
  /* foo is a function pointer */
1:   (*foo)(void) = evil();
2:   void alarm_handler(){
3:       foo = good;
4:   }
5:   int main(){
6:       int i;
7:       signal(SIGALRM, alarm_handler);
8:       alarm(1);
9:       for (i=0; i < 1000; i++){
10:      /* this should take
              more than 1 second */
11:     }
12:     foo();
13:  }
```

Figure 6.3: Asynchronous signal handler

edge after every instruction in the main execution trace where there is possibility of control divergence, but this imposes unnecessary computational complexity to the analysis.

Figure 6.3 shows a sample piece of code where the handler for the `SIGALRM` signal is set to the function `alarm_handler`. Similar to the code in Figure 6.1(b), the call to `alarm()` function at line 7 registers a `SIGALRM` signal to be sent to the program in one second. Depending on the speed of the execution, the signal may arrive somewhere between lines 9-13. Moreover, a function pointer `foo` is initialized to the `evil()` function which resembles malicious activity and `foo` is called at line 12 of the code. If the signal `SIGALRM` arrives before the call to the `foo` function, `alarm_handler` gets executed which results in `foo` pointing to the function `good()`, otherwise, the `evil()` function gets executed by calling function `foo`. The side effect in this code example is the assignment to the `foo` variable in such a way that if the handler is executed before line 12, because of the assignment of `good()` to `foo`, the malicious activity is not exposed. Otherwise, `foo` points to `evil()` meaning that if the handler executed after line 12, another execution path is taken.



(*a*) Static CFG of the code, the gray box in the main code resembles one basic block

(*b*) Augmented CFG for asynchronous signal handler

Figure 6.4: CFG Augmentation for asynchronous signal handler

To handle asynchronous signals, in general, we need to take into account the side effects of executing the handler code. Figure 6.4 tries to visualize this situation where, as opposed to synchronous signals where the control flow edge is at a fixed location (in the code), for asynchronous signals, control transfer points are determined by possible impacts of the handler on the execution. As shown in Figure 6.4(a), variable `X` is defined globally where it can be accessed from both the main code and the signal handler. Suppose that the gray box in the main code is where a signal can be delivered to the process meaning that the control could be transferred to the signal handler at any point in the gray area. The variable `X` is accessed multiple times in the gray area, while the signal handler function modifies the variable `X`. Modification to `X` may have impact on the main code where `X` is used, i.e. points where there is an edge to the handler in Figure 6.4(b). Figure 6.4(b) shows the augmented CFG for the code and handler of Figure 6.4(a).

To handle asynchronous signals we are not necessarily proposing to use symbolic execution to identify alternative execution paths that might be because of the signals delivered to the process. Nevertheless, symbolic execution can be helpful in some

cases, such as when the argument to the `kill` function (that is used to send arbitrary signals to processes) is symbolic. This results in activation of different functions in the receiving process if the process has registered different handlers for different signals. Symbolic execution can help identifying inputs that lead to different handlers being called in the execution by symbolically determining arguments to the function `kill`. Asynchronous signals are generally generated externally which are then delivered to the process. For an attacker to exploit these mechanisms provided by the OS, they need to arrange for the signals to be delivered to the process somehow, perhaps by forking multiple threads and having the threads to communicate with each other using signals. This complex logic that is necessary for an attack to succeed makes the attack detectable and perhaps predictable.

The code examples that are shown in this section use `SIGALRM` signal as to show how to use asynchronous signals for control transfer purposes. However the proposed solution is generalizable to the other types of asynchronous signals. Example of such asynchronous signals that can be used in a similar way are those of used for inter-process communications such as `SIGCHILD` and `SIGUSR1`. `SIGCHILD` is raised when a forked child of a process dies and `SIGUSR1` is left for user defined handlers.

6.3.3 Attack Model

As with other techniques that rely on symbolic execution and SMT solvers, our approach is limited by the theoretical boundaries of SMT solvers [118, 142]. An attacker can also add complexity to the code in order to add complexity to the path constraints. This can increase the processing time for the back-end SMT solver to check satisfiability of the path condition. A related attack is to use code that produces too much work for the analysis. For example by using opaque symbolic memory pointers (symbolic pointers that are not satisfiable), the analysis needs to send a lot of queries to the SMT solver which slows down the analysis (Chapter 5). However, note that in the context of identifying the possible exception-based control-flow behaviors of a program, the back-end SMT solver is used only for those exceptions for which exception handlers have been registered during execution and

for those instructions that can give rise to such exceptions, not for all executed instructions. Registering custom handlers for all of the exceptions does not affect the analysis significantly because only a small subset of these exceptions can be generated by an instruction, such as `SIGILL` or `SIGSEGV` in Linux. Other signals are mainly used for inter-process communication and may not provide a reliable mechanism to obfuscate control transfers in the code.

6.4 Implementation

We have implemented a prototype system based on Intel PIN tool [88] which is a dynamic binary instrumentation tool. Our system works on the binary level and so we do not need access to the source code. The analysis is based on two main components: 1) dynamic taint analysis; and 2) symbolic execution. Our taint analysis uses bit-level granularity for taint propagation discussed in Section 2.3. Our symbolic execution engine is a layer on top of the taint analysis engine which interfaces with the Z3 [45] SMT library to solve symbolic expressions. We have implemented concolic testing that runs the input program with the given inputs and analyses the execution trace observed along the execution path. Both taint analysis and symbolic execution analysis are implemented to work on the `x86` instruction set.

For dynamic taint analysis, we use the `objdump` tool on Linux to produce disassembly and create a CFG of the disassembled code. The post-dominator information is then extracted by analyzing the CFG which specifies post-dominators for conditional branches in the code. For signal handlers, however, we use PIN APIs to intercept user's registered signal handlers to analyze them. Our current implementation does not analyze the handler code statically because static analysis of binary code is hard in general and does not always guarantee correct results [86] due to code obfuscation or self-modification that are common among malware code. We instead take a dynamic approach where we intercept the handlers at runtime and capture the implicit flow by marking tainted those data flows that are because of any exception-based control transfers. We realize that with this approach it is possible

to miss some opportunities for symbolic execution. The alternative assumptions we make here are quite realistic based on what we have observed in the experimental evaluations on real malicious codes.

To find specific user handlers that are registered by the user code, the tool intercepts calls to the `sigaction` and `signal` system calls that are used to register user handlers for given signals. Using the addresses of the signal handlers that are registered by the user we are able to augment the static CFG and add explicit control flow edges wherever the symbolic execution engine finds a satisfiable input that would cause an instruction to raise a particular exception.

6.5 Experimental Evaluations

We evaluated our prototype tool against a variety of different test programs and malicious codes. We then compared our results with current state-of-the-art available tools in both dynamic taint analysis and symbolic execution. We ran the analysis on a Linux platform running Ubuntu 14.04 with 4 Intel Xeon `CPUs` (3.3 GHz) and 1 Tera-byte of RAM.

For both dynamic taint analysis and symbolic execution we used two sets of programs for our evaluations:

- The first set consists of three sample test programs that use exception-based control transfers to obfuscate branch points in the code. All three programs define exception handlers that will redirect the control flow, and have some input value that is used to conditionally trigger the exception. Additionally, all three samples propagate some tainted data in the signal handlers to resemble the idea of implicit information flow. These three programs are:

    1. *invalid-memory* that uses user input to create a memory pointer that is a valid memory address if the input satisfies some conditions, otherwise an invalid pointer.

2. *invalid-opcode* that overwrites `nop` instructions in the code by an invalid opcode if the user input does not satisfy the conditions resulting in a `SIGILL` in the code.
3. *divide-by-0* that performs a division where the dividend is computed from input. The dividend is zero for inputs not satisfying the condition and non-zero otherwise.

- The second set consists of six exploit codes written in `C` that have behaviors similar to our generated test programs. These are proof-of-concept exploits that are written for various vulnerabilities and are publicly available through web-sites such as `http://www.exploit-db.com`. All of the samples overwrite signal handlers for particular signals where two of the samples use variations of the implicit flow showed in Figure 6.3. The exploits are mentioned by their corresponding CVEs, if there were any, in the rest of this section. From all the exploit samples in Table 6.2 as well as `CVE-2004-1235` in Table 6.1, the first four register a handler function for the `SIGALRM` signal and use it to redirect control flow or to propagate some data; and the last test-case uses `SIGSEGV` for these purposes.

The rest of this section presents our results compared to other systems for dynamic taint analysis and symbolic execution against our evaluation test-cases.

6.5.1 Dynamic Taint Analysis

We use Dytan [35] the only dynamic taint analysis system that is available publicly. Dytan implements implicit taint propagation through control flow edges. For the test-cases that contained some form of implicit taint propagation through the signal handlers, we ran experiments with Dytan and our tool where the results are presented in Table 6.1. Dytan failed to identify any implicit flow on all the test-cases while our tool was able to successfully identify the implicit flows that were caused by exception-based control transfers.

	Dynamic Taint Analysis	
Prgm	Dytan	Our System
invalid-memory	×	✓
invalid-opcode	×	✓
divide-by-0	×	✓
CVE-2004-1235	×	✓
CPL-kernel-exploit	×	✓

Table 6.1: Dynamic taint analysis results

6.5.2 Symbolic Execution

In order to evaluate our ideas against state-of-the-art symbolic execution engines, we picked a handful of available symbolic execution engines and compared their results with ours. The symbolic execution engines that we used are *KLEE* [22], *S2E* [34], *FuzzBall* [16] and *angr* [119]. KLEE and S2E are automated test-case generation symbolic execution engines that maximize the code coverage, and FuzzBall and angr are symbolic execution frameworks. KLEE needs to have access to source code while the others work at the binary level. Test-cases were annotated to introduce symbolic variables for KLEE, S2E and angr. FuzzBall however accepts arguments which instructs the tool to mark memory locations or registers as symbolic at certain point in the execution.

	Symbolic Execution Engines				
Prgm	KLEE	S2E	FuzzBall	angr	Our System
invalid-memory	✓	✓	×	×	✓
invalid-opcode	×	×	×	×	✓
divide-by-0	✓	✓	×	×	✓
remote-file-exploit	×	×	×	×	✓
CVE-2005-2772	×	×	×	×	✓
CVE-2011-4029	×	×	×	×	✓
CVE-2014-3153	×	×	×	×	✓
CPL-kernel-exploit	×	×	×	×	✓

Table 6.2: Symbolic execution results

Prgm	*Workload*	*Normal Exec. Time (s)*	*Analysis Exec. Time (s)*	*Total Query Time (s)*	*No. Queries*	*Avg. Query Time (s)*
MD5	1	0.003	36.664	21.794	745	0.039
	2	0.005	72.558	43.566	1485	0.029
Gzip	1	0.047	159.204	34.127	1631	0.020
	2	0.094	273.846	33.546	1628	0.020

Table 6.3: Cost analysis of our prototype tool

Table 6.2 summarizes the analysis results. KLEE only was able to find alternative inputs and execution paths for *invalid-access* and *divide-by-0* programs, however, KLEE did not execute the code in the handler function meaning that it was not able to explore paths that existed within the signal handler set for `SIGFPE` . S2E was able to find inputs leading to the execution paths for the *invalid-access* and *divide-by-0*, as we were expecting, however it failed on the last exploit, `CPL-kernel-exploit` which sets a signal handler for `SIGSEGV` after running for a few hours. We were not able to run KLEE on the last exploit too since the code uses inline assembly that is not implemented in KLEE. angr and FuzzBall were not able to find alternative execution paths or alternative inputs for any of the test-cases and failed. Our tool however, was able to detect exception-based control transfers and so was able to find appropriate inputs that cause different execution paths for the instructions causing exception-based control transfers.

For the exploit samples that use `alarm` system call, we also ran experiments with the argument to the `alarm` symbolic. This is apparently problematic for angr and FuzzBall since they produced error messages indicating that they do not handle system calls with symbolic arguments. KLEE is able to make system calls when the argument is symbolic[1] but did not find any alternative paths or inputs. S2E is able to trigger the function handler for `SIGALRM` if the argument to the `alarm` is made symbolic, however we believe this is due to the internal logic the `alarm` function that allows S2E to construct path constraints and suggest alternative input values as argument to `alarm`. Moreover, in all the samples the argument to `alarm` is a

[1]This is possible only with the option `--allow-external-sym-calls` given to KLEE

constant and it does not make sense to require the user to manually find and make the arguments to these function symbolic specially when there are many samples that need to be analyzed.

6.5.3 Performance Overhead

To measure the performance overhead of our system, we ran our tool on two computation intensive programs and recorded the overheads imposed by our tool as well as the SMT solver. We chose *Md5* program from `http://people.csail.mit.edu/rivest/Md5.c` and *Gzip* form `spec2000int`, added a simple signal handler for `SIGSEGV` signal and ran each of these programs under different workloads where workload 2 is twice as large as workload 1. The reason behind choosing these programs was to test our tool against programs with more realistic sizes than our test-cases and get a better sense of performance overheads imposed by our tool on real programs.

The results are presented in Table 6.3. *Normal Execution Time* is the time programs ran natively while *Analysis Execution Time* shows the time taken by the analysis. *Total Query Time* is that amount of analysis time that was spent in the SMT solver, *number of Queries* are total number of queries sent to the Z3 SMT solver and the *Average Query Time* is the average time spent for each query in the SMT solver in each workload. As it can be seen from the table, the slowdown caused by the analysis varies by the programs. Especially if the program has more instructions that potentially can raise exceptions (memory accesses in this case), more time is spent in the SMT solver which slows down the analysis. Moreover, larger average query time for *Md5* program compared to *Gzip* suggests that the complexity of the path constraints and how memory pointers are computed in the code impacts the overall performance.

Although according to the Table 6.3, the analysis overhead is considerable, there is lot of room for optimization. For instance, we can use a caching mechanism to cache the results of recently solved queries and compare them with new ones before sending them to the solver. This is beneficial specially for symbolic memory access

checks, that is the most common type of queries sent to the solver because memory access patterns are usually sequential and happen in a loop. If, for instance, the symbolic part of the pointer is the same while an offset is changing, using the cached results of previously sent queries will improve the overall performance of the system.

6.6 Related Work

Many researchers have investigated symbolic code execution; see the survey by Schwartz *et al.* [114]. An important application of this approach is in analysis of malicious and/or obfuscated code [12, 13, 159, 21, 93, 117, 42, 157]. However, these works generally do not explicitly address the challenges that arise in analyzing obfuscated code, which is especially prevalent in malware.

We are not aware of any other work in the research literature discussing the issues confronting symbolic execution or dynamic taint analysis when dealing with programs that contain exception-based control transfers. Our approach to find exception-based control transfers are similar to those used in existing systems such as KLEE, S2E or Mayhem. These systems use symbolic execution to guide the execution of the program under the analysis and find different combination of inputs that cause the program to crash [24, 22] which then can be combined with other reasonings to automatically generate exploits [30]. This is different from our work that recognizes program faults as an obfuscation technique to obscure control flow transfers.

The use of exceptions to obfuscate control flow is well known [107]. Malware have long used a simple instance of this approach to obfuscate direct unconditional jumps, by constructing and dereferencing a null pointer. In commonly encountered malware, this obfuscation is typically used to bypass ordinary anti-virus detectors rather than to propagate information through implicit flows. However, it is straightforward to modify this code to use exception-based control transfers to hinder dynamic taint propagation and symbolic execution.

A number of researchers have described security-related applications of dynamic taint analysis [97, 66, 67, 74, 150]. Clause *et al* [35] and Schwartz *et al.* [114] discuss

general frameworks for dynamic taint analysis, but do not address issues arising from implicit flows in obfuscated code in general, and exception-based control transfers in particular. The problems arising from dynamic taint analysis of code containing implicit information flows is discussed by Cavallarro *et al.* [27].

A number of researchers have looked into the problem of analysis of exception-handling behavior of programs. This work typically focuses on explicit exception-management mechanisms (throw-catch statements) at the source code level [121, 111, 31, 72, 120, 19]. We are not aware of any work on reasoning about exception behavior at the binary level.

CHAPTER 7

CONCLUSIONS

In this dissertation, we explored the idea of a generic automatic deobfuscation approach that minimizes the assumptions about the underlying obfuscation techniques being analyzed. We showed that current standard program analysis techniques that are used to understand and characterize program semantics do not work on obfuscated code. The reason is that these analyses are designed to handle normal compiler-generated code that complies with certain characteristics where these characteristics do not necessarily hold in obfuscated code. Instead of tailoring the deobfuscation algorithm to specific characteristics of a particular obfuscation technique, e.g. the structure of the virtual machine in emulation-obfuscation, we try to extract the logic of the program by finding input-to-output computation achieved by applying data and control flow analysis to the program.

It was shown in Chapter 2 that taint analysis, a well-known and highly useful data flow analysis technique, suffers from imprecision when applied to obfuscated code. We showed how this problem can be addressed by refining and tuning different parameters: propagating taint at bit-level granularity, using distinct taint labels for the sources of the taint, and using operation functional semantics in propagating taints from source to destination operands. Empirical studies in Section 2.4 showed that the presented approach is more precise in finding the flow of tainted data. Moreover, experimental results in Section 4.3.3 showed that the new approach is quite effective in finding the input-to-output mapping when deployed in the deobfuscation algorithm.

Chapter 3 discussed a twist in another widely used and well-studied program analysis, control dependence analysis, that is crucially important in identifying the input-to-output mapping of the deobfuscation process. We showed that standard control-dependence analysis that is computed based on CFGs and post-dominator relationships in the CFG, is not precise enough to capture the entire logic in interpre-

tive systems including virtualized-obfuscated code. Our solution was to incorporate different components involved in the computations–involving the input program, interpreter and JIT compiler–to correctly identify the control dependencies. Experimental results in Section 3.5 showed that the new concept of control dependency introduced in Chapter 3 is more precise for client applications such as dynamic slicing and detecting implicit information flow. Moreover, experimental results in Sections 4.3.1.1, 4.3.1.2 and 4.3.2 show the over-approximation algorithm to compute the new notion of control dependency is able to capture the logic of the original unobfuscated code in virtualized-obfuscated binaries as well as ROP programs.

Using the revisited taint and control flow analyses in Chapters 2 and 3, we proposed a deobfuscation technique (Chapter 4) that identifies the input-to-output mapping in the computation and simplifies this mapping dynamically. Experimental results in Section 4.3 showed the deobfuscation technique is successful in undoing the obfuscation for virtualization-obfuscation and ROP programs. Experiments using sophisticated commercial obfuscation tools indicate that our approach is effective in stripping out the obfuscation and extracting the logic of the original code.

To address the code coverage issue of deobfuscation approach, we equipped the analysis with a symbolic execution engine that builds path constraints over an execution trace and finds alternative inputs that cause alternative execution paths to be taken if supplied to the program. We showed in Chapters 5 and 6 that using vanilla symbolic execution analysis is not practical for obfuscated code and does not produce accurate results. We proposed a way to mitigate the problem using a combination of fine-grained bit-level taint analysis and architecture-aware constraint generation. For exception-based control transfer obfuscations we proposed run-time CFG augmentation that identifies unexpected control flow edges by instructions that might disrupt the normal control flow of the code. Our approach employs symbolic execution to reason about the possibility of an exception when the instruction to be executed can potentially raise one. We then used the augmented CFG to identify alternative execution paths and potential implicit flows. Experiments in Sections 5.5 and 6.5 indicate that the proposed approaches significantly improve the efficacy

of symbolic execution as well as dynamic taint analysis on obfuscated code and allows the analysis to identify implicit flows in malware that were not being detected previously.

7.1 Limitations and Future Work

7.1.1 Size of Execution Trace

Our approach requires an execution trace collected from the executed program. Aside from anti-analysis and anti-tracing techniques that might be protecting the binary from dynamic analysis, the execution trace size could be an issue. The attacker can use techniques to make the trace size arbitrarily large (e.g. [77]). One possible future direction of research would be to examine the possibility of applying the deobfuscation process on a recorded CFG. Unlike an execution trace, the number of basic blocks in the code are limited[1] which can significantly reduce the size of an execution trace where only basic blocks are recorded.

7.1.2 Input/Output Obfuscation

In our deobfuscation approach, we made an assumption that the significant computation is the one that computes the input-to-output mapping. The deonfuscation then identifies and simplifies this mapping. In other words, we assume that only the computation involving the input-to-output mapping is obfuscated not the input/output itself. However, knowing this, an attacker can pollute the computation using irrelevant inputs and/or outputs, for example by inserting unnecessary and irrelevant system calls into the code. This can be mitigated by manually choosing the input/output sources but detecting legitimate input/output system calls in general might be a hard problem.

Furthermore, new computations can be added by the obfuscation such that they involve inputs and/or outputs but do not affect the final values. For instance, an

[1]It is possible to generate new basic blocks at run-time using self-modification where in this case, the time the binary is being monitored could be limited.

attacker can encrypt and then decrypt the input before using it in the original computation. There is no easy way to identify the fake computation part of the code automatically so this might add to the complexity of the simplified code. However, adding arbitrary irrelevant computations to the original code degrades the performance of the obfuscated code which limits the utilization of this kinds of obfuscations by the adversary.

Finally, those parts of the CFG that are influenced by the inputs can be modified by the obfuscation. Our deobfuscation technique simplifies the control flows that are not influenced by the input since the target does not change depending on the input. An attacker can insert arbitrary conditional branches in the obfuscated program that did not exist in the original code where the target basic blocks have the same functionality with slightly different codes (one can use identity functions [143] to implement this kind of obfuscation). The obfuscated code can choose one of the targets at runtime randomly. The analysis needs to identify codes that are semantically equivalent but are different in syntax in order to be able to merge two different execution paths.

7.1.3 Taint Labels

In Section 2.3 we proposed a bit-level precise taint analysis that uses distinguished taint labels for computations that involve condition flags register. Using distinct taint labels helps to accurately track each individual condition flag bit that was influenced by input and prevent taint explosion. This was specially helpful for virtualization-obfuscated binaries where the condition flags were used by obfuscation tool to implement and obfuscate conditional branching, but the same idea could be applied generally to any input value. In such a case, the analysis should be able to individually track each bit of input data that might impose a significant overhead to the analysis. It also gives the attacker an opportunity to degrade the analysis.

Appendices

APPENDIX A

Proofs of i-Control-Dependency Theorems

THEOREM 3.3.1. *Given two instructions I and J in an execution of an interpretive system $\mathcal{I}(\mathbb{I}, \mathbb{J})$ on an input program P, where both $\mathcal{I}$ and P satisfy the realizable paths assumption, $J \overset{semantic}{-\!\!\rightarrow} I$ implies $J \overset{interpretive}{-\!\!\!\!\rightarrow} I$.*

Proof. The proof is by induction on the number k of JIT-compilation rounds performed by $\mathcal{I}$ on the input program.

Base case. The base case is for $k = 0$, i.e., when there is no JIT compilation. The proof in this case is by contradition. Suppose that $J \overset{\text{semantic}}{-\!\!\rightarrow} I$ and $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$. From Definition 3.3.3, $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$ is equivalent to (1) $J \overset{\text{static}(\mathcal{I})}{-\!\!\not\rightarrow} I$; and (2) $\Gamma(J) \overset{\text{static(P)}}{-\!\!\not\rightarrow} \Gamma(I)$. We consider these two cases separately:

Case 1. For (i), based on Definition 3.2.1 if $J \overset{\text{static}(\mathbb{I})}{-\!\!\not\rightarrow} I$, then I is post-dominated by J in the interpreter $\mathbb{I}$, which means that in every execution trace of $\mathcal{I}$(P), either both instructions I and J or none of them should be executed. Combining the realizable paths assumption with Definition 3.3.1, this means that $J \overset{\text{semantic}}{-\!\!\not\rightarrow} I$.

Case 2. If $J \overset{\text{static(P)}}{-\!\!\not\rightarrow} I$, then $\Gamma(I)$ is post-dominated by $\Gamma(J)$ in the input program P. Given the realizable paths assumption, this means that for every execution of the $\mathcal{I}$(P), either both $\Gamma(I)$ and $\Gamma(J)$ (and hence I and J) or none of them should be executed. This simply means that $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$ and according to (1) above, $J \overset{\text{semantic}}{-\!\!\not\rightarrow} I$.

It follows from this that $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$ implies that $J \overset{\text{semantic}}{-\!\!\not\rightarrow} I$. This contradicts the hypothesis that $J \overset{\text{semantic}}{-\!\!\rightarrow} I$ and $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$. Thus the theorem holds.

Inductive case. Assume that the theorem holds after k rounds of JIT-compilation and consider the program after $k+1$ rounds of JIT-compilation. Suppose that the $(k+1)^{st}$ round of JIT-compilation causes a set of new instructions **J**to be added to the interpretive system.[1] Suppose that $J \overset{\text{semantic}}{-\!\!\rightarrow} I$; we want to establish that this implies that $J \overset{\text{interpretive}}{-\!\!\!\rightarrow} I$. Depending on which parts of the code I and J belong to, we have the following cases:

1. $I \notin$ **Jand** $J \notin$ **J**, i.e., neither I nor J were added in the $(k+1)^{st}$ round of JIT compilation. It follows from the induction hypothesis that $J \overset{\text{interpretive}}{-\!\!\!\rightarrow} I$ in this case.

2. $I \notin$ **J and** $J \in$ **J**:

   (a) $\Gamma(I) \neq \perp$ and $\Gamma(J) \neq \perp$: this means that I and J originated in the code for instruction handlers in the interpreter. $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$ therefore means that $\Gamma(J) \overset{\text{static(P)}}{-\!\!\not\rightarrow} \Gamma(I)$. This means that $\Gamma(I)$ is post-dominated by $\Gamma(J)$ in the input program. From the realizable paths assumption, this means that $J \overset{\text{semantic}}{-\!\!\not\rightarrow} I$. This is a contradiction. We therefore conclude that $J \overset{\text{interpretive}}{-\!\!\!\rightarrow} I$.

   (b) $\Gamma(I) = \perp$ and $\Gamma(J) \neq \perp$: since $I \notin$ **J**, it belongs to the interpreter code. For the interpreter to transfer control to the JIT-compiled code, there must be an instruction K in the interpreter that transfers control to the JIT-compiled code. Given Definition 3.3.1 and the realizable paths assumption, this implies that J post-dominates K. Suppose $J \overset{i\text{-control}}{-\!\!\not\rightarrow} I$, this means that $I \overset{i\text{-control}}{-\!\!\not\rightarrow} K$ so for every execution of $\mathcal{I}$(P), either I, K and subsequently J (because J post-dominates K) or none of I, K and J are executed together. This is contradictory to the hypothesis where $J \overset{\text{semantic}}{-\!\!\rightarrow} I$ thus $J \overset{\text{interpretive}}{-\!\!\!\rightarrow} I$ holds.

3. $I \notin$ **J and** $J \in$ **J**: this is similar to the previous case.

[1] Some instructions may be removed as well, e.g., because they are dead or unreachable. However, instructions that are removed in this way will not be considered for any subsequent control dependence queries, so we do not consider them.

4. $I \in$ **Jand** $J \in \mathbf{J}$. This implies that $\Gamma(I) \neq \bot$ and $\Gamma(J) \neq \bot$. $\Gamma(J) \overset{\text{static(P)}}{-\not\rightarrow} \Gamma(I)$. This means that $\Gamma(I)$ is post-dominated by $\Gamma(J)$ in the input program, which means that $J \overset{\text{semantic}}{-\not\rightarrow} I$. This is a contradiction. We therefore conclude that $J \overset{\text{interpretive}}{\longrightarrow} I$.

Hence in every case, $J \overset{\text{semantic}}{\longrightarrow} I$ implies $J \overset{\text{interpretive}}{\longrightarrow} I$ and thus that the theorem holds after $k+1$ rounds of JIT compilation.

It follows that the theorem holds for all $k \geq 0$. $\square$

THEOREM 3.3.2. *Given an interpretive system $\mathcal{I}$ and an input program P that both satisfy the realizable paths assumption (see Section 3.2.7), $J \overset{interpretive}{\longrightarrow} I$ implies $J \overset{semantic}{\longrightarrow} I$.*

Proof. Based on the Definition 3.3.3, $J \overset{\text{interpretive}}{\longrightarrow} I$ implies that: (1) $J \overset{\text{static}(\mathcal{I})}{\longrightarrow} I$; or (2) $\Gamma(I) \overset{\text{static(P)}}{\longrightarrow} \Gamma(J)$. We consider each case separately:

Case 1. $J \overset{\textit{static}(\mathcal{I})}{\longrightarrow} I$**.** From the definition of static control dependence (Definition 3.2.1), we can conclude two facts:

- From one of the successors of I—call it namely C_J—execution will eventually reach J; and for all instructions K on paths from I to J through C_J such that $K \neq I, J$, J postdominates K. Let S_I^+ be the set of execution traces of $\mathcal{I}$(P) that include I and C_J: these executions necessarily include J, and since J postdominates all of the other instructions K that lie on paths from I to J through C_J, it follows that J occurs after K on each execution trace along these paths. Finally, from the realizable paths assumption, S_I^+ is non-empty.
- J may not be executed for the successor(s) of I other than C_J. Let S_I^- be the set of execution traces that do not include C_J. It follows that some of these traces do not include J.

This is equivalent to the definition of semantic control dependency. It follows that $I \overset{\text{semantic}}{---\!\!\rightarrow} J$.

Case 2. Suppose that $\Gamma(I) = I'$ and $\Gamma(J) = J'$ (and so $I \in \Gamma^{-1}(I')$ and $J \in \Gamma^{-1}(J')$) in the input program. Based on Definition 3.2.1, $\Gamma(I) \overset{\text{static(P)}}{---\!\!\rightarrow} \Gamma(J)$ means that in the input program P, for one of the successors of I', namely $C_{J'}$, the execution will eventually execute J', and J' is not executed for other successors of I'. Let S_I^+ be the set of execution traces that include instructions in $\Gamma^{-1}(C_{J'})$ and hence include $\Gamma^{-1}(J')$ and so J because J' post-dominates $C_{J'}$ in the input program. Similarly, S_I^- be the set of execution traces that do not include $\Gamma^{-1}(C_{J'})$ and hence do not include $\Gamma^{-1}(J')$ and J. This is equivalent to the definition of semantic control dependency and so we have $I \overset{\text{semantic}}{---\!\!\rightarrow} J$.

So for both cases $J \overset{\text{interpretive}}{-----\!\!\rightarrow} I$ implies $J \overset{\text{semantic}}{---\!\!\rightarrow} I$ under the realizable paths assumption. $\square$

APPENDIX B

Source code for the TinyRISC interpreter

We show below the code for the *tinyRISC* emulator we used for some of our experiments with multi-level emulation. The main interpreter loop is implemented by the function `interp()`. The reason we implemented this function using a series of `if-else` statements is that we found that none of our commercial obfuscation tools applied emulation-obfuscation to code containing a `switch` statement: instead, in this case they would simply perform entry-point obfuscation.

```
======================== FILE: interp.h ========================
/*
* Opcodes
*/
typedef enum {
  MOV, ADD, SUB, MUL, DIV, IF_GE, IF_GT, IF_LE, IF_LT, IF_EQ, IF_NE, JMP, PRINT, HALT
} Opcode;

/*
 * Operand types
 */
typedef enum {
  REG, MEM, IMM, INDIR, NONE
} OperandType;

typedef struct operand {
  OperandType type;
  int  value;
} Operand;

typedef struct instr {
  Opcode opcode;
  Operand src1, src2, dest;
} Instr;

#define NREGS 8
#define NMEM 1024

======================== FILE: interp.c ========================
#include <stdio.h>
#include <stdlib.h>
```

```
#include <stdbool.h>
#include "interp.h"

int Regs[NREGS];
int Memory[NMEM];
extern Instr byteCodeArray[];
extern int nInstrs;

int OperandValue(Operand *op) {
  if (op->type == REG)        return Regs[op->value];
  else if (op->type == MEM)   return Memory[op->value];
  else if (op->type == IMM)   return op->value;
  else if (op->type == INDIR) return Memory[Regs[op->value]];
  else {
    fprintf(stderr, "ERROR [OperandValue]: Unknown operand type %d\n", op->type);
    exit(-1);
  }
}

/*
 * WriteBack(dest, val) -- write val into destination.
 */
void WriteBack(Operand *op, int val) {
  if (op->type == REG)        Regs[op->value] = val;
  else if (op->type == MEM)   Memory[op->value] = val;
  else if (op->type == INDIR) Memory[Regs[op->value]] = val;
  else {
    fprintf(stderr, "ERROR [WriteBack]: Illegal operand type %d\n", op->type);
    exit(-1);
  }
}

/*
 * interp(ip) -- interpret the program starting at the instruction at index ip.
 */
void interp(int ip) {
  Instr *currInstr;
  int op1, op2, result;
  while (true) {
    currInstr =  &(byteCodeArray[ip++]);
    if (currInstr->opcode == MOV) {
      WriteBack(&(currInstr->dest), OperandValue(&(currInstr->src1)));
    }
    else if (currInstr->opcode == ADD) {
      op1 = OperandValue(&(currInstr->src1));
      op2 = OperandValue(&(currInstr->src2));
      result = op1 + op2;
      WriteBack(&(currInstr->dest), result);
    }
    else if (currInstr->opcode == SUB) {
```

```
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  result = op1 - op2;
  WriteBack(&(currInstr->dest), result);
}
else if (currInstr->opcode == MUL) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  result = op1 * op2;
  WriteBack(&(currInstr->dest), result);
}
else if (currInstr->opcode == DIV) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  result = op1 / op2;
  WriteBack(&(currInstr->dest), result);
}
else if (currInstr->opcode == IF_GE) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  if (op1 >= op2) {
    ip = OperandValue(&(currInstr->dest));
  }
}
else if (currInstr->opcode == IF_GT) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  if (op1 > op2) {
    ip = OperandValue(&(currInstr->dest));
  }
}
else if (currInstr->opcode == IF_LE) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  if (op1 <= op2) {
    ip = OperandValue(&(currInstr->dest));
  }
}
else if (currInstr->opcode == IF_LT) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  if (op1 < op2) {
    ip = OperandValue(&(currInstr->dest));
  }
}
else if (currInstr->opcode == IF_EQ) {
  op1 = OperandValue(&(currInstr->src1));
  op2 = OperandValue(&(currInstr->src2));
  if (op1 == op2) {
    ip = OperandValue(&(currInstr->dest));
```

```
      }
    }
    else if (currInstr->opcode == IF_NE) {
      op1 = OperandValue(&(currInstr->src1));
      op2 = OperandValue(&(currInstr->src2));
      if (op1 != op2) {
        ip = OperandValue(&(currInstr->dest));
      }
    }
    else if (currInstr->opcode == JMP) {
      ip = OperandValue(&(currInstr->src1));
    }
    else if (currInstr->opcode == PRINT) {
      op1 = OperandValue(&(currInstr->src1));
      printf("%d\n", op1);
    }
    else if (currInstr->opcode == HALT) {
      return;
    }
  }
}

/*
 * init_mem() : initialize the contents of memory (in this case for binary search).
 */
void init_mem() {
  int search_key, nvals, i;
  if (scanf("%d", &search_key) != 1) {
    fprintf(stderr, "Could not read search key -- aborting\n");
    exit(1);
  }
  if (scanf("%d", &nvals) != 1) {
    fprintf(stderr, "Could not read no. of values -- aborting\n");
    exit(1);
  }

  /*
   * read in the array of values to search
   */
  for (i = 0; i < nvals; i++) {
    if (scanf("%d", &(Memory[4+i])) != 1) {
      fprintf(stderr, "Could not read array value for i = %d\n", i);
      exit(1);
    }
  }

  /*
   * initialize search parameters
   */
  Memory[0] = search_key;     /* the value to search for */
```

```
  Memory[1] = nvals;          /* the number of values in the array being searched */
  Memory[2] = 4;              /* the start index of the array being searched */
  Memory[3] = 4+nvals-1;      /* the end index of the array being searched */
}


int main() {
  init_mem();
  interp(0);
  return 0;
}
```

REFERENCES

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 280–290, June 1998.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[3] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *ICSM*, volume 93, pages 348–357. Citeseer, 1993.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 2007.

[5] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proc. Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, Jan. 1983.

[6] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[7] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[8] G. Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, pages 102–110, 2002.

[9] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[10] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 13–24, 2010.

[11] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[12] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, Feb. 2009.

[13] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1), Aug. 2006.

[14] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: A trace-based jit compiler for cil. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725, 2010.

[15] G. Bilardi and K. Pingali. A framework for generalized control dependence. *ACM SIGPLAN Notices*, 31(5):291–300, 1996.

[16] Bitblaze. FuzzBALL: Vine-based Binary Symbolic Execution.

[17] S. Blazy, S. Riaud, and T. Sirvent. Data tainting and obfuscation: Improving plausibility of incorrect taint. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 111–120. IEEE, 2015.

[18] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proc. 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, 2011.

[19] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 1–12. ACM, 2009.

[20] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[21] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection: Countering the Largest Security Threat*, volume 36, pages 65–88, 2008.

[22] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[23] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software*, pages 2–23. Springer, 2005.

[24] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[25] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[26] K. Casey, M. A. Ertl, and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):37, 2007.

[27] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Stony Brook University, Stony Brook, New York*, 2007.

[28] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[29] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions, Malware and Vulnerability Analysis (DIMVA)*, July 2008.

[30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[31] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 620–625, 2001.

[32] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. ACM Conference on Computer and Communications Security*, pages 559–572, 2010.

[33] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attack. In *Proc. 21th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

[34] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2011.

[35] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[36] J. Clause and A. Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 249–260. ACM, 2009.

[37] L. R. Clausen. A java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[38] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. 1998 IEEE International Conference on Computer Languages*, pages 28–38.

[39] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Sciences, The University of Auckland, July 1997.

[40] K. Coogan, G. Lu, and S. Debray. Deobfuscating virtualization-obfuscated software: A semantics-based approach. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 275–284, Oct. 2011.

[41] V. S. Costa. Optimising bytecode emulation for prolog. In *Principles and Practice of Declarative Programming*, pages 261–277. Springer, 1999.

[42] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 25–36, Oct. 2006.

[43] R. Cytron, J. Ferrante, and V. Sarkar. Compact representations for control dependence. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 337–351, 1990.

[44] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proc. 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 40–51, Mar. 2011.

[45] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[46] A. Decker, D. Sancho, M. Goncharov, and R. McArdle. Ilomo: A study of the ilomo/clampi botnet. Technical report, Trend Micro, Aug. 2009.

[47] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[48] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[49] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 51–62, Oct. 2008.

[50] S. Drape et al. Intellectual property protection using obfuscation. *Proceedings of SAS 2009*, 4779:133–144, 2009.

[51] W. Drewry and T. Ormandy. Flayer: Exposing application internals. *WOOT*, 7:1–9, 2007.

[52] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[53] N. Falliere. Inside the jaws of Trojan.Clampi. Technical report, Symantec Corp., Nov. 2009.

[54] H. Fang, Y. Wu, S. Wang, and Y. Huang. Multi-stage binary code obfuscation using improved virtual machine. In *Information Security*, pages 168–181. Springer, 2011.

[55] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[56] P. Ferrie. Anti-unpacker tricks–part one. *Virus Bulletin*, 4, 2008.

[57] P. Ferrie. Prophet and loss. *Virus Bulletin*, Sept. 2008. `www.virusbtn.com/virusbulletin/archive/2008/09/vb200809-prophet-loss`.

[58] M. A. Francel and S. Rugaber. The relationship of slicing and debugging to program understanding. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 106–113. IEEE, 1999.

[59] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 263–276. Springer, 1997.

[60] E. Gagnon and L. Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *Compiler Construction*, pages 170–184. Springer, 2003.

[61] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[62] V. Ganesh and T. Hansen. STP. `https://github.com/stp/stp`.

[63] S. Ghosh, J. D. Hiser, and J. W. Davidson. Matryoshka: strengthening software protection via nested virtual machines. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 10–16. IEEE, 2015.

[64] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, June 2005.

[65] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[66] C. Hauser, F. Tronel, L. Mé, and C. J. Fidge. Intrusion detection in distributed systems, an approach based on taint marking. In *Proc. 2013 IEEE International Conference on Communications (ICC)*, pages 1962–1967, 2013.

[67] C. Hauser, F. Tronel, J. F. Reid, and C. J. Fidge. A taint marking approach to confidentiality violation detection. In *Proc. 10th Australasian Information Security Conference (AISC 2012)*, Jan. 30 2012.

[68] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.

[69] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[70] X. Hu, T.-C. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proc. ACM Conference on Computer and Communications Security*, pages 611–620, Nov. 2009.

[71] Intel. Intel 64 and ia-32 architectures software developer's manual. 2015.

[72] J.-W. Jo and B.-M. Chang. Constructing control flow graph for java by decoupling exception flow from normal flow. In *Computational Science and Its Applications–ICCSA 2004*, pages 106–113. Springer, 2004.

[73] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[74] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[75] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proc. Fifth ACM Workshop on Recurring Malcode (WORM 2007)*, Nov. 2007.

[76] J. C. King. Symbolic executin and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[77] C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296. ACM, 2011.

[78] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24. ACM, 2006.

[79] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[80] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. 13th USENIX Security Symposium*, Aug. 2004.

[81] A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968, 2005.

[82] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *ACM SIGARCH Computer Architecture News*, 20(2):46–57, 1992.

[83] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[84] Laurelai. Partial stuxnet source decompiled with hexrays. `https://github.com/Laurelai/decompile-dump/blob/master/output/016169EBEBF1CEC2AAD6C7F0D0EE9026/016169EBEBF1CEC2AAD6C7F0D0EE9026.c`, 2010.

[85] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[86] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[87] K. Lu, D. Zou, W. Wen, and D. Gao. deRop: removing return-oriented programming from malware. In *Proc. 27th. Annual Computer Security Applications Conference (ACSAC)*, pages 363–372, Dec. 2011.

[88] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, Chicago, IL, June 2005.

[89] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 198–209. IEEE, 2004.

[90] S. P. Midkiff. Automatic parallelization: an overview of fundamental compiler techniques. *Synthesis Lectures on Computer Architecture*, 7(1):1–169, 2012.

[91] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, 2015.

[92] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[93] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[94] G. Myles and C. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[95] S. Nanda, W. Li, L. Lam, and T. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, 2006.

[96] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[97] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[98] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[99] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proc. 1998 International Conference on Computer Languages*, pages 132–142, 1998.

[100] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Proc. 26th. Annual Computer Security Applications Conference (ACSAC)*, pages 49–58, Dec. 2010.

[101] Oreans Technologies. Code virtualizer: Total obfuscation against reverse engineering. `www.oreans.com/codevirtualizer.php`.

[102] Oreans Technologies. Themida: Advanced windows software protection system. `www.oreans.com/themida.php`.

[103] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2006.

[104] L. Pitt and M. K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993.

[105] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. *ACM SIGPLAN Notices*, 33(5):291–300, 1998.

[106] A. Podgurski, L. Clarke, et al. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on*, 16(9):965–979, 1990.

[107] I. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proc. Usenix Security 2007*, pages 275–290, Aug. 2007.

[108] T. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, New York, N.Y., Jan. 1995. ACM Press.

[109] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. A framework for understanding dynamic anti-analysis defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 2. ACM, 2014.

[110] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):27, 2007.

[111] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[112] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2, Mar. 2012.

[113] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[114] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[115] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, Sept. 2005.

[116] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *proc. ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[117] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proc. 2009 IEEE Symposium on Security and Privacy*, May 2009.

[118] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. 15th Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[119] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[120] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 348–357. IEEE, 1998.

[121] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[122] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74. ACM, 2009.

[123] J. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[124] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proc. of the 4th International Conference on Information Systems Security*, Dec. 2008.

[125] B. Spasojević. Code deobfuscation by optimization. `https://events.ccc.de/congress/2010/Fahrplan/events/4096.en.html`.

[126] V. Srinivasan and T. Reps. Slicing machine code. Technical Report TR-1824, Computer Sciences Department, University of Wisconsin, Madison, WI, Oct. 2015.

[127] J. Stafford. *A Formal, Language-independent, and Compositional Approach to Interprocedural Control Dependence Analysis*. PhD thesis, Boulder, CO, USA, 2000.

[128] J. Stoy. *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT, 1977.

[129] StrongBit Technology. EXECryptor – bulletproof software protection. `www.strongbit.com/execryptor.asp`.

[130] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 50–57. ACM, 2003.

[131] Symantec. Internet security threat report, 2015.

[132] V. Thampi. Udis86: Disassembler Library for x86 and x86-64. `http://udis86.sourceforge.net/`.

[133] S. Thibault, C. Consel, J. L. Lawall, R. Marlet, and G. Muller. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[134] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[135] Tora. Devirtualizing FinSpy. `http://linuxch.org/poc2012/Tora, DevirtualizingFinSpy.pdf`.

[136] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proc. 12th IEEE Working Conference on Reverse Engineering*, pages 45–54, Nov. 2005.

[137] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[138] VMProtect Software. VMProtect – New-generation software protection. `www.vmprotect.ru/`.

[139] VX Heavens. Vx heavens, 2011. `http://vx.netlux.org/`.

[140] W32.Netsky.AE. `www.symantec.com/security_response/writeup.jsp?docid=2004-102522-4640-99&tabid=2`.

[141] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.

[142] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pages 210–226. Springer, 2011.

[143] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[144] Wei Ming Khoo. Taintgrind: a Valgrind taint analysis tool. `https://github.com/wmkhoo/taintgrind`.

[145] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352—357, July 1984.

[146] W. worm. `www.cert.org/historical/advisories/CA-2003-20.cfm`.

[147] T. Wouters, J. Yasskin, and C. Winter. unladen-swallow: A faster implementation of python. `https://code.google.com/p/unladen-swallow/`.

[148] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 185–195. ACM, 2007.

[149] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[150] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical report, Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005.

[151] B. Yadegari and S. Debray. Bit-level taint analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 255–264. IEEE, 2014.

[152] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ser. CCS*, volume 15, pages 732–744, 2015.

[153] B. Yadegari and S. Debray. Control dependence analysis of interpretive systems. In *under review*, 2016.

[154] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015.

[155] B. Yadegari, J. Stepehns, and S. Debray. Analysis of exception-based control transfers. In *under review*, 2016.

[156] H. Yin and D. Song. *Automatic Malware Analysis: An Emulator Based Approach*. Springer, 2012.

[157] H. Yin and D. Song. Analysis of trigger conditions and hidden behaviors. In *Automatic Malware Analysis*, SpringerBriefs in Computer Science, pages 59–67. 2013.

[158] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[159] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, 2007.

[160] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1116–1127. VLDB Endowment, 2007.