

A Security Architecture for Survivability Mechanisms

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Chenxi Wang

© Copyright by

Chenxi Wang

All Rights Reserved

October 2000

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

Author

This dissertation has been read and approved by the Examining Committee:

John C. Knight (Dissertation Advisor)

James French (Committee Chairman)

George R. Blakley, III.

John McHugh

Peter G. Neumann

William A. Wulf

Yacov Haimes (Minor Representative)

Accepted for the School of Engineering and Applied Science:

Dean Richard W. Miksad, School of Engineering and Applied Science

Acknowledgements

When the dissertation is finally written and sealed, the days of agonizing over half-baked ideas, struggling through writer's block, and worrying about when (and if) all this is going to end all seem rather distant. The many people who helped me in this long and sometimes frustrating process deserve special thanks, and they are many.

First and foremost, I am indebted to the two advisors in my graduate career: Bill Wulf and John Knight. Bill had the misfortune of coaching me through the early years of my graduate study when I was an awkward twenty-year-old who knew nothing. I learned many things from Bill, whose vision, vast knowledge, and impeccable insights have been and will continue to be an inspiration to me. John's invaluable guidance is the driving force behind this dissertation research. As an advisor and also a friend, John believed in me and guided me through the most difficult times throughout this work. I thank John for his superb patience in putting up with me. I shall miss his humor and the many drag-down and knock-out research meetings with him.

I am grateful to Jonathan Hill for the many hours he devoted in implementing the core of the One-way Translation compiler. Jonathan is a dream partner without whom this project would not have been completed as smoothly and successfully as it has been.

Jack Davidson deserves special thanks for his brilliant insights and in-depth knowledge of programming languages that helped shape this research.

I am also thankful to my thesis committee. In addition to my advisors, the committee consisted of Bob Blakley, Jim French, Yacov Haimes, John McHugh, and Peter Neumann. They provided helpful feedback and suggestions on this work.

I am particularly grateful to Peter Neumann who believed in me and encouraged me throughout the frustrating process of thesis writing. Peter has acted as a mentor to me in many ways, and I benefited tremendously from his experience, wisdom and guidance.

Special thanks go to Bob Blakley who has been a mentor and a friend to me. Bob's empathy and encouragement helped me to persevere through many difficult times.

Gratitude also goes to the many friends in and out of Charlottesville who made my life in the past a few years a particularly memorable experience: John, for his patience and understanding. Glenn, Nuts, Anh, and Karine, for keeping me in line and for the many wonderful dinner parties and get-togethers. Lisa, for indulging my girlish instincts. Chenyang and XiaoYuan, for putting me up many times as an unexpected dinner guest. My other friends, Kate Stoddard, Marv Schaefer, Dave Evans, Anil Somayaji, Matt Elder, Sally McKee, Tongtong Zhang, Jane Prey, Micky Lo and the diva-in-training Sarah Wells, all enriched my life in their own unique ways, and I thank them for that.

And finally, I thank my parents for their unconditional love, never-ceasing support, and their artful ways of breeding my independence and free thinking. It is them who instilled in me the inspiration and drive for excellence, and I shall always be grateful.

Chenxi Wang, October 2000.

Abstract

In survivability management systems, some management entities reside on application hosts that are not necessarily trustworthy. The integrity of these software entities is essential to the security of the network management scheme. In this talk, I present a novel framework to facilitate software security against malicious execution environments.

The approach consists of two fundamental techniques: a) Incorporating diversity in the deployment and the design of the program such that impersonation or intelligent tampering attacks require extensive analysis of the program; and b) one important aspect of program analysis, namely static analysis, is deterred by the incorporation of aliasing and further degeneration of the program control flow. It is shown that analyzing the transformed programs statically is an NP-hard problem. Theoretic bounds on approximate analysis methods are also provided. The transformations are implemented in a C compiler. Program performance results are presented. Empirical experiments with existing analysis tools showed that static analysis for the transformed programs are hindered to a significant degree.

CHAPTER 1	1
INTRODUCTION	1
1.1. MOTIVATION	1
1.2. THE MALICIOUS HOST PROBLEM	2
1.2.1. DIFFERENT FACETS OF THE MALICIOUS HOST PROBLEM	3
1.2.2. THE INPUT-SPOOFING PROBLEM.....	7
1.3. DESIGN GOALS	9
1.4. SOLUTION OVERVIEW	10
1.5. DISSERTATION ORGANIZATION	12
CHAPTER 2.....	13
BACKGROUND AND PROBLEM CONTEXT.....	13
2.1. CRITICAL INFRASTRUCTURE SYSTEMS	14
2.1.1. LARGE SCALE.....	14
2.1.2. HETEROGENEITY.....	15
2.1.3. EXTENSIVE USE OF COTS COMPONENTS.....	17
2.2. THE SURVIVABILITY ARCHITECTURE	18
2.2.1. A CONTROL SYSTEM CONSTRUCT.....	18
2.2.2. THE NETWORK PROBE PROGRAMS.....	21
2.3. THE MALICIOUS HOST PROBLEM AND OTHER SECURITY ISSUES	22
2.4. THREATS AND ATTACK SCENARIOS	25
2.4.1. DENIAL-OF-SERVICE ATTACKS.....	25
2.4.2. INTELLIGENT TAMPERING AND IMPERSONATION.....	27
2.5. CAPABILITIES OF INTRUDERS	29
2.5.1. NETWORK INTRUDERS.....	29
2.5.2. MALICIOUS INSIDERS.....	30
2.5.3. PRIVILEGED USERS	31
2.6. FUNDAMENTAL CHALLENGES	33
2.6.1. VERIFICATION OF EXECUTION RESULTS.....	34
2.6.2. FINITE STATE SPACE FACILITATES PROGRAM ANALYSIS.....	35
2.7. SUMMARY	35
CHAPTER 3.....	36
THE SOLUTION FRAMEWORK.....	36
3.1. A COMPLEXITY ARGUMENT	37
3.1.1. INFORMATION DIVERSITY.....	41
3.1.2. INFORMATION COMPLEXITY.....	43
3.1.3. INPUT -TO-OUTPUT STATE INFLATION	46

3.2. PUTTING IT ALL TOGETHER	49
3.3. IMPLEMENTATION STRATEGY.....	50
CHAPTER 4.....	52
ONE-WAY TRANSLATION	52
4.1. A MODEL OF SEMANTICS-PRESERVING TRANSFORMATION.....	53
4.2. ONE-WAY TRANSLATION PROCESS.....	55
4.2.1. BEHAVIORAL TRANSFORMATIONS.....	56
4.2.2. INTERNAL TRANSFORMATIONS.....	60
4.3. INTRA-PROCEDURAL TRANSFORMATIONS	61
4.3.1. THE FUNDAMENTALS OF INTRA-PROCEDURAL ANALYSIS.....	61
4.3.2. DEGENERATION OF THE STATIC PROGRAM CONTROL-FLOW	63
4.3.3. ALIASING AND DATA-FLOW ANALYSIS.....	68
4.3.4. OBSTRUCTING INTRA-PROCEDURAL ANALYSIS—PUTTING IT TOGETHER.....	75
4.4. INTER-PROCEDURAL CODE TRANSFORMATION.....	77
4.4.1. FUNCTION-CALL TRANSFORMATIONS.....	79
4.4.2. FUNCTION POINTER ALIASING.....	80
4.4.3. INTER-PROCEDURAL ALIASES.....	87
4.5. SUMMARY	89
CHAPTER 5.....	90
THEORETICAL EVALUATION	90
5.1. AN NP-COMPLETE ARGUMENT	90
5.2. PRACTICAL COMPLEXITY MEASURES	94
5.3. INTRA-PROCEDURAL ALIAS ANALYSIS	97
5.3.1. PROCESSING POINTER ASSIGNMENT STATEMENTS.....	98
5.3.2. ANALYZING THE COMBINATORIAL EFFECT	101
5.4. INTER-PROCEDURAL ALIAS ANALYSIS	105
5.5. ITERATIONS OVER THE PCG.....	111
5.5. PUTTING TOGETHER THE COMPLEXITY ARGUMENT	113
CHAPTER 6.....	115
IMPLEMENTATION	115
6.1 DESIGN GOALS	116
6.2 THE SUIF COMPILER.....	117
6.3 IMPLEMENTATION IN SUIF PASSES	118
6.3.1 FUNCTION SIGNATURE UNIFICATION.....	119
6.3.2 CONTROL-FLOW FLATTENING.....	124
6.3.3 INTER-PROCEDURAL ALIAS INSERTION.....	125
6.4 PREPROCESSING	129

6.4.1	VARIABLE DECLARATION MOTION.....	130
6.4.2	FUNCTION SIGNATURE PREPROCESSING.....	130
6.5	CORRECTNESS DISCUSSION	131
6.6	IMPLICATION ON DEBUGGING	132
CHAPTER 7.....	133	
EMPIRICAL EVALUATION	133	
7.1. EVALUATION CRITERIA.....	133	
7.2. PERFORMANCE OVERHEAD	135	
7.2.1.	IMPACT OF BRANCH TRANSFORMATIONS ON PERFORMANCE	136
7.2.2.	IMPACT OF PERVASIVE ALIASING.....	141
7.2.3.	IMPACT OF FUNCTION CALL STRUCTURE MODIFICATIONS.....	144
7.3. PERFORMANCE AND PRECISION OF STATIC ANALYSIS TOOLS	146	
7.3.1.	EXPERIENCE WITH NPIC	147
7.3.2.	EXPERIENCE WITH PAF.....	148
7.4. SUMMARY	149	
CHAPTER 8.....	151	
TRANSFORMATIONS AND DYNAMIC ANALYSIS	151	
8.1. THE FUNDAMENTALS OF DYNAMIC ANALYSIS	152	
8.2. THE EFFICIENCY OF PROFILING AND TRACING	154	
8.3. EXECUTION PROFILING.....	160	
8.4. PROGRAM TRACING	162	
8.4.1	STATE INFLATION.....	163
8.5. BLACKBOX ANALYSIS AND STATE INFLATION	165	
8.6. SUMMARY	166	
CHAPTER 9.....	167	
REVISIT THE BIG PICTURE	167	
9.1. RECAPPING THE PROBLEM: EXTENDING THE TRUST BOUNDARY OF THE NETWORK SURVIVABILITY ARCHITECTURE.....	167	
9.2. A SYSTEM-LEVEL SOLUTION FOR A SYSTEM PROBLEM	169	
9.3. THE OTHER PIECES IN THE PUZZLE.....	172	
9.4. SUMMARY	177	
CHAPTER 10	178	
RELATED WORK	178	
10.1. CODE OBFUSCATION WORK	178	
10.2. SECURITY OF MOBILE AGENTS	181	
10.2.1.	MOBILE CRYPTOGRAPHY.....	181
10.2.2.	TIME-LIMITED BLACKBOX SECURITY	182

10.2.3.	SERVER REPLICATION.....	182
10.3.	TAMPER RESISTANT SOFTWARE.....	183
10.4.	OTHER RELATED WORK.....	184
10.5.	SUMMARY.....	185
CHAPTER 11	186
CONTRIBUTIONS AND CONCLUSION	186
11.1.	CONTRIBUTIONS.....	186
11.2.	WHERE DO WE GO FROM HERE?	188
11.3.	THE FINAL CONCLUSION.....	189
REFERENCES	190

FIGURE 2.1. THE BANKING INFRASTRUCTURE	16
FIGURE 2.2. A CONTROL SYSTEM FOR THE BANKING INFRASTRUCTURE	20
FIGURE 2.3. A CLOSE LOOK AT THE CONTROL SYSTEM	21
FIGURE 3.1. EFFORT SPACE OF PROGRAM ANALYSIS	38
FIGURE 4.1. CREATION OF A ONE-WAY TRANSLATED PROGRAM	55
FIGURE 4.2. AN EXAMPLE WHILE LOOP AND ITS CFG	63
FIGURE 4.3. INDIRECT BRANCHING EXAMPLE.....	65
FIGURE 4.4. DISMANTLING HIGH-LEVEL CONSTRUCTS	66
FIGURE 4.5. TRANSFORM TO INDIRECT CONTROL-TRANSFERS	67
FIGURE 4.6. A FLATTENED CONTROL-FLOW	68
FIGURE 4.7. EXAMPLE ILLUSTRATING INDEX COMPUTATION	72
FIGURE 4.8. INTRODUCING ALIASES THROUGH POINTERS.....	74
FIGURE 4.9. AN EXAMPLE PROGRAM CALL GRAPH	78
FIGURE 4.10. FUNCTION CALL VIA FUNCTION POINTERS.....	80
FIGURE 4.11. EXAMPLE ILLUSTRATING UNIFYING FUNCTION SIGNATURES.....	81
FIGURE 4.12. FUNCTION SIGNATURE MODIFICATION - ORIGINAL CODE SEGMENT	83
FIGURE 4.13. FUNCTION SIGNATURE MODIFICATION – MODIFIED CODE.....	84
FIGURE 4.14. A PCG WITH FALSE EDGES	86
FIGURE 4.15. ALIASING THROUGH SIDE EFFECTS.....	88
FIGURE 5.1. AN EXAMPLE ALIAS RELATION GRAPH.....	95
FIGURE 5.2. POINTER ASSIGNMENT STATEMENT	99
FIGURE 5.3. TRANSFER FUNCTION FOR $PI = QJ$	100
FIGURE 5.4. ALGORITHM FOR DEREFERENCING POINTER VARIABLES	101
FIGURE 5.5. AN EXAMPLE MEET NODE.....	103
FIGURE 5.6. THE FORWARD BINDING AND BACKWARD BINDING PROCESS.....	106
FIGURE 5.7. A FORWARD BINDING ALGORITHM – F CALLS G AT CALL SITE Q	107
FIGURE 5.8. AN EXAMPLE ILLUSTRATING FORWARD AND BACKWARD BINDING.....	107

FIGURE 5.9. A BACKWARD BINDING ALGORITHM	109
FIGURE 5.10. ALGORITHM FOR CONSTRUCTING THE PCG.....	111
FIGURE 6.1. AN EXAMPLE ILLUSTRATING CUTTING PARAMETERS	122
FIGURE 6.2. AN EXAMPLE ILLUSTRATING ADDING PARAMETERS	123
FIGURE 6.3. AN EXAMPLE ILLUSTRATING GLOBAL-TO-LOCAL ALIASES	127
FIGURE 6.4. AN EXAMPLE ILLUSTRATING PARAMETER ALIASES	127
FIGURE 6.5. ALIASING THROUGH FUNCTION CALL SIDE EFFECTS	129
FIGURE 7.1. EXECUTION TIME WITHOUT OPTIMIZATION	137
FIGURE 7.2. EXECUTION TIME WITH OPTIMIZATION	138
FIGURE 7.3. EXECUTION TIME OF THE ORIGINAL PROGRAMS	138
FIGURE 7.4. EXECUTABLE SIZE WITHOUT OPTIMIZATION	139
FIGURE 7.5. EXECUTABLE SIZE WITH OPTIMIZATION.....	140
FIGURE 8.1. A REGULAR CONTROL-FLOW GRAPH IN AND THE FLATTENED VERION	156
FIGURE 8.2. AN EXAMPLE ILLUSTRATING EDGE VS. BLOCK INSTRUMENTATION	157
FIGURE 8.3. EXAMPLE ILLUSTRATING EDGE PROFILING TO IDENTIFY DEAD CODE	160
FIGURE 8.4. EXAMPLE ILLUSTRATING LOOP UNROLLING TO DETER EXECUTION PROFILING.....	161
FIGURE 8.5. EXAMPLE ILLUSTRATING STATE INFLATION	164
FIGURE 9.1. A REVIEW OF THE SURVIVABILITY ARCHITECTURE.....	169
FIGURE 9.2. A PSEUDO-CODE EXAMPLE ILLUSTRATING POSSBILE VERIFICATIONS OF THE EXECUTION ENVIRONMENT	174

Chapter 1

Introduction

This dissertation describes the design, implementation, and analysis of an approach to the problem of software security in untrustworthy execution environments. This approach is novel in that it combines the principles of diversity and information complexity to prevent analysis and tampering of software components. The attractive properties of the solution approach include platform independence, ease of use, and flexibility with respect to performance trade-offs. Furthermore, this is the first software protection mechanism with demonstrable security strength supported by both theoretical and empirical complexity measures.

1.1. Motivation

The advent of computer networks has given rise to new computational environments and computational models. Remote execution, distributed computing, and code mobility are no longer unfamiliar terms. These modern computational models bring great flexibility and new promises to the world of computing. However, accompanying the expanded potential comes a set of security implications that were not present when computation was carried out largely on local, stand-alone machines.

First, when programs are executed in a remote environment, assurance must be provided that the execution environment will be protected from any malicious behavior of the incoming program. This is commonly known as the *malicious code* problem [32] [56].

A related problem is the protection of code from malicious execution environments. Since the environment is responsible for the program's execution, there appears to be precious little the program can do to protect itself from disclosure, tampering and incorrect execution. Protecting code from untrustworthy environments is by far the more difficult security problem. It is known as the *malicious host* problem [16][32].

Much research has been devoted to the malicious code problem, including proof-carrying code [65], policy-directed code safety [31], artificial playgrounds for mobile agents [81] and many others [10][26][36]. The malicious host problem has not been investigated with nearly the same rigor and intensity. Despite the existence of partial solutions [74][78] and the effort of some preliminary investigations [40][41], the present defense techniques against malicious hosts have remained largely ad hoc and lack a theoretic and algorithmic underpinning. As new applications such as Legion [25], Parabon [34] and the like exploiting the potential of distributed computing become more pervasive, the ability to run trusted code on potentially untrustworthy hosts is more pressing than ever.

1.2. The Malicious Host Problem

This work considers the malicious host problem within the context of a particular application—network monitoring and management systems. Large computer networks such as national infrastructure systems are often the carriers of mission-critical

applications whose successful operation is of paramount importance. To ensure the survival of the critical applications, the network is often monitored and managed by the likes of a network management system [39][75][77].

A network monitoring and management system monitors the state of the entire network, performs network status analysis, and—if necessary—initiates real-time system reconfiguration. Such a system is essentially a distributed application. The front end of the application is composed of *probe programs*, which execute on network hosts and collect local information. There is a clear need for running the probes as securely and reliably as possible. Since the underlying hosts on which these probes execute are not necessarily trustworthy (hence, in part, the need for monitoring), this problem falls into the malicious host category.

An in-depth look at the characteristics of these network management schemes is presented in Chapter 2. In the remaining part of this chapter, I examine the malicious host problem at an abstract level.

1.2.1. Different Facets of the Malicious Host Problem

There are two distinct classes of applications for which the issue of malicious hosts is relevant. In the first class, remote execution occurs solely for the purpose of resource utilization. This class of applications can be described as follows:

Alice (the target program) has an algorithm f , a program P that implements f , and some data x . Bob (the host) has the resources to execute P . Alice wishes to use Bob's

resources to compute $f(x)$ (or Bob wishes to execute $f(x)$ for Alice). In this case, Alice wants some assurance that $f(x)$ is correctly executed. Additionally, it may be desirable to keep the algorithm f and data x secret from Bob.

The second class of applications (the network management application belongs in this class) can be described as follows:

Alice has an algorithm f and a program P that implements f . Bob has the data x . Alice wishes Bob to execute P on input x . Alice wants some assurance that $f(x)$ is correctly executed, and in most cases, Alice wants to keep the algorithm f secret from Bob.

In the second class of applications, a correct execution of $f(x)$ depends on the integrity of f as well as the input data x . It is clear that handling the malicious host problem is significantly more difficult for the second class of applications due to the input data problem—Bob can potentially lie about x . It should also be clear that any solution to the second class of applications also solves the malicious host problem for the first class of applications.

This dissertation deals with the malicious host problem in the second category, but has close associations with the first. For the purpose of discussion, let me put aside the input data problem for a moment (I revisit this issue in Section 1.2.2), and concentrate on the protection of code integrity and algorithm privacy.

In order to execute the target program, the host must have access to the program code and states. (It has been proposed that some limited forms of functions can be executed in an

encrypted form [74], but the scheme does not generalize to general-purpose software)

The host, if malicious, can affect the program in the following ways:

- **Denial-of-service:** Bob can deny Alice's execution altogether.
- **Algorithm-privacy Attack:** Bob can analyze Alice's program and steal algorithm f .
- **Execution-integrity Attack:** Bob can modify Alice's code or data in such a way that the result $f(x)$ is invalid. Note that this attack is different from denial of service. In this case, the result $f(x)$ falls in the range of acceptable outputs, however, it is not an acceptable output given x . Ramifications of such attacks may vary depending on the context of the application. In some cases, an invalid $f(x)$ simply constitutes a computational error, while in other application domains the consequence may be far more severe. In the case of network management systems, carefully miscalculated values of $f(x)$ may have a ripple effect on the decision making of the management scheme. This class of attacks is identified as *intelligent tampering* attacks (more on this subject is discussed in Chapter 2).

This work does not deal with host denial-of-service attacks. The system model, which I elaborate in the next chapter, assumes ready detection of denial of service. The solution mechanism described in this dissertation therefore is specially tailored to enforcing algorithm privacy and execution integrity.

To some extent, the issue of algorithm secrecy and execution integrity can be dealt with independently of the input x . For example, it is possible to develop a mechanism that, given a set of x 's and $f(x)$'s, the malicious host cannot easily produce $f(y)$ on y without

actually executing the program (algorithm privacy achieved). On the other hand, execution integrity can be viewed as a function of algorithm privacy; that is, if the algorithm of computation is not disclosed, tampering with the execution will be essentially impossible (except in the case of random tampering).

Consider an example of computing the function

$$f(x) \equiv (ax + c) \bmod m$$

where a , c , x , m are all integers. When the values of a , c and m are 2, 9 and 7, respectively, the output

$$f(x) \equiv (2x + 9) \bmod 7 \tag{1}$$

expands the entire set of positive integers. When a , c and m have the values 4, 10 and 8, respectively, the set of output integers

$$f(x) \equiv (4x + 10) \bmod 8 \tag{2}$$

include only those integers that are congruent to 2 modulo 7 or 6 modulo 7, regardless of what x might be. In other words, the value of $f(x)$ can be represented as:

$$f(x) = \{y \mid y = 7i + 2 \text{ or } y = 7i + 6\}$$

where i is a random integer.

Assuming that there is a way to hide the value of a , c and m such that they are not immediately obvious without extensive code analysis, tampering with the computation in

the case of (2) can result in a value that does not fall in the correct output range.

1.2.2. The Input-spoofing problem

The algorithm-secrecy and execution-integrity issues must be considered in conjunction with the input data when Bob has no interest to supply a genuine x , and when execution with a spurious x is detrimental to the law-abiding entities involved even when neither the algorithm secrecy nor execution integrity is compromised. This is called the *input-spoofing* problem.

Consider an example where Bob executes a test version of Alice's software P , and x is Bob's local clock reading. Alice has in her best interest to stop the execution of P when the designated test period expires while Bob may want to supply a spurious x in order to continue to use the test software for free.

In this scenario, Bob can easily get what he wants by manipulating his local clock value x . This is not particularly difficult because where x (the clock value) is stored and how it is retrieved are easily identifiable. Consider for a moment that the checking of the clock value is not performed as a single, distinguishable operation. Rather, it is part of the regular algorithm execution. For example, assume the execution of P requires the following interaction protocol with a trusted server:

```

Bob → Alice:    { Alice, Bob, data field D, timestamp}
Alice:          if ( mytimestamp - Bob's timestamp > threshold value )
                  Do not send anything back
                  else
                    Alice → Bob {Bob, Alice, Computed data D', mytimestamp}

```

And consider the following steps as part of program P :

```

S1:    v1 = getSystemClockValue ( )
        v2 = getSystemClockValue ( )
S2:    v3 = getSystemClockValue ( )
        ...
S3:    conduct the above protocols with Alice using timestamp v2
S4:    if (v1 > storedDownloadTime + test period)
        stop execution
        else continue

```

In the above pseudo-code example, code blocks S1 and S2 read the system clock value multiple times. One of these values is used later in the program (in block S3) in the protocol messages to Alice, and Alice performs a freshness check on Bob's timestamp. Another value is used at other points in a comparison operation to check the expiration of the test period. In order to get his data computed by Alice, Bob must demonstrate the freshness of messages by supplying a real time stamp. However, in order to continue to use the test program beyond the expiration period, Bob must supply a stale time stamp for comparison.

This all boils down to one essential point: Bob must perform a program analysis on P to determine which variable reads in S1 and S2 will be used for communicating with Alice and which will be used for staleness comparison (note Bob may not know how many variable reads there are or how the values are used later). This is almost equivalent to performing an analysis to deduce the algorithm of P , which I have identified earlier as the algorithm-secrecy problem. Note how the modified version of program P is much harder to spoof than a simple check of the local system clock.

The point of this example is that if spoofing input x requires solving the algorithm-

secrecy or execution-integrity problem, then techniques to ensure the latter can be used to counteract input spoofing. However, there are applications where this is not possible. For those applications, protection of only algorithm secrecy and execution integrity will not be effective since the malicious host can bypass the protection by supplying a spurious input x . This thesis deals with the first type of applications, for which it is possible to construct algorithms in such a way that input spoofing hinges on breaking the algorithm privacy or execution integrity.

1.3. Design Goals

Based on the discussion of the malicious host problem, the following design goals are identified for the solution.

- **Host architecture independence.** The solution mechanism must not depend on any architecture-specific features. In other words, the solution must be appropriate for use with a wide variety of common host architectures. However, in some cases the program we seek to protect might make use of platform-specific functionality, in which case the protection mechanism may be inherently platform-dependent.
- **High-level language independence.** The basic principles of the mechanism must not rely on language-specific attributes. When appropriate, language-specific techniques can be exploited to implement an underlying concept that is source-independent.
- **Efficiency.** The mechanism must not result in an unacceptable performance slow-down on the part of the target program. What is considered acceptable and what is unacceptable may vary from application to application.

- **Ease of Use.** Ideally, the solution mechanism should be fully transparent to application programmers and users of the system. However, in some cases, the programmer's input helps to identify the appropriate level of trade-off between performance and protection. The mechanism should provide the flexibility for programmers to customize the set of code transformations that are appropriate for their applications. Beyond that, it should be fully automatic.

1.4. Solution Overview

The solution described in this document is comprised of several techniques that collectively contribute to the goal of software protection. The premise is that software tampering requires program-specific information. To acquire this information, some form of program analysis is needed. The solution techniques, therefore, aim to increase the complexity of the program analysis.

The core approach is comprised of two principles: *information diversity* and *complexity*. Information diversity is achieved by the use of specific forms of design diversity while information complexity is realized with functionality-preserving code transformations to make the program less analyzable and therefore more difficult to manipulate. The code transformations are implemented in a *One-way Translation* compiler that produces diverse and obfuscated programs.

The compiler's most basic code modifications aim to degenerate the target program's static control-flow. This is accomplished by changing the statically determinable control transfers into dynamic branching statements. The purpose of these transformations is to

hinder the most basic form of program analysis—building the static control-flow graph of the program. Nearly all forms of static analysis rely on the program control flow being statically determinable [35][59]. Taking this basic assumption away cripples the fundamental premise of many static analysis techniques.

Furthermore, difficulties in data-flow analysis are introduced by the creation of pervasive aliasing throughout the program. Alias resolution is known to affect many data-flow problems, and precise alias resolution has been shown to be inherently difficult.

These program modifications introduce two fundamentally interdependent difficulties. First, program control-flow is made data-dependent. Resolution of control-flow in such a form requires data-flow analysis. Second, data-flow analysis is made difficult by withholding the static control-flow information, and by the introduction of pervasive aliasing.

Each of these transforms is performed automatically by the One-way Translation compiler, implemented as a source-code translating extension of a traditional compiler. Experiments are conducted to evaluate the efficacy of such a scheme. In addition, a theoretical argument is supplied to attest to the security strength and benefits of the said transforms.

The thesis of this dissertation is that static analysis of programs can be deterred, under a certain set of assumptions, with acceptable cost and demonstrable strength. Additionally, some of the techniques designed to deter static analysis extend nicely to constructing potential countermeasures for dynamic analysis.

1.5. Dissertation Organization

This dissertation is structured as follows: In Chapter 2, I discuss background material and the problem context. Chapter 3 describes the underlying philosophy and the solution framework. In Chapter 4, I describe the One-way Translation techniques to deter static analysis. In Chapter 5, I discuss the theoretical evaluation of the proposed techniques. A set of empirical evaluations on the techniques is presented in Chapter 6. In Chapter 7, I describe an implementation of the One-way Translation scheme in an ANSI C compiler. Chapter 8 switches gears and presents a preliminary investigation into the issue of defending against dynamic analysis. Chapter 9 examines the individual techniques within the system context and explains how they collectively form a system solution to the software protection problem. Chapter 10 discusses related work. Finally, in Chapter 11, I conclude the dissertation with a discussion of future research directions.

Chapter 2

Background and Problem Context

In this chapter, I establish the problem context and architectural environment for the dissertation.

This research is initially inspired by the need to secure a survivability architecture for critical infrastructure systems. Survivability architectures represent a class of network management schemes retrofitted onto an underlying system to ensure its continuing services in the event of errors, failures, and malicious attacks. It is of extreme importance that the architecture itself be adequately protected. For the type of infrastructure system that is of concern, threats are likely to be many and the stakes of security breaches are likely to be high. Malicious-host problems, for example, represent a significant class of security threats against the survival of the management architecture itself.

The characteristics of the survivability architecture and the underlying system shape the specifics of this research and the solution mechanism. This chapter describes those characteristics and investigates the types of security threats and intruder capabilities with respect to the malicious-host problem. Finally, I conclude this chapter with a discussion of the fundamental challenges toward achieving secure execution of trusted programs in

potentially hostile environments.

2.1. Critical Infrastructure Systems

Today's critical infrastructure systems are large, complex information systems. Often distributed across wide geographic regions, these systems are different from the fully-connected, open nature of the Internet: they are typically large-scale private networks, and they are typically connected in a point-to-point fashion. For clarity of illustration, the discussions in this chapter are framed in the banking and finance infrastructure, although many of the characteristics are general and are common for other infrastructure systems.

The following sections describe the essential characteristics of a national infrastructure system and the ways in which they impact the design of a survivability mechanism as well as the protection of the mechanism itself.

2.1.1. Large scale

Figure 2.1 depicts a high-level view of the banking and finance infrastructure with respect to the check-clearing functionality. The internal network within an organization is represented by solid lines and interconnections between different financial institutions are depicted in dashed lines.

Assume a check is deposited at a Citibank branch where the beneficiary customer banks, and the account where the check intends to draw funds resides with Chase Manhattan. From the moment the check is deposited to the final settling of the accounts, the entire

operation involves the Citibank branch office, the internal system within Citibank (including the Citibank processing center), the extranet between Citibank and the Federal Reserve Check-clearing center, the Federal Reserve internal System, extranet between Federal Reserve and Chase (where the writer of the check banks) and the Chase internal system.

This scaled-down picture of the banking infrastructure in Figure 2.1 consists of a large number of geographically dispersed computing nodes that belong to different authority domains and organizational structures. The entire banking infrastructure minimally contains tens of thousands of computing hosts connected via a large network. These hosts and their interconnections provide the computation, data storage, and communication that are needed to provide services, as in the case of the check-clearing function.

The exact topology of the infrastructure network is not important, although such networks are often not fully connected. For example, there is usually no need for every teller's desktop to be connected to the central bank processing center—they might be connected to a branch server which serves as the gateway to the backend processing center. Similarly, interconnections between different organizations may be sparse (as in the case between Citibank and Chase).

2.1.2. Heterogeneity

Another defining characteristic of critical infrastructure systems is the degree of heterogeneity. First, the system itself is typically composed of subsystems with diverse operational environments and policies. For example, Citibank's internal system is sure to

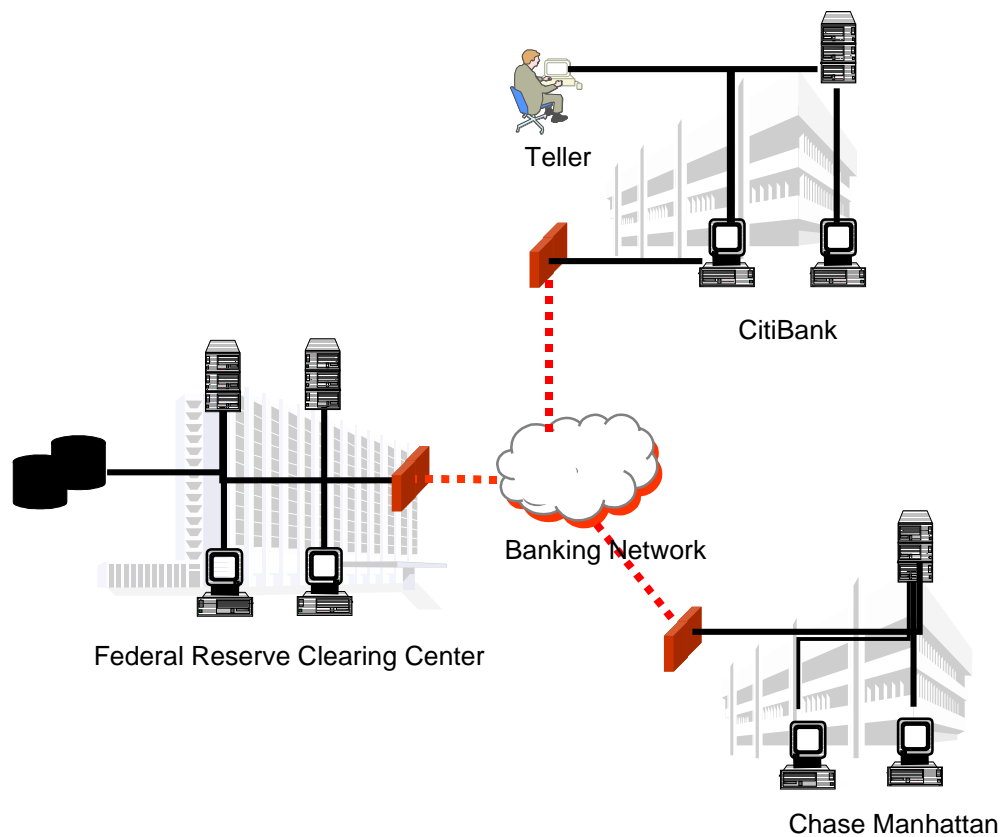


Figure 2.1. The banking infrastructure

have a different layout and be managed differently than the one in Chase or the Federal Reserve. From a security standpoint, this degree of heterogeneity implies incongruity in the policies and mechanisms employed; some sites will be more easily penetrated than others. A universal protection mechanism across the board is clearly impractical and perhaps infeasible.

Second, critical applications usually do not consist of only similar programs performing similar functions. Rather, the programs running on different hosts often serve distinct purposes, and they must cooperate in some form of sequential processing in order to

provide desired services.

Consider again the example of check clearing in Figure 2.1, the bank teller's desktop initiates a deposit request with the destination bank's routing number and account number. This request is then processed and queued at the Citibank processing center. A bundled request is sent from the Citibank processing center to the Federal Reserve clearing house, where it settles the accounts between two banks by transferring funds in the Federal Reserve account database. The final results are sent back to individual banks before they settle their member accounts. In this example it is clear that the software running on the teller's desktop, the program executing at the bank's processing center and the software the Federal Reserve system employs perform very different functions, but they all have to operate in order for the check-clearing process to operate smoothly.

A direct consequence of this phenomenon is that functionality is not uniformly distributed across the system; some computing nodes provide services that are more important than the services provided by others. Since system survivability is concerned with the survival of critical functionality on the system-level rather than the survival of individual hosts or subsystems, it is therefore necessary to correlate and integrate information from many different sources to obtain system-level knowledge. This implies complexity in enforcing network-wide management policies as well as inherent difficulties in securing the management architecture itself.

2.1.3. Extensive use of COTS components

The software employed in infrastructure systems tends to be large and to make extensive

use of both legacy and Commercial-Off-The-Shelf (COTS) components. This characteristic means that any mechanism retrofitted to the management of the system must consider the impact of COTS and legacy software, especially the uncertainty factor they bring in in terms of security and reliability.

The extensive use of COTS and legacy software also implies that the characteristics of the operational environment will be determined largely by the applications—retrofitting such systems with survivability mechanisms is particularly difficult, for it is not practical to mandate drastic changes to existing system architectures (such as demanding changes to the network topology) or software (such as demanding that the applications be rewritten).

2.2. The Survivability Architecture

It is in the context just described that survivability mechanisms have been proposed [45][62][77]. In this section I describe a survivability architecture developed at the University of Virginia. My work henceforth is carried out in the context of this architecture.

2.2.1. A Control System Construct

A critical component in the survivability architecture is the *control system* construct. Introduced as an external entity to manage the infrastructure system, the control system operates in parallel with the infrastructure system. Figure 2.2 depicts a simple control system for the banking infrastructure. In this picture, the control system is composed of three groups of computing nodes (*A*, *B* and *C*) in the upper left corner.

The control system's primary function is real-time monitoring and management of the system operation. Because of the scale and the complexity of the underlying system, each control server is responsible for managing a portion of the network. In Figure 2.2, the control server *A* manages two network points in the Federal Reserve System, *B* manages the Citibank system and *C* manages both Citibank and Chase Manhattan.

In addition to the control servers, the control system also includes a set of monitoring and actuating programs (hereafter referred to as *probes*) that are responsible for collecting local information and carrying out reconfiguration commands. Figure 2.3 shows a more detailed monitor-and-control architecture.

In Figure 2.2, each control server has a set of probes under its control. The control servers execute analysis algorithms based on the information gathered by the probes. If necessary, the servers issue reconfiguration commands to the probes, which then carry out the necessary local actions. An example is illustrated in Figure 2.3. In this example, control server *A* communicates with the probe program (represented by the red dot) on a Federal Reserve host. If the probe reports that this particular clearing center is experiencing delays or other capacity errors, *A* forwards this information to *B* and *C*, which in turn instruct probes under their control to reroute requests to a different site.

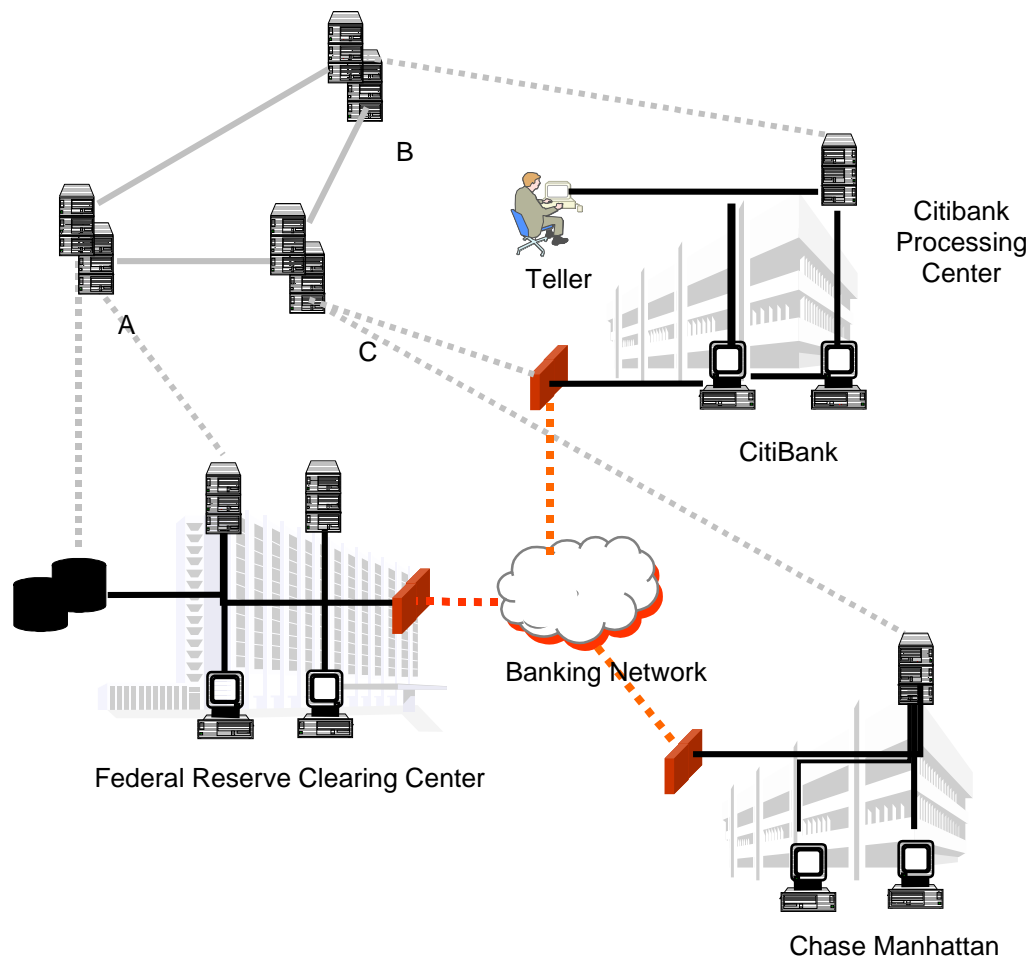


Figure 2.2. A control system for the banking infrastructure

Note that the control hosts are interconnected—they do not need to be fully connected, although sparse connections will complicate data sharing. Also note that the control servers are physically separate from the infrastructure system. There are several advantages to such a design. First, executing control algorithms locally on application hosts can be a significant resource drain—running them exclusively on the control servers is beneficial for efficiency reasons. Second, there tend to be fewer numbers of control

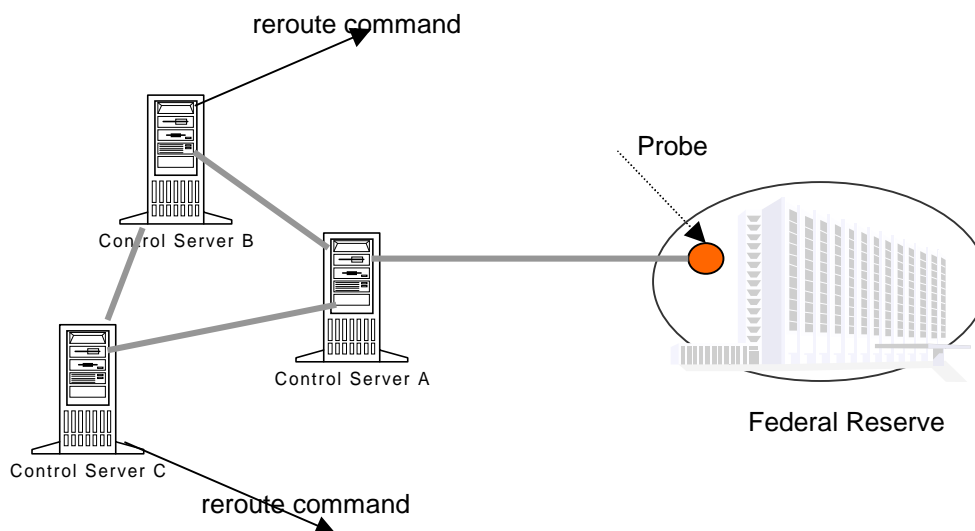


Figure 2.3. A close look at the control system

servers than application hosts, therefore it is possible to have dedicated control servers, and to secure and configure them independently of the remainder of the system. This allows rigorously controlled execution environments for the control servers, and consequently, enhanced server security and assurance.

2.2.2. The Network Probe Programs

In this architecture, the probe programs reside on the infrastructure hosts. They perform two primary functions.

- Collect raw system state information and generate status reports that are forwarded to the control servers for processing.
- Process reconfiguration commands and initiate appropriate local actions. Examples of dynamic reconfigurations include shutting down network connections, dynamic

process migration, active load balancing, etc. A more detailed discussion on dynamic reconfiguration can be found in Elder *et al* [27].

The probe programs are initially dispatched from the trusted control servers. While executing remotely, they keep a high level of interaction with the control servers. Communications between the probes and the control servers follow prescribed protocols that may include timing mechanisms (e.g. timeouts), predetermined data formats, and designated hand-shaking sequences. It is important to note that adhering to the interfacing protocol is considered expected behavior of the probe program.

2.3. The Malicious Host Problem and Other Security Issues

The survivability architecture, as described above, gives rise to a series of security concerns. These security concerns can be grouped into three loosely-defined categories.

- **Protection of the control servers.** The control servers execute the control analysis algorithms, and therefore are at the heart of the whole control mechanism. The servers and the control software must be protected from malicious attacks.
- **Protection of communication.** Monitoring and control communications occur over the network. This network traffic needs to be authenticated, and protected against tampering, eavesdropping, traffic insertion and deletion attacks.
- **Protection of the probes.** The probe process needs to be protected so that the data gathering and the actuating part of the mechanism can be trusted.

These protection issues are closely related; that is, techniques used to secure one part of the mechanism may impact the protection of other parts. For example, cryptographic methods are often used to protect network communication. However, cryptography is not sufficient if the cryptographic keys stored on the communicating hosts are not adequately protected.

Because of the multitude and the complexity of the security issues, the following four simplifying assumptions are made to help focus the task of this research:

- **Trustworthy control servers.** The control servers, physically separated from the rest of the system, are *dedicated* to performing the algorithmic part of the control mechanism. Securing the control servers requires careful system administration (e.g., exercising strict control on what software is allowed to execute on the servers) and good physical security (e.g., restricted access). For the purpose of this discussion, I assume that the control servers are secure and trustworthy.
- **Secure communications.** I assume that all network communications occur over authenticated channels equipped with cryptographic protections, and that the cryptographic techniques are sufficient, in this context, to protect the freshness, integrity and privacy of the communication¹. It should be noted that the use of

¹ This only assumes communication security. It does not assume that the cryptographic keys are necessarily secure at both ends.

cryptography does not necessarily prevent denial-of-service attacks—network traffic can be deleted and communication channels can be jammed.

- **Trusted survivability software.** Many real-world security problems arise not because of flaws or oversight in the design of the security mechanism, but rather because of errors in the software implementation. It is not my objective, in this work, to address security flaws of the latter kind. Throughout this work, I assume that the survivability software is trusted in such a way that it operates as expected (if not compromised), and it does not contain any malicious flaws that will lead to a compromise of the survivability mechanism. This assumption allows my work to focus on security threats from the environment as well as general protection issues that are not implementation specific.
- **Un-enhanced application hosts.** The infrastructure system is a network enterprise with tens of thousands of computing hosts. It is virtually impossible to enhance the security of every application host. In the context of this work, I assume that application hosts are running in their normal operational mode—no special enhancement other than the security mechanisms that are already in place for the applications. This assumption has many implications. In particular, it implies that the application software and the infrastructure hosts might contain security flaws, and that they might be vulnerable to a variety of security attacks.

The assumption of secure control servers and secure network communications brings focus to the *protection of the probe programs running on untrustworthy application hosts*. The task of probing and actuating constitute the basis of analysis and system

management. Therefore it is of paramount importance that the execution of these operations be reliable and trustworthy. Since the probes (or some essential parts of them) must reside on the monitored host to perform their functions, and since the monitored hosts are assumed vulnerable to security attacks, direct corruption of the probe programs is a distinct possibility. Note that this falls under the malicious-host category identified in Chapter 1.

Like any other security mechanisms, solutions to this problem must be considered in terms of the threats and attack scenarios that are likely to be present. In the following sections, I elaborate in depth the types of security threats and attacker capabilities that I will consider in this work. The series of solutions, presented in later chapters, represent targeted responses to those threats.

2.4. Threats and Attack Scenarios

The discussion in this section is framed within the context of the survivability architecture. However, many issues are of general concern, and therefore are applicable in other contexts where there is a need to protect software components from malicious hosts. I should stress that the discussion here is concerned with protection of the survivability mechanism. General security threats aimed at the underlying infrastructure system itself are not considered in this discussion.

2.4.1. Denial-of-service attacks

By denial-of-service, I mean the termination or obstruction of the probe execution on an

application host. This attack is possible if the perpetrator has the appropriate privileges to stop the execution, or if they are able to monopolize system resources. If successful, denial-of-service attacks would disable the monitoring and management capability on the target host.

A similar effect can be achieved by obstructing communications between the probes and the control servers. For this the perpetrator might flood the network with unwanted traffic or simply delete communication packets if they have access to routers or gateway machines.

Denial-of-service attacks in general are impossible to prevent. Detection, however, is often easier. In this system, the probe program maintains a high level of interaction with the trusted server. A *non-action* for an extended period of time, or messages not following the prescribed protocols, signifies abnormal behavior.

In practice, it might not always be possible to discern whether a detected non-action is due to a host-based denial-of-service or disruption on the communication line. For the present discussion, it is sufficient to note that while a widespread denial-of-service attack would defeat the survivability mechanism and render the entire system unmanageable, isolated local occurrences of such attacks are within the class of errors the survivability mechanism is designed to manage (consider the case of a host crash). It is therefore the designer's responsibility to choose whatever mechanisms deemed necessary to react to such a situation. For this reason, I do not consider denial-of-service attacks in this work.

2.4.2. Intelligent Tampering and Impersonation

This category refers to attacks for which an intruder attempts to spoof the trusted servers by impersonating or tampering with the legitimate probe program.

Intelligent Tampering. Intelligent tampering refers to scenarios in which the intruder modifies the program or data in some specific way that allows the program to continue to operate in a seemingly unaffected manner, but on corrupted data or state. For example, overwriting data buffers with data of the correct format but different values is an example of intelligent tampering. Using this definition, tampering with the software in a random way (e.g. overwriting random bits in the memory) does not constitute an intelligent tampering attack. It may result in denial-of-service however, since the tampered program or data can cause execution to fail.

Impersonation. An impersonation attack is similar to intelligent tampering in that the attacker seeks to establish a rogue version of the legitimate program. Impersonation, however, is primarily concerned with emulating the observable behavior of the original program, while intelligent tampering often involve direct modifications of the internal specifics of the program such as its code or data.

This dissertation is primarily interested in defending against intelligent tampering and impersonation attacks, because of the following reasons.

- These attacks are the most difficult to detect. Unlike denial-of-service attacks, the result of an intelligent tampering or impersonation attack is not always obvious; if the attacker has detailed knowledge of what the software is supposed to do and the

appropriate privileges to instantiate a malicious copy, he or she can replace the original program and make the replacement virtually undetectable.

- Such attacks have the potential to inflict substantial damage. For example, a carefully coordinated attack on a selected set of monitors would cause the control mechanism to reach an inaccurate view of the state of the network and arrive at an erroneous reactive decision that may lead to deterioration of services. In this case, the intruder can manipulate the control mechanism to perform malicious tasks on a network-wide scale. This is far more dangerous than a typical network intrusion whereby an adversary may be able to compromise a node (albeit possibly an important one) but further compromise beyond that usually requires additional effort or resources.

The discussion on malicious hosts in Chapter 1 identifies three facets to the problem: Algorithm secrecy, execution integrity and input spoofing. All of these are relevant in this context. Note that intelligent tampering is an attack against the execution integrity of the target program. Both intelligent tampering and impersonation require knowledge about the target program—either about the external behavior or the internal specifics of the program. Acquiring this knowledge constitutes a compromise (or at least partial compromise) of the algorithm privacy of the target program. For example, consider a scenario in which the intruder's objective is to forge probe messages to the control server. Assume all messages are signed with the probe program's private key. The knowledge the intruder must acquire, among others, is the private key of the probe program. In order to discover the location or the content of the key, the intruder might need to learn, for example, what cryptographic algorithm is in use and how the key is used in the implementation.

The discussion in Chapter 1 also argued that the input-data problem is solvable only if feeding malicious input data to the program requires breaking the algorithm privacy or the execution integrity first. This becomes one of the solution criteria that is discussed in the next chapter.

The important point here is that in order to launch an intelligent tampering and impersonation attack, the intruder must obtain the knowledge necessary to do so. The ways in which this knowledge can be acquired depend heavily on the intruder's capability, and this is the topic of discussion for Section 2.5.

2.5. Capabilities of Intruders

Three categories of intruders, classified by their respective capabilities, are likely to be present in the context of the survivability architecture. Listed in the order of increasing level of capability, they are: *Network Intruders*, *Malicious Insiders*, and *Privileged Users*. This classification is similar to the one used by Aucsmith in his Integrity Verification Kernel (IVK) work [4]. This section discusses each category within the context of the survivability architecture described earlier.

2.5.1. Network Intruders

This category refers to intruders who do not have direct access to the host where the monitor program executes. These intruders access the system through network entry points, and can eavesdrop on the communication line and insert and delete network traffic.

A typical network intruder is bound by communications protocols and other network-based security mechanisms (e.g., firewalls, network access control, etc.). Their mission is to either breach the host security perimeter (i.e., getting in from outside) or interpose between communicating parties by forging or replaying communication messages. This class of intruder can be dealt with by the use of correctly designed and implemented security protocols and proper host administration. This work does not consider network intruders, for they cannot affect host software directly.

It should be noted that network intruders can gain further access by exploiting flaws in communication protocols or network security mechanisms. For example, a successful buffer overrun attack may render more privileges to the intruder as a result. In doing so, a network intruder may become a malicious insider or a privileged user who possesses significantly more powerful capabilities than a typical network intruder.

2.5.2. Malicious Insiders

This category refers to intruders who have control of some program running on the targeted host. These intruders could be legitimate users, or an outsider who has gained illegal access to the host system.

Malicious insiders have access to some system resources, and they can manipulate the programs under their control or introduce Trojan-horse programs to inflict damage to other applications or the underlying host. An example of a malicious insider is someone who has obtained the password of other users and is now able to read and write the private data and programs in the compromised accounts.

Malicious insiders are intruders without the “root privilege”. Actions of a malicious insider can be greatly limited by the use of properly designed access control mechanisms, competent intrusion detection tools and careful administration. At worst, intruders in this category can cause denials of service or instantiate malicious software such as virus or Trojan-horse programs to effect damage to the host or other programs.

However, actions of malicious insiders do not directly undermine the security of the host system (e.g. they generally do not compromise the operating system). For the purpose of this discussion, I assume that malicious insiders are still bound by the operating system and its security mechanisms. To defeat malicious insiders, this work adopts the principle of diversity to reduce software uniformity, which is often the cause of successful virus or Trojan-horse attacks [33]. More on software diversity is discussed in chapter 3.

2.5.3. Privileged Users

Adversaries in this category have direct access to the host on which the target program is running. Specifically, they may possess the following privileges:

- Access to private memory of other user or system processes
- Access to source code of the target program
- The ability to introduce and execute random software on the host
- The ability to manipulate and replace system software

Read access to the host memory implies that the adversary can obtain the binary image of

a loaded executable. That includes code as well as data associated with it. Write access gives the perpetrator the ability to modify raw memory bits.

Out-of-bound avenues exist for a determined intruder to acquire a copy of the program source code. This suggests that software protection should not, and cannot, rely on the obscurity of the source program. However, I posit that knowledge of the source program does not necessarily imply a direct compromise or immediate knowledge of the running binary program. That is, an executable generated from a known source, aside from being functionally equivalent to the source, could contain extensive syntactic or semantic differences from the source such that impersonation or intelligent tampering of the running program would still require analysis of the executable. This premise is the cornerstone of much of the dissertation, and I elaborate in Chapter 3 on why the premise stands and how it can be exploited as a basis to devise software protection mechanisms.

The ability to introduce and execute random software implies that the intruder may have access to specialized software analysis tools such as debuggers, decompilers, and system diagnostic utilities. They can perform analysis online such as system diagnostics, or offline such as blackbox testing, execution emulation, and break-point-based debugging.

The ability to manipulate and replace system software suggests that the host security mechanisms such as those provided by the operating system can be compromised potentially. This suggests that any mechanism deployed to protect the software should not depend solely on the authenticity or security of the host operating system. This assumption is of course the most troublesome—once an intruder has compromised the operating system, he or she may have near complete control of the platform, and their actions are

therefore limited only by available resources.

There is, however, one restriction on the intruder capabilities—he or she may not substitute or install hardware on the host system. Altering hardware configurations requires physical access to the host system. It is reasonable to assume that such access is difficult to obtain. This assumption discourages online hardware-aided analysis and allows the possibility of special hardware-based security solutions.

At this level of sophistication, the adversary has access to ample system resources and a great deal of knowledge of how the system works. Security attacks from these adversaries are the most powerful and also the most difficult to defeat. In fact, no security mechanism exists and none could be developed that will provide protection against such adversaries in the absolute sense—there is no solution against perpetrators with unbounded resources.

What I aim to do in this work is to:

- Increase the technical difficulty to deter security attacks by malicious insiders and privileged users
- Understand and provide a theoretical basis to determine what benefits each protection mechanism affords in order to make informed decisions.

2.6. Fundamental Challenges

Many difficulties exist in dealing with the malicious-host problem in the context just described. In this section, I outline two fundamental challenges that I believe are key obstacles to software protection. The solution framework detailed in the next chapter

presents a suite of approaches that collectively constitute a comprehensive solution to these challenges.

2.6.1. Verification of Execution Results

Ensuring execution integrity of programs requires that there exist some means to verify the result of the execution. However, the means through which the verification can be performed may not be readily available.

Consider a program whose purpose is to factor a large number. This program is dispatched to execute on a remote machine. In this case, the result of the computation can be easily verified. Hence any tampering of the program can be detected with a high-degree of confidence.

The same is not necessarily true when execution of the program depends on local states, in which case the program execution integrity and input from the local state are both factors that need to be considered². It is clear that results of the execution are not verifiable if local input is required and the input cannot be authenticated in some way.

In the context of the survivability architecture, the probe program reports sensing values to the control server. Verifying whether the sensing value indeed represents the system status is a fundamental challenge. Whilst an unencrypted heartbeat message can be easily

² This is the input-data problem identified in Chapter 1.

faked, a more sophisticated reporting mechanism would require the ability to hide a secret in the software from its hosting system, a task that is extremely difficult.

2.6.2. Finite State Space Facilitates Program Analysis

In theory, programs can have an unbounded number of states. However, the reality is that normal programs spend most of their time in a limited state space. This implies that given enough time and resources, an intruder will be able to learn, through program analysis, enough relevant information to understand the program behavior.

The second challenge arises because of finite state spaces. In other words, how can a finite amount of information withstand analysis with bountiful resources?

2.7. Summary

The discussion in this chapter recasts the malicious-host problem in the survivability architecture context. The fundamental challenges and the threat scenarios are the topic of discussion of the ensuing chapters, which detail a solution framework to tackle the software security problem in malicious environments.

Chapter 3

The Solution Framework

This chapter presents an overview of a solution to the problem of software protection in malicious environments. The discussion in Chapter 2 posited that program-specific information is needed for intelligent tampering or impersonation. This information represents the key to compromising the algorithm secrecy or the execution integrity of the target program.

In order to acquire this program-specific information, I further postulate that some form of program analysis is required. The solution framework described in this chapter builds on the notion that program analysis can be made expensive if the information that an intruder seeks is difficult to procure. This is not a new idea; previously proposed code obfuscation work is based on the same principles [19][20][40][41]. The differences between my work and the code obfuscation studies are as follows:

- This work aims to protect the network application as a whole instead of considering only individual programs.
- This work is supported by both theoretical and empirical complexity measures.

3.1. A Complexity Argument

Program analysis is an operation, and as an operation it has a certain level of complexity associated with it. As discussed in Section 2.5, absolute protection of software in untrustworthy environments is impossible. The objective of this work, therefore, is to devise mechanisms to increase the computational complexity of program analysis, and to measure and understand at a theoretical level the effectiveness of each mechanism.

It should be noted that the notion of *computational complexity* is used in a slightly unconventional sense here. Traditionally, computational complexity is dominated by the order of growth of the algorithm—an operation that is of the order $O(n)$ is considered more efficient and less complex than one that is of $O(n^2)$. The traditional model trivializes the lower order terms of the running-time formula and the constant coefficient of the leading term. In my work, it is not the order of growth alone that is of interest. Instead, this work is concerned with the input size of the problem, the constant coefficients in the running-time formula and the order of growth—any parameter that may potentially affect the *operational complexity* of the analysis procedure. For instance, supposing the time complexity of a program analysis algorithm can be expressed as:

$$an^x + bn^{x-1} + \dots + c,$$

where n is the input size, a practical defense mechanism might aim to raise the order x , the input size n , or the most significant coefficient a .

In the spirit of this operational complexity model, the effort space of program analysis

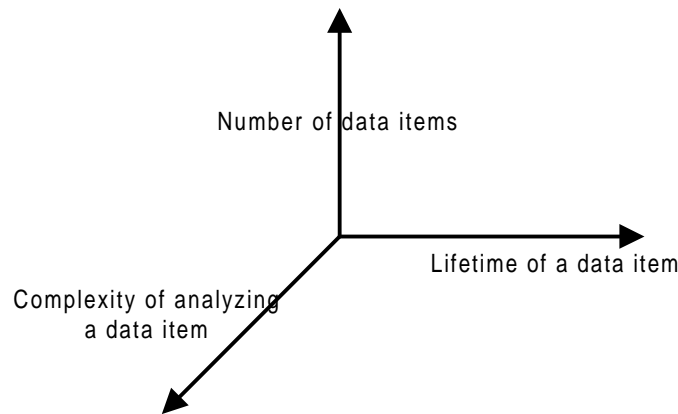


Figure 3.1. Effort space of program analysis

can be described in a multi-dimensional composition shown in Figure 3.1. Assuming the program-specific information the intruder seeks to acquire can be represented as a set of data items, and further assuming that the effort to analyze each data item is independent of the others, the dimensions in this complexity space are:

- **Number of data items:** Intuitively, the more data items there are, the more complex it is to collect the information. For example, compromising the network-wide survivability architecture may require the control of a collection of probe programs at different locations. If the probes are diverse programs, independent analysis efforts will have to be expended to analyze each program. This is an example of increased complexity by increasing the amount of information the intruder must procure.
- **The complexity of analyzing a data item:** There is a cost in complexity in analyzing each information item. It is also intuitive that this complexity constitutes a significant factor in the overall difficulty of program analysis.

- **Lifetime of a data item:** Operational complexity, when rated against available resources, can be measured in terms of time. Increasing the amount of information and the complexity of analyzing each information item can be viewed as efforts to increase the time required for an attack, while limiting the lifetime of the information serves as a complementary tactic—it imposes a time bound within which the attack must consummate. In other words, the shorter the information lifetime, the more resources are required for the analysis, and hence the more difficult the attack is.

These different dimensions determine the complexity of program analysis. Using this model, techniques accentuating one or more of the dimensions yield increased difficulties for the analysis. The basic concepts of the solution framework are based on this complexity model. They are:

- **Information Diversity:** On a network-wide scale, probe programs employ different algorithms, communication protocols, and identification secrets, at different locations and different times. In other words, information diversity is manifested in two forms—*spatial* and *temporal* diversity—in the deployment of the probe programs. Spatial diversity across the network increases the amount of information an intruder must acquire if their goal is to compromise the network-wide survivability mechanism. Temporal diversity serves as a means to limit the information lifetime and consequently induce greater analysis complexity.
- **Information Complexity:** Each probe program undergoes unique code transformations to obscure its critical information. These transformations aim to increase the complexity of information analysis and thus make the program more

difficult to manipulate. From an information theory perspective, these transformations increase the entropy of the target program. Therefore the process of deducing useful information from the program is made more complex. When combined with temporal diversity, this mechanism provides a powerful way to obstruct program analysis.

- **State Inflation:** This concept aims specifically to address the finite state space challenge raised in Chapter 2. State inflation includes a set of mechanisms to expand the program state space, particularly where the program input-to-output relationship is concerned. Black-box analysis, for example, can be thwarted since it relies on the target program having a relatively simple input-to-output behavior.

The concept of information diversity is made possible by incorporating some very specific forms of design diversity in the probe programs. This, along with information complexity and state inflation, is implemented in a *One-way Translation* compiler. The One-way Translation process is capable of transforming a single source program into any number of functionally-equivalent but structurally-varied versions of binary programs. More on the One-way Translation compiler is discussed in Chapter 4.

The general approach is outlined as follows:

- The trusted control servers generate the probe programs, each a product of the One-way Translation process.
- The (diverse) probes are installed at various points throughout the network.
- Each probe program is periodically refreshed with a functionally equivalent version.

Precisely how this happens is detailed in the following sections and the ensuing chapters.

3.1.1. Information Diversity

Diversity is an important engineering principle in building dependable systems. For example, in the design of an aircraft, geographic diversity is often used in the layout of hydraulic lines—each of the redundant lines feeding control surfaces pass through different parts of the fuselage and wings. This design helps to ensure dependable operation by tolerating certain perturbations in the environment.

Incorporating diversity into the design of secure systems helps to reduce vulnerabilities that arise from uniform designs that are often the source of replicated flaws and class attacks [33]. Two forms of diversity—*spatial* and *temporal* diversity—are particularly useful in software protection.

Spatial Diversity: By spatial diversity, I mean the deployment of diverse software versions (e.g., probe programs) at different locations throughout the network. Spatial diversity can be particularly effective in thwarting class attacks—a type of attack that is based on exploitation of the same software and/or configuration flaws [4][33]. For example, most script-driven attacks capitalize on a particular set of known flaws, and the same attack may be replicated successfully on thousands of computing nodes. This is particularly problematic in a COTS-heavy environment—one flaw in a COTS software program may affect tens of thousands of machines on which the software is deployed.

When diverse copies of the software are deployed in a networked system as in the context

of the network-probing architecture, if one program is compromised the same attack may not work on others. The intruder must invest significantly more effort if the goal is to corrupt the network-wide survivability mechanism. Viewed in the context of the solution space described earlier, spatial diversity increases the *amount* of information an intruder must analyze for an attack.

Temporal Diversity: Temporal diversity refers to periodic variations of the software characteristics over time. Temporal diversity serves as a means to limit the lifetime of information, and thus the time window for a particular attack.

As an example, suppose that after obtaining the binary image of an executable program P , an intruder attempts to perform a systematic state-space search to reverse engineer the program. If this effort completes after time period ΔT , the result might lead to a successful tampering or impersonation attack against P . However, if the properties of P change within ΔT ; that is, if P is replaced with P' , the information obtained at the end of ΔT might prove to be ineffective if used against P' .

Temporal diversity implies dynamic changes—a property or a data element may only be valid or have security-related consequences for a limited time. This is not a new concept; password and key aging are rooted in the same principle. They are based on the assumption that a certain amount of time is needed for brute force methods to break a static password or key, and these methods attempt to defeat such attacks by updating the password or key periodically. Applying the idea of temporal diversity to software protection, however, is a novel approach. In this work, temporal diversity is realized with

periodic replacement and reorganization of the binary program and its properties.

To facilitate spatial and temporal diversity, this work employs some specific forms of design diversity in the development of the probe program. Design diversity, in a traditional sense, is the use of different designs within several programs that implement the same specification. It has been employed in various forms of fault-tolerant software including recovery blocks and N-version programming [76].

The types of design diversity employed in this work are different from general design diversity in that this work is interested in specific, algorithmic changes of program characteristics to promote diversity. In this sense, the differences between the different versions can be measured and reasoned about. The same does not hold for general design diversity since it allows unrestricted forms of variations.

3.1.2. Information Complexity

The purpose of program analysis is to deduce certain information from the target program. The complexity of this process depends on the form of the program (e.g., whether it lends itself well to analysis) and what the target information might be.

Program analysis comes in many forms. For example, analysis of the program can take place on a static copy of the code or dynamically during an execution. Static analyses can reveal a great deal about the structure, the algorithm, and also the dynamic behavior of a program, and they are more efficient than dynamic analyses for which interpretation on-the-fly is often required. Dynamic analyses, of course, reveal the ultimate run-time

information about a program. However, conducting program analysis at run-time is expensive, and it often relies on some form of static information to guide its analysis [6][59]. For these reasons, my work is primarily focused on defending against static analysis of programs. One form of dynamic analysis—black-box testing—is considered in Chapter 9.

A comprehensive static analysis on a program requires, as a minimum, the following information [35]:

- Control-flow information (this includes intra-procedural control flow and inter-procedural function calls)
- Relevant data-flow information

Control-flow information provides knowledge on the program execution flow, which is often used as the basis of further analyses. Data-flow information provides knowledge about the possible “modification, preservation and usage” of certain data quantities [35]. Examples of target data quantities include variables, instructions and memory locations.

The complexity of static analysis depends on the complexity of acquiring the control flow and data flow information. A goal of the one-way translation, therefore, is to incorporate techniques to obstruct control-flow and data-flow analysis. Some of the techniques discussed in this research are:

- **Masking control flow:** The control flow of the program can be masked by insertion and restructuring of control constructs. By adding nonfunctional code and breaking

and reorganizing existing control constructs, the program control flow can become arbitrarily complex.

- **Masking code or data content:** Data representations, as well as code constructs, can be restructured in such a way that it will be difficult to recover its original content or even its intent. For example, variables can be divided into subparts, and computations on the variable can be replaced with corresponding computations on the subparts and an operation to construct the correct result. Similar techniques can be applied to arrays, statements and subroutines.
- **Masking code and data location:** Some flow analysis techniques rely on code generation conventions such as the placement of local variables, etc. This is particularly useful when a version of the source code is available and the intruder seeks to match the binary program with the initial source. This type of analysis can be thwarted by breaking code generation conventions and employing randomization in code or data allocation. Furthermore, certain types of semantics-preserving transformations such as function inlining and parameter restructuring can be used to obfuscate function signatures and locations.
- **Masking data usage:** One of the primary functions of data flow analysis is to determine the usage of data—where and how they are used in the program. Data usage provides critical information to facilitate program tampering. Data aliases, for example, can complicate the analysis of data usage. Similarly, indirect addressing and pointer manipulation can be used to mask information on data usage.

These techniques aim to obscure information contained in the program. The premise is

that by obfuscating, the resulting program will be more difficult to analyze, thus more difficult to manipulate and impersonate.

It is important to raise the question of the different objectives between conventional program analysis and what is being discussed in this work. The former has the goal of code improvement, thus a more aggressive, global analysis tactic is desirable. The latter, however, intends to gain specific knowledge to allow targeted program manipulation, and therefore does not necessarily require as ambitious or as comprehensive an analysis strategy. While that may prove to be true in some cases, the ultimate objective of this work is to make the task of program analysis *as difficult as possible*. In other words, the techniques employed here must include an effort to force the use of the most advanced analysis techniques possible. For example, distributing critical information throughout the entire body of the program requires a global analysis to gather the necessary information. In so doing, static analysis will be ineffective if not used in its most aggressive form, or not applied to the entirety of the program.

3.1.3. Input-to-output State Inflation – Increasing the Complexity of Blackbox

Analysis

A blackbox analysis analyzes the program input-to-output behavior without delving into the internal specifics of the program. The goal of blackbox analysis is to gain insights into the program's input-to-output behavior in order to emulate its behavior.

If the state space of the program regarding input and output is simple, with relatively low effort the intruder can deduce the input-to-output mapping and impersonate the behavior

of the legitimate program. For example, consider a probe program for which there are three basic input states: UP, DOWN, and DEGRADED, and the program outputs an integer 0, 1, and 2, respectively, for each of the input states. In this case, a simple blackbox testing would suffice in revealing the entire input-output state space.

To protect against blackbox analysis, the technique of *input-to-output state inflation* can be used. The purpose of state inflation is to increase the complexity in the program's input-to-output mapping such that the exact mapping cannot be easily deduced. Again, the effectiveness of the scheme should be measured in terms of the amount of information an intruder might be able to gather within a prescribed time frame. Strictly speaking, state inflation is another way to promote information complexity. It is singled out as a separate technique partially due to its application against blackbox analysis and partially for convenience in discussion.

The benefit of input-to-output state inflation is perhaps best illustrated with an example. Consider again the example of the probe program that operates on three basic input states: UP, DOWN, and DEGRADED. The program generates output integers 0, 1, and 2, respectively, based on the input state. Now instead of generating one of the three integers, the probe applies the following algorithm to generate three series of numbers x_1, x_2 , and x_3 such that:

$$x_1 = \{x / x \in \text{integer, and } E(k, x) \bmod 3 = 0 \}$$

$$x_2 = \{x / x \in \text{integer, and } E(k, x) \bmod 3 = 1 \}$$

$$x_3 = \{x / x \in \text{integer, and } E(k, x) \bmod 3 = 2 \}$$

$E(k, x)$ is a one-way function, and k represents a key that the probe shares with the

trusted server. Instead of transmitting 0, 1, or 2 across the network, the probe transmits a randomly chosen x from the appropriate series of numbers (e.g. x_1 for UP, x_2 for DOWN, and x_3 for DEGRADED). The receiver of the x 's can then compute:

$$Status = E(k, x) \bmod 3$$

to obtain the status information. An observer, not knowing k , will not be able to determine which x 's correspond to which state simply by observing the input and output relationship.

While this example is reminiscent of encryption, what is important here is that the one-to-one mapping between the input and the output is replaced with a one-to-many mapping. Note that there are an arbitrarily large number of output values for each input state, which will appear essentially random to an outside observer.

Dynamic analysis such as blackbox analysis is based on information obtained by observing program execution. Each state transition during the program execution disseminates a certain amount of information into its environment. Over time the aggregate of this information may be sufficient for an observer to determine the entire state space of the program. This particular state inflation technique attempts to expand the complexity of the input-to-output relationship of the program. Consequently, the average amount of information provided by each observable mapping will decrease, and more effort must be expended to gather an equivalent amount of information.

3.2. Putting It All Together

The previous sections present a brief overview of a suite of techniques, each contributing to a specific dimension in the solution space. It is important to note that no single technique will suffice in defending against the very capable adversary described in Chapter 2. For example, the state inflation example in Section 3.1.3 will be ineffective if someone can simply read the key k out of memory. Ultimately, it is the combination and interactions of the various techniques that provide the necessary apparatus to ensure the secure execution of software in untrustworthy environments.

Here I review briefly one of the assumptions stated in section 2.5.3, namely making allowances for possible knowledge of the original source program. The implication is that protection mechanisms should not rely on the obscurity of the source program. In that discussion, I posited that knowledge of the source program does not necessarily imply immediate knowledge of the running executable. Note how this premise starts to become clear with the solution techniques described above; the one-way translation process implements code transformations that embody the concept of information diversity and complexity. These transformations introduce structural and even semantic differences in the resulting binary programs. Therefore knowledge of the source does not necessarily imply direct knowledge of the binary program unless the exact transforms the program underwent are learned.

In essence, my solution mechanism employs a specialized translation process during which an input source program is transformed according to a set of strategies that incorporate design diversity and information complexity. The result of the translation

process is a set of functionally-equivalent but structurally-varied binary versions. These different versions are then put to use in a network setting. Collectively, they form a resilient defense frontier to protect a network application from malicious underlying hosts. The specifics of this translation process, along with the various transformation techniques, are described in Chapter 4.

An integral piece of this work is the security strength of the solution techniques. In other words, what assurance can you get if you employ these techniques to protect your programs? Without provable assurance, the latest and greatest software protection mechanisms may be only one step away from being outwitted by new countermeasures. In Chapter 5, I present proofs and complexity arguments to reason about the security strength of each of these techniques. The security analysis draws inspiration from the rich body of work in the field of programming languages, and serves as the theoretical foundation of this research.

3.3. Implementation Strategy

The method of transforming programs to incorporate design diversity and information complexity is a task that could potentially be performed on the source program directly by a programmer. The problem with such an approach is that it is a complex, error-prone task. Furthermore, programmers should not be burdened with extra programming tasks that do not contribute directly to the essential goal of the program. For these reasons, I choose to implement the code transformations automatically by means of compiler extensions. This strategy allows perhaps the most efficient and least intrusive integration

into the regular program development cycle.

The extended compiler is designated to perform the following tasks:

- A random set of built-in transformations driven by a random seed.
- Programmer-specified code transformations.

As a result, a source program is translated into an array of different binary programs, each with a different set of behavior and internal representations. The code transformations are implemented as source-to-source modifications. The resulting source program can be compiled subsequently into any low-level machine representation that is desired. A detailed-look at the compiler implementation is presented in Chapter 6.

Chapter 4

One-way Translation

This chapter presents the design of the One-way Translation process—a compiler-based approach to achieve design diversity and information complexity. The core of One-way Translation is the semantics-preserving transformation of programs in such a way that the reverse transform cannot be determined without the expenditure of tremendous resources. Formally, One-way Translation can be described as follows:

Let TR be the translation process, such that $P \xrightarrow{TR} B$ translates a source program P into a binary program B . TR is a one-way process if the time taken to reconstruct P from B is greater than a specific constant T .

Note that there exists a loose analogy between One-way Translation and cryptography: Cryptographic schemes operate under the premise that encryption is easy to perform but the reverse process is computationally expensive (without the key).

One of the objectives of this research is to develop software protection just as one would

with cryptographic methods—much in the same way that the strength of a cryptographic algorithm depends on the key length, the strength of a software protection mechanism should be easily deduced based on some well-defined characteristics.

To achieve One-way Translation, program transformations promoting information complexity are employed. Throughout this chapter, I describe a suite of these transformations. The key idea behind the transformations is to make the two essential forms of program analysis—control-flow and data-flow analysis—strongly and ubiquitously co-dependent. The result of this co-dependence is increased analysis complexity, hence the *One-wayness* of the translation.

4.1. A Model of Semantics-preserving Transformation

Before delving into the details of the One-way Translation process, I first present a model of semantics-preserving transformations used in this work, as its meaning departs slightly from the traditional definition of functional equivalence.

In traditional compiler parlance, *semantics-preserving* transformations preserve the input-output behavior of the program. In other words, the program, before and after the transformation, must produce the exact same results if given the same input [59]. Semantics-preserving transformations yield programs that are considered functionally equivalent.

It should be noted that this definition of functional equivalence is often violated in the actual practice of compilation. For example, the result of commutative operations such as

addition should not depend on the order of the operands. However, reversing the order of the operands in an addition operation sometimes can lead to different results due to possible rounding errors [1].

This work employs a slightly relaxed notion of functional equivalence. Function is not defined in terms of input-output relations. Instead, it is defined as a set of high-level specifications, defining the tasks to be performed. Different implementations, if they fulfill the tasks specified, are considered functionally-equivalent irrespective of their input-output behavior. For example, if a program's functionality is to report the temperature of the day, two different programs—one reporting the temperature in Fahrenheit and the other in Celsius—both accomplish the specified task, and are considered functionally equivalent despite the fact that their input-output behavior is quite different.

Under this definition of functional equivalence, code transformations may affect the internal structure of a program, as well as its external behavior. Transforms affecting the external behavior of the program may alter the program signature. Also note that the traditional definition of functional equivalence is subsumed by the new definition; that is, program transforms that are considered semantics-preserving in the traditional sense are clearly semantics-preserving under the new definition.

The new notion of functional equivalence is application specific—the set of semantics-preserving transformations for one application may or may not preserve functionality for others. For example, replacing a DES encryption algorithm with an implementation of RC4 could be a functionally equivalent transformation for the purpose of encryption, but

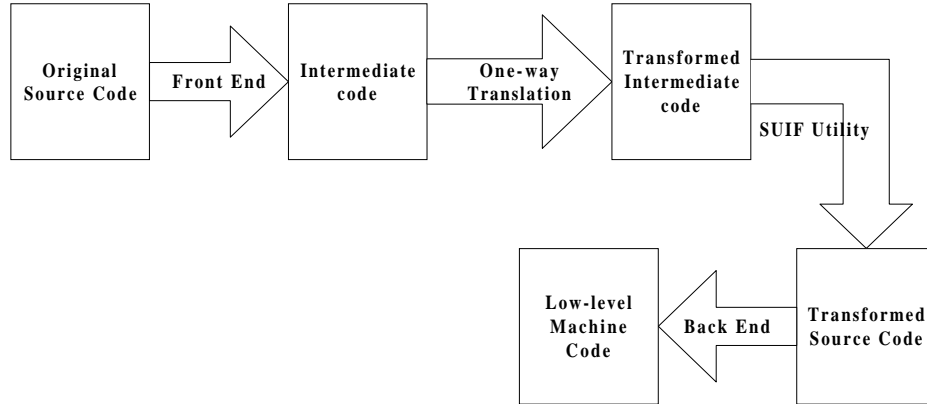


Figure 4.1. Creation of a one-way translated program

would be meaningless where encryption is not concerned. The specification of tasks will have to be derived from the domain knowledge of the application. Exactly how that can be achieved is beyond the scope of this dissertation. For the purpose of this discussion, I simply assume that there exists a set of different implementations for a specification that are considered functionally equivalent.

The notion that there exists an equivalence class of programs that differ not only in terms of internal representation but also in external behavior is fundamental to the One-way Translation idea. This equivalence class embodies the idea of software design diversity, and how the diversity is achieved is detailed in the next sections.

4.2. One-way Translation Process

The One-way translation process transforms a high-level representation of a program to one of the many versions of programs that are in the same equivalence class. The transformations are performed by a source-code translator as part of the compilation

process. Figure 4.1 depicts the creation of a one-way translated program.

The goal of the transformations is to promote design diversity and analysis complexity. Two general forms of transformations are employed for this purpose: *Behavioral transformations*, and *Internal transformations*.

4.2.1. Behavioral Transformations

Behavioral transformations alter the observable behavior of a program. As discussed in section 4.1, some forms of behavioral transformations are considered semantics-preserving in this context. Specifically, the following categories of transformations are applied to the target program in order to provide design diversity.

Change of algorithms (functions): Algorithms for implementing a specific task can be transformed to an alternative algorithm fulfilling the same task. This transformation is based inevitably on user specifications—the programmer specifies a pool of interchangeable implementations, and the compiler simply chooses one at random during compilation.

To simplify the implementation, I consider alternative algorithms in the context of functions; that is, the algorithm to be replaced should be contained in a set of well-defined local functions. Cases such as parallel algorithms are not considered.

A complication arises in handling the implementation of function calls. For example, when function *func1(a, b)* is replaced with *func2(x, y, z)*, the call site should change accordingly to ensure the correct parameter passing. To that end, a simple algorithm is

used. In this algorithm, programmers are required to specify the name of each function that is in an interchangeable set to include the prefix "*choosei_*", where *i* is an integer indicating which set these functions belong. For instance, supposing *func1(a, b)* and *func2(x, y, z)* are in the same interchangeable list, and *func3(m, n)* and *func4(p, q)* are in another list, the four functions would end up in these forms:

$$\begin{aligned} \textit{Func1}(a, b) &\rightarrow \textit{choose1_func1}(a, b) \\ \textit{Func2}(x, y, z) &\rightarrow \textit{choose1_func2}(x, y, z) \\ \textit{Func3}(m, n) &\rightarrow \textit{choose2_func3}(m, n) \\ \textit{Func4}(p, q) &\rightarrow \textit{choose2_func4}(p, q) \end{aligned}$$

Additionally, at the call site the programmer is required to change the function call to the name "*choosei*" followed by a union set of all the possible parameters, with the use of an "_" separating the parameter sets for different functions³. For example, the call to *func1(a, b)* is changed to

$$\textit{choose1}(a, b, _ x, y, z)$$

One other simplification is made—only functions with the same return type and no side effects can be used interchangeably. This restriction can be lifted, with more efforts required on the programmer's part.

The intermediate form of the program retains most of the variable and function names. It is then a simple process to match up the call sites with the appropriate, randomly chosen

³ Let's hope no programmer makes a habit of using "_" as variable names.

functions.

This is by far the most labor-intensive task for the programmer, because it requires the programming of alternative implementations for the same functionality, and placing the hooks for modifying function calls. However, the extra labor incurs no more cost than a typical *N*-version programming⁴ [76]. In addition, because the interchangeable implementations are used in a very limited sense—only well-defined functions can have alternatives, the average programming cost should be lower than general *N*-version programming.

Change of interfacing protocol: The probe program interfaces with the trusted servers via a predetermined protocol. This protocol can also change, in an installation-unique fashion. To facilitate this, the compiler chooses at random a protocol (or a unique instance of the protocol) from an available pool of protocols supplied by the programmer.

The challenge in changing protocols as opposed to changing local functions is that the former involves multiple entities. This implies appropriate changes must be made in both communicating parties. In practice, coding protocol changes in multiple parties can be accomplished by bundling up the programs of the communicating parties in a single module and implement code transformation passes to operate on both parties the same time.

⁴ This cost is possibly less than *N*-version programming, because different versions of implementation are

Consider the example of replacing a DES secret key-sharing scheme with an IDEA implementation. In this case, the encryption and decryption facilities on both sides need to be updated accordingly. First of all, the programmer must rewrite the affected functions much in the same way as described above. For example, on the probe side,

DES_encryption (parameter_list) , and
IDEA_encryption(parameter_list)

are replaced by

export0_choose1_DES_encryption (parameter_list), and
export0_choose1_IDEA_encryption(parameter_list)

The "*export*" prefix indicates that this is an interfacing function. The integer following "*export*" ("0" in this example) identifies the particular pair of operation (e.g., encryption/decryption). Accordingly, on the server side, there are two decryption functions in the following form,

export0_choose3_DES_decryption (parameter_list), and
export0_choose3_IDEA_decryption(parameter_list)

The compiler first performs a preprocessing step in which the different modules in the programs are examined and the *export* functions with the same integer are found. This step builds an external data structure, which contains references to the function signatures

needed for only portions of the program.

and information on which operation the different signatures belong.

During the transformation stage, the compiler walks through the intermediate representation of the different modules, consulting with the external data structure in making selections. For example, if the compiler chooses IDEA encryption on the probe side, it knows to choose the decryption function with the same name ("IDEA") and the same export number ("0").

Protocol changes also require effort on the programmer's part to supply the implementation of the different protocol steps and to line up the protocol steps so the resulting program can function correctly.

Change of input-output behavior: The program's input-to-output behavior can be encoded in an arbitrarily complex mapping. More on this subject is discussed in Chapter 8.

4.2.2. Internal Transformations

This type of transformation affects the internal representation of the program. It does not necessarily change the program's external behavior. For example, changing the order of non-interfering instructions alters the binary program's internal code structure, but leaves the result of execution unaffected. The following transformations are applied to the program in order to induce difficulties in control-flow and data-flow analysis:

- **Intra-procedural Transformation:** This type of transformation affects the intra-procedural program analysis. Specifically, two main strategies are employed: a) the

degeneration of the program control-flow via transformations of static branches to register-based dynamic branching statements, and b) the liberal introduction of data aliases.

- **Inter-procedural Transformation:** Inter-procedural program analysis entails investigation of the program call structure and reasoning about the effect of function calls on specific data quantities. The inter-procedural transformations employed in this work include three aspects: a) function calls are transformed to indirect calls via function pointers, b) aliases to function pointers are created so as to further obscure the program call structure, and c) inter-procedural aliases—aliases created due to function calls—are introduced to consequential data items. Collectively, these techniques ensure that analysis on these data items would require a full-up inter-procedural analysis on a heavily degenerate call structure.

The remainder of this chapter focuses on the intra- and the inter-procedural part of the code transformations. The subject of behavioral transformation are revisited in Chapter 8.

4.3. Intra-procedural Transformations

In this section, I first describe the fundamentals of intra-procedural analysis in order to set the context. From there, I go on to describe the code transformations that are conceived specifically to deter intra-procedural analysis.

4.3.1. The Fundamentals of Intra-procedural Analysis

Intra-procedural analysis can be classified into two categories: *flow-sensitive* and *flow-*

insensitive [38][59]. Flow-sensitive algorithms consider program control-flow information and, in general, yield more precise results than flow-insensitive algorithms. Flow-insensitive algorithms, without control-flow information, must settle with a solution that summarizes over all possible control-flow paths. For this reason, flow-insensitive analysis is generally more efficient, but less precise.

Precision of program analysis is defined as how far the reported analysis results are from actuality. Static analysis cannot always determine the realizable paths of the program, hence the analysis is only an approximation [35].

When program analysis is performed with the goal of acquiring knowledge for software manipulation, flow-insensitive analysis is too imprecise to be useful [38]. The discussion hereafter is based on flow-sensitive analysis. A typical flow-sensitive analysis entails two essential steps:

- Build the Control-Flow Graph (CFG) of the program (a step that is commonly referred to as control-flow analysis). A CFG consists of nodes (which are basic blocks) and edges (which indicate control transfers between blocks). A CFG provides control-transfer information of the procedure.
- Perform data-flow analysis on the target data quantities over the CFG.

Typically, control-flow analysis constitutes the first stage of analysis—it provides control transfer information that is essential for subsequent data-flow analysis. Without this information, data-flow analysis is restricted to the basic-block level only and is fundamentally ineffective for programs where data usage is dependent on program

control-flow.

The central strategy of the intra-procedural code transformations is to conceal the explicit program control-flow and thereby hinder both control-flow and data-flow analysis. This strategy is realized in two sets of code transformations made to the program:

- Degeneration of the static program control flow
- Introduction of pervasive aliasing

4.3.2. Degeneration of the Static Program Control-flow

Control-flow analysis encodes and makes explicit the flow of control in a program. Intra-

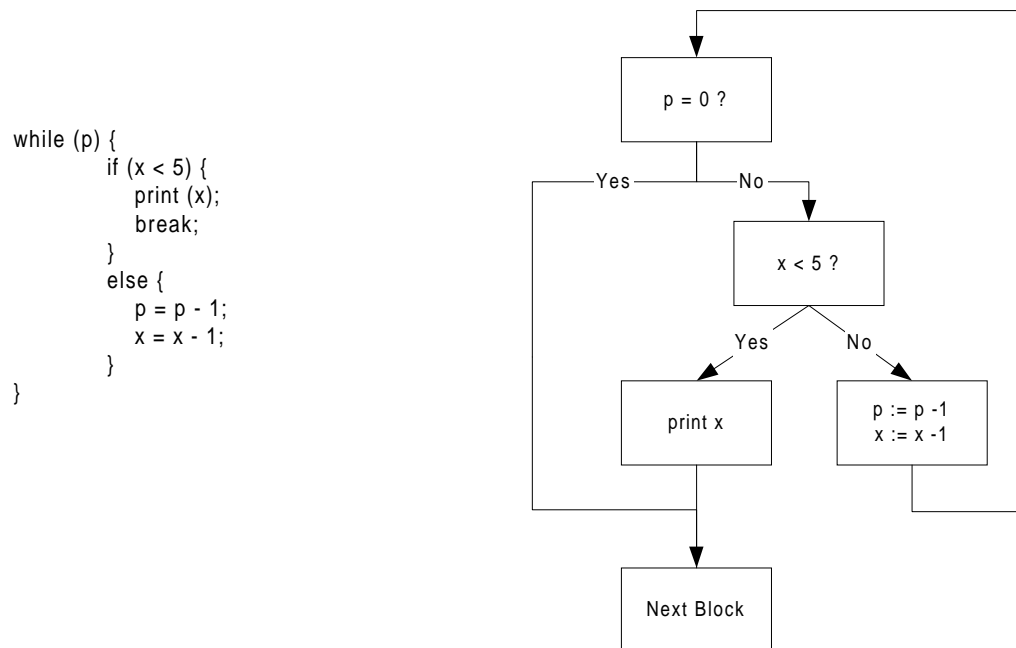


Figure 4.2. An example while loop and its CFG

procedural control-flow analysis constructs the CFG for each procedure as follows: consecutive statements within the procedure are partitioned into *basic blocks* such that once the first statement of the block is executed, all statements in the block are executed sequentially. Program control is transferred to another block once every statement in the current block has been executed.

Formally, a CFG is a triple $G = (N, A, s)$, such that N is a set of vertices representing basic blocks, A is a set of arcs between blocks, and s is the starting vertex in the graph. An arc(x, y) from node x to node y indicates that program control can potentially transfer from block x to block y . There exists at least one path from the starting node to every other node in the graph. Figure 4.2 shows an example CFG for a *while* loop segment.

Real-world programs tend to have control-flows that can be easily discerned, as this is encouraged for program clarity and enforced by high-level language constructs. In such a program, branch instructions and targets are easily identifiable. Thus constructing the CFG is a straightforward operation of complexity $O(N)$, where N is the number of basic blocks in the procedure.

Now consider the code segment in Figure 4.3 in which branch instructions are indirect jumps whose target addresses are not known statically. In this example, the instruction at S12 is an indirect branching statement whose branch target is contained in register 1. In order to determine to which location this instruction will branch, a static analyzer must examine the code to deduce where the content of register 1 is defined last (instruction S1 in this case). What just happened here is a *use-and-def* analysis in which a *use* of a variable (whose content is held in register 1) is identified and its latest *definition* (at S1) is

```

S0:      load r1, 5
S1:      add r1, r2, r3
...
S12:     jump r1

```




Figure 4.3. Indirect branching example

found [59]. The dashed line in Figure 4.3 illustrates the *use* to *def* information chain.

When control-transfers in the program are organized in a data-dependent fashion (such as in the indirect branching example), construction of the CFG is no longer a simple $O(N)$ operation on the order of basic blocks; data-flow analysis such as finding the use-and-def chains for certain data quantities must be conducted to determine the precise program flow.

It is widely known that many data-flow problems do not have efficient solutions in the presence of certain program characteristics such as general aliasing [59]. Some problems have been proven to be NP-complete [49][61]. This difficulty in data-flow analysis suggests that if the program control-flow is data-dependent, and that if the data-flow problem in resolving the control-flow can be made difficult, it is then straightforward that determining the precise program control-flow is a difficult process, and that the program control-flow is effectively degenerate.

To make the program control-flow data dependent, I employ a technique called *control-flow flattening*. This technique is performed in two steps. In the first step, high-level control structures are decomposed into an equivalent *if-then-goto* construct. This transform is illustrated in Figure 4.4 in which the sample program in Figure 4.4(a) is

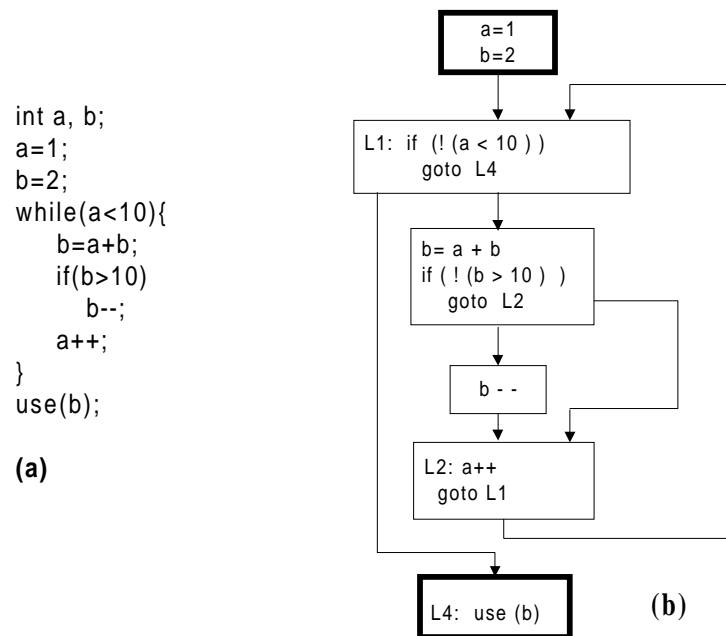


Figure 4.4. Dismantling high-level constructs

transformed into the structure in Figure 4.4(b).

Once all high-level control constructs are converted into this uniform *if-then-goto* structure, the second step is to modify the *goto* statements such that the targets of the *goto*'s are determined dynamically. This is accomplished by loading from the content of data variables instead of using direct jump labels. In C, I model this by replacing each *goto* statement with an entry to a *switch* statement, and assign the switch control variable dynamically in each code block to determine which block is to be executed next. This transform (on the same sample program shown in Figure 4.4) is depicted in Figure 4.5.

With these transformations, direct branches are replaced with data-dependent instructions. As a result, the flow graph that can be obtained from static branch targets degenerates to a common form shown in Figure 4.6. I will refer to such a degenerate flow

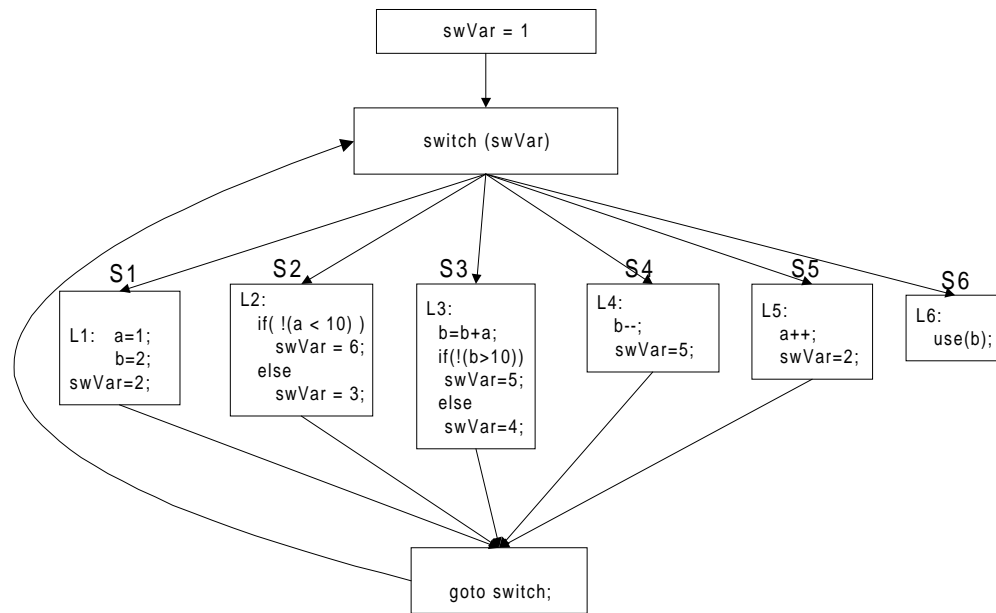


Figure 4.5. Transform to indirect control-transfers

graph informally as *flattened*.

Note that once the program control-flow has been flattened, the static scoping information is absent from the program. It is no longer possible to perform a simple textual pass through the program and determine its scoping (e.g., from examining the stack frame pointers, etc.). Furthermore, flattening the control-flow implies that basic blocks do not need to appear in memory in the general order of execution—they can be organized randomly.

Recall the notions of flow-sensitive and flow-insensitive analysis described earlier in this chapter. A flow-insensitive analysis does not rely on control-flow information to conduct its analysis. In other words, flow-insensitive analyses perceive the program as a collection of basic blocks and ignore the inter-relations among the blocks. Hind *et al.* [37]

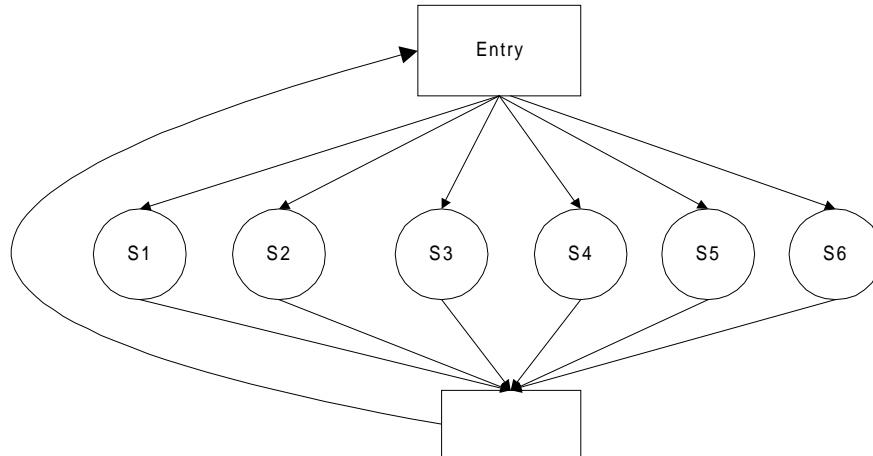


Figure 4.6. A flattened control-flow

showed in their work that a flattened control-flow form much like the one shown in Figure 4.6 is equivalent to the control-flow perceived by a flow-insensitive analysis.

With a flattened control-flow, a flow-sensitive analysis cannot provide any increased precision than a flow-insensitive analysis unless semantics information from the program (such as the branch targets) is available. To obtain this semantics information, the attacker must conduct data-flow analysis on the data quantities that are referenced in computing the branch targets. The next section explores yet another set of code transformations with the objective of impeding these data-flow analyses.

4.3.3. Aliasing and Data-flow Analysis

After flattening of the program control flow, constructing the CFG becomes a data-flow problem on the modification and usage of the data quantities that lead to the definitions of the branch targets.

In the example shown in Figure 4.5, the values of the switch control variable *swVar* are assigned dynamically with a constant assignment statement in each block. A constant propagation analysis [59] combined with a use-and-def analysis on the value of *swVar* would quickly reveal, for each block, what the branch target is for the next block, and consequently reveal the entire control-flow graph.

This suggests that flattening of the static control-flow alone is not sufficient—further hindrance of the data-flow analysis is essential. To achieve this, I enlist two additional modifications to the program; *index computation*, and *aliasing*.

Index Computation: Consider the code segment in Figure 4.7(a). A *use-def* analysis on the value of *swVar* (contains branch target information) is straightforward (the dashed line indicates the *use-def* information chain). Now consider the code segment in Figure 4.7(b) in which a global array “*g[]*” is introduced and the value of *swVar* is computed through the elements of the array. Replacing the constant assignments in Figure 4.7(a) with complex expressions involving array elements implies that the static analyzer must first deduce the array values before the value of *swVar* can be determined.

Index computation involving elements of aggregates, such as arrays or complex data structures, is difficult to analyze statically [35][59], especially when the values of the aggregate elements do not remain constant throughout the execution. Most static analyzers simply assume that a reference to any element of the aggregate is a reference to the entire structure, and that potentially all elements can be changed if an assignment is made to any element of the aggregate.

The following four transformations are applied to the program in order to take advantage of index computation:

- A global array— $g[]$ —is introduced. The elements of this array are used by the program in data computation, including the computation of branch targets. A certain subset of the array is initialized according to the following rules:
 - Every n th element in the array contains a data value x such that $x \equiv c \bmod j$ (" \equiv " represents a congruence relationship). These elements are called the *black* elements since they contain useful information.
 - The constant values of c , n and j are contained in three randomly-selected array elements.
 - The other elements of the array are called the *white* elements, and they contain random values.
- Computation of the branch targets (essentially an integer in this case) is performed through the use of black elements. For example, supposing the branch target in question is integer "2", and assuming that the constant c is 1—every black element is thus congruent to 1 modulo the constant j , integer 2 can be calculated as:

$$(\text{black element \#1} + \text{black element \#2}) \bmod j$$

It should be clear that the values of the black elements can compose arbitrarily complex expressions to compute any arbitrary integer values.

- After each computation of the branch target, a random set of black elements are overwritten with values in the same congruence class. For example, supposing every tenth element of the array contains a data value x such that $x \equiv 1 \pmod{47}$, further assuming that the twentieth element of the array $g[20]$ has the value 95, $g[20]$ can be overwritten with the value 246, which is in the same congruence class as 95 modulo 47.
- After each computation of the branch target, a random set of white elements are overwritten with random values.
- A new subroutine called *Redistribute* is added to the program. The purpose of *Redistribute* is to reorganize the black and white elements, and it is called periodically throughout the execution. *Redistribute* overwrites the value of n and computes a new set of black values which are then stored in the new locations of black elements.

With these modifications, the program can calculate branch targets dynamically using arbitrarily complex expressions involving index computations—consider calculating subscripts of black elements using black elements themselves. In particular, since the array elements do not remain constant throughout the execution, a naive static analyzer that does not perform value interpretation is all but hopeless in deducing any useful information in this case.

Even a more aggressive analysis algorithm that is specially tuned to analyze arrays or aggregate data structures, or even one that incorporates a limited form of interpretation (such as constant propagation), must deal with yet another difficult feature of programs—

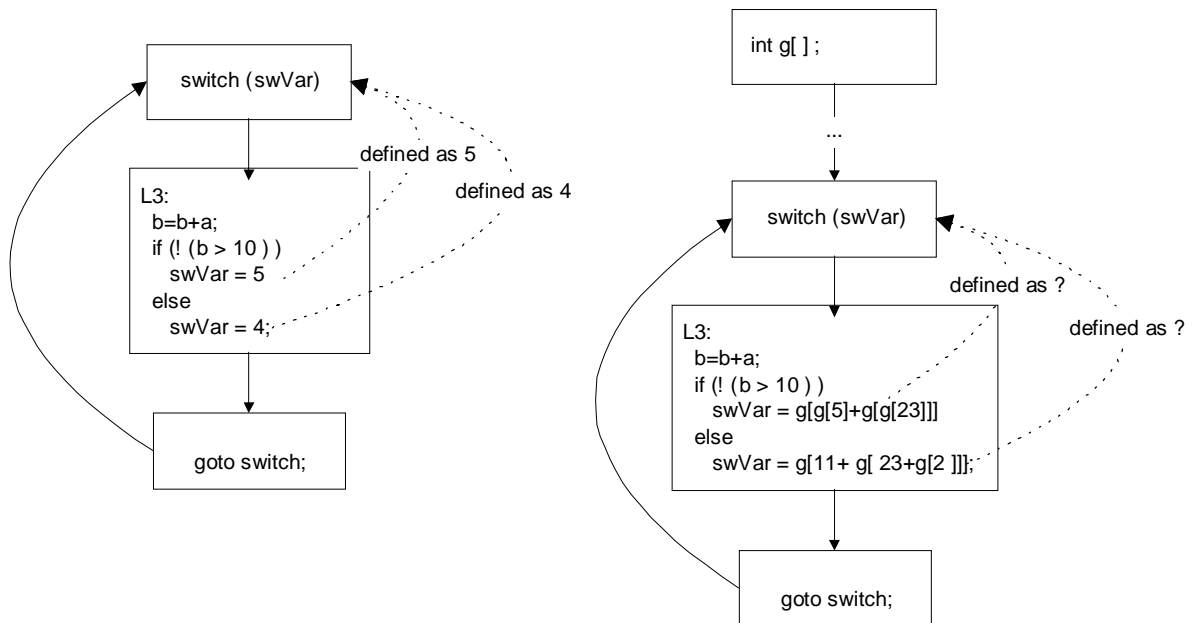


Figure 4.7. Example illustrating index computation

the existence of aliases. Aliases, a primary complexity factor in data-flow analysis, is the focus of the next set of code transformations.

Aliasing: Aliases happen when two or more names refer to the same memory location. Alias detection is essential to data-flow analysis. For example, consider the following code segment,

```

S0:    i = 0;
S1:    *p = 1;
S2:    while (i < 5)
    {
        *p = *p + i;
        i = i + 1;
    }
S3:    f1(i);

```

If $*p$ is an alias of i at $S0$, the *while* loop would only execute twice, and the value of i

would be 7 at S3. If $*p$ is not aliased to i at S0, the while loop would execute 5 times, and i 's value would be 5 at S3. Without the alias information, it is impossible to determine the precise value of i statically at program point S3.

Many classical data-flow problems are known to be difficult to solve precisely because of aliases [49][61]. Accurate alias detection, in the presence of general pointers and recursive data structures, is known to be undecidable [49], and that is the key reason why any data-flow problem influenced by aliasing is fundamentally difficult.

The code transformations described below introduce non-trivial aliasing in order to influence the analysis of data quantities (in this case, the data quantities of interest include the ones relevant to the computation of the branch targets).

- A random number of pointer variables to common data types (such as *int*, *char*, etc) are introduced as both global and local variables⁵.
- Assignments to these pointer variables are made such that the pointer variables are aliased to existing data variables as well as elements of aggregate structures (e.g., the global array elements).
- Computations on certain data quantities are replaced with references through the aliased pointers. For example, if variable $*p$ is aliased to data variable a , the operation

⁵ Exactly how many pointer variables are introduced can be specified by the programmer

- $c = a + b$ can be replaced with $c = *p + b$. Furthermore, if a pointer variable is aliased to an array element, references to other elements in the array can be replaced with references through the pointer variable and the appropriate pointer arithmetic operations. For example, if variable $*p$ is aliased to array element $g[10]$, a reference to $g[35]$ can be substituted with $*(p + 25)$.
- Artificial basic blocks are created such that they contain spurious computations on any relevant data quantities through the aliased pointers. As much as possible, uses of the pointers and their definitions are placed in different basic blocks.

Some of these basic blocks will execute in all traces of the program, and others are simply dead code. Since the static analyzer does not know which blocks actually execute, and since definition of the pointers and their uses are placed in different code blocks, the analyzer will not be able to deduce which definition is in use at each use of the pointer—all pointer assignments will appear live.

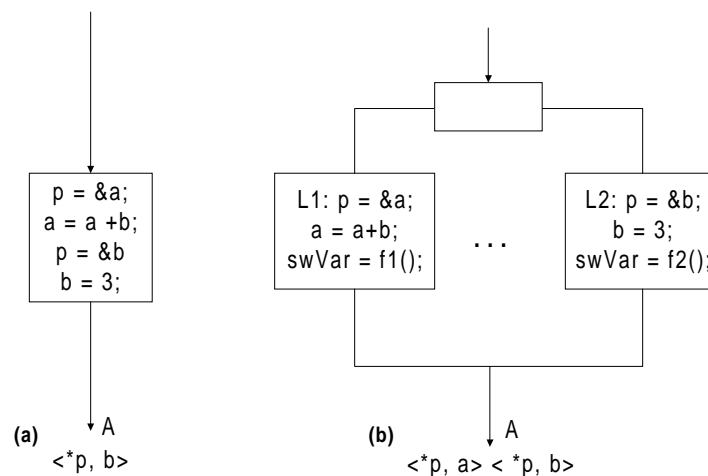


Figure 4.8. Introducing aliases through pointers

For example, consider the code segment in Figure 4.8(a). It can be seen that in Figure 4.8(a) the second definition of the variable p carries to point A in the program. However, if the code segment in Figure 4.8(a) is decomposed into two blocks in Figure 4.8(b) and the transition between blocks is obfuscated using the *flatten-and-jump* technique described earlier ($f1$ and $f2$ denote complex expressions), the static analyzer, not knowing which block executes first, will report both alias relations $\langle *p, a \rangle$ and $\langle *p, b \rangle$ holding at point A.

Intra-procedural aliases, when introduced in combination with the degeneration of static control-flow, induces difficulties in the precise determination of alias relations. As a result, a static analyzer will report imprecise alias relations. With a sufficient number of aliases, the analysis will resolve an array element to a large set of possible values (because of imprecise aliasing). This in turn implies that, at each use, the switch variable can take on a large set of values.

4.3.4. Obstructing Intra-procedural Analysis—Putting It Together

The above sections detailed a strategy to alter intra-procedural control-flow and ways to deter the static determination of control-flow targets. This strategy flattens the program control-flow to a ubiquitous data-dependent flow structure, and at the same time, introduces data aliases to further complicate the control-flow analysis.

The end results of the transformations are:

- Flow-sensitive analysis can never be more precise than flow-insensitive analysis

(since the required static control-flow information is missing).

- Flow-insensitive analysis is made ineffective by the introduction of data quantities whose usage is flow dependent (e.g., aliasing).

It can be argued that if an adversary can capture the initial value of *swVar*, he can then find the first block to be executed, and from there the next block can be identified, and so forth. Doing so might recover some of the original control-flow. While this requires interpretation and simulation of the code in order to identify the next block, the interpretation needs to be done only once for each block. As a result, the complexity of this analysis lies somewhere between static analysis and a full execution trace, with analysis time being proportional to the number of blocks in the program.

One way to counteract the above technique is by unrolling loops and introducing semantically equivalent blocks that will be chosen randomly during execution. This will make the cost of recovering the program control-flow comparable to a full simulation. Additionally, the initial computation of *swVar* can be erased from memory once it is used to avoid unnecessary exposure of information.

A side effect of these transformations is that the basic blocks no longer need to appear in the general order of execution—they can be organized in any random order. This serves as a cheap obfuscation technique that can be used to deter naive pattern matching analysis.

4.4. Inter-procedural Code Transformation

Function invocations can affect the usage of data quantities. To study program behavior, analysis must be performed at the intra- as well as the inter-procedural level.

An inter-procedural data-flow analysis relies on the function invocation relationships to determine, among other things, the static information propagation paths among procedures, which are necessary in reasoning about inter-procedural data references.

The function invocation relations of a program can be encoded in a graph called the Program Call Graph (PCG). Formally, a PCG is a triple (N, e, p) where; N is the set of functions in the program such that $N = \{p_1, p_2, \dots, p_n\}$, p is the function that contains the program entry point; and e is a set of directed edges such that if (p_i, p_j) is an element of e , there exists at least one call from function p_i to p_j . Figure 4.9(a) shows a program skeleton in which function f calls g , g calls h and i , and i calls j and g . The PCG for this program is shown in Figure 4.9(b). The entry function f is marked by the bold circle.

Construction of the PCG is a straightforward process when all function calls are explicit—a simple textual pass over the program will suffice. In the presence of function pointers (or function parameters and function variables), however, a call-site may not be bound statically to a unique function. Thus the process of constructing the PCG is more complex.

Several approaches to building the PCG in the presence of function pointers exist, with varying degrees of complexity and precision [30][54]. An approach that simply assumes

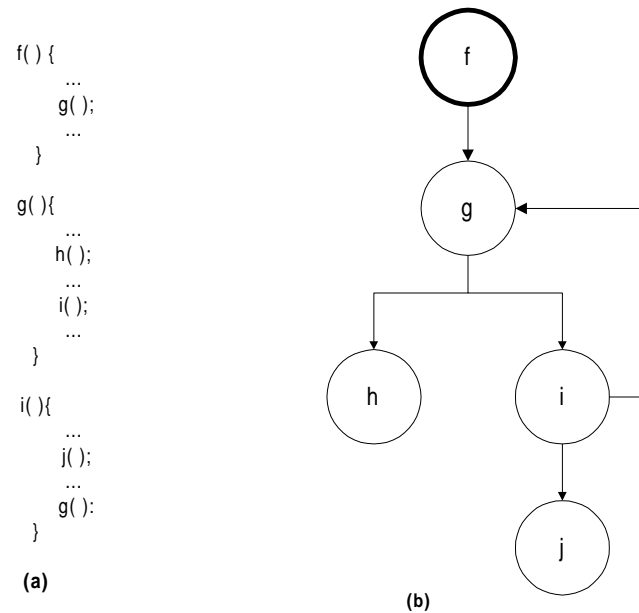


Figure 4.9. An example program call graph

that an invocation through a function pointer may invoke any function in the program requires a single pass over the program, and thus is the least costly with the least precision. An approach that takes into account only the functions that have been instantiated requires a flow-insensitive traversal of the program, and it generally provides a better precision than the previous approach.

A more precise PCG can be obtained by conducting pointer alias analysis prior to building the final PCG. The alias analysis helps to restrict the number of functions invocable from a call site to the set of functions that are aliased to the function pointer at that particular point. In other words, construction of a precise PCG hinges on no other than alias analysis of the function pointers.

It should be clear by now that if function pointers are treated in the same manner as other

pointers, the techniques described in section 4.3.3 can be applied to create aliases for function pointers. Doing so would reduce the precision of the PCG representation, and ultimately impede the process of inter-procedural data analysis.

This section describes a set of code transformations based on the above observation. The basic strategy behind these transformations is as follows:

- Function calls are modified to calls-through-function-pointers
- Aliases to function pointers are created and function signatures are unified to allow pervasive aliasing
- Alias-inducing techniques are applied to generate inter-procedural aliases

4.4.1. Function-call Transformations

To see why the presence of function pointers complicates the task of PCG constructions, consider the two code segments in Figure 4.10 that implement equivalent functionality. In Figure 4.10(b), `ptr`, `fptr1`, and `fptr2` are function pointers. To determine the target of the indirect call on line S6 in Figure 4.10(b), the analysis must determine the set of functions to which `fptr1` and `fptr2` are aliased at S6. This requires knowledge of the alias relations that hold on entry to `func1`. Without this information, the function pointer `ptr` on line S6 cannot be bound to any particular function at analysis time.

Transforming function calls to calls-through-function-pointers is straightforward. It requires creation and initialization of the function pointer variable, and the corresponding changes at the call site. This step does not require any changes to the function signature.

<hr/> <pre> S1: func1() { S2: if (x > 4) S3: func2(); S4: else S5: func3() S6: }</pre> <hr/>	<hr/> <pre> S1: func1(fp1, fptr){ S2: if (x > 4) S3: ptr = fp1; S4: else S5: ptr = fp2; S6: ptr;}</pre> <hr/>
(a)	(b)

Figure 4.10. Function call via function pointers

4.4.2. Function Pointer Aliasing

To further complicate inter-procedural data-flow analysis, function pointer aliases are introduced to the program.

There is a twist to the creation of function pointer aliases. Unlike data variables, functions tend to have distinct signatures. Thus each function pointer of a certain type can only be bound to one function. This in itself is not sufficient to fool a static analyzer of any intelligence, for it is a trivial task to match up invocations (the number and the types of parameters) with function signatures. For this reason, the compiler, prior to creating function pointer aliases, performs an operation to unify function signatures so that all functions in the program conform to only a small number of distinct signatures. The following paragraphs discuss function signature unification first, and then the introduction of function pointer aliases.

Unifying function signatures: Consider the code segment in Figure 4.11(a). There are two function invocations in *func1*. Both destination functions *p* and *q* return an integer. However, *p* has just one integer parameter while *q* takes two parameters, one integer and

one float. One way to unify the two signatures is to modify p's signature to add a float parameter as depicted in Figure 4.11(b). The bold letters indicate the added variable and parameter.

Requiring all functions to conform to the superset of all parameter lists, as shown in Figure 4.11, is the simplest and most intuitive way to unify function signatures. However, it has a few drawbacks. For example, it is possible for such a scheme to result in unreasonably long function signatures; consider n one-parameter functions, each with a distinct parameter type, the resulting signature would have n parameters. Long function signatures incur unnecessary cost, and are best avoided. Optimization is clearly possible. In Chapter 6 where implementation of the One-way Translation compiler is discussed, I will describe in detail some of the optimization techniques I have developed to facilitate

<hr/> <pre> func1() { int x, y, z; float f; y = p(x); z = q (x, f); } int p(int para1) { ... } int Q(int para1, float para2) { ... } </pre> <hr/>	<hr/> <pre> func1() { int x, y, z; float f1, f2; y = p(x, f2); z = q (x, f1); } int p(int para1, float para2) { ... } int Q(int para1, float para2) { ... } </pre> <hr/>
(a) original	(b) transformed

Figure 4.11. Example illustrating unifying function signatures

efficient unification of function signatures.

A number of other issues need to be considered when modifying function signatures and the corresponding invocations. Some of those are discussed below:

- **Return type:** In order to unify functions with different return types, a *void* type is used in the transformed function signatures. An explicit cast back to the original type at the function invocation and another cast before the function returns complete the transform.
- **Complex parameters:** Complex parameters such as structures, arrays or functions are replaced with void pointers. Consequently, function invocations and references to the original parameters inside the function are modified as follows:
 - A direct reference to the original parameter inside the called function is replaced with an indirect reference via the void pointer parameter, following an explicit cast to allow the void pointer to point to the original parameter type.
 - At the function invocation, the original actual parameter is replaced with the void pointer variable which holds the addresses of the original parameter.

Figure 4.12 and Figure 4.13 illustrate an example of function signature modification. The original program code is shown in Figure 4.12, and the transformed code is in Figure 4.13.

The bold letters in Figure 4.13. indicate modified or inserted variables or statements. Note that the signature of *func1* in line S5 in Figure 4.12 is modified such that instead of taking a structure variable, the transformed *func1* takes a void pointer as shown by line S5 in Figure 4.13. Also note that within the calling function, the original structure variable *r1*, declared on line S15 in Figure 4.12, is replaced with a void pointer *void *r1*.

The bold letters in Figure 4.13 indicate modified or inserted variables or statements. Note that the signature of *func1* in line S5 in Figure 4.12 is modified such that instead of taking a structure variable, the transformed *func1* takes a void pointer as shown by line S5 in

```

S1:    structure {
S2:        int field1;
S3:        char filed2;
S4:    } records;

S5:    int func1 ( int x, records r, char *ch) {
S6:        int y;
S7:        y = r.field1;
S8:    }

S9:    char func2 (int x, char *c) {
S10:        ...
S11:    }

S12:    int calling_fun ( ) {
S13:        int x, y, z;
S14:        char *m, *n;
S15:        records r1;
S16:        z = func1 (x, r1, m)
S17:        func2 (y, n);
    }

```

Figure 4.12. Function signature modification - original code segment

Figure 4.13. Also note that within the calling function, the original structure variable *r1*, declared on line S15 in Figure 4.12, is replaced with a void pointer *void *r1*. *r1* is initialized on line S16 in Figure 4.12, and then passed to *func1* as a parameter. Inside *func1*, the reference to *r1.field1* on line S7 in Figure 4.12 is replaced with *(record *) r1* → *field1* on line S6 in Figure 4.13.

The operation of unifying function signatures results in a large number of functions inside the program with an identical signature. Observe that with these transformations, a function pointer can refer to a large number of otherwise distinct functions.

```

S1:    structure {
S2:        int field1;
S3:        char filed2;
S4:    } records;

S5:    int func1 ( int x, void *r, char * ch) {
S6:        int y;
S7:        y = (records *)r -> field1;
S8:    }

S9:    char func2 (int x, void *r, char *c) {
S10:        ...
S11:    }

S12:    int calling_fun ( ) {
S13:        int x, y, z;
S14:        char *m, *n;
S15:        void *r1, *r2;
S16:        r1 = (void *)new (records);
S17:        r2 = (void *)new (records);
S18:        z = (int) func1 ( x, r1, m );
S19:        func2 ( y, r2, n );
S20:    }

```

Figure 4.13. Function signature modification – modified code

The number of distinct function signatures in a program is an application specific choice. As an extreme, every function could be transformed to a single signature, in which case only one type of function pointer is required. Alternatively, functions can be grouped into a small number of groups, each with a distinct signature. The advantage of the latter scheme is efficiency. In general, a large portion of the functions in a program contains a similar list of parameters with common types (e.g., integers or floats). It is then more economical to group these functions together to form a new signature. Since the new signature has a significant overlap with any of the original ones, it has less an impact on the run-time space and time complexity than a full signature unification.

Function-pointer aliasing: When function pointers are treated in the same manner as other pointers, the aliasing inducing techniques described in section 4.3.3 can be applied to introduce aliases for function pointers.

Specifically, function pointer aliases are created using the following methods:

- A number of function-pointer variables are declared—they have the same type since function signatures are unified.
- Assignments to the function pointers are made such that some of the assignments are located in spurious blocks (these assignments will never be executed).

Since the function signatures are unified, a function pointer can point potentially to any function in the program. At an extreme when there is only one function signature in the program, there will be an edge in PCG from a function that contains a call site to all functions in the program. Of course, some of the PCG edges are real and some are simply

spurious.

Spurious edges in the PCG increase the complexity and reduce the precision of the interprocedural analysis. (Information propagation in between functions must be analyzed between function pairs for which there exists no information propagation path from one to the other.) Figure 4.14 illustrates such an example. Figure 4.14(a) is the original call structure whose corresponding PCG is depicted in Figure 4.14(b). The post-transform PCG, as perceived by a static analyzer, is illustrated in Figure 4.14(c) where the dashed

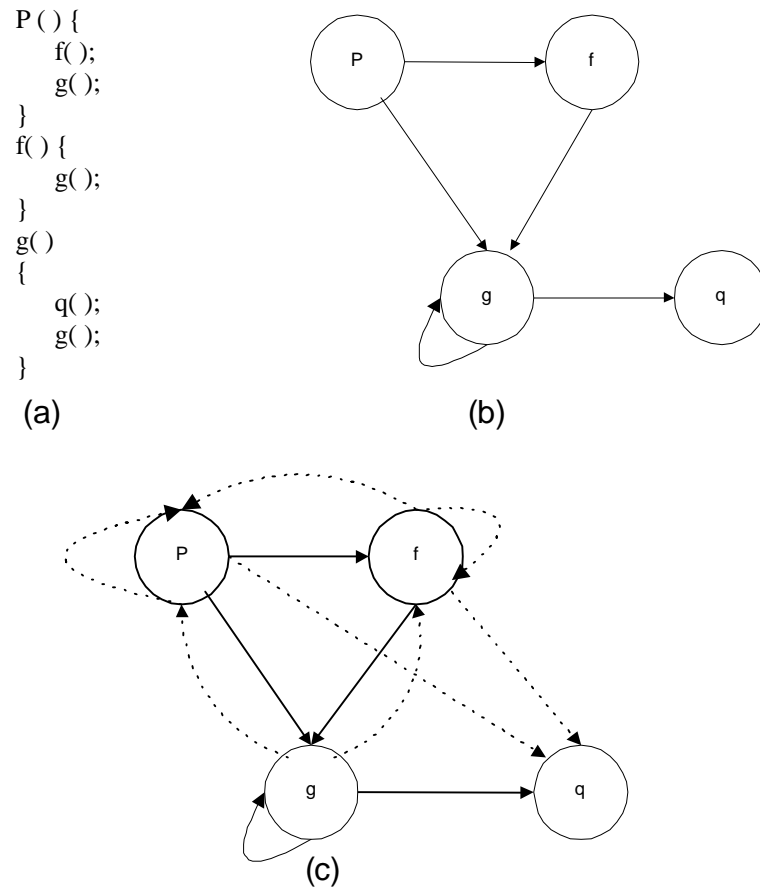


Figure 4.14. A PCG with false edges

lines represent artificial edges added due to function pointer aliasing.

4.4.3. Inter-procedural Aliases

Under the conjecture that the PCG of the program is degenerate and that precise information propagation paths among functions are not retrievable statically, I describe a set of techniques to further thwart static analysis by introducing inter-procedural aliases—aliases whose meaningful resolution requires none other than inter-procedural data-flow analysis.

Inter-procedural aliases generally happen by parameter passing and the accessibility of non-local stack locations. More precisely, the One-way Translation compiler introduces inter-procedural aliases in the following ways:

- *Global and local reference aliases:* Addresses of global variables are passed to functions as parameters. Inside the function, the global variables and the corresponding formal parameters become aliases.
- *Parameter aliases:* Aliases between formal parameters are created by binding the same address to two or more pointer parameters.
- *Aliasing through return values:* If the called function returns the address of a variable visible in the calling function, and the return value is assigned to a different pointer variable, the two variables become aliases upon return from the invoked function.
- *Alias through side effects:* When the address of a pointer variable is bound to a

parameter, this variable can be altered inside the called function. If, inside the called function, the variable is assigned the address of another pointer variable that is also visible in the calling function, the two pointer variables become aliases upon return from the called function. Such an example is shown in Figure 4.15.

As the above discussions illustrate, function invocation can result in aliases in both the called and calling functions. This is due to inter-procedural information propagation between function calls—there is a *forward* parameter binding process in which information propagates from the calling function to the called function, and this information results in new aliases in the latter. Similarly, there is a *backward* binding process in which information propagating from the called function back to its caller prompts new alias relations in the calling function.

The presence of inter-procedural aliases requires inter-procedural analysis. Analysis at the inter-procedural level requires the program call structure information in order to propagate static information among functions. However, the indirect function calls and the function unification transformation obscure the true information propagation paths

```

f( ) {
    int *i, *j;
    g (&i, &j);
    ...
}
g (int **a, int **b) {
    a = b;
}

```

-----// *i and *j become aliases here

Figure 4.15. Aliasing through side effects

among functions. Thus the information needed to bind arguments and parameters properly is not readily available.

4.5. Summary

In this chapter I present a suit of code transformations at the intra- and inter-procedural level to deter static analysis. These transformations are based on two strategies:

- Degeneration of the static program control-flow
- Pervasive aliasing

On one hand, degenerate control-flow arises from transforming direct control-transfers to data-dependent branches and subsequently flattens the static control-flow. Consequently, control-flow analysis is converted to a ubiquitous data-flow problem (the problem of determining the branch targets). On the other hand, data-flow analysis is made more difficult by the introduction of pervasive aliasing throughout the program.

Note that transformations such as the ones described in this chapter are essentially one way (with respect to time period T)—undoing the transforms requires minimally the resolution of aliasing and restoring the static control flow, both are difficult problems.

Through these transformations, the two facets of static analysis—control-flow and data-flow analysis—become strongly and ubiquitously co-dependent. The result of this co-dependence is increased analysis complexity. Exactly to what degree analysis complexity is affected is the topic of discussion for the next chapter—theoretical evaluation.

Chapter 5

Theoretical Evaluation

In this chapter, I present the theoretical evaluation of the efficacy of the code transformations described in chapter 4. An NP-complete proof is presented to show that in a general case, determining indirect branch targets statically for a transformed program is an NP-complete problem. I follow the NP-completeness proof with a discussion of the practical complexity in program analysis, given the said transforms. In particular, I investigate the complexity of alias approximation methods since aliasing is the basis for the claim of analysis difficulty.

5.1. An NP-complete Argument

I have thus far conjectured that the difficulty of discerning indirect branch target addresses is influenced by aliases in the program. In this section, I support this claim by presenting a proof to show that determining precise indirect branch addresses statically is an NP-complete problem in the presence of general pointers.

Theorem 1: In the presence of general pointers, the problem of determining precise indirect branch target addresses is NP-complete.

Proof: To prove a problem NP-complete, it suffices to show a polynomial-time reduction from a known NP-complete problem to the target problem. In this case, I choose the 3 SATisfiability (3SAT) problem as the known NP-complete problem [21], and show a reduction from 3SAT to that of determining precise indirect branch targets.

This proof is a variation of the proof originally proposed by Myers in which he proved that various data-flow problems are NP-complete in the presence of aliases [61]. Landi later proposed a similar proof to prove that alias detection is NP-complete in the presence of general pointers [50].

Consider the 3-SAT problem such that

$$\bigwedge_{i=1}^n (V_{i1} \vee V_{i2} \vee V_{i3}) ,$$

where $V_{ij} \in \{v_1, \dots, v_m\}$, and v_1, \dots, v_m are propositional variables whose values can be either *true* or *false*. The 3-SAT problem states it is NP-complete to determine whether an arbitrary 3 satisfiability formula is satisfiable.

The reduction is shown in the code below. The branch target address is located in the array element $A[*true]$. The *if* conditionals are not specified—the assumption is that all paths are potentially executable.

```

L1:      int *true, *false, **v1, **v2, ... **vm, *A[];
L2:      A[*true] = &f1();
L3:      if (-)
            $\overline{v_1 = \&true; v_1 = \&false}$ 

```

```

else
     $v_1 = \&true; \overline{v_1} = \&false$ 
if (-)
     $v_2 = \&true; \overline{v_2} = \&false$ 
else
     $v_2 = \&true; \overline{v_2} = \&false$ 
...
if (-)
     $v_n = \&true; \overline{v_n} = \&false$ 
else
     $v_n = \&true; \overline{v_n} = \&false$ 

L4:  if (-)
         $A[** \overline{v_{11}}] = \&f2()$ 
        else if (-)
             $A[** \overline{v_{12}}] = \&f2()$ 
        else
             $A[** \overline{v_{13}}] = \&f2()$ 

        if (-)
             $A[** \overline{v_{21}}] = \&f2()$ 
        else if (-)
             $A[** \overline{v_{22}}] = \&f2()$ 
        else
             $A[** \overline{v_{23}}] = \&f2()$ 
        ...
        ...
        if (-)
             $A[** \overline{v_{n1}}] = \&f2()$ 
        else if (-)
             $A[** \overline{v_{n2}}] = \&f2()$ 
        else
             $A[** \overline{v_{n3}}] = \&f2()$ 

L5:
```

Code segment L1 declares the variables and an array $A[]$. v_1, v_2, \dots, v_m are doubly dereferenced pointer variables. L2 assigns $A[*true]$ to the address of $f1$.

A path from L3 to L4 represents a truth assignment to the propositional variables for the 3-SAT formula. Here the assignment to *true* is represented as an alias relationship

$\langle *v_i, true \rangle$, and the alias $\langle *v_i, false \rangle$ represents assigning *false* to variable v_i .

If the truth assignment for the particular path from L3 to L4 satisfies the 3-SAT formula, then every clause contains at least one literal that is true. This means that there exists at least one path between L4 and L5 on which the value of $A[*true]$ is never reassigned. Consider choosing the path that goes through the true literal in every clause. In every clause it assigns $A[*false]$ to $\&f2$ since every variable $*v_{ij}$ on that path is aliased to *false*.

If the truth assignment renders the formula not satisfiable, then there exists at least one clause, $(V_{i1} \vee V_{i2} \vee V_{i3})$, for which every literal is *false* (i.e., all the literals in the clause are aliased to *false*). This implies that $\overline{*v_{ij}}$ is aliased to *true* for this clause. Because every path from L3 to L4 must go through the following statement

```

If (-)  $A[\overline{*v_{i1}}] = \&f2()$ 
  else if (-)  $A[\overline{*v_{i2}}] = \&f2()$ 
    else  $A[\overline{*v_{i3}}] = \&f2()$ 

```

Therefore, at program point L5, $A[*true]$ must point to the address of $f2$.

The above code segment shows that 3-SAT is satisfiable if and only if the branch target address contained in $A[*true]$ is the address of $f1$. This proves that 3-SAT is reducible in a polynomial time to the problem of finding precise indirect branch target addresses.

5.2. Practical Complexity Measures

The NP-completeness proof above presents a theoretical measure for the analysis of general case programs. However, such a proof does not guarantee the average case complexity—for any given program, the complexity of analysis could be well within the grasp of a static analyzer (because of its size, characteristics, etc.)

In the remaining sections of this chapter, I examine the complexity measures for practical program analysis within the context of the code transforms described in Chapter 4. In particular, I concentrate on the complexity of alias approximation methods since alias analysis influences the complexity of other data-flow problems.

In practice, alias analysis is conducted in an approximation manner—the results reported by the alias analyzer may not be precisely the same as the actual alias relations, but usually contain a conservative estimate of the reality (a superset of the actual alias relations). How far the reported results are from the actual alias set is called the *precision* of alias analysis.

The discussion here is concerned with the efficiency of the alias analysis as well as its precision. Note that an analysis algorithm may be able to reach a conclusion quickly if precision is not of a major concern. A clear example is to trivially report that every variable is aliased to every other variable—such an analysis would take very little time but would report with the least precision.

The complexity of alias analysis has been examined at a great length in various studies

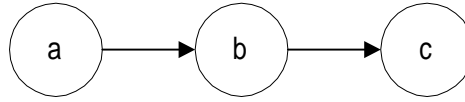


Figure 5.1. An example alias relation graph

[37][49][50]. In the discussion here, I distill from the previous research the essential parameters that affect alias analysis and explore the effect of the code transformations with respect to these parameters.

Before delving into the details of analysis, I introduce a number of conventions and terminology that are used in the subsequent discussions.

The “Points-to” representation: The discussion hereafter is based on the *points-to* abstraction of alias information [30]. Using this representation, x points-to y if x contains the address of y . The “Points-to” representation is commonly regarded as more compact and efficient than the explicit alias pair representation [37]. For example, the alias relations shown in Figure 5.1 can be represented with two points-to tuples $\langle *a, b \rangle$ and $\langle *b, c \rangle$, while in the explicit representation, this alias graph is represented by $\langle *a, b \rangle$, $\langle **a, c \rangle$, $\langle *b, c \rangle$, and $\langle **a, *b \rangle$ ⁶.

Complexity Variables: Listed below are the variables used throughout the complexity

⁶ The relationship $\langle **a, c \rangle$ and $\langle **a, *b \rangle$ can be inferred from $\langle *a, b \rangle$ and $\langle *b, c \rangle$ of the points-to representation

analysis discussion.

1. $NL(p)$ is the set of non-local variables that are visible in function p . In C , $NL(p)$ is the set of global variables.
2. $LOCAL(p)$ is the set of local variables of function p .
3. $LOCAL_{av}$ is the number of average local variables of all functions.
4. $PARAM(p)$ is the set of formal parameters of function p .
5. $PPARAM(p)$ is the set of formal parameters of function p that are pointers.
6. $PPARAM(p, i)$ is the i th formal parameter of function p that is a pointer.
7. $ARGU(q, p)$ is the set of arguments of call site q that calls p .
8. $ARGU(q, i, p)$ is the i th argument of function call site q that calls p .
9. $PARGU(q, p)$ is the set of pointer arguments of call site q that calls p .
10. $PARGU_{av}$ is the average number of pointer arguments of all function calls.
11. $ALIASE(a, d)$ is the set of aliases of object a after d levels of dereference.
12. AR is the number of alias relations currently holding. AR_{max} and AR_{av} denote the maximum and the average number of alias relations holding at any time, respectively.
13. AR_{sg} is the number of alias relations holding for an object due to a single level of dereference.
14. AR_{mul} is the number of alias relations holding for an object due to an arbitrary level of dereference

15. F is the number of functions in the program.
16. S is the average number of pointer assignment statements in a function.
17. B is the average number of blocks in a function.
18. FC is the number of function calls in the program. In the presence of function pointers
19. FCE is the number of edges in the PCG. FCE can be larger than FC when function pointer calls are allowed.
20. FAR is the average number of alias relations for a function pointer object, $FAR < F$

In section 5.3, I investigate the complexity of intra-procedural alias approximation methods. The discussion extends to the inter-procedural level in section 5.4. In section 5.5, I summarize the practical complexity discussion.

5.3. Intra-procedural Alias Analysis

Determining the range of possible aliases in a program is essential to decipher the behavior of the program statically—where and how a variable might be accessed carries information about the algorithm the program employs, and therefore is important to intelligent tampering or impersonation attacks. This section examines the complexity of intra-procedural alias analysis.

Intra-procedural analysis requires the examination of the statements within a function and the subsequent combinatorial analysis (if any) over their effect on aliasing. To be more specific, the intra-procedural phase entails the following operations:

- Process each pointer assignment statement to determine the effect of this particular statement on the alias relations.
- Analyze the combinatorial effects of the individual statements (either in a flow-sensitive or insensitive manner)

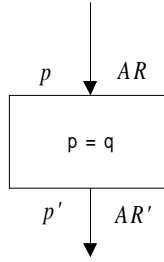
The above two steps may be repeated multiple times, for the inter-procedural analysis could result in new information that needs to be processed before the alias set converges. The discussion in this section is initially focused on intra-procedural analysis only, assuming the information propagated from other functions remains static. The effect of function calls will be examined in the inter-procedural alias analysis discussion.

5.3.1. Processing Pointer Assignment Statements

The only kind of statements where alias information might be modified is pointer assignments of the form:

$$Ptr = Expression(Q)$$

where the evaluation of $Expression(Q)$ returns an address of a memory location (either static or dynamically allocated), which is subsequently placed in pointer object Ptr . Assuming p and p' are the program points immediately before and after the pointer assignment statement, and AR is the set of alias relations holding at p , processing the assignment statement produces the set of alias relations AR' holding at p' (see Figure 5.2).



A transfer function $f : AR \xrightarrow{Stmt(Q)} AR'$ maps the pre-statement alias relations AR to the post-statement AR' . There exists a set of well-known transfer functions that handles different statements. Some are more complex than others [59]. These transfer functions specify a set of rules via which alias relations are modified. For example, the transfer function for the statement $p = q$ where p and q are both pointers is such that,

$$AR' = AR - (\text{any alias of } *p) + (*p, \text{deref}(q, I))$$

This transfer function kills the alias relations of $*p$ (since p is reassigned) and adds the alias relations between $*p$ and anything that q points to (since p is now pointing to whatever q points to). In Figure 5.2, if $AR = \{ \langle *p, a \rangle \langle *p, b \rangle \langle *q, c \rangle \}$, after the statement “ $p = q$ ”, AR' is $\{ \langle *p, c \rangle \langle *q, c \rangle \}$, and the alias relations $\langle *p, a \rangle \langle *p, b \rangle$ are killed by the statement.

More generally, for a statement of the form

$$Pi = Qj \tag{1}$$

where Pi denotes a pointer object i levels of dereferences away from P , and Qj is a

```

Transfer function f {
    if (deref(p, i) must alias to pointer object c)
        AR' = AR - (any alias relation of c) + <*c, deref(q, j + 1)>
    else
        AR' = AR + { <*a, b> | a is in deref(p, i) and b is in deref (q, j + 1) }
}

```

Figure 5.3. Transfer function for $P_i = Q_j$

pointer object j levels of dereferences away from object Q , the transfer function can be computed as shown in Figure 5.3.

The $deref(p, i)$ function in Figure 5.3 returns the set of objects that are i levels away from object P (for the alias relations shown in Figure 5.1, $deref(a, 2)$ will return c). The algorithm of $deref(p, i)$ is based on the points-to representation, and it is described in Figure 5.4.

The derefencing algorithm is similar to the alias query algorithm presented in Hind *et al.* [37]. The average and worst case time complexity of such an algorithm is respectively,

$$O(ARsg_{av}^{derefLevel}), \text{ and } O(ARsg_{\max}^{derefLevel})$$

where $ARsg_{av}$ and $ARsg_{\max}$ denote the average and the maximum number of alias relations for an object due to a single level of dereference.

```

function deref (p, derefLevel)
{
    if (derefLevel = 0)
        AliasSolution  $\leftarrow$  AliasSolution + p;
    else
        for each alias relation (*p, target) or (p, target), do
            deref (target, derefLevel -1)
        end do
    end if
}

```

Figure 5.4. Algorithm for dereferencing pointer variables

Most pointer assignment statements can be decomposed into a linear combination of statements with the form shown in (1). To process such a statement two calls to *deref* are required, as demonstrated by the transfer function in Figure 5.3. The result from calling *deref* is an alias set whose size is bound by AR_{mul} , and pairing the results of calling two *deref* is $O(AR_{mul} * AR_{mul})$. Therefore the complexity of applying the transfer function to process a pointer assignment statement is roughly,

$$O(ARsg_{\max}^{derefLevel} + ARmul_{\max} * ARmul_{\max}), \quad (2)$$

for the worst case and

$$O(ARsg_{av}^{derefLevel} + ARmul_{av} * ARmul_{av}),$$

for the average case.

5.3.2. Analyzing the Combinatorial Effect

In a flow-insensitive algorithm, the effect of individual statements on aliasing is simply

summarized together to represent the combinatorial effect. Using such an algorithm, the time complexity of a single intra-procedural analysis phase is,

$$O(S * F * ARsg_{\max}^{derefLevel} + ARmul_{\max} * ARmul_{\max}),$$

where S denotes the average number of pointer assignment statements in a function, and F denotes the number of functions in the program.

In a flow-sensitive analysis, the program control-flow is taken into account when conducting analysis. Alias relations are killed if assignments to the same destination occur sequentially, and they are combined at program points called *meet nodes* [18]. Meet nodes are actual or abstract nodes in the flow graph where different program flows meet. An example meet node is shown in Figure 5.5—node d is a meet node where the *yes* and the *no* branch of the conditional statement in a meet.

At each meet node, a union operation on the alias relations is performed, and the resulting alias relations are fed as the input to the meet node. Time complexity of this operation is bounded by the worst case measure,

$$O(\text{Number of meet nodes} * ARmax),$$

where $ARmax$ is the largest number of alias relations holding in an alias set at any given point.

If the flow graph is fully flattened (see the discussion in section 4.3.2), there exists only one meet node in each function—the switch statement node (see Figure 4.5). Therefore the worst-case complexity of combining alias relations at the meet nodes is $O(F * ARmax)$.

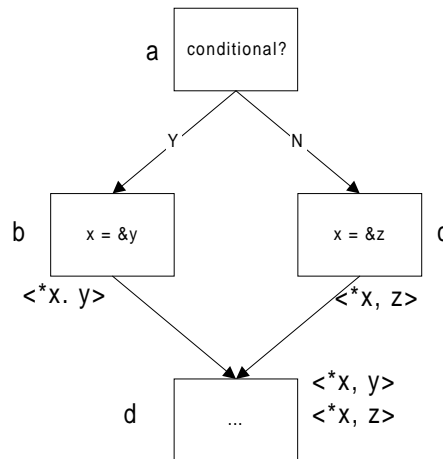


Figure 5.5. An example meet node

Armax), where F stands for the number of functions in the program.

If there exists any back edges in the CFG, a flow-sensitive analysis will attempt to iterate over the graph until the alias set converges. Each iteration consists of two steps:

- Use the alias results of the previous iteration (meet-over-all-path solution for one iteration) as input to the new iteration.
- For each control-flow path, compute the output alias set using the input and the transfer functions along the path.

In a fully degenerate flowgraph, alias relations cannot be killed based on flow information between blocks. Therefore the above steps simply become computing the meet-over-all-block alias set and disseminating that as input to each block for the next iteration. In other words, the flow sensitive analysis algorithm is essentially made flow insensitive (with respect to flow between blocks) during each iteration over the CFG.

The number of times an analysis iterates over the CFG is a function of the structure of the

flowgraph. In a fully flattened CFG, the minimum number of iterations required over the CFG is two—one to compute the alias information for each block, and one to disseminate the union of this information over each block⁷. The maximum number of iterations is bound by $O(B * ARmax)$ where B denotes the average number of blocks in a function, and $ARmax$ denotes the maximum number of alias relations holding in an alias set. The worst case measure of $O(B * ARmax)$ assumes each alias relation in $ARmax$ causes each block to generate new alias information which prompts yet another iteration.

It should be noted that while much of the discussions here focus on algorithm complexity, the issue of analysis precision remains important. Supposing there are n alias relations holding at the exit of each block, and further assuming there are m basic blocks in the function, in a fully degenerate flowgraph, there should be at least $n*m$ alias relations that hold at the exit of the function. This resulting alias set reflects the effect of all possible paths on aliasing. Note if the graph is not fully degenerate (i.e., some flow information is available), the final alias set should be smaller than $n*m$ since some of the alias relations will be killed in analyzing blocks that execute in sequence.

Finally, the complexity of a single flow-sensitive intra-procedural analysis pass is,

$$O(2 * (S * F * ARsg_{av}^{derefl_{level}} + ARmul_{av} * ARmul_{av}) + F * ARav), \quad (3)$$

⁷ This is a trivial lower bound

as a trivial lower bound, and

$$O((B * AR_{max}) * (S * F * AR_{sg_{max}}^{derefLevel} + AR_{mul_{max}} * AR_{mul_{max}}) + F * AR_{max}),$$

for the worst case.

5.4. Inter-procedural Alias Analysis

Inter-procedural alias analysis primarily handles information propagation between functions. In particular, two information propagation processes are needed for every function call: *forward binding* and *backward binding*. Forward binding maps the set of aliases holding at a call site in function f into aliases holding on entry into the called function g . Backward binding maps aliases holding at the exit of the invoked function g into aliases holding immediately following the execution of g in f . Figure 5.6 depicts this process.

The forward and backward binding operations, with respect to alias propagation, directly affect the complexity of the inter-procedural alias analysis. In this section, I investigate these binding operations and their complexity. The complexity discussion continues in the context of a C-like language with pass-by-value parameters and pointers.

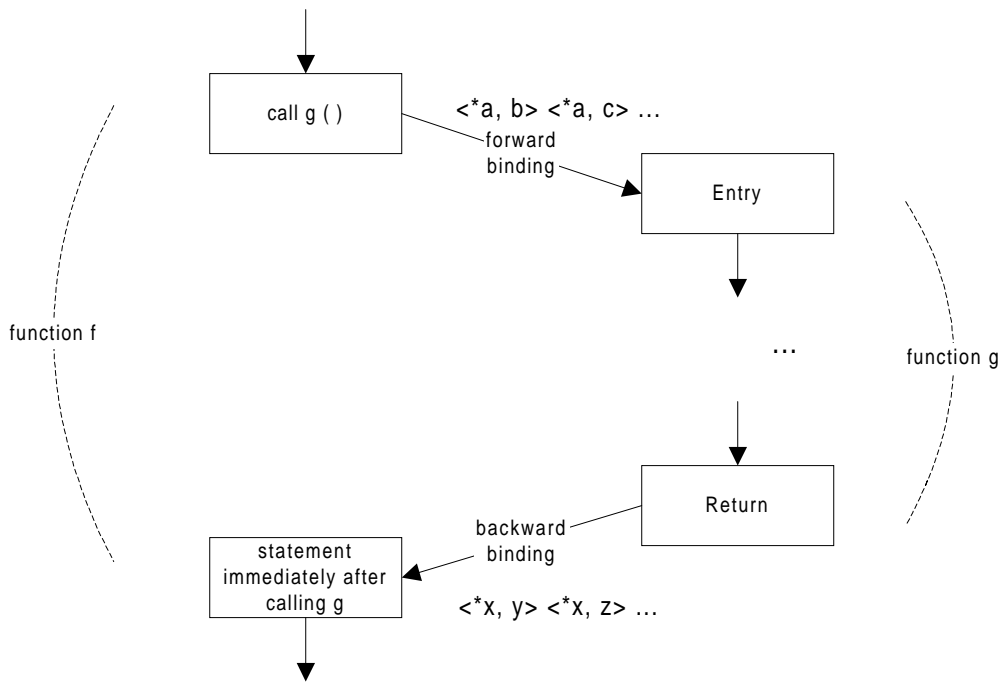


Figure 5.6. The forward binding and backward binding process

Forward Binding: An algorithm for forward binding is described in Figure 5.7. As shown in the algorithm, at the entry to function g , actual pointer parameters in the alias relations are replaced with the corresponding formal parameters. For example, consider the code segment in Figure 5.8. Alias relations $\langle *x, a \rangle$ and $\langle *y, a \rangle$ hold at the call site where f calls g . At the entry of g , the forward binding process replaces the actual parameter x and y with m and n ; that is, the alias relations $\langle *x, a \rangle$ and $\langle *y, a \rangle$ become

$$\{ \langle *m, a \rangle, \langle *n, a \rangle \}$$

as a result of the forward binding process.

```

forwardBinding ( f, g, q ) {
    for each pointer variable a in ARGU(q, g)
        such that a = ARGU(q, i, g), do
            replace <*a, x> in AR with
                <*PARAM(g, i), x> (or <PARAM(p, i), x>)
        end do
    end forwardBinding }

```

Figure 5.7. A forward binding algorithm – f calls g at call site q

This forward binding algorithm in Figure 5.7 is devised based on the *points-to* alias abstraction, and it is otherwise general in the sense that the complexity of the algorithm does not depend on any special implementation of data structures. The algorithm examines each pointer argument of the call, finds all its alias relations and replaces them with the corresponding formal parameter.

The average and worst case time complexity for the forward-binding process are,

```

int a, b;
f( ) {
    int *x, *y;
    x = &a;
    y = &a;
    g (x, y);
}

g(int *m, int *n) {
    int *x;
    x = &b;
    n = x;
    *m = *x + 1;
}

```

----- <*x, a> <*y, a> hold at this point

Figure 5.8. An example illustrating forward and backward binding

respectively, $O(PARGU(q, p) * AR_{av})$, and $O(PARGU(q, p) * AR_{max})$, where $PARGU(q, p)$ is the set of pointer arguments at call site q that calls p , and AR_{av} , AR_{max} denote the average and the maximum number of alias relations holding during execution.

Backward Binding: An algorithm for backward binding, in the presence of pass-by-value and pointer arguments, is described in Figure 5.9.

The backward binding algorithm discards each alias relation that involves a local variable or a non-pointer parameter, and replaces the formal parameters in the remaining alias relations with the corresponding actual parameters. Thus time complexity of the backward binding process is,

$$O (AR_{av} * (LOCAL(g) + PPARAM (g) + NL(g)))$$

for the average case, and

$$O (AR_{max} * (LOCAL(g) + PPARAM (g) + NL(g)))$$

for the worst case.

To see how the backward binding algorithm works, consider again the example in Figure 5.8. The set of alias relations holding at the exit of function g is $\{ \langle *m, a \rangle, \langle *n, b \rangle, \langle *x, b \rangle \}$. The backward binding algorithm eliminates the alias relation $\langle *x, b \rangle$ since x is only local to g , and replaces m and n with the variable x and y in f 's scope. The alias relations holding at the return from g are,

$$\{ \langle *x, a \rangle, \langle *y, b \rangle \}$$

Backward and forward: For each function call, there is a cost of forward and backward binding which propagate alias information back and forth between the called and calling function. For a single function call, the cost of the inter-procedural information propagation procedure is:

$$O (AR_{av} * (LOCAL(g) + PPARAM (g) + NL(g) + PARGU (q, p)))$$

for the average case, and

$$O (AR_{max} * (LOCAL(g) + PPARAM (g) + NL(g) + PARGU (q, p)))$$

for the worst case (g is the called function at call site q , and p is the calling function).

Thus the average overall time complexity for the inter-procedural analysis is,

$$O (FCE * AR_{av} * (LOCAL(g) + PPARAM (g) + NL(g) + PARGU (q, g))) \quad (4)$$

Where FCE is the number of edges in the PCG.

```

BackwardBinding (g, f, q) {
  For each(x, y) in current AR, do
    If either x or y is in Local(g), or in PARAM (g) but not in PPARAM(g)
      discard <x, y> from AR
    else
      replace <x, y> where x is PPARAM(g,i) with <ARGU(q, i, g), y>
    end if
  end do
end backwardbinding }

```

Figure 5.9. A backward binding algorithm— g returns to f after call site q

An important step in inter-procedural alias analysis is determining the structure of the PCG. In other words, the analysis must determine between which functions the forward and backward binding process is to take place. In a normal program, this process is simple since the PCG is mostly static and can be constructed in constant time as a preprocessing step⁸. However, in the presence of function pointers, the exact structure of the PCG is not known statically. Thus an approximation step is required to build a PCG on which inter-procedural analysis can progress.

The simplest strategy to follow in this—which I consider as the worst case behavior—is to assume that the destination of a function call is the set of all functions in the program. This strategy can be refined by the use of the number and types of parameters passed only when variable parameter lists are not used.

A more precise strategy is to determine the alias set of the function pointer at the call site and deem that the invocable set of functions. Such a strategy is detailed in Emami *et al.*[30]. The skeleton of the algorithm is described in Figure 5.10. This algorithm requires an inter-procedural analysis to determine the alias set of each function pointer when an indirect call site is encountered. In other words, building of the PCG has just been converted into a data-flow problem. The time complexity of this process is:

$$\text{The number of indirect call sites} * (\text{the number of aliased function} * \text{the cost of processing each call})$$

⁸ The constant time only applies to programs without recursive calls.

```

S1:  Building the PCG {
S2:      Process_call (null, main) ; }
S3:  Process_call (calling, called) {
S4:      forward_binding (calling, called)
S5:      for each indirect call site with fptr, do
S6:          conduct an intra-procedural pass to determine the alias set for fptr
S7:          for each function func in fptr's alias set, do
S8:              Process_call (this function, func)
S9:              update the alias set for fptr (and other function pointers)
S10:         backward_binding (called, calling)
S11: }

```

Figure 5.10. Algorithm for constructing the PCG

which translates to,

$$FC (F * (forward_binding + backward\ binding + the\ cost\ of\ one\ intraprocedural\ pass))$$

when each function pointer is aliased to the set of all functions, and

$$FC (FAR * (forward_binding + backward\ binding + the\ cost\ of\ one\ intraprocedural\ pass))$$

when the function pointer alias set is FAR , where $FAR < F$.

5.5. Iterations Over the PCG

Described in section 5.3 and 5.4 are practical complexity measures for intra and inter-procedural alias analysis. The complexity formulas (1) and (2) represent cost of a single intra- or inter-procedural analysis pass. However, an aggressive analysis algorithm might attempt to iterate over the PCG and repeat these analysis passes in an effort to gain more precision. Such an iterative method might yield more accurate results at the price of being

more costly in terms of both running time and state space. One of the state-of-the-art alias analysis algorithms—the NPIC algorithm (by Hind *et. al.*)—indeed employs an iterative method which interleaves the inter- and the intra- phase of the analysis [37]. The NPIC algorithm is generally regarded as one of the most aggressive and more sophisticated algorithms. It has been reported that NPIC is capable of producing more precise results than most other algorithms in the same category.

The number of iterations over the PCG is determined by how quickly the alias set reaches convergence. Unfortunately, it is impossible to conjecture a lower bound for the number of iterations since that depends on the alias computation within each function.

In the worst case, every alias relation can cause the algorithm to perform an iteration over the PCG. Since each function may have up to AR_{max} alias relations holding in their entry and exit alias set, the maximum number of iterations over the PCG is

$$O (AR_{max} * F),$$

where AR_{max} is the maximum number of alias relations holding at any time and F is the number of functions.

During each iteration, the intra-procedural and the inter-procedural phase must be repeated. Thus the overall complexity of iterating over the PCG and repeating the intra and inter-procedural analysis is bounded by the worst-case measure,

$$\begin{aligned} &O ((AR_{max} * F) * ((FCE * AR_{max} * (LOCAL + PPARAM + NL + PARGU) \\ &+ (FC (FAR * (AR_{max} *(LOCAL + PPARAM + NL + PARGU)))) \end{aligned}$$

$$\begin{aligned}
& + ((B * AR_{max}) * S * F * (AR_{sg_{max}}^{derefLevel} + AR_{mul_{max}} * AR_{mul_{max}}) \\
& + F * AR_{max}))))
\end{aligned} \tag{5}$$

5.5. Putting Together the Complexity Argument

The complexity measures described in section 5.3 and 5.4 represent the time requirement for a comprehensive alias analysis. The measures showed that the parameters affecting alias resolution include the following:

- Size of the alias set AR (this includes function pointer aliases)
- Levels of pointer dereferencing
- Size of the parameter list (this also include the size of local variable lists)
- Size of the program (number of blocks, functions, pointer assignment statements, etc.)
- Number of edges in the PCG (as perceived by the static analyzer)
- Number of function calls (distinct static call sites)

The complexity of alias analysis can be made worse by increasing each of these parameters, and the effect of increasing the parameters can be easily reasoned about using complexity measures #1 through #5.

Note that the transformations introduced in Chapter 4 affect several complexity parameters directly. For example, the parameter FCE in complexity measure #4

represents the number of edges in the program call graph as perceived by the static analyzer. The discussion in Section 4.4.2 demonstrated how false static edges can be added to the PCG to confuse a static analyzer.

The worst-case measures are rarely accurate indicators of performance. In practice, the number of times a function is visited and a transfer function is evaluated maybe far from the worst-case behavior. However, consider a program with a flattened control-flow, a degenerate call structure and an abundant number of aliases. A function might be visited each time a function call site is encountered (consider a fully connected program call graph), and a basic block will be analyzed each time a branch instruction is met (consider a fully degenerate flow graph). In such a case, the analysis complexity is close to the worst-case estimate. Such an analysis will eventually converge and report a set of results that include as many spurious alias relations as it possibly can; in several occasions when tested against automated analysis algorithms, the analysis algorithm halts with a report that every pointer variable is aliased to every memory location that appears on the right hand side of assignment statements [79] (also see Chapter 7).

The complexity studies in this chapter confirm that the degeneration of the program control flow and call structure render a data-dependent program flow structure that can be made difficult to analyze. The analysis difficulty arises primarily due to the fact that the control-flow and data-flow analyses are made strongly co-dependent by the code transforms. This ubiquitous co-dependence is the source of a) reduced analysis precision, and b) increased complexity of both control-flow and data-flow analyses.

Chapter 6

Implementation

In this chapter I describe the implementation of a prototype of the One-way Translation mechanism. With the help of my colleague, Jonathan Hill, I have implemented the prototype in a SUIF *C* compiler [2]. The code transformations are implemented as SUIF passes that operate on the SUIF intermediate representation of the program. The transformed intermediate code is then translated back into a *C* source code, which can be compiled independently for any target platform.

At this level of automation, the programmer is required to specify some transformation parameters such as the level of control-flow flattening and aliasing. Fortunately, this step does not appear to be terribly burdensome since it involves no more than setting the values of a few parameters (easy default values are available). In this chapter, I describe the implementation of this One-way Translation compiler, and the specific design choices that I have made.

6.1 Design Goals

Based on the discussions in the previous chapters, the following design goals are identified for the One-way Translation compiler.

- Platform-independence
- Modular support for different code transforms
- Easy extension to other high-level languages

For the prototype compiler, I selected ANSI C as the target language and the SUIF2 compiler from Stanford as the infrastructure to implement code transformations [2]. The compiler takes an input C program, performs code transformations and produces as output a new ANSI C code. The resulting C code can then be compiled using any ANSI C compiler. Since the code transforms operate on the SUIF representation which is largely architecture independent, these choices meet the first requirement, *platform-independence*.

The types of code transformation may evolve as new protection techniques surface. It is important therefore to have modular support for incorporating additional transformations. SUIF provides a programming environment that allows modular development, easy integration, and easy interoperation of different compiler passes. These features made SUIF an ideal candidate for implementing the compiler.

Many source-to-source translation tools exist between ANSI C and other languages such as C++, Pascal and Fortran. Programs written in those languages can be translated into

ANSI C before undergoing the code transforms. Therefore the choice of ANSI C as the target language allows extensions to other languages. Finally, the SUIF team is designing a JAVA front-end that, when available, can be incorporated into the One-way Translation compiler to handle JAVA programs.

It should be noted that while some of the code transformation techniques appear to depend on specific high-level language features such as C's explicit pointer manipulation, the fundamental concepts can be duplicated at a lower level that is independent of the high-level language. For example, pointer manipulations can be achieved at the assembly level even when the high-level language does not provide language support for such manipulations. I choose to implement the prototype in C, at the source level, for two reasons: *platform independence* and *ease in implementation*. The choice of my target language does not preclude the application of the fundamental techniques to programs written in other high-level languages.

6.2 The SUIF Compiler

The SUIF compiler is designed to provide modular support to facilitate research in compilation techniques [2]. The compiler uses an intermediate representation called SUIF and a set of well-defined capabilities to manipulate the SUIF representation. The attractive properties of the SUIF compiler include the provision of useful abstractions and frameworks for developing compiler passes and a programming environment that allows different passes to easily inter-operate.

The intermediate representation produced by the SUIF front end is a complete abstract

syntax tree of the program. Having a complete syntax tree to work with is essential for making global modifications of the program. Typically, code manipulations are implemented as SUIF passes that operate on the syntax tree. The passes are then made into shared libraries that can be imported dynamically during compilation. Different passes can be implemented independently of each other and later combined together to realize a particular functionality.

6.3 Implementation in SUIF passes

In this section, I describe the implementation issues of the fundamental set of transformations performed by the prototype compiler.

Throughout this section, I make certain assumptions about the input code that help to simplify the implementation of the transformations. These assumptions are:

- There are no function parameters in function signatures. Although the transforms make extensive use of function pointers, function parameters in the original program complicate many of the code transformations and make it difficult to maintain the original semantics of the program. Therefore the prototype implementation, as it stands now, does not handle programs with function parameters.
- A limit is placed on the number of parameters a function can have, and that is typically a small integer (i.e., extra long function signatures are disallowed).

It should be noted that although these assumptions appear to be limitations of the current

prototype implementation, they are not fundamental to the principles underlying the code transforms—they are enlisted for efficiency and simplification reasons only.

6.3.1 Function Signature Unification

One of the fundamental transformations performed by the One-way Translation compiler is the function-call modification in which direct function calls are converted to function pointer calls. To facilitate such a transformation, function signatures need to be unified so that only a small number of distinct signatures exist in a program (see the discussion in section 4.4).

Unifying function signatures inevitably means creation of new parameters and return types. Care must be taken so that the unification process does not incur unnecessary run-time costs. Chapter 4 discussed a simple scheme that unifies function parameter lists to a superset. Here I present a more sophisticated and less costly scheme to unify function signatures.

First, a destination signature length is chosen. This specifies the number of parameters the unified signature will have. The destination signature length can be specified as an option by the programmer. Alternatively, the compiler chooses a length that is the average of all signature lengths in the program.

Second, the parameter types are chosen, and this is done using the following algorithm:

- The compiler walks through the program and collects all data types that appear as

parameters in function calls. Non-primitive types such as arrays or data structures are replaced with a *void** type. Each data type has a rating associated with it, and each time a type is encountered, its rating is incremented by “1”. For example, in examining the following function signatures:

func1 (int, int, char)

func2 (int, char)

*func3 (int *)*

the type *int* receives a “3” rating, while *char* receives a “2”, and *int** receives a “1”.

- The compiler chooses *n* data types with the top ratings as the final parameter types (*n* is the destination signature length). If the number of distinct types is less than *n*, the compiler adds additional parameters by traversing from the top of the list, adding a second parameter of the highest-rated type, then the second highest, and so on. This process continues until the desired signature length is reached. For example, consider a destination signature with three parameters. If the type collection has rendered the following information: *int*(20), *char*(13), the final signature is then (*int, int, char*).

Now the function signatures, bodies and the calls are modified as follows:

- For each parameter that is not included in the final function signature, a global variable of that type is created. At each call site to the function, the value of the original argument is written to the global variable. Inside the function body, references to the original parameter are replaced with references to the global variable (see example in Figure 6.1).

- In Figure 6.1, function func2's signature is changed from *(int, int, char)* to *(int, char)*. To facilitate this change, a global variable `_para2` of the type *int* is created. Before calling func2, `_para2` is initialized to contain the value of the original argument, `para2`. Inside func2, the original reference to `para2` is replaced with a reference to the global variable `_para2`.
- For each added parameter, if its type is the same as an original parameter, the same value is bound to both the original and the new parameter. Figure 6.2 illustrates such an example. In Figure 6.2, function func2's signature is changed from *(int, char)* to *(int, int, char)*. At the call site in func1, the same value is passed to `para1` and the new `para2` since they are both integers. Inside func2, one of the references to `para1` is replaced with a reference to `para2`. The modified code is indicated with bold text.

When an added parameter also adds a new type to the parameter list, this new parameter needs to be referenced inside the function. Otherwise it is a trivial task to find the artificial parameters since they are not used in the function (or the optimizer may decide to omit this parameter since it is not used). If this new parameter happens to be a pointer type, and if there happens to be a global variable that is of the type this pointer points to, the compiler passes the address of the global variable to this new parameter at the call site. Inside the function, references to the global variable are randomly replaced with references to the new parameter. Furthermore, if the new parameter is a pointer type that can be made to point to either a local variable or another formal parameter of the function inside the function, this parameter is assigned the address of these variables, and computations are changed accordingly.

```

func1( ) {
    int para1, para2;
    char para3;
    ...
    call func2(para1, para2, para3)
}

```

```

      :
      call
      func2
      ▼

```

```

func2 (int para1, int para2, char para3) {
    x = para1;
    y = para2;
    z = para3;
}

```

(a) Original program

```

int _para2;           --- global variable
func1( ) {
    int para1, int para2;
    char para3;
    ...
    _para2 = para2;    --- initialize global
    call func1(para1, para3)
}

```

```

      :
      call
      func2
      ▼

```

```

func2 (int para1, char para3) {
    x = para1;
    y = _para2;
    z = para3;
}

```

(b) Transformed program

Figure 6.1. An example illustrating cutting parameters

```

func1( ) {
    int para1;
    char para3;
    ...
    call func2(para1, para3)
}
      .
      call
      func2
      ▼
func2 (int para1, char para3) {
    x = para1;
    y = para3;
    z = para1;
}

```

(a) Original program

```

func1( ) {
    int para1;
    char para3;
    ...
    call func1(para1,para1 , para3)
}
      .
      call
      func2
      ▼
func2 (int para1,int para2 , char para3) {
    x = para1;
    y = para3;
    z = para2;
}

```

(b) Transformed program

Figure 6.2. An example illustrating adding parameters

If the new parameter is not a pointer type, the compiler checks to see if there exists a global variable of the same type. If the answer is no, a global variable of the type is created. If there is no reference to the global variable inside the function, spurious operations involving the global variable and the new parameter are created. Before the function returns, the global variable is restored to its original value. In the case where there already exist legitimate operations involving the global variable, the operation is replaced with a copy from the global variable to the artificial parameter, the operation performed on the parameter, and a copy back to the global variable.

6.3.2 Control-flow Flattening

As described in Section 4.3.2, degeneration of the program control-flow is central to the One-way Translation idea. To support this transform, the compiler first builds the control-flow graph of every procedure, then it converts it to a universal structure with the form shown below,

```
While ( )
{
  Switch ( )
  {
  }
}
```

In this structure, the blocks of the switch cases are the basic blocks of the function. The compiler provides an option for the programmer to specify the extent to which the function control-flow is to be degenerate. Replacing direct branches with data-dependent branches has a cost in run-time. Therefore fully flattening the control-flow may not be an economical choice. (Chapter 7 investigates the cost associated with the control-flow

flattening operation.)

When the programmer specifies the percentage of the branches to be transformed, it is a random choice on the compiler's part to choose which branch statements to transform. This can obviously be improved by the adoption of a random selection policy of some intelligence. For example, it is advantageous to transform branches that will not be heavily executed during the execution. This may require a preprocessing step to identify loops, an optimization that is not currently considered in the prototype implementation.

Another transform associated with the control-flow flattening is the introduction of artificial basic blocks. In the current implementation, these blocks contain only pointer assignment statements in order to confuse the alias analyzer, and they are never executed. The number of artificial blocks to be inserted can also be specified by the programmer, or left to the default (50% increase of the number of basic blocks). To implement the addition of the artificial blocks, the compiler makes a pass through the function and notes the number of block labels and the jump instructions that utilize them. A new set of labels is then generated including the ones for the new blocks. The labels are organized in such a way that labels for the artificial blocks are interleaved with the legitimate ones. The jump instructions are changed accordingly. Note this step is performed before the flattening takes place.

6.3.3 Inter-procedural Alias Insertion

One of the major transforms the One-way Translation compiler performs is the creation of inter-procedural aliases. The discussions in Section 4.4.3 described several different

types of inter-procedural aliases and how they might be introduced in the program. In this section, I revisit this transform with the focus on implementation rather than mechanisms.

Section 4.4.3 identifies four different ways aliases can be created as result of function calls. They are: *global-to-local* aliases, *parameter* aliases, *side-effect*, and *return-value* aliases. The discussion below considers these cases in turn.

Global-to-local aliases: To create global-to-local aliases, the compiler performs the following steps.

- Perform one pass over the program to find all references to global variables in the program.
- For each function that contains at least one reference to a global variable, its signature is modified such that a new pointer parameter is introduced for each reference to a distinct global variable.
- At the call site, this new pointer parameter is bound to the address of the corresponding global variable.
- Inside of the called function, the new parameter and the global variable become aliases. References to the global variable can be replaced with dereferences via the pointer parameter.

```

int i, j;
int f() {
    i = 1;
    j = g ();
}
g () {
    i ++;
    i ++;
}

```

(a) original program

```

int i, j;
int f() {
    i = 1;
    j = g ( &i );
}
g (int* a ) {
    *a = *a + 1;
    i ++;
}

```

(b) transformed program

Figure 6.3. An example illustrating global-to-local aliases

Note that more than one parameter can be introduced for each global reference, in which case there are more than two ways to reference the global variable inside the called function. The compiler allows the programmer to control the transform by specifying the percentage of the global references that should be modified and the extent to which pointers to a global variable are replicated. Figure 6.3 depicts an example transform

```

f () {
    int i, j;
    g ( &i)
}
g (int *a)
{
    *a = *a + 1;
    return *a;
}

```

(a) original program

```

f () {
    int i, j;
    g ( &i, &i);
}
g (int *a, int *b)
{
    *a = *b + 1;
    return *b;
}

```

(b) transformed program

Figure 6.4. An example illustrating parameter aliases

where a pointer parameter to a global variable is passed to a function, and one reference to the global variable inside the called function is replaced with a reference through the pointer parameter. The bold text indicates modified code.

Parameter aliases: When two or more pointer parameters are bound to the same address, they become aliases inside the called function. In order to create aliases of this type, it suffices to replicate existing pointer parameters. Figure 6.4 illustrates such an example. In Figure 6.4, function *g* is called with two identical addresses. Inside *g*, *a* and *b* are aliases to each other.

To accomplish this transform, the compiler performs a pass over the program, and identifies all the pointer parameters. The programmer can specify the percentage of the pointer parameters to be replicated (the default is zero percent), and the extent to which they are replicated (one or more duplicates). Function signatures are then modified to include the new parameter(s). Call sites are modified accordingly to bind the same address to the new parameter(s).

Intra-procedural aliases through inter-procedural means: One of the transforms described in Section 4.3.3 is the creation of intra-procedural aliases. In addition to straightforward pointer manipulations (pointer assignments and arithmetic operations), aliases can be created as a result of function calls (i.e., using inter-procedural mechanisms to create intra-procedural aliases, this possibility is first discussed in Section 4.4.3).

```

f( ) {
    int *i, *j;
    g (&i, &j);
    ...
}
g (int **a, int **b) {
    *a = *b;
}

```

-----// i and j become aliases here

Figure 6.5. Aliasing through function call side effects

In order to create aliases between two pointer objects inside function *f*, the compiler chooses a destination function *g*, which is called by *f*. Two additional parameters (both addresses of pointers) are created in *g*'s signature, one for each of the pointer objects. At the call site, the addresses of the two pointers are passed as arguments to the newly created parameters. Inside *g*, *g* simply assigns the address contained in one object to another, thus creating an alias relationship. Such an example is shown in Figure 6.5.

6.4 Preprocessing

Several preprocessing steps are performed before the code transformations take place. In this section, I examine these preprocessing steps. In theory the preprocessing is not integral to the fundamental code transformations, its only function is to simplify the transforms. However in practice the preprocessing operations play an important part in enforcing the correctness and efficacy of the transforms.

6.4.1 Variable Declaration Motion

In ANSI C, scoping rules state that variables defined in an outer scope are visible from the inner scopes, but not vice versa. When the control-flow is flattened, the scopes disappear. What this means is that variables defined in outer blocks are no longer visible to the inner blocks since they have been raised to the same level. To circumvent that, prior to block flattening, the compiler performs an operation, called *variable declaration motion*. This operation makes a pass over all the basic blocks within a function and moves all the variable definitions to the outermost scope of the function. The variables are renamed to unique strings to avoid naming conflicts.

6.4.2 Function Signature Preprocessing

In order to simplify the unification of function signatures, the compiler performs a preprocessing step to replace non-primitive parameters such as structures or arrays with a *void ** type. The implementation of this step scans the entire program and locates all function signatures that contain non-primitive parameters. When such a signature is encountered, the particular parameter in question is replaced with a new parameter of the type *void **. Inside the function, references to the original parameter are replaced with dereferencing the new pointer parameter, following a cast to make the pointer point to the original type. The compiler then performs another pass, and finds all corresponding call sites. At each function that contains such a call site, a new local variable of the type *void ** is created, and it is assigned to the address of the non-primitive data object prior to the call. At the call site, the physical call is modified such that instead of passing the non-primitive data object, the code passes the pointer object instead. An example of function

signature preprocessing is illustrated in Figure 4.12 and Figure 4.13 in Chapter 4.

6.5 Correctness Discussion

When a program is transformed during compilation, there is always the issue of whether the transformation is performed correctly—whether the resulting code preserves the semantics of the original program. Formally proving the correctness of the transforms is beyond the immediate scope of this dissertation. In fact, the compiler community has not solved the general correctness question regarding the more traditional code transforms performed by optimizing compilers.

While the issue of correctness is not dealt with directly, I point to a few research efforts that address the various aspects of the correctness problem. Translation validation, for example, is a technique designed to check the result of each compilation against the source program and to detect errors on-the-fly [68]. Nacula proposed a practical framework for translation validation within which small instances of code transforms (described as a pass) can be validated [64]. Nacula showed that it is possible to implement a practical translation validation infrastructure with about the effort typically required to implement one compiler pass. Since in this work the transformations are implemented as compiler passes, I believe that while it may not be possible to prove that the compiler is always correct, it is possible to check the correctness of each compilation using translation validation techniques.

6.6 Implication on debugging

Programs that are transformed using the methods described in this work do not follow traditional program constructs and hence are particularly hard to debug. The transforms incorporate program features that are generally considered bad programming practice, such as the liberal use of *gotos* and *pointers*. While it is nearly impossible to debug such programs, the strategy I advocate is to debug the transformer (in this case the compiler) and ensure that each transform is applied and implemented the way it is intended.

Chapter 7

Empirical Evaluation

In this chapter I present the empirical evaluation of the code transforms described in the previous chapters. The essential functionality of the One-way Translation compiler was implemented and tested on a variety of applications. The evaluation consists of performance studies of the transformed applications and experimental results when measured against existing software analysis tools.

7.1. Evaluation Criteria

Based on the nature of the code transformations and their intended use, the following set of evaluation criteria was developed:

- Performance overhead of the transformed program
- Performance of analysis tools when tested on the transformed program
- Precision of the automated analysis

Performance overhead: The code transformations developed in this work make significant modifications to the target program. They transform the basic structure of the program, introduce pointers and aliases, insert artificial blocks and modify the function call behavior. These transforms will have considerable impact on the program's run-time performance, and it is important to understand what that impact might be.

The notion of performance overhead considered here includes two facets: *execution-time slow down* and *code-space expansion*. Both are examined in the experiments.

It is important to note that the fundamental reason that a program might want to undergo such transformations is to protect the program's operation (which translates to knowledge of its internal algorithm, data structures, etc.) from malicious environments. Therefore the security strength of the mechanism is of primary concern, not the performance penalty.

Efficiency of static analysis tools: In practice, program analysis will be conducted with automated tools. It is therefore of interest to see how the transformations measure up against existing analysis tools. To that end, I choose to experiment with alias analysis tools for two reasons. First, aliasing is a fundamental technique underlying the One-way Transforms, so the difficulty in alias analysis constitutes a baseline estimate. Second, alias analysis is a well-defined form of program analysis—automated tools for other types of program analysis are less likely to be as mature or as general. By efficiency here, I mean the running time of the tool before the analysis reaches closure and terminates.

Precision of alias analysis: Precision of alias analysis indicates how accurate the analysis result is with respect to the true alias relations.

Both analysis precision and the efficiency of analysis are important criteria. They represent the two facets of the analysis quality. An analysis may reach closure quickly, but if the results are imprecise, it will be of little use. On the other hand, accurate analysis results, if not reached within a reasonable amount of time, are equally ineffective, especially with time-varying obfuscations. I consider both analysis efficiency and precision in the experiments and in analysis of the results.

7.2. Performance Overhead

In considering the performance overhead of the transforms, a related factor is compiler optimization. Two issues regarding optimization are of interest. First, since the code transformations are performed at the source-code level, it is essential to understand the impact of optimization on the transformed program. In particular, the issue of whether the optimizer will undo the code transformations is a major concern. Second, it is also of value to examine the impact of the code transformations on the effectiveness of compiler optimizations.

With these issues in mind, experiments to measure the transformed program performance were conducted. An ideal target program for the experiment is of course the network probe program described in Chapter 2. However, the survivability architecture, along with the probing mechanism, is still in heavy development and therefore is not suited for this experiment.

The target programs that were used in these experiments included three programs from the SPEC95 benchmark program suit and STIDE, an intrusion detection program from

the University of New Mexico. The three SPEC programs are *Compress95*, *Go* and *LI*. The choice of the three SPEC programs is based on their respective characteristics: *Go* is a program that implements *Wei Qi*, the Chinese board game. *GO* is specially branch intensive. *Compress95* is a loop intensive compress algorithm. *LI* is a LISP interpreter program with heavy nesting structures. STIDE performs primarily sequential processing on large input files. These programs embody a wide range of program characteristics, and are good representatives of real-world programs. Table 7.1 describes the five test programs and their characteristics.

The platform for the experiments is a 50MHZ, 4-CPU SuperSPARC with 512 MB of RAM, running Sun OS 5.51. Tests with and without optimizations (with the *gcc -O* option) were performed.

Program	Description	Size(bytes)	Number of static branches
Compress	A compress algorithm	106,440	459
GO	Chinese board game	699,172	10,857
LI	LISP interpreter	118,980	2728
STIDE	Intrusion Detection Program	819,072	1868

Table 7. 1. Test programs (source before transformations)

7.2.1. Impact of Branch Transformations on Performance

The first set of experiments is designed to show the impact of control-flow flattening. These experiments are performed with no artificial blocks inserted and no changes made

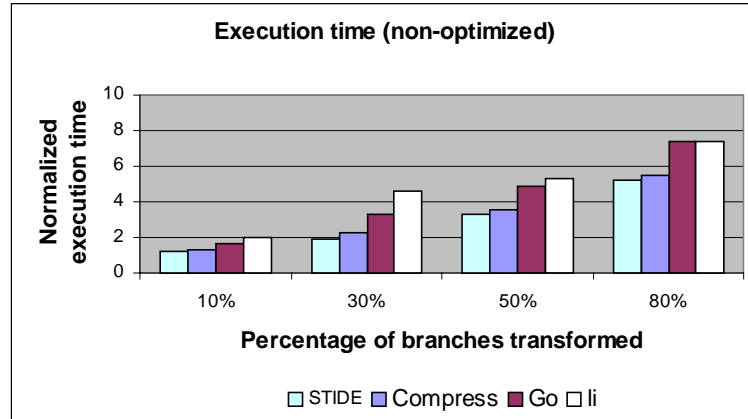


Figure 7.1. Execution time without optimization — branch transformation only, no artificial blocks added, no function unified

to the function call structure. Dynamic computations of the branch targets are performed through index computation via a global array. Three array accesses, on average, are made for each branch target computation.

Figure 7.1. shows the (normalized) execution time of the test programs without optimization. As can be seen in Figure 7.1, the performance slow-down increases exponentially with the percentage of transformed branches in the program. With a 50% branch transformation, the average performance slows down by about a factor of 4.

This result is intuitive. In this transform, each direct branching statement is replaced with one direct branch (to the switch statement) and one indirect branch (through the jump table). This substitution, along with the array accesses, consists of expensive operations that cause considerable performance overhead at run-time.

An interesting case is *Compress* which has the least number of static branches. Due to its loop-intensive nature, *Compress*'s branches are executed often at run-time, and this

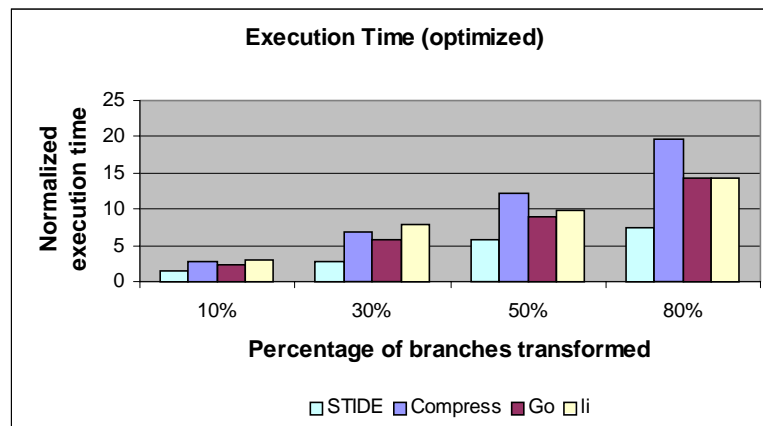


Figure 7.2. Execution time with optimization

explains why the performance slowdown of *Compress* is comparable to that of *STIDE*'s.

Figure 7.2 shows the execution time with optimization. The other parameters remain the same. Across the board, the performance slowdown is much worse with optimization. On average, a factor of 4 increase for 50% branch transformation in an non-optimized version has turned into an increase by a factor of 9.57 when optimized. Again, *Compress* is the interesting case. Figure 7.3 shows the execution time of the original programs, with

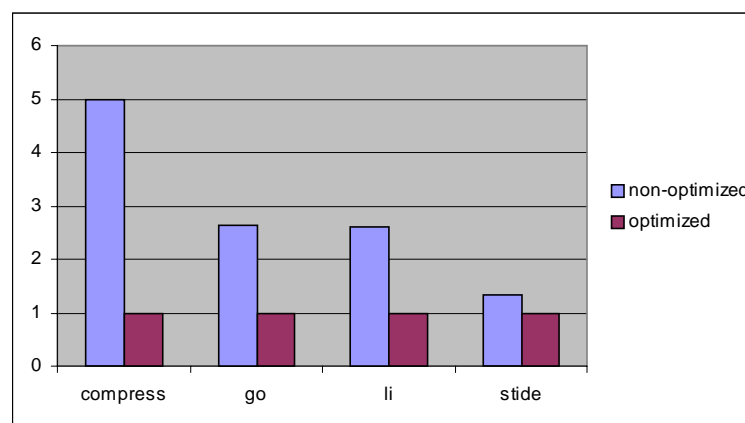


Figure 7.3. Execution time of the original programs, with and without optimization

and without optimization. Observe how compiler optimization performed best on *Compress* among the untransformed programs—a whopping 80% decrease in the execution time due to optimization. However, optimization performed much worse on the transformed *Compress*. As *Compress* is a loop-intensive program, what we observed was that the transforms disabled certain forms of loop or loop kernel optimization.

This is encouraging because an optimizer is essentially a program analyzer. Although its goal is code improvement, it nevertheless follows the general principles of program analysis. The fact that the transforms were able to obstruct the analysis performed by the optimizer is another empirical result supporting the thesis that the code transformations are working.

In the case of *STIDE*, optimization before and after the transforms had little impact on its performance. This is mainly due to its sequential processing nature—the program spends most of its time executing large blocks rather than branching between blocks.

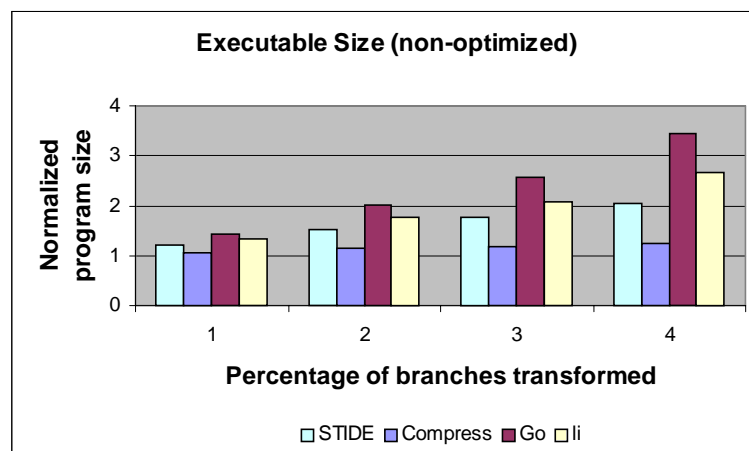


Figure 7.4. Executable size without optimization

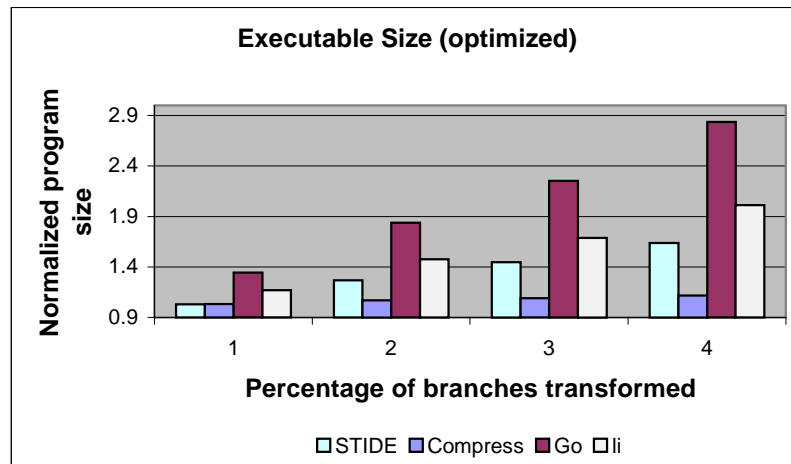


Figure 7.5. Executable size with optimization

Figure 7.4 and Figure 7.5 depict the code size expansion, with and without optimizations. As can be seen in Figure 7.4 and Figure 7.5, code size expansion is fairly limited with branch-only transformations. With an 80% branch transform, the average program size increased about 84% in the non-optimized version and 50% when optimized. *GO*, being the one with the largest number of branches, experienced the largest code expansion; for an 80% transform of branch instructions, the executable size of *GO* increased by 200%. Conversely, *Compress* experienced the least increase in size since it had the least number of branch instructions.

At present, a random algorithm is used to choose which branch to transform. An obvious future improvement is to employ some form of intelligent policy to perform the following: a) identify the regions of the program that require greater protection from static analysis, and b) perform transformation on the less-often-executed branches for better performance results.

7.2.2. Impact of Pervasive Aliasing

As described in Chapter 4 and Chapter 5, aliasing is fundamental to the security strength of the code transformations. This set of experiments is performed to investigate the impact of pervasive aliasing on the program performance.

In these experiments, the following four types of alias were introduced:

- For every function that contains references to global variables, the global variable doubled as a formal parameter. References to the global variable inside the function were randomly replaced with references to the parameter.
- In every function, a random number of pointer variables were introduced (a number between 1 and the number of existing local variables). These variables were pointers to common data types (a preprocessing step was used to determine the most commonly used data types).
- The compiler randomly assigned these pointers to local variables and global variables; at least one variable was aliased to the global array that contained data values to compute branch addresses. A subset of these assignments were performed inter-procedurally (see the discussion on inter-procedural aliasing in Section 4.4 and Section 5.4 for details). Subsequently, existing computations were updated with references through these pointers.
- A small number of basic blocks (a random number between 1 and half of the existing basic blocks in the function) were introduced. These blocks were never executed, but they contained spurious assignments to the pointer variables.

After the transform, the programs were executed and performance results were collected for a 15-run average. Table 7.2 and Table 7.3 contain the execution time and code size for the non-optimized programs.

	10%		30%		50%		80%	
	Before	After	Before	After	Before	After	Before	After
Compress	762	802	1365	1611	2153	2839	3292	4501
Go	522	680	1017	1697	1491	3708	2274	6435
LI	1192	1442	2787	4152	3178	6006	4424	9378
STIDE	506	608	765	1215	1328	2354	2122	4398

Table 7.2. Execution time (in seconds) without optimization, with variable branch transforms, before and after aliasing

	10%		30%		50%		80%	
	Before	After	Before	After	Before	After	Before	After
Compress	113	140	121	160	125	177	132	189
Go	1002	1478	1417	2126	1807	2675	2402	3300
LI	223	330	296	412	350	560	448	657
STIDE	983	1232	1253	1786	1449	1876	1679	2314

Table 7.3. Executable size (in Kbytes) without optimization, with variable branch transforms, before and after aliasing

Aliasing incurred further performance slowdown as shown in Table 7.2. However, the impact of aliasing was not evenly distributed. In the case of *Compress*, the presence of aliases generated a further performance slowdown between 5% and 30%, while aliasing caused the performance of *GO* to drop another factor of 2 or 3. As *GO* had many basic blocks, and as some of the aliases and alias-related computations were introduced on a per-block basis, aliasing had the most effect on *GO*'s performance—for a 50% branch transform, *GO*'s performance overhead increased from a factor of 5 to beyond a factor of

10. As expected, *Compress* and *STIDE* had comparatively less performance overhead due to aliasing than the other programs. *STIDE* spent most of its time in sequential computations, so the cost of the pointer assignments and indirect addressing was amortized while *Compress* has fewer branch computations that were affected by aliasing.

On average, the increases in program size due to aliasing were between 12% and 50%. This increase was more predictable since the aliasing scheme inserts a known number of basic blocks and extra instructions.

Table 7.4 and Table 7.5 contain the performance data and code size for the optimized programs. The performance results in Table 7.4, along with the ones in Table 7.2, show that optimizations continue to be hindered—the presence of aliasing decreased the reduction in execution time produced by optimization.

Aliasing appeared to be the most effective in counteracting optimization when the program control-flow was not completely degenerate (in which case optimization could perform fairly well without aliasing). As the program control-flow became more degenerate, significantly less reduction in execution time induced by optimization was observed.

Applying optimizations here (with full-scale aliasing and control-flow flattening) produced negligible differences in the program size as demonstrated by the data in Table 7.5. (average 5% decrease in size). This further implied that the artificial blocks that were placed in the program had evaded the notice of the optimizer—another encouraging piece of evidence that the aliasing scheme was taking effect, at least to the degree of impeding

compiler optimizations.

	10%		30%		50%		80%	
	Before	After	Before	After	Before	After	Before	After
Compress	332	648	812	1167	1457	2074	2354	3719
Go	285	428	682	1364	1054	2940	1685	4200
LI	676	1108	1845	3324	2262	4550	3282	7815
STIDE	404	502	648	1104	1300	2073	2040	4311

Table 7.4. Execution time (in seconds) with optimization, with variable branch transforms, before and after aliasing

	10%		30%		50%		80%	
	Before	After	Before	After	Before	After	Before	After
Compress	140	127	160	150	177	170	189	179
GO	1478	1430	2126	2012	2675	2487	3300	3187
LI	330	321	412	389	560	541	657	645
STIDE	1232	1204	1786	1743	1876	1828	2314	2248

Table 7.5. Executable size (in Kbytes) with aliasing, before and after optimization

7.2.3. Impact of Function Call Structure Modifications

The last set of experiments consisted of applying function signature unification and function pointer calls to obscure the program's call structure. The modifications were applied on top of the already degenerate control-flow and the aliasing structure. For the purpose of these experiments, a file-specific signature unification scheme (with the

exception of the *main* function) was applied⁹.

Table 7.6 and Table 7.7 show the code size and execution time of the programs, before and after the function call modification, with a 50% flattening of the control-flow. As shown in Table 7.6, the function call transformations did little to affect the code size for the target programs. In the case of *Compress* where function signatures were similar to begin with, the modifications made virtually no difference in terms of code size.

The modifications to the function call structure also had relatively little impact on the execution time of the programs as shown in Table 7.7. The program on which the modifications had the most impact was *LI*, a program with many small data-processing routines that were called often during execution. Since each one of those calls was now made through a function pointer, performance took a hit to accommodate the indirect addressing. *STIDE*, with its relatively fewer number of function calls, appeared largely unaffected.

These results are not surprising. To begin with, the test programs were well-suited to function signature unification—most functions came with a short parameter list (between one and four parameters) and with parameters of common types (e.g., `int`, `char`, `char*`). Furthermore, the function call modifications altered the parameter-passing behavior of the program, but it did not affect the number of functions or the frequency of function

⁹ Choosing not to unify function signatures across file boundaries is driven only by the need for a speedy implementation, not by the fundamental techniques.

calls at run-time. Lastly, the function-call modifications modified function signatures to a fixed length (i.e., a fixed number of parameters). Any parameter outside of this length is made into a global variable that can be accessed by the called function. The SPARC architecture uses a set of register windows for parameter-passing. As long as the number of function parameters did not exceed the number of registers in the window, a few extra parameters imposed a limited run-time cost.

	Compress	Go	LI	STIDE
Before	125244	1807962	350500	1449757
After	125386	2381103	490707	1622720

Table 7.6. Executable size (in Kbytes) before and after function call transformations (non-optimized, 50% branch transform)

	Compress	Go	LI	STIDE
Before	2153	1491	3178	1328
After	2325	2326	7023	1726

Table 7.7. Execution time (in seconds) before and after function call transformations (non-optimized, 50% branch transform)

As most real-world programs are composed of functions with relatively short parameter lists and common parameter types, the cost of the function call modifications can be expected to be moderate.

7.3. Performance and precision of static analysis tools

In this set of experiments, I report the results of experimenting with existing alias analysis tools. The purpose of these experiments was to investigate how well the code transformations hold up against automated analysis tools. Although a positive outcome

(i.e., one that renders analysis tools ineffective) is not in any way conclusive or complete in arguing for our case, a negative result (i.e., the tools can defeat the code transformations) would certainly be a definitive piece of evidence that the transformations have not succeeded.

Alias analysis for the purpose of code optimization has been extensively researched. However, comprehensive and user-friendly tools are scarce to come by. The notable tools developed in the research community include IBM's NPIC tool [37] and Rugter's PAF toolkit [66]. The following subsections discuss experience with both.

7.3.1. Experience with NPIC

NPIC employs a sophisticated inter-procedural alias analysis algorithm, and it represents the state-of-the-art in terms of general alias analysis. Unfortunately, IBM no longer maintains and distributes the tool. The experience with NPIC was therefore limited to small, semi-automated experiments conducted with the NPIC algorithm.

NPIC uses an algorithm that performs iterative analysis phases interleaving the inter-and intra-procedural analysis. Every time new aliasing information is generated by an intra-procedural phase, it is propagated to its successor functions which then repeat their intra-procedural analysis, and so on, until the alias set converges.

With the help of my colleagues, I conducted a limited number of experiments with the NPIC algorithm on small programs. These experiments, to the extent that a semi-automated analysis would allow, revealed that little accuracy was achieved when the

analysis terminates.

In a particular instance where index computation and aliasing were used to compute branch targets, NPIC started out in an iteration indicating that the elements of the global array could contain a number of possible values. As the iterations went on, this information was never refined to be more specific. Furthermore, alias relations identified in later iterations increased the set of possible values that the array elements were deemed to have. The algorithm eventually terminated and claimed that the elements of the global array were changed an arbitrary number of times, therefore they could contain potentially any value. Computations involving the array elements were deemed unanalyzable. This in turn implied that any basic block in the program could be the potential predecessor of a large number of blocks. Alias information propagation among those blocks therefore did not get easier and alias relations were never refined.

7.3.2. Experience with PAF

PAF implements a flow-sensitive, inter-procedural alias analysis algorithm. In the experiments we conducted with PAF, the toolkit analyzed small programs successfully but failed to handle some of the larger ones including the SPEC benchmarks.

We tested PAF on a wide range of small programs that contain either extensive looping constructs or branching statements. In all of the test cases, PAF terminates quickly reporting the largest possible number of aliases in the program (the worst possible precision). In a program with n distinct pointer assignments per block and k basic blocks, PAF reports $n*k$ alias relations. Because of the size of the test programs, negligible

differences in the pre and post-transformation analysis times were observed.

The precise reason for PAF's failure to process large programs was unclear. It is possible that the code transformations fundamentally disabled PAF's analysis strategy, and caused it to spill over its internal states and stop. It is also possible that the structure of the programs after the transformations trivially invalidated some assumptions PAF relied on for its analysis. In either cases, investigating the root of the problem would require substantial expertise into the development of the PAF tool which we do not currently have. In at least one test case, PAF failed to process a large, untransformed program, which lead us to believe that the implementation of the toolkit might be somewhat flawed.

In all the test cases that we were able to complete with PAF (with small test programs), PAF terminated rather quickly with a set of very imprecise analysis results. Investigation into the analysis results indicated that PAF failed to resolve aliases across the flattened basic blocks. The flow-sensitive analysis algorithm PAF employs essentially conducted a flow-insensitive analysis and terminated. This was a case of an efficient analysis with imprecise results.

7.4. Summary

The empirical results discussed in this chapter are admittedly quite limited. Nevertheless, they offer a glimpse into the fundamental impact of the code transformations. The essential observations derived from the experiments are summarized below:

- Performance of a program can slow down exponentially with the number of indirect branches and aliases in the program. Therefore the tradeoff between performance and the protection strength must be considered.
- Function call modifications have a relatively low impact on program performance.
- Compiler optimizations are essentially ineffective when the code transformations are applied.
- The technique of control-flow flattening and making the control-flow and data-flow co-dependent present a fundamental difficulty that existing analysis algorithms lack the sophistication to handle.

As a final note to conclude the empirical evaluation chapter, I want to stress that in the case of evaluating security mechanisms, empirical measures, although important in their own right, should be treated with healthy suspicion. Demonstrating protection strength against hacking tools (or in this case program analysis tools) only speaks for the specific instances, and in many cases the results do not generalize (as demonstrated by many years of security protocol design). Theoretical evaluations, on the other hand, are much more dependable, and should be treated with more importance (see Chapter 5).

Chapter 8

Transformations and Dynamic Analysis

This chapter explores the issue of dynamic analysis and the impact of the code transformations on dynamic analysis. Defending against dynamic analysis is not the main focus of this dissertation, and so this chapter serves only as an initial investigation into possible defense strategies and an exploration of future research directions.

In this chapter I first examine the fundamentals of dynamic analysis. In particular, I explore three different techniques: *profiling*, *tracing* and *blackbox testing*. I then investigate the ways in which the code transformations described in the previous chapters affect the nature of dynamic analysis. The major contributions of this chapter are as follows:

- I show that, with a degenerate static control-flow, optimal algorithms for placing instrumentation code for profiling and tracing can perform no better than brute-force placement methods.
- I propose the notion of state inflation and investigate a rudimentary architecture based on the state inflation to deter tracing and blackbox analysis.

- I show, by providing examples, that the one-way transformed program sports a structure that lends itself well to state-inflation techniques.

The remainder of this chapter is organized as follows: Section 8.1 provides background material on dynamic analysis. Section 8.2 shows how optimal tracing and profiling strategies are effectively disabled by the program transformations described in previous chapters. Section 8.3 examines frequency-based program profiling and potential countermeasures. Section 8.4 and Section 8.5 investigate the notion of state-inflation and its application in defending against program tracing and blackbox analysis.

8.1. The Fundamentals of Dynamic Analysis

Dynamic analysis refers to techniques designed to analyze a program during execution. These techniques include, but are not limited to, *profiling*, *tracing* and *blackbox analysis*.

Profiling and tracing are essentially two different forms of white-box analysis in which the internals of the program are open for examination. Profiling examines the frequency of events during execution (e.g., the number of times a basic block is visited) while tracing is primarily concerned with the ordering of events (e.g., whether instruction i is executed before instruction j). Blackbox analysis, on the other hand, analyzes the program's input-to-output behavior and usually does not consider the internal specifics of the code.

When dynamic analysis is performed with the purpose of program tampering or impersonation, the target of the analysis is usually to deduce knowledge of the following

subjects:

- The operational semantics of the program (e.g., which routines handle which tasks and how they interact)
- The manipulation semantics for a particular set of data quantities (e.g., how they are accessed and manipulated by the program)
- The I/O behavior of the program (e.g., how the program interacts with its environment and with the home server)

Because dynamic analysis follows an actual execution path rather than hypothesizing statically how execution might take place, information acquired from dynamic analysis is more accurate than static analysis. In an ordinary program, and with some information about the source program known a priori, dynamic analysis can be as easy as finding a known operation (e.g., the operation of retrieving a cryptographic key from memory) and the rest is history.

For a one-way transformed program, however, the rules of the game are somewhat changed; recall the discussion on diversity and the behavioral transformations in Section 4.2.1 and Section 4.2.2. The operational semantics and the I/O behavior of the program are subject to change for each compilation and subsequent installation. Therefore knowledge of the original source program and the high-level functionality may or may not help in slimming down the information the analysis must collect at run-time.

This observation suggests that dynamic analysis of the one-way transformed programs

should be performed on the full-scale of the program; short cuts such as looking for known behavior patterns or operations are presumed ineffective. Discussions hereafter in this chapter is based on this premise.

Dynamic analysis can be attempted either online, on a legitimate execution, or offline, as in the case of a simulated execution. As a general rule of thumb, more intrusive and hence more aggressive forms of analysis (such as specially instrumenting the program for the purpose of profiling or tracing) are possible only with an analysis conducted offline.

8.2. The Efficiency of Profiling and Tracing

Program profiling and tracing can help discern the run-time behavior of a program. Profiling typically counts occurrences of an event (or events) during a program's execution while tracing is concerned with the order of dynamic events [5][6][8][47].

Profiling and tracing are often accomplished by instrumenting the target program to include profiling and tracing instructions. For profiling, the instrumentation code increments a counter that records how many times an event (e.g., the execution of an edge or a block) happens during execution. In tracing, the code writes a unique token to a trace file whenever it is encountered.

Instrumentation has an overhead in terms of both execution time and code size. Therefore for performance purposes, it is desirable to deploy a minimum amount of instrumentation, at suitable locations in the program to minimize the overhead.

Much research has been performed on finding optimal solutions for placing profiling and tracing instrumentation [6][47][51][72]. The result is a set of spanning-tree based algorithms for instrumentation placement. These algorithms compute a spanning tree of the program's control-flow graph and place the instrumentation code on the graph edges that are not in the tree. Ball and Larus [6], Ramamoorthy *et. al.*[72] and Knuth [47] all showed that instrumenting the program using the spanning tree method results in the most efficient profiling or tracing mechanism (i.e., the least overhead) for the underlying control-flow graph. In the following paragraphs, I show that the code transformations described in previous chapters prohibit the use of these optimal instrumentation strategies.

The spanning-tree based methods assume a statically determinable control-flow based on which to compute the optimal instrumentation strategy. These schemes become expensive when control-flow cannot be determined statically. For example, consider the flowgraph in Figure 8.1. The graph on the left is the original flowgraph while the one on the right is the degenerate flowgraph after flattening (no artificial basic blocks inserted). For the original flowgraph, the most efficient set of edge instrumentation determined by the spanning tree method can be found on edge $a \rightarrow c$, $b \rightarrow d$, $d \rightarrow e$, $f \rightarrow g$ and $g \rightarrow h$ (the instrumented edges are marked with black dots). If profiling, the counts for uninstrumented edges can be deduced from the measurements obtained from the instrumented edges. For example, the execution count for edge $a \rightarrow f$ can be computed as $(f \rightarrow g - (a \rightarrow c + b \rightarrow d))$. When tracing, execution paths can be derived from the instrumented traces. For example, the execution (a, c, d, f, g, i) can be regenerated from

the trace $\{a \rightarrow c, f \rightarrow g\}$ and the execution (a, f, g, i) is represented by the trace $\{f \rightarrow g\}$. In general, Kirchoff's flow law (e.g., $a \rightarrow f = f \rightarrow g - (a \rightarrow c + b \rightarrow d)$) can be used to deduce the counts for unmeasured edges and therefore minimize the amount of instrumentation that is needed.

In the flattened flowgraph shown in Figure 8.1(b), there is no flow information between basic blocks. The instrumentation therefore must be placed on all edges between the ENTRY node and the basic blocks. For example, for execution (a, c, d, f, g, i) , the program would have generated the trace $\{Entry \rightarrow a, Entry \rightarrow c, Entry \rightarrow d, Entry \rightarrow f, Entry \rightarrow g, Entry \rightarrow i\}$.

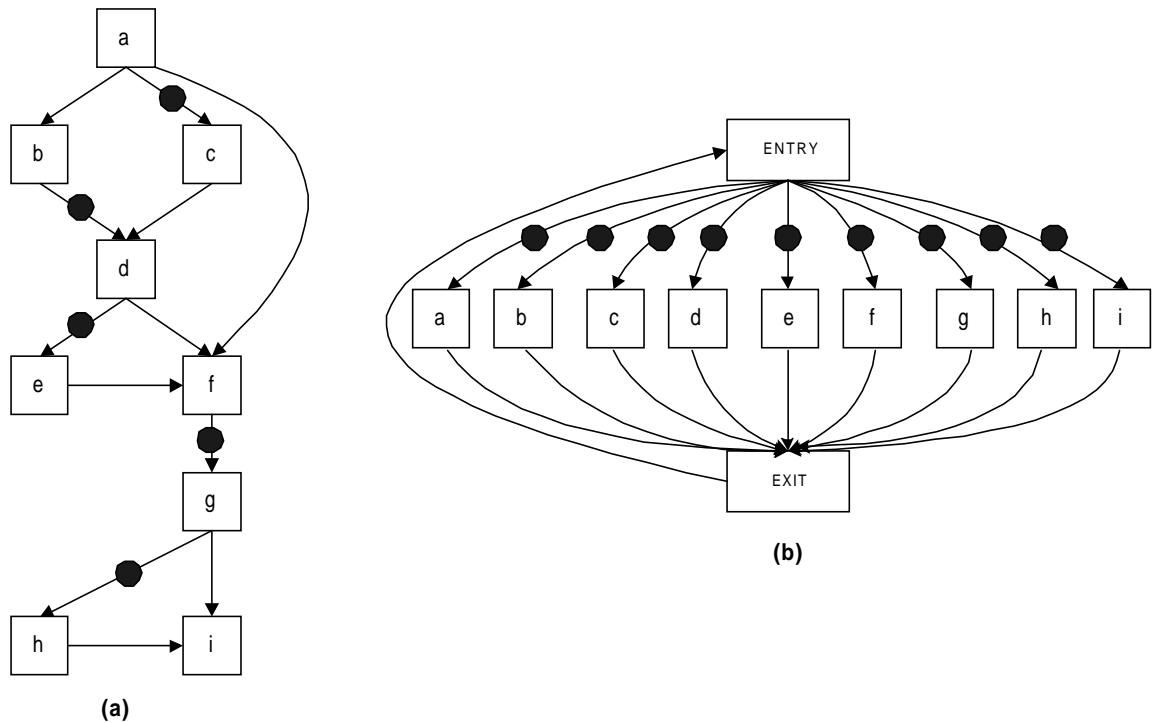


Figure 8.1. A regular control-flow graph in (a) and the flattened version in (b)

Instrumentation can also be placed strictly with blocks instead of edges. In such a scheme, a counter (or a trace value) is associated with a basic block, and it is incremented (or written out to a trace file) each time the block is executed. With a structured flowgraph such as the one in Figure 8.1(a), it is not necessary to instrument every basic block. For example, a counter associated with basic block g would also indicate the number of times that block f has been executed. Hence there is no need to use a separate counter with f . The same is not true when the flowgraph is flattened—the only way to measure the occurrences of any basic block is to attach a specific counter to it.

An interesting twist to the profiling and tracing problem is that, for structured flowgraphs, an optimal edge instrumentation never has a higher cost than an optimal block instrumentation [6]. Consider the example in Figure 8.2 where the figure on the left is the original flowgraph with each edge marked with its execution frequency. The optimal block instrumentation for tracing is to instrument blocks a and b . This instrumentation will result in a cost of 40 since block a will execute 30 times and b 10

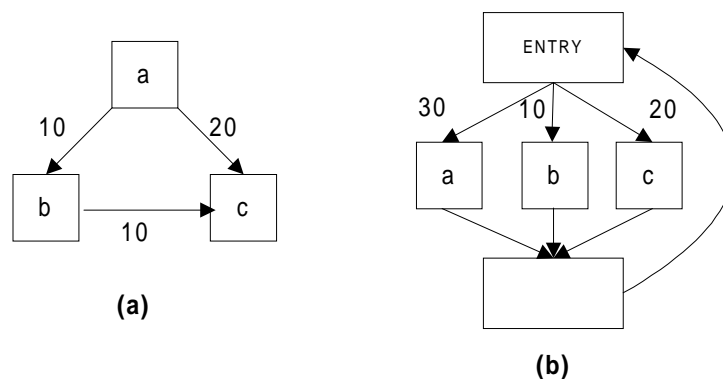


Figure 8.2. An example illustrating edge vs. block instrumentation

times. Note the cost can be reduced if edge instrumentation is used instead. The optimal solution for edge instrumentation would call for only edge $a \rightarrow b$ (or $b \rightarrow c$) to be instrumented, resulting in a cost of 10 ($a \rightarrow b$ executes 10 times).

When the program's static control-flow is degenerate, edge instrumentation no longer has any advantage over block instrumentation. Figure 8.2(b) shows the flattened version of the flowgraph in Figure 8.2 (a). For this flowgraph, edge instrumentation needs to occur on every edge from the entry node to a , b and c . This instrumentation results in a cost of $30 + 20 + 10 = 60$, exactly the same as in block instrumentation which would yield a cost of 60 as well (30 for a , 10 for b , and 20 for c).

Knuth [47] showed that a set of edge instrumentation E_{inst} solves the profiling and tracing problem if and only if $(E - E_{inst})$ contains no cycle, where E is the set of edges in the control-flow graph. Ball and Larus took this result a step further and showed that the optimal solution to the edge placement problem is one such that the set of uninstrumented edges— $(E - E_{inst})$ —constitutes a maximum spanning tree of the graph. Thus, the minimum number of edges that need to be instrumented for profiling and tracing is $|E| - (|V| - 1)$, where $|E|$ is the number of edges and $|V|$ is the number of blocks. In a flattened graph, it is straightforward that a safe and sufficient solution to profiling and tracing would call for $|V|$ edges to be instrumented (one for each block).

A control-flow graph of a normal program usually has almost twice as many edges as basic blocks, thus the difference in the number of instrumentation instructions between the brute-force instrumentation method (recording every block) and the optimal

placement is not significant. However, the real importance lies in the cost in terms of execution time caused by instrumentation. Ball and Larus conducted several experiments in their 1994 paper to investigate optimal profiling and tracing mechanisms [6]. They measured execution times for the uninstrumented program, the naively instrumented program (instrumenting every edge or block) and one that is optimally instrumented using the spanning tree algorithm. Their experimental results showed that, on average, naive instrumentation produced an overhead of 10-225 percent over the original program, while the overhead of optimal instrumentation lay somewhere between 5-91 percent. In other words, for some programs, naive instrumentation can be more than twice as expensive as optimal placement.

Ball and Larus observed in their experiments that programs with large basic blocks and fewer conditional branches tend to lend themselves well to profiling and tracing (i.e., less instrumentation overhead). This is because in those programs large blocks dominated the execution of the program and the cost of instrumentation was better amortized. In their experiments, the programs for which naive instrumentation produced an overhead of more than 150% over the original programs all came with small block sizes and a large number of conditional branches. This observation is useful since it points out that profiling and tracing can be made more expensive by increasing the number of conditional branches and reducing the size of basic blocks. Note how this fits nicely with the control-flow degeneration technique—basic blocks can be arbitrarily broken up by adding branches and every branching statement is essentially conditional because the switch target is set dynamically.

In summary, a degenerate static control-flow prohibits the static determination of optimal tracing and profiling strategies. Consequently, both profiling and tracing must be performed in the most rudimentary and costly fashion—every edge (or block) must be instrumented in order to capture enough information for effective profiling or tracing.

8.3. Execution Profiling

This section investigates execution profiling and potential counter measures. Conventional profiling typically considers the frequency of events only, not the order of events. Thus information gathered from profiling alone is too coarse to be directly useful in intelligent tampering and impersonation attacks. However, profiling can help to identify program portions that are heavily executed (e.g., hot paths), and such information can be used in deciphering the transformed program by comparing its behavior with the

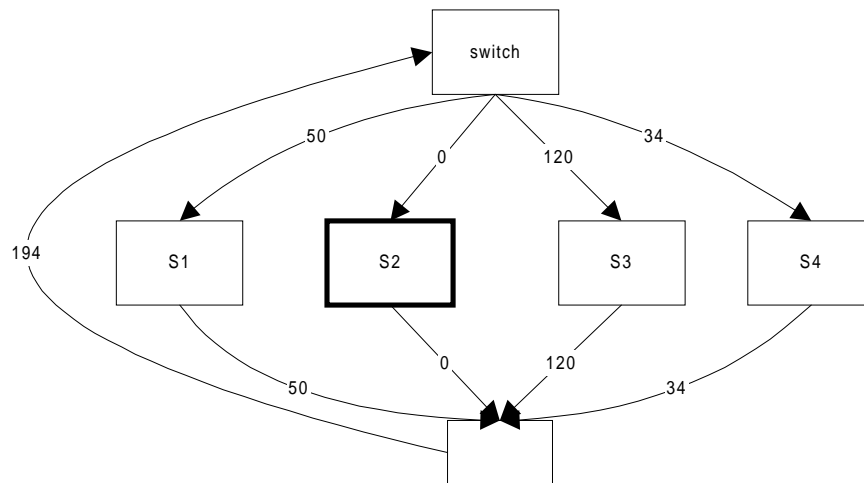


Figure 8.3. Example illustrating edge profiling to identify dead code

original program. Hot path information can also be used to improve the quality of data-flow analysis as illustrated by Ball and Larus [7].

A direct application of program profiling is to identify and eliminate dead code. Consider, for example, a flattened control-flow graph in Figure 8.3. Each edge in this graph is labeled with its frequency from profiling. It is easily seen from the profiled data that code block S2 is likely to be dead code since its frequency count is zero. Such information can be used to remove dead code and possibly determine the actual control-flow of the program. Another potential application of profiling is to identify heavily executed basic blocks (such as loop bodies). Even in a degenerate form, loop bodies can be spotted easily with profiling statistics.

Assuming the goal of program profiling is to help determine the program control-flow using execution counts, one way to defeat such frequency-based profiling is to introduce

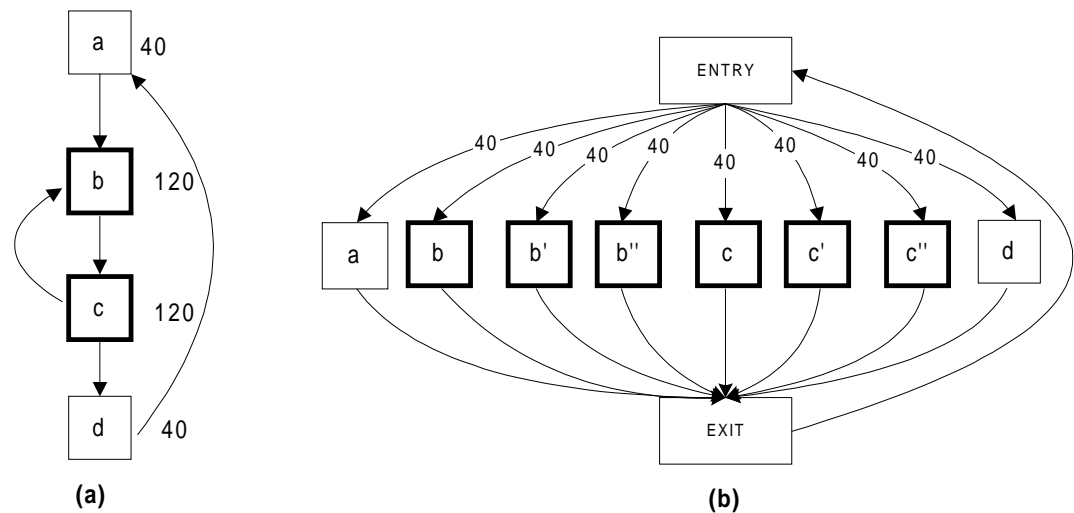


Figure 8.4. Example illustrating loop unrolling to deter execution profiling

additional blocks that actually execute. For example, in Figure 8.4, the original flowgraph in (a) is degenerated to the form in (b) and the loop body *b* and *c* in (a) are unrolled into three different basic blocks each in (b) (in bold lines). Each of these loop-body blocks is a slightly obfuscated version of the original block (in order to defeat simple pattern matching). The profiled data in (a) clearly identifies block *b* and *c* as the *hot block*—one that is heavily executed. As a contrast, the frequency counts in (b) do not convey nearly as straightforward a story.

Another strategy to thwart frequency-based profiling is changing the original execution frequency of existing blocks to a form that does not lend itself well to the identification of hot blocks or hot paths. For example, consider again the flowgraph in Figure 8.4(b). Instead of having block *b* and *c* replicated in three functionally-equivalent blocks, the program can have block *a* short-circuit itself and execute 120 times (for example) before letting the control transfer to other blocks. Such a scheme is easy to implement since the control transfer is set dynamically by each block. This of course depends on whether the computation in *a* can be repeated without affecting the outcome of the program (e.g., idempotent operations).

8.4. Program Tracing

Unlike profiling, program tracing records the sequence of executed basic blocks, and therefore is capable of conveying finer-grain information than profiling. Program tracing can also be accomplished using program instrumentation [6][51].

The primary challenge of program tracing is to determine what information to record and

when enough information has been recorded for the purpose of tracing [6]. The first task of tracing is the unambiguous identification of the paths traversed. With that, data-reference tracings on particular data quantities can be performed.

Clearly, the more instrumentation there is in the program, the more information the tracing mechanism will be able to collect. For a flattened control-flow such as the one shown in Figure 8.1(b), it is straightforward to show that instrumenting every edge from the entry node (the switch node) to all of its successors (every block in the procedure) meets the requirement of unique identification of execution paths.

The aliasing mechanism developed in the earlier chapters to obscure static data manipulation is not sufficient to obstruct dynamic data-reference tracing. A data-reference trace records the addresses and values of data accessed, and thus is able to reveal the true modification, reservation and usage of data quantities at run-time.

In the remainder of this section, I discuss a technique called *state-inflation* as a potential countermeasure to deter dynamic code tracing, and I provide a preliminary investigation into how state-inflation might be achieved within the context of the one-way code transformations.

8.4.1 State Inflation

The idea of state-inflation is straightforward. Consider for example, the program segment depicted in Figure 8.5(a). The program execution from point A to point B follows the path {S1, S2, S3}, and the computation on this particular path is such that $\{b \leftarrow x, a \leftarrow b\}$.

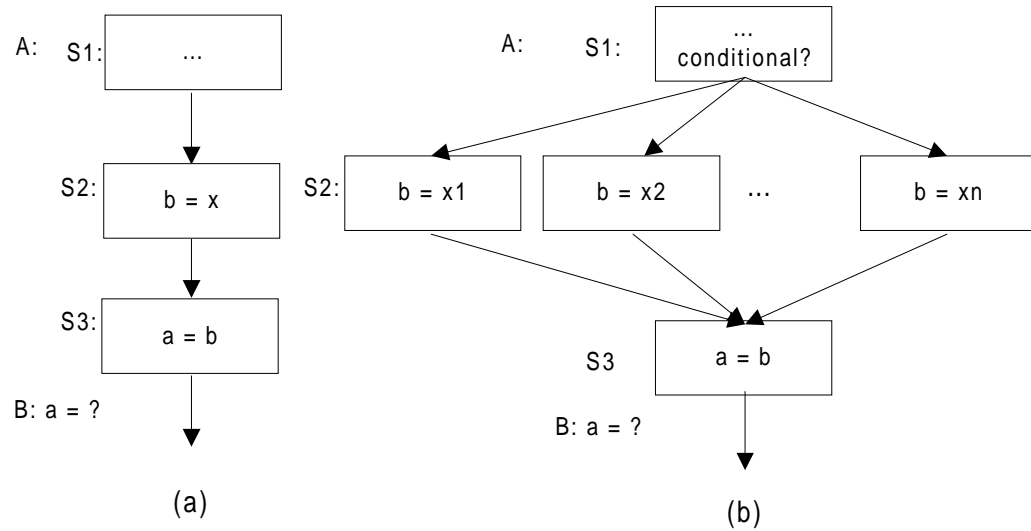


Figure 8.5. Example illustrating state inflation

Therefore at program point B, the value contained in variable *a* is *x*. Now consider the program segment in Figure 8.5(b). Here a conditional statement is added to S1 and a series of blocks, each assigning a different value to *b*, take S2's place. Which block gets executed depends on the evaluation of the conditional in S1.

The purpose of this transformation is to replace the state: $b \leftarrow x$ with a set of equivalent states: $\{b \leftarrow x1, b \leftarrow x2, \dots b \leftarrow xn\}$. Each time S1 is executed, it selects one of the blocks to execute randomly. These states are only equivalent from the perspective of the transformer (subsequent computations using any of the values of *b* are considered equivalent). From an analyzer's stand point, these different blocks represent different states as they reflect the outcome of the conditional evaluation. Note that in the transformed code, there are *n* different paths between point A and B as opposed to the single one in the original program.

Using program tracing to deduce the operational semantics of code transformed this way is challenging, for each time the segment is executed potentially a different path is followed. This results in an explosion of states that the tracing analysis must keep.

Note how this state-inflation transform can be realized easily with the flattened control-flow: the conditional in S1 can be implemented as setting the branch target to one of the n values, determining which block to execute and hence which value the variable a will have at program point B.

State-inflation can be extended to the entire program by identifying a series of execution steps and applying the inflation technique to each step. For an execution between program point $p1$ and $p2$ which consists of n sequential steps, and assuming each step is inflated to m different incarnations of the original step, there are then m^n different paths from $p1$ to $p2$.

Furthermore, the idea of replicating blocks can be extended to functions such that the destination of a call site can be one of n functionally equivalent implementations of the original function.

8.5. Blackbox Analysis and State Inflation

The final dynamic analysis technique I want to examine is blackbox testing. Unlike profiling or tracing, blackbox analysis does not consider the internal specifics of the program. Instead, it analyzes the program's input-to-output relations in an attempt to emulate the program's run-time observable behavior.

It is clear that if the program has a relatively simple state space (i.e., the input-to-output mapping is easily deducible), a blackbox analysis can lead to successful impersonations of the legitimate program.

Once more, the notion of state inflation can be useful to impede blackbox analysis. The techniques discussed in the previous section can be adopted to affect the input-output behavior of the program. It is not difficult to see how a one-on-one mapping between the input and the output might be inflated to a one-to-arbitrarily-many mapping using the inflation techniques, in which case a simple blackbox analysis will not be sufficient to deduce the program state space and thus emulate its behavior.

8.6. Summary

In this chapter, I offer some preliminary insights into the effect of the One-way Transformations on performing dynamic analysis of programs. The specifics of program profiling, tracing and blackbox analysis are discussed. The investigation here shows that the code transformations deter the application of some of the optimal dynamic analysis strategies. I also propose the concept of state inflation as a potentially fruitful strategy to deter dynamic analysis, and I argue that the flat structure of the transformed program fosters the easy incorporation of state inflation techniques. However, to adopt the principle of state inflation and to assess its implications more accurately, a more comprehensive and in-depth investigation is required.

Chapter 9

Revisit the Big Picture

From Chapter 4 to Chapter 8, I have taken you through detailed descriptions focusing on various individual techniques. Now is the time to tie these pieces together and revisit the big picture. In this chapter, you will see how the techniques fit together to form a system-level solution to protect network management systems, and in particular to defend against the malicious host problem identified in the introductory chapters.

9.1. Recapping the Problem: extending the trust boundary of the network survivability architecture

The original motivation for this work arises from the need to protect a survivability architecture for critical infrastructure systems. The architecture, recapped in a high-level overview in Figure 9.1, provides means to ensure the survivable operations of the underlying application system. Details of this architecture and the infrastructure system it manages are discussed in Chapter 2. It is worth reviewing a number of essential characteristics of this system, as they inspired the work that became the topics of discussion for much of this dissertation.

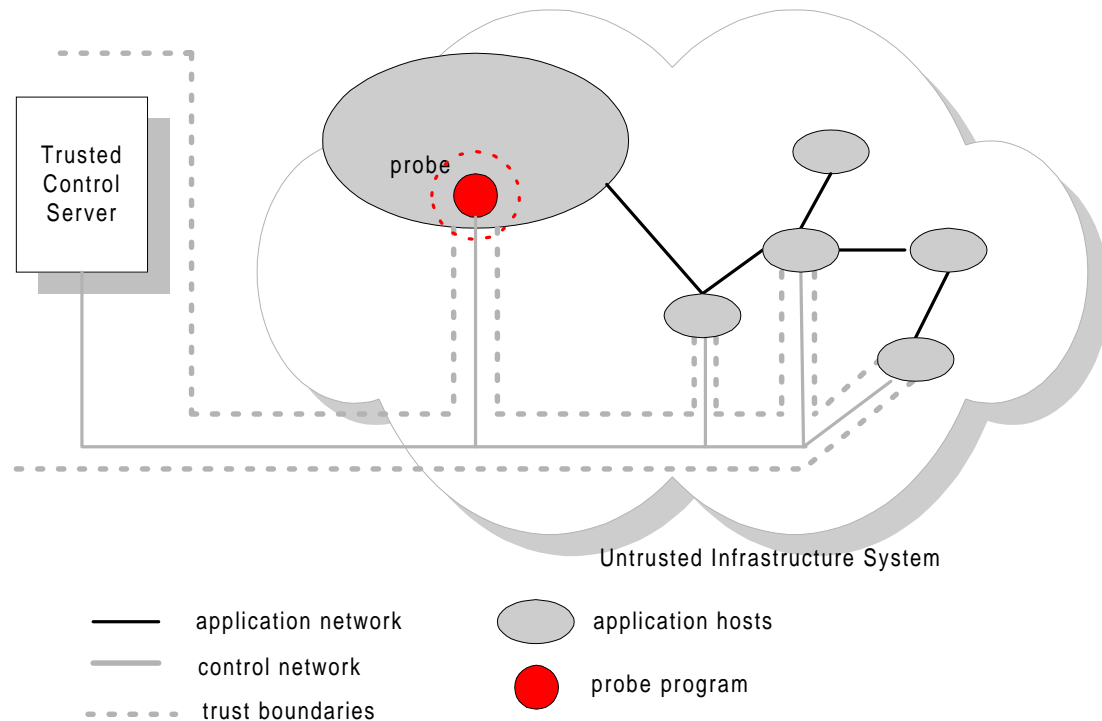


Figure 9.1. A review of the survivability architecture

The survivability mechanism is vital to the continuing operation of the application system. Therefore, assurance must be provided that the mechanism itself is protected from corruption and malicious attack. Investigation into the trustworthiness of the survivability architecture yielded the trust boundary depicted in Figure 9.1.

In this figure, the control server is trusted, and so is the communications network between the control server and the application hosts (assumed secure with cryptographic means). The trust boundary that includes the control server and the network reaches only to the perimeter of the application hosts (shown in Figure 9.1). The application hosts cannot be presumed hardened with the same measures as the control server, and thus are vulnerable to security attacks. This implies that the probe programs, an integral part of the survivability architecture, execute outside the trust boundary in an untrustworthy

environment.

To ensure the secure operation of the survivability mechanism, the probe programs must be included inside the trust boundary. The problem therefore becomes how to extend the trust boundary (as shown by the red dashed circle in Figure 9.1) and ensure the secure execution of trusted software on untrustworthy platforms.

9.2. A System-level Solution for A System Problem

The general problem of software security in malicious environments is characterized as the *malicious host* problem in Section 1.1. Stated in the usual computer security parlance (*Alice* represent the home server and *Bob* is the remote host), the malicious host problem consists of three facets¹⁰:

- **Algorithm secrecy:** The algorithm privacy problem is stated such that Alice wants to execute program P on Bob, and that she does not want to disclose to Bob the algorithm that is implemented by P .
- **Execution integrity:** The execution integrity problem is such that Alice wants to execute P on Bob, and she wants to be assured that, given a particular input x , P is correctly executed,
- **Input spoofing:** The input spoofing problem states that Alice wants Bob to compute

¹⁰ The denial-of-service problem is omitted here for reasons stated in Section 2.4.1.

P on some data x that Bob has. Alice wants to ascertain that Bob has not lied about the input x .

The discussion in Chapter 2 argues that there is no solution to the input-spoofing problem if input spoofing does not require learning the computational algorithm or compromising the execution first. In other words, if the effect of computing P on a corrupted x will not give him away, Bob can make up any x that he wishes. When input spoofing does require analysis of the computation itself, the input-spoofing problem is subsumed by the algorithm privacy and execution integrity problems.

The malicious host problem also arises in several other application contexts [25][34]. However, it has thus far been viewed in an isolated context: namely, protection of one program against its hosting machine. The majority of traditional code obfuscation work is a manifestation of this school of thought.

In this work, the malicious-host problem is viewed as a system issue. The approach hereafter derived represents a system-level solution. Consider again the target application as a set of probe programs in the survivability architecture. These programs are distributed across the network (i.e., executing on different network hosts) and they interact together to achieve a collective functional goal. To compromise the collective system functionality, an intruder must corrupt (or compromise) a designated number of software components (noted as m out of total n programs). To counteract such an attack, three inter-related mechanisms are derived:

- **One-way Translation:** The One-way Translation mechanism described in Chapter 4

provides a means to generate diverse and difficult-to-analyze software versions. Analysis of the resulting program versions requires a certain amount of time as indicated by the complexity metrics in Chapter 5.

- **Temporal Diversity:** The probe programs are refreshed periodically. Analysis of a probe program must be completed within its lifetime or the information acquired during analysis will be useless.
- **Spatial Diversity:** Diverse versions of the probe programs are deployed across the network. Compromising the network-wide probing mechanism requires independent effort spent on attacking each probe program.

The One-way Translation mechanism facilitates the diversity mechanisms by generating functionally equivalent but otherwise diverse program versions that are deployed across the network (spatial diversity) and chronologically along the time line (temporal diversity).

The above mechanisms have a network-wide effect: Assuming the interval at which a program is refreshed (with a different version) is T , one can derive a probability p based on the complexity metrics such that p represents the probability that an analysis of the program can be completed within T . Further assuming that compromising the network-wide functionality requires the compromise of m (out of n total) software components (e.g, the probe programs), it is then straightforward that the probability of a successful attack is $P = p_1 * p_2 * \dots * p_m$, where p_i is the probability of completing the analysis of program i within T . Note that P is smaller than any of the individual p_i 's

If identical copies of the program are deployed across the network, the overall probability of a successful attack is simply p_i , since $\forall i, j$ such that $i \neq j$, p_{ij} , the conditional probability of a successful attack on program i given a successful attack on program j is assumed to be 1.

With this network-wide perspective, the collective, system-level security requirement can be decomposed into very specific complexity goals for each program. Subsequently, code transformations can be tailored and applied to meet those goals. When the goals are met, the trust boundaries are extended, in a time-limited fashion, to the trusted software that is executing on untrustworthy hosts.

9.3. The Other Pieces in the Puzzle

In addition to the core mechanisms, the solution framework relies on a certain number of assumptions. This section examines some of the assumptions and explores their implications.

Trusted program deployment: The solution mechanism calls for the trusted server to update the remote program on-the-fly with a different version of the software. This requires the trusted dynamic deployment of programs. Trusted deployment means two things:

- The program is properly instantiated (up and running as intended)
- The program is instantiated at the intended host.

Knowing whether the program is instantiated is perhaps easier than knowing where exactly it is instantiated. Consider that the communications between the remote program and the home server follows a certain timing mechanism. After the program is dispatched, the home server expects it to follow through with a particular hand-shake sequence that indicates to the server that the program has been instantiated (or at least is executing to the completion of the hand-shake protocol).

Knowing where the program is executing is a much harder problem. Given the possibility that the program can be hijacked to execute on a different host, what guarantee can one have if this update-on-the-fly method is employed?

Several ways to alleviate this problem exist with varying degrees of assurance and cost. One possibility is using hardware-based identification schemes such that a hardware device on the intended host performs mutual authentication with the arriving program. This scheme is costly and the issue of possibly spoofing the hardware is still of concern.

Another approach lies in having the program perform some kind of active checking to verify properties of its execution environment: for example, if the program could be compiled with knowledge of the invariants in the intended execution environment (e.g., memory layout of the operating system kernel) or in the monitored applications. A checking mechanism similar to the code depicted in Figure 9.2 can be used.

In the program segment in Figure 9.2, a series of checksum operations are performed. Each checksum operation reads a set of memory locations from the host followed by a *compare-if-equal* operation. Some of these checksums are computed on known invariant

locations and others are simply spurious computations on random memory locations. It is readily seen that the checksum operations compose a bit-pattern calculation (e.g., status1, status3, status6, status n-1) that can potentially be used to indicate the execution paths followed by the remote program.

Furthermore, a fail-and-stop mechanism can be implemented to stop the program execution if the checks do not return satisfactory results. For example, the program can overwrite the black elements of the global data array (see the discussion in Section 4.3.3) and disable further execution (since branch targets are computed using the black elements of the array).

For the program in Figure 9.2, a naive attack that simply returns *yes* (or *no*) for the *compare-if-equal* operation will not work. To circumvent the checking mechanism, the intruder must understand the invariant behavior of the host system or duplicate the entire execution environment, is a substantial undertaking.

Of course, this type of checking requires the ability to identify host-unique invariants,

```

S1:    v1 = checksum(memI1, memI2, ... memIn);
S2:    if (v1 == value1)
S3:        send status1
S4:    else
S5:        send status2
S6:    ...
S7:    v2 = checksum(memJ1, memJ2, ...memJm);
S8:    if (v2 == value2)
S9:        send status3
S10:   else
S11:        send status4

```

Figure 9.2. A pseudo-code segment illustrating possible verifications of the execution environment

thus only specific and very limited forms of checks can be performed. Note that if it were indeed possible for programs to actively verify their execution environments, one form of dynamic analysis—namely analysis of an offline execution on a separate host—could be severely discouraged.

Input spoofing: The discussion thus far has argued that solutions for the input spoofing problem only exist when input spoofing requires learning of the algorithm or tampering with the program execution. However, the discussion did not identify under what conditions or characteristics of applications that input spoofing is inherently associated with (or can be made to associate with) the issue of algorithm privacy or execution integrity.

There are clearly cases in which input spoofing requires very little analysis (if any). In those scenarios, both the semantics of the input data and the manners in which it is collected by the program are easily identifiable (e.g., consider the well-known case of the program checking to see if its installation CD ROM is currently in the drive). There are also cases where collection of the input is performed in a manner that is both hard to identify and difficult to spoof (e.g., consider a network packet sniffer as the data collector). The challenge of course lies in identifying whether the former applications can be modified to resemble the latter kind in terms of data collection. A study on the general data-collection characteristics will be particularly beneficial in that regard, and it will also help to determine the class of applications for which the current work is applicable with respect to the input spoofing problem.

Dynamic Analysis: Chapter 8 considers issues associated with dynamic program

analysis. In particular, the application of state inflation as a potential defense strategy was investigated. The basic principle of state inflation is to decrease the amount of information contained in each state transition in order to impede dynamic analysis. Another approach based on the same principle is the use of multi-threaded computational models. Multi-threaded programs are inherently more difficult to analyze. The incorporation of multiple threads presents a significant challenge to tampering and impersonation attacks based on program analysis.

For example, when a program consists of multiple threads executing in parallel, what an intruder can learn from an offline, simulated execution is fairly limited since the offline execution is different from the legitimate execution. Furthermore, observing one legitimate execution reveals little about the next n executions due to the non-deterministic nature of multi-threaded programs.

A potential drawback of the multi-threaded model is that although the incorporation of concurrent threads introduces the element of non-determinism, each combination of the thread executions must be valid. The intruder thus needs to compromise only one execution to fool the trusted server. For this method to work, there must be a way for the trusted server to distinguish the current, truly legitimate execution from others. It is not clear how this could be implemented.

Furthermore, a design to support multiple threads must include a utility to translate sequential programs into concurrent threads. This is not a trivial undertaking if the technique is aimed at general applications. Some applications lend themselves well to implementations in concurrent tasks while others are inherently sequential. Transforming

a sequential implementation into concurrent threads requires the identification of tasks that can be executed in parallel—a job that is beyond the capabilities of the current code translator.

9.4. Summary

This chapter revisits the problem context that originally motivated this research. The techniques discussed so far were examined in this context. The key point of this chapter is that the various techniques introduced in this dissertation are not ad hoc methods, but rather specific pieces to the solution of a specific system-level problem. The One-way Translation mechanism and the temporal and spatial diversity techniques were developed for the collective goal of protecting the network control infrastructure. I examine the overall network effect of the One-way Translation and the diversity schemes. I point out the pieces that are not covered currently by this investigation, and identify future research directions.

Chapter 10

Related Work

In this chapter I examine related work in the general area of software protection. Whenever possible, comparisons between my work and the existing approaches are offered.

10.1. Code Obfuscation Work

The first category of related work is in the area of copyrights and intellectual rights management. Managing the rights of intellectual properties sometimes calls upon the service of *code obfuscation* techniques to obscure the algorithm or the implementation details of software components. Code obfuscation work to date [19][20][82] has been focused on this class of applications to prevent reverse engineering.

Collburg *et. el.* [19][20] conducted several studies on specific forms of code obfuscation techniques. They classified obfuscation transformations into four general types: layout, control, data, and preventive transformations. A key contribution of Collburg's work is the concept of *opaque predicates*. Opaque predicates are program predicates, which have

some property known a priori to the obfuscator, but difficult for an analyzer to deduce.

The fundamental problem with Collburg's work and with other code obfuscation work [23] is that they are not supported by any quantifiable metrics. In his 1997 paper, Collburg identified four main criteria that can be used to measure the quality of an obfuscation transformation: *potency*, *resiliency*, *stealth*, and *cost*. These criteria are developed based on software complexity metrics. It is widely known that software complexity metrics do not reflect the real program complexity. For example, metrics used by Collburg include the Munson metrics for software [60]. Munson metrics consider that, for example, a multi-dimensional array is more complex than an array of only one dimension. This is perhaps true in some cases, but it does not hold in cases where a multi-dimensional array is a more natural choice for data representation. The main reason that the use of software complexity metrics has not been satisfactory is that these metrics are by nature quality metrics, and as such they do not imply quantifiable results. Users are still left to wonder about the assurance offered by the various schemes and how many of the transformations one needs to apply to obtain a certain level of security.

Other code obfuscation work has been undertaken primarily with the goal of obfuscating Java bytecode since bytecode programs carry more source information than native binary code[23]. The focus of the code obfuscation work has been to obscure information (i.e., algorithm privacy), rather than prevention of tampering.

My work differs from the code obfuscation techniques in three significant aspects:

- In my work, a theoretical framework is offered with the code transformations. The

complexity metrics in Chapter 5 serve as a concrete means to measure the effect on complexity for each transformation. Users can reason about the strength of the transformations using these complexity metrics.

- The fundamental techniques afforded by my work render the entire program data-dependent (no static control-flow and function calls). Tampering of such a software component requires analysis of the program behavior. In that regard, my work deals with prevention of tampering as well as protection of the algorithm privacy.
- Despite the aforementioned studies on various code obfuscation schemes, to the best of my knowledge, there does not exist a comprehensive implementation of the obfuscation transformations in the public domain. The One-way Translation compiler is the first working code translator with publicized transformation techniques with which empirical studies of the techniques can be performed.

There are also a few commercial companies developing products in the general area of intellectual rights management and software protection [14][43]. Cloakware's JAVA obfuscator is the only other known implementation for which some preliminary performance data has been published [15]. Their results showed an average increase of execution time by more than ten-fold beyond the original execution time. My performance results are significantly better. Commercial companies rarely publish anything of any significance in the public domain. A sound comparison between my approach and the commercially available products therefore cannot be readily obtained.

10.2. Security of Mobile Agents

Another class of related applications exists in the area of mobile computing and the protection of mobile agents against potentially hostile hosts. Several research efforts are noteworthy in this area. The following subsections examine them in turn.

10.2.1. Mobile Cryptography

Sanders *et. al.* proposed the notion of executing programs in an encrypted form [74]. This approach, *called mobile cryptography*, is able to perform limited forms of computation with encrypted functions using composition.

Sander's approach is based on homomorphic encryption schemes. In a nutshell, the mobile cryptography technique can be summarized as follows,

- Homomorphic encryption schemes exist for the evaluation of polynomials. Sanders *et al.* identified an additively homomorphic encryption scheme such that evaluation of a computation can be derived from a homomorphic computation with encrypted functions.
- Sanders also identified that function composition can be used to prevent the malicious host from discovering the secret key of the agent, since the decomposition of multivariate rational functions is a hard problem in general.

The mobile cryptography work has the flavor of achieving true blackbox security with theoretically provable strength. However, their mechanism as it stands now is only applicable to polynomial and rational functions. It remains to be seen whether similar

techniques can be extended to general software.

10.2.2. Time-limited Blackbox Security

Independently of the temporal diversity idea in this work, Hohl proposed the idea of time-limited blackbox security [40]. Time-limited Blackbox Security recognizes the fact that there is no absolute protection—given enough time and resources, any protection mechanism (including encryption) can be broken. Software protection, therefore, should be based on a more restricted form of blackbox security such that the blackbox execution is guaranteed only for a certain time period.

Hohl’s approach is basically obfuscation with a timestamp: programs are first “messed up” (i.e., obfuscated) such that the executing host cannot “read”, “understand”, or “modify” the program’s code and data for a certain known time interval. After this interval, attacks are possible, but they would have no effect since the agent program would have been expired.

Hohl offered little suggestion as to how this blackbox security (even for a limited time) can be achieved, let alone evaluated. His illustrative “messaging-up” algorithms are simple obfuscation techniques that, again, are not supported by quantifiable measures.

10.2.3. Server Replication

Yet another approach to mobile-agent security is the use of *reference states* [41]. This approach proposes that the use of a trusted, reference host to check the execution results on an untrusted host. The approach is straightforward, assuming the attack does result in

differences in the agent's states. However, such an approach is not always practical—it relies on the existence of a completely duplicated execution environment.

Other work based on server replication includes the use of fault-tolerant computing. Minsky *et al.* proposed an approach in which the execution of a software component is duplicated on a set of independent, identical servers [58]. Every execution step is performed in parallel by all hosts, and results are reached by a majority voting mechanism among the hosts. Results from one step is fed as input to the next step which starts another round of parallel execution and majority voting. Server replication and majority voting are applicable only for specific applications (i.e., critical computations for which tolerance of random faults is the main objective).

10.3. Tamper Resistant Software

Aucsmith's work on the Integrity Verification Kernel (IVK) [4] at Intel is of direct interest, because it is a similar attempt to solve the general software protection problem. Aucsmith also coined the phrase of *Tamper Resistant Software*. An IVK is a critical code section consisting of multiple cells (code segments). At run-time, all the cells are encrypted except the one that is currently executing. The executed cell becomes encrypted again after execution while the next cell decrypts.

Run-time encryption and decryption requires significant performance overhead. Although the author did not provide detailed performance data, this method is recommended only for protection of small portions of code segments.

The encryption and decryption processes of IVK are carried out on the host that is not trusted. Although the key used for encryption and decryption is exposed only for a short time, it is possible that an adversary could obtain the key and subsequently compromise the entire execution. Furthermore, building the IVK at the first place requires considerable user intervention in the identification and isolation of the critical code segments that may need to be specially armored.

10.4. Other Related Work

In addition to general software protection, a number of other related work has implications in software security.

Devanbu and Stubblebine proposed a mechanism to protect the integrity of data structures (e.g., stacks and queues) on hostile platforms [22]. Their method uses a digital signature chain, starting from an initial signature that is stored on a trusted device, to verify the integrity of the data and data operations. Each data item is stored alongside a signature that is computed by the trusted device. A signature checking operation by the trusted device is required for each retrieval of a data item. This work is designed primarily for allowing a trusted device to store large amounts of data at a remote host and have assurance that the data will not be tampered with.

The Immunix project at Oregon Graduate Institute (now with Wirex) includes an effort to build survivable operating systems through diversity specialization [69]. The focus of Immunix is to thwart class attacks due to replicated software flaws. The claim is that the specialization method allows the system to guard the validity of the operating-system

software, both statically and dynamically. It is unclear, and the designers of Immunix provided little hint, how diversity is enforced and whether diversity techniques can be effective when applied to complex software such as operating systems.

10.5. Summary

Although much of the One-way Translation work is in the general spirit of code obfuscation, my work arrived at the problem with a different motivation and generated a solution that is both more general and more quantifiable than the previous approaches. My work also produced the first working prototype of a software-hardening compiler, which translates ordinary programs to a difficult-to-analyze version, and the difficulties are supported by concrete complexity metrics.

Chapter 11

Contributions and Conclusion

In this dissertation I have presented an approach to the problem of software security in untrustworthy execution environments. This chapter reviews the fundamental contributions of this work. Directions for future research are discussed in Section 11.2.

11.1. Contributions

First and foremost, I identified and proposed an approach to the software security problem from a system perspective. The software security problem is stated as ensuring the execution integrity and algorithm privacy of a software in untrustworthy environments. The various aspects of the approach, diversity schemes and One-way Translation, are derived from considering the software protection issue within the context of the application system rather than treating it in isolation. As such, the approach is more systematic and comprehensive than the previous work in this area.

I have described the underlying principles of the approach, based on a framework of complexity measures. I have identified a set of fundamental program characteristics that, when instigated, yield the desired complexity and at the same time, are well suited to the

requirement of quantifiable strength.

I described the detailed techniques derived from the framework, and demonstrated their security strength from a theoretical standpoint and assessed their feasibility with empirical experiments. A working prototype of the described mechanism—an augmented *C* compiler—has been implemented, and tested on representative benchmarks.

The experiments using the prototype implementation provided empirical assessments of the techniques with encouraging results. For one thing, the experiments demonstrated that software security measures can be implemented by automated tools such as the one described in this work, with a relatively low level of user intervention and adjustable system overhead. Furthermore, the performance results from the experiments significantly outperformed the only reference point currently available [15].

The prototype implementation is portable and highly extensible. Built-in support for programmer specified transformations is provided. In addition, the mechanism is general. The core ideas—control-flow flattening and aliasing—can be applied to any program irrespective of its programming language and underlying host architecture¹¹.

¹¹ This is true on an abstract level. In practice, some of the mechanisms cannot be implemented at the source level, however, they can be accommodated at a lower level of program representation.

11.2. Where do we go from here?

Other issues: In the final analysis, a few other issues need to be addressed to round out the overall approach. As discussed in Chapter 9, the issues of secure program upload and communications between the trusted home server and the remote program should be considered. In addition, an investigation into the input-spoofing problem could determine the class of applications for which this problem can be provably subsumed by the algorithm security and the execution integrity problems. This class of applications is also the class of applications for which this work will be useful.

Network deployment: There is also merit in assessing the feasibility of the approach in a real, production network environment. For example, how many probes should be deployed throughout the network and how often they should be refreshed have impact on the security mechanism, yet they are network-wide issues and should be considered in that context. Other issues include scalability due to state management on the part of the trusted home servers should also be appraised in the network environment.

Other applications: The approach, as stated here, has potential applications in other environments such as the protection of mobile agents and market-based distributed computing. A detailed investigation into the application contexts is needed to determine the feasibility of adopting the approach in those environments.

Language extensions: Support for additional programming languages such as C++ and JAVA represent useful extensions to the current work. Implementation of language extensions will be the subject of future work. The basic paradigm should be extensible to

other languages, but the implementation strategy might have to be adapted to suit the other language environments.

11.3. The Final Conclusion

In summary, I note the following about the research documented in this dissertation:

- Software security via code transformations can be accomplished with automated tools.
- The theoretic and algorithmic underpinning of the code transformations must be considered in order to provide demonstrable protection.
- Software protection should not be considered in isolation; the mechanisms described in this work provide a system approach to the problem of software protection. The various techniques must work in unison to acquire the desired benefit.

References

- [1] A. Aho, R. Sethi, J. Ullman, "Compilers, Principles, Techniques, and Tools". Addison-wesley Publishing Company.
- [2] G. Aigner, et al. "The SUIF2 Compiler Infrastructure". Documentation of the Computer Systems Laboratory, Stanford University.
- [3] G. Ammons, J. Larus. "Improving Data-flow Analysis with Path Profiles". In the Proceeding of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal Canada, June 17-19, 1998.
- [4] D. Aucsmith. "Tamper Resistant Software". In the Proceedings of the first information hiding workshop, Cambridge, England, 1996.
- [5] T. Ball, J. Larus. "Efficient Path Profiling", In the proceeding of MICRO-29, December 2-4, 1996, Paris, France.
- [6] T. Ball, J. R. Larus. "Optimally Profiling and Tracing Programs". ACM Transactions on Programming Languages and Systems, Vol 16, No. 4, July 1994, pp1319-1360.
- [7] T. Ball, J. Larus. "Improving Data Flow Analysis with Path Profiling". In the Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, 1998. pp 72-84.
- [8] T. Ball, P. Mataga, and M. Sagiv. "Edge Profiling versus Path Profiling: The Showdown", In the Proceeding of the 1998 ACM Conference on the Principles of Programming Languages, pp134-148. San Diego. California.

- [9] J. Banning, "An Efficient Way to find the Side effects of procedure calls and the aliases of variables". In the proceeding of the Sixth Annual ACM Symposium on the Principles of Programming Languages. pp 29-41. January 1979.
- [10] S. Berkovits, J. Guttman, V. Swarup. "Authentication for Mobile Agents", in: Giovanni Vigna (Ed.): Mobile Agents and Security. pp 114-136. Springer-Verlag, 1998.
- [11] C. Cifuentes. "Structuring Decompiled Graphs. Personal Communication. Proceedings of the International Conference on Compiler Construction (CC'96), Lecture Notes in Computer Science 1060. Linkoping, Sweden. 22-26 April 1996, pp 91-105.
- [12] C. Cifuentes and KJ Gough, "Decompilation of Binary Programs", Software - Practice & Experience. Vol 25(7), July 1995. Pp 811-829.
- [13] C Cifuentes, M Van Emmerik, and N. Ramsey, *The Design of a Resourceable and Retargetable Binary Translator*. Proceedings of the Sixth Working Conference on Reverse Engineering, Atlanta, USA, October 1999, IEEE-CS Press, pp 280-291.
- [14] Cloakware Systems. <http://www.cloakware.com>.
- [15] Cloakware Systems. "Building Cloakware". Five minute presentation at the 2000 IEEE Symposium of Security and Privacy. May, 2000. Berkeley, California.
- [16] D. Chess. "Security issues in mobile code systems". in: Giovanni Vigna (Ed.): Mobile Agents and Security. pp 1-14. Springer-Verlag, 1998.
- [17] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, S. Staniford-Chen, R. Yip, D. Zerkle. GrIDS: "*A Graph-Based Intrusion Detection System*". National Information System and Security Conference, Baltimore, 1997.
- [18] J. Choi, R. Cytron, J. Ferrante. "Automatic Construction of Sparse Data Flow Evaluation graphs". In 18th Annual ACM Symposium on the Principles of Programming Languages. pp55-66.
- [19] C. Collberg, C. Thomborson, D. Low. "Breaking Abstractions And Unstructuring Data

- Structures”, IEEE International Conference on Computer Languages, Chicago, May 1998.
- [20] C. Collberg, C. Thomborson, and D. Low. “A Taxonomy of Obfuscating Transformations”. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
 - [21] T. Cormen, C. E. Leiserson, R. Rivest, “Introduction to Algorithms”. The MIT Press, 1993. Tenth edition.
 - [22] P. Devanbu and S. Stubblebine. "Stack And Queue Integrity On Hostile Platforms". Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, May 1998.
 - [23] D. Dyer. “Java Decompilers Compared”. <http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html>, June 1997.
 - [24] D. M. Dhamdhere, U. Khedker, "Complexity of Bidirectional Data-flow Analysis". In the proceedings of the 20th ACM's Conference on Principles of Programming Languages (POPL), pp397-408. Januaray, 1993. South Carolina. USA.
 - [25] “Divide and Conquer”, Economist, July 29, 2000, pp. 77-78.
 - [26] G. Edjlali, A. Acharya, V. Chaudhary, “History Based Access Control for Mobile Code”, in: Jan Vitek; Christian Jensen (Eds.): Secure Internet Programming, LNCS 1603, Springer-Verlag, pp. 413-432, 1999.
 - [27] M. Elder, J. C. Knight, "Dynamic Reconfigurable Systems". CS Technical Report, CS-00-19. Department of Computer Science, University of Virginia.
 - [28] M. Elder, J. Knight, “Security Attacks on Critical Infrastructure Systems”. Computer Science Technical Report. CS-98-23.
 - [29] B. Elspas, K. Levitt, R. Waldinger, and A. Waksman, “An assesement of techniques for proving program correctness”, Computer Surveys, June 1972.
 - [30] M. Emami, R. Ghiya, L. Hendren. “Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers”, SIGPLAN 94, pp242-256. June 1994. Orlando, Florida USA.

- [31] D. Evans and A. Twyman. "Flexible Policy Directed Code Safety". In the proceeding of the 1999 IEEE Symposium on Security and Privacy, Oakland, California, May 9-12, 1999.
- [32] W. Farmer, J. D. Guttman, and V. Swarup. "Security for Mobile Agents: Issues and Requirements". In Proceedings of the 19th National Information System Security Conference, pp591-597. Baltimore, Maryland.
- [33] S. Forrest, A Soma. "Building Diverse Computer Systems". In the Proceedings of the 1996 Hot Topics of Operating Systems.
- [34] A. Grimshaw, A. Ferrari, F. Knabe, M. Humphrey, "Legion: An Operating System for Wide-Area Computing". *IEEE Computer*, 32:5, May 1999. Pp29-37.
- [35] M. Hecht. "Flow Analysis of Computer Programs". Elsevier North-Holland, New York. 1977.
- [36] M. Hennessey, J. Riely, "Type Safe Execution of Mobile Agents in Anonymous Networks", in: Jan Vitek; Christian Jensen (Eds.): Secure Internet Programming, LNCS 1603, Springer-Verlag, pp. 95-116, 1999.
- [37] M. Hind, M. Burke, P. Carini and J. Choi. "Inter-procedural Pointer Analysis". *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 4, July 1999, pp 848-894.
- [38] M. Hind, A. Pioli. "Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses". Research Report 21251, IBM T.J. Watson Research Center.
- [39] M. A. Hitunen and R. D. Schlichting. "Adaptive Distributed and Fault-Tolerant Systems" *International Journal of Computer Systems Science and Engineering*, vol. 11, No. 5, pp. 125-133, September 1996.
- [40] F. Hohl. "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts". In *Lecture Notes in Computer Science*, vol. 1419, Mobile Agents and Security. Edited by G. Vigna. Springer-Verlag, 1998.
- [41] F. Hohl, "A Framework to Protect Mobile Agents by Using Reference States". In: *Proceedings of*

the 20th International Conference on Distributed Computing Systems (ICDCS 2000).

- [42] F. M. Ingels. "Information and Coding Theory". Intext Educational Publishers, 1971.
- [43] InterTrust. <http://www.intertrust.com>.
- [44] R. Keller, U. Holzle. "Binary Component Adaptation", Computer Science Technical Report. Department of Computer Science, University of California at Santa Barbara. TRCS-97-20
- [45] J. Knight, K. Sullivan, M. Elder, C. Wang. "Survivability Architectures: Issues and Approaches" In Proceedings: DARPA Information Survivability Conference and Exposition. IEEE Computer Society Press. Los Alamitos, CA, January 2000, pp. 157-171.
- [46] J. Knight, M. Elder, J. Flynn, P. Marx, "Summary of Three Critical Infrastructure Systems". Computer Science Technical Report, CS-97-27, Department of Computer Science, University of Virginia.
- [47] D. Knuth, and F. Stevenson. "Optimal Measurement Points For Program Frequency Counts". BIT 13, pp313-322, 1973.
- [48] W. Landi, B. Ryders. "A Safe Approximate Algorithm for Interprocedural Pointer Analysis". CS Technical Report, Rutgers University, 1991.
- [49] W. Landi. "Undecidability of Static Analysis". ACM Letters on Programming Languages and Systems, Vol. 1, No. 4. December 1992, pp 323-337.
- [50] W. Landi. "Interprocedural Aliasing in the Presence of Pointers". Ph.D. Dissertation, Rutgers University, 1992.
- [51] J. R. Larus. "Efficient Program Tracing". Computer, Vol 26. No. 5. May 1993. Pp52-61.
- [52] T. Lunt. "Detecting Intruders in Computer Systems". In the conference record of the 1993 Conference on Auditing and Computer Technology. 1993.
- [53] S. Macrakis. "Protecting source code with ANDF". OSF public document, 1993

- [54] T. Marlowe, W. Landi, B. Ryder, J. Choi, M. Burke, and P. Carini. *Pointer-induced Aliasing: A Clarification*. ACM SIGPLAN Notices, 28(9), pages 67-70, September 1993.
- [55] T. Marlowe, B. Ryder. "An efficient hybrid algorithm for incremental data-flow analysis". In the Proceedings of the seventeenth annual ACM Symposium on the Principles of Programming Languages. 1990. pp 184-196.
- [56] G. McGraw, G. Morrisett. "Attacking Malicious Code". Final Report of the Malicious Code InfoSec Science and Technology Study Group to the InfoSec Research Council.
- [57] C. Meadows. "Detecting Attacks On Mobile Agents". In Proceedings of the DARPA workshop on Foundations for secure mobile code, Monterey CA, USA, March 1997.
- [58] Y. Minsky, R. van Reness, F. Schneider, S. Stoller. "Cryptographic support for fault-tolerant computing", In the proceeding of the Seventh ACM SIGOPS European Workshop. Pp109-114. Connemara, Ireland. September, 1996.
- [59] S. Muchnick. "Advanced Compiler Design Implementation". Morgan Kaufmann Publishers, 1997.
- [60] J. C. Munson and T. M. Kohshgoftaar. "Measurement of Data Structure Complexity". Journal of Systems Software, 20:217-225, 1993.
- [61] E. Myers. "A Precise Inter-procedural Data Flow Algorithm". In the conference record of the Eighth Annual ACM Symposium on Principles of Programming Languages. Williamsburg, VA. January, 1981. pp219-230
- [62] P. Neumann, "Practical Architectures for Survivable Systems", Report for the Army Research Lab. 2000.
- [63] P. Neumann, Computer-Related Risks. ACM Press, New York, and Addison-Wesley, Reading, MA, 1994.
- [64] G. Necula, "Translation Validation for an Optimizing Compiler", In the proceeding of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation. Vol 35, No.

5. May 2000, Vancouver, British Columbia, Canada.

- [65] G. Necula, "Proof-Carrying Code". In the Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France, January, 1997.
- [66] The Prolangs Analysis Framework (PAF). Rutgers University.
<http://www.prolangs.rutgers.edu/public>
- [67] Report of the Defense Science Board Task Force On Information Warfare—Defense (IW-D), Office of the Secretary of Defense. November 1996. Available at <http://www.jya.com/iwd.htm>.
- [68] A. Pnueli, M. Siegel, and E. Signerman. "Translation validation". In Bernhard Steffen, editor, Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS'98, Vol. LNCS1384, pp151-166. Springer, 1998.
- [69] C. Pu, A. Black, C. Cowan, J. Walpole, A Specialization Toolkit to Increase the Diversity in Operating Systems. Proceedings: 1996 ICMAS Workshop on Immunity-based systems. Nara, Japan. December 1996.
- [70] Report of The Presidential Commission on Critical Infrastructure Protection, 1997.
- [71] P. Porras, P. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances". In the proceeding of the 1997 National Information Systems Security Conference. Baltimore, 1997.
- [72] C. V. Ramamoorthy, K. H. Kim, W. T. Chen, "Optimal Placement of Software Monitors Aiding Systematic Testing", IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December, 1975. Pp403-411.
- [73] R. Rivest, A. Shamir, Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21,2, February 1978, pp120--126.
- [74] T. Sander, C. Tschudin. "Protecting Mobile Agents Against Malicious Hosts". Lecture Notes of

Computer Science, Edited by G. Vigna. Vol. 1419. Mobile Agents. 1998. Springer-Verlag.

- [75] D. Schnackenberg. "IDIP Concept Document". Boeing, Personal Communication.
- [76] D. Siewiorek, and R. Swarz. "The Theory and Practice of Reliable System Design". By Digital Press, 1982.
- [77] K. Sullivan, J. C. Knight, X. Du, and S. Geist, "Information Survivability Control Systems". Proceedings of the International Conference of Software Engineering, 1998.
- [78] G. Vigna, "Cryptographic Traces for Mobile Agents", in Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, June 1998. Edited by G. Vigna.
- [79] C. Wang, J. Hill, J. Knight, J. Davidson. "Protecting Network Probe Programs Against Untrustworthy Hosts". Computer Science Department Technical Report, CS-00-12.
- [80] C. Wang, J. Knight. "A Framework to Run Trusted Software on Untrustworthy Platforms", Computer Science Department Technical Report, CS-99-15. March 1999.
- [81] B. Yee. "A Sanctuary for Mobile Agents". Technical Report CS97-537. Computer Science Department, University of California in San Diego, USA.
- [82] T. Murayama, M. Manbo, E. Okamoto. "A Tentative Approach to Constructing Tamper Resistant Software". In the Proceedings of the New Security Paradigms Workshop, 1998. Great Landale, UK. pp. 23-33.