

# **Protection against overflow attacks**

by

Ge Zhu

A dissertation submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of

**DOCTOR OF PHILOSOPHY**

**Major: Computer Science**

Program of Study Committee:  
Akhilesh Tyagi (Major Professor)  
Johnny S. Wong  
Soma Chaudhuri  
Wallapak Tavanapong  
James A. Davis

Iowa State University

Ames, Iowa

2006

Copyright © Ge Zhu, 2006. All rights reserved.

UMI Number: 3229145

### INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

**UMI<sup>®</sup>**

---

UMI Microform 3229145

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

Graduate College  
Iowa State University

This is to certify that the doctoral dissertation of

Ge Zhu

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

# TABLE OF CONTENTS

ABSTRACT .....	vii
CHAPTER 1. INTRODUCTION .....	1
1.1 Internet threats .....	1
1.1.1 Threats.....	1
1.1.2 Buffer overflow attacks.....	2
1.2 Current research against overflow attacks .....	4
1.3 Our contribution and thesis organization.....	6
CHAPTER 2. Function Pointer Protection.....	8
2.1 Introduction.....	8
2.2 Direct buffer overflow vs. indirect buffer overflow .....	10
2.2.1 Direct buffer overflow example.....	10
2.2.2 Indirect buffer overflow example .....	11
2.3 Related work on overflow attacks .....	12
2.3.1 StackShield .....	13
2.3.2 StackGuard.....	13
2.3.3 Efficient Detection .....	14
2.3.4 Automated Detection .....	14
2.3.5 Annotation-assisted Static Checking .....	14
2.3.6 Cleanness Checking of String Manipulations.....	15
2.3.7 Backwards-compatible Bounds Checking .....	15
2.3.8 Type-assisted Dynamic Buffer Overflow Detection.....	15
2.3.9 Libsafe and Libverify.....	16
2.3.10 IBM GCC extension .....	16

2.3.11 PointGuard .....	17
2.3.12 Non-executable User Stack.....	19
2.4 Function pointer protection.....	19
2.4.1 Potential function pointers .....	20
2.4.2 Function pointer encryption/decryption.....	21
2.5 XOR encryption.....	22
2.5.1 Key selection.....	22
2.5.2 Implementation .....	23
2.5.3 Security testing.....	24
2.5.4 Performance .....	25
2.6 RC5 encryption.....	27
2.6.1 RC5 .....	28
2.6.2 Implementation .....	29
2.6.3 Result .....	29
2.6.4 Performance .....	32
2.7 Conclusion and future work.....	33
2.8.1 Hidden key .....	34
2.8.2 Dynamic linking.....	34
CHAPTER 3. CONTROL FLOW CHECKING AUTOMATA .....	35
3.1 Introduction.....	35
3.2 Trust automata .....	38
3.3 Embedded framework.....	41
3.3.1 Trust policy of control flow graph .....	41
3.3.2 Electronic commerce examples .....	48
3.3.3 Program control flow violation examples.....	52

3.4 Support for traditional trust management .....	54
3.4.1 Trust algebra .....	55
3.5 Related work on program level anomaly detection .....	57
3.5.1 Fast Automaton-Based Method .....	57
3.5.2 Intrusion Detection via Static Analysis.....	57
3.6 Future work.....	58
3.6.1 Separate monitor .....	58
3.6.2 Transaction level trust.....	59
3.6.3 Trust function.....	59
3.7 Conclusions.....	59
CHAPTER 4. STREAM AUTOMATA .....	60
4.1 Introduction.....	60
4.2 Related work of formal model on monitoring .....	64
4.3 Stream automata .....	64
4.3.1 Enforcement actions.....	67
4.4 Enforceable policy examples and the power of stream automata.....	71
4.4.1 Power of stream automata: TM based model.....	76
4.4.2 Power of stream automata: informal reasoning .....	80
4.5 Buffer overflow & honeynet examples.....	85
4.5.1 Program monitor example.....	85
4.5.2 Honey wall policy scenarios .....	89
4.5.3 Project SAFE .....	94
4.6 Conclusions.....	99
CHAPTER 5. CONCLUSION AND FUTURE WORK.....	101
5.1 Conclusions.....	101

5.2 Future work.....	102
BIBLIOGRAPHY.....	104
ACKNOWLEDGEMENTS.....	112

## ABSTRACT

Buffer overflow happens when the runtime process loads more data into the buffer than its design capacity. Bad programming style and lack of security concern cause overflow vulnerabilities in almost all applications on all the platforms. As a common vulnerability, buffer overflow accounts for more than 20% of the public vulnerabilities reported in Common Vulnerabilities and Exposures (version 20040901).

Buffer overflow attack can target any data in stack or heap. A common target of overflow attack is return address stored in the stack during runtime. By overflowing the return address, the attacker could redirect the program control flow when the current function returns. Many solutions have been proposed to protect return address from being overflowed, like StackGuard, StackShield et cetera. However the current solutions ignore the overflowed targets other than return address.

Function pointer, for example, is another possible target of overflow attack. By overflowing the function pointer in stack or heap, the attacker could redirect the program control flow when the function pointer is dereferenced to make a function call. To address this problem we implemented protection against overflow attacks targeting function pointers. During compiling phase, our patch collects the set of the variables that might change the value of function pointers at runtime. During running phase, the set is protected by encryption before the value is saved in memory and decryption before the value is used. The function pointer protection will cover all the overflow attacks targeting function pointers.

To further extend the protection to cover all possible overflowing targets, we implemented an anomaly detection which checks the program runtime behavior against control flow checking automata. The control flow checking automata are derived from the source codes of the application. A trust value is introduced to indicate how well the runtime

program matches the automata. The attacks modifying the program behavior within the source codes could be detected.

Both function pointer protection and control flow checking are compiler patches which require the access to source codes. To cover buffer overflow attack and enforce security policies regardless of source codes, we implemented a runtime monitor with stream automata. Stream automata extend the concept of security automata and edit automata. The monitor works on the interactions between two virtual entities: system and program. The security policies are expressed in stream automata which perform Truncation, Suppression, Insertion, Metamorphosis, Forcing, and Two-Way Forcing on the interactions. We implement a program/operating system monitor to detect overflow attack and a local network/Internet monitor to enforce honeywall policies.

## CHAPTER 1. INTRODUCTION

Network security has been an active research area for almost two decades. Buffer overflow, as an attack technique, has been studied and researched since the beginning of Internet threats. In this chapter, we give a brief introduction to Internet threats and buffer overflow attacks. Then we talk about the existing approaches against overflow attacks and their shortcomings. Based on existing work, we present our contribution and organization of the thesis.

### 1.1 Internet threats

#### 1.1.1 Threats

Internet was technically born on January 1<sup>st</sup>, 1983 when ARPANET officially changed to use TCP/IP protocol [Wiki, 2006]. At its birth, Internet included only a few hundred machines and served a few hundred users from the government, institutes and universities [Whitehouse, 1996].

As a popular media for the modern world, Internet has been growing at a very dramatic speed. Within the last twenty-two years, Internet users have grown from less than 1,000 to over 1,000,000,000 as of January, 2006 [Stats, 2006]. People depend heavily on Internet for working, shopping, entertaining, socializing, living, and studying.

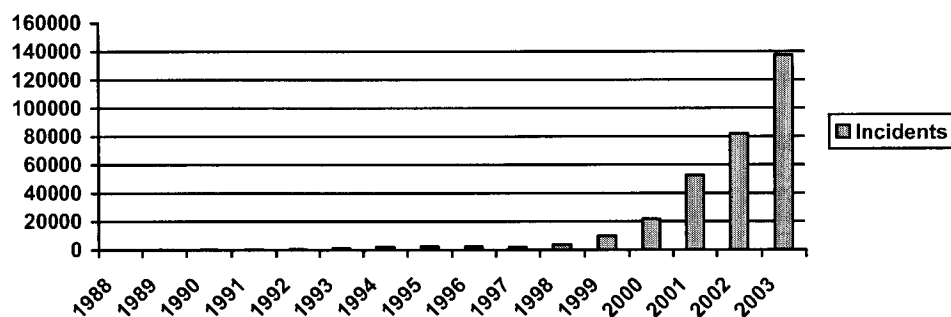
However, Internet has never been a secure and safe place to surf. Worm, virus, spam, phishing, adware etc. threaten Internet users everyday. The number of Internet incidents reported to CERT has almost doubled every year from 1988 to 2003 [CERT, 2005].

Internet attacks affect thousands to millions of machines and cause millions to billions of dollars worth of damage. Take two of the most famous worms as examples:

1. Morris Worm – one of the earliest Internet worms – was launched by Robert T. Morris on November 2, 1988 from MIT. The worm infected 6,000 major UNIX

machines, which accounted for 10% of the Internet machines at that time and caused the damage estimated at 1,000,000 dollars. [Wiki, 2006]

2. Code-Red Worm II – one of the most recent worms – was launched on August 4<sup>th</sup>, 2001. Code-Red Worm II infected over 359,000 machines on Internet within 14 hours [Moore *et al.*, 2001] and Code-Red Worm caused total damages estimated at 2,600,000,000 dollars [Lemos, 2003].



**Figure 1.1 Incidents Reported to CERT**

Today the software installed on computers usually consists of millions of lines of source codes. The vulnerabilities are inevitable on all the platforms. Among them, buffer overflow is the most popular one which the Internet attacks take advantage of.

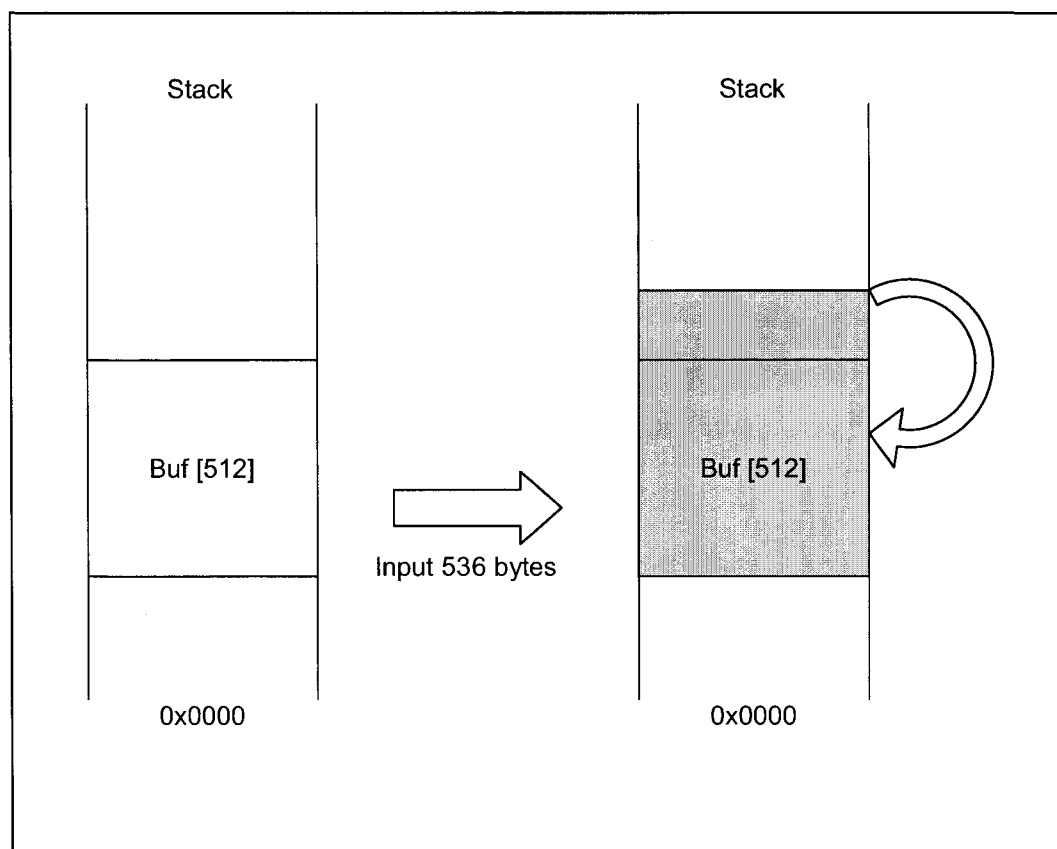
### **1.1.2 Buffer overflow attacks**

Buffer overflow attack happens when the runtime program tries to load into the buffer more data than its design capability. Bad programming style and lack of security concern cause overflow vulnerabilities in almost all applications on all the platforms. If a hacker successfully exploited buffer overflow vulnerability, the hacker might be able to modify system configuration, set up backdoor and run random program with administrator privilege.

Many attacks utilize buffer overflow vulnerabilities. According to CVE version 20040901 (Common Vulnerabilities and Exposures), there were 3,053 total vulnerabilities

from 1999 to 2004. Among them, 644 were overflow attacks, which accounts for 21% of all attacks.

The Morris Worm and Code-Red Worm mentioned above both utilize buffer overflow vulnerabilities to spread. We use Morris Worm to demonstrate how a buffer might be overflowed and how the attacker can take advantage of it.



**Figure 1.2 Morris Worm Overflow Exploit**

One target of Morris Worm – *BSD Fingerd v4.3* – gets input using a standard c library *gets()* function call. *gets()* will put the input into the buffer (size of 512 bytes) in the *fingerd* runtime stack without checking the boundary. Morris Worm exploits this vulnerability by sending a carefully designed input string of 536 bytes to it. The input string will fill the buffer first and then overflow the memories adjacent to the buffer in the stack. As

the return address is stored adjacent to memory of local variables in the stack, the input string has a chance to overflow the return address. Thus, when the current function returns, the overflowed return address is loaded and the program jumps to the middle of the stack, where the crafted instructions in the input string get executed. As most of *fingerd* daemons run with root privilege, Morris Worm could execute random instructions on the victim machine. [Spafford, 1988]

## 1.2 Current research against overflow attacks

Significant research has been done against overflow attacks. We categorize them by the program phase where the proposed solutions are applicable. Usually there are two phases for all the programs: compiling and running. For the compiling phase, the compiler turns the program in high level language into binary codes. For the running phase, the operating system loads the program and allocates memory of text, data and stack for the process and then executes it.

Compiling phase solutions include safe language, static analyzer and compiler modification [Younan *et al.*, 2004]. The approaches in this category require access to the source codes and do not work for legacy binary codes.

1. Safe language is designed to eliminate the possibility of overflow attack from the very beginning. Program annotations and special structures help compilers to detect possible overflow danger at the early phase. The approach requires the program be implemented in safe language.
2. Static analyzer scans the existing program source code to locate the possible overflow conditions. Due to the incomplete information from static analysis, static analyzer might generate both false positives and false negatives.

3. Compiler modification generates extra checking codes for the program during compiling. At runtime the codes could check overflow conditions, protect important variables or enforce certain security policies etc.

Running phase solutions include sandboxing, anomaly detection and environment modification [Younan *et al.*, 2004]. The running phase solutions have the advantage that they do not need the access to the source codes.

1. Sandboxing enforces the “Principle of Least Privilege”. The runtime process will not be protected from overflow attacks. The sandbox grants only necessary privilege to process and limits the possible harm ability of exploited process.
2. Anomaly detection derives the normal behaviors of process from runtime collecting or static analysis. The runtime process is monitored against the normal behaviors. If certain deviation between the runtime process and normal behavior is detected, a corresponding action will be taken, like terminating the process, sending an alert to the administrator, or logging process behaviors etc.
3. Environment modification includes the modifications to the operating system, library, and hardware. These modifications might disable codes running other than DATA section, randomize memory layouts or change the implementation of library etc.

For easier comparison with our approaches, we will discuss the detail of related works in Chapter 2, 3 and 4.

Majority of the current solutions have ignored the advanced overflow techniques that target objects other than the return address. Overflow attacks could target any data in the stack or heap. The overflowed data will change the runtime behavior of the program. A carefully crafted change can finally lead to attacker’s code.

Take function pointer as one example: an attacker overflows a function pointer with the starting address of attacking codes. When the function pointer is dereferenced during

runtime, the attacking codes get executed. Biondi *et al.* describes an overflow attack targeting at function pointer in Skype [Biondi *et al.*, 2006].

### 1.3 Our contribution and thesis organization

How to protect against the advanced overflow attacks is the main problem that this thesis tries to solve. We take three different approaches to address this problem.

Chapter 2 introduces an encryption based protection for potential function pointers through a compiler patch. This protection gathers the information on potential function pointers set at compiling phase and inserts patch codes to encrypt/decrypt function pointers. At runtime, the patch codes will encrypt the function pointer before putting it into memory and decrypt it before loading it from memory. The technique protects the advanced overflow attacks targeting function pointers. The general idea applies to other targets of the process also. The system overhead brought by the function pointer protection is lower than 5% for simple encryption algorithm.

Chapter 3 introduces a control flow integrity enforcement technique through compiler patch. This technique tries to solve the overflow problem from an anomaly detection point of view. The basic idea is to enforce program runtime behavior according to the source code design. While all the overflow attacks succeed in changing program runtime behavior, we address a subset of those attacks which change the program runtime behavior within program codes. The control flow enforcement technique derives control flow graph from the compiling phase to enforce the program runtime behavior according to control flow graph. The monitor generates a program level trust for the program execution according to how well the program behavior matches the control flow checking automata. In such a way, if the attacker exploits and changes the behavior of the process, the mismatch between the program behavior and trust automata will lower the trust level to alert the administrator.

Both function pointer protection and control flow enforcement are compiling phase techniques, which require access to source codes and are not capable of protecting legacy codes. We present the protection of legacy codes in Chapter 4 with a powerful runtime monitor which enforces stream automata. This work enforces security policies on the interactions between two entities that communicate with each other. As these two entities are abstract objects, stream automata could be applied to all the cases where interacting objects are involved. Considering the two entities as program and operating system, we implement the stream automata monitor to enforce program level security policy and detect overflow attacks. Considering the two entities as server and network, we implement Stream Automata Firewall Engine to enforce the security policy of the network flows.

Chapter 5 will summarize and outline the future work.

## CHAPTER 2. FUNCTION POINTER PROTECTION

In this chapter, we target indirect buffer overflow attacks that overflow a buffer in memory to re-point a function pointer to the attacker's program. This type of attack could bypass most of the current stack protection mechanisms. Our proposed approach encrypts a function pointer before it is put into the memory and decrypts it before it is taken from the memory. Each function pointer is encrypted with a unique key that is randomized for each program run. This leads to two desirable properties: (1) orthogonality of key space, (2) zero incremental knowledge gain for the adversary between two attacks on two different program runs. The key space orthogonality does not allow a one key compromise to propagate to other function pointers. The “zero knowledge gain” forces the adversary to compromise all (or most of) the keys in the same run. This is difficult since runtime key randomization leads to a  $2^{32}$  iteration brute force attack on each key for a 32-bit architecture. This scheme was incorporated into GCC-3.0 on RedHat 7.0 Linux distribution. The performance overhead of this scheme is below 4.5% on Apache web server version 1.3.22 with WebStone 2.5 as benchmark.

### 2.1 Introduction

Buffers or arrays are designed to hold up to a certain amount of data. A buffer overflows if a write access to an array element beyond its declared boundaries is attempted. Such an access overwrites some other program variable that is allocated in the vicinity of the overflowing buffer. Although, the data segments vulnerable to buffer overflow attacks are created accidentally through programming error or oversight, the buffer overflow exploits are very deliberately researched, thereby making it an increasingly common type of security attack on data integrity. In buffer overflow attacks, the extra data may contain code designed to trigger specific actions, in effect sending new instructions to the attacked computer that could, for example, damage the user's files, change data, or disclose confidential information.

Buffer overflow attacks are said to have arisen because the C programming language supplied the framework, and poor programming practices supplied the vulnerability.

We distinguish between two categories of buffer overflow attacks: direct and indirect buffer overflow attacks. Direct buffer overflow attacks use direct mechanisms to modify a program counter bound address. The indirect buffer overflow attacks, on the other hand, use indirect means of getting to the program counter such as tampering of a function pointer. We define and describe direct and indirect buffer overflow attacks in Section 2.2.

Many research projects have proposed techniques to prevent buffer overflow attacks, such as run-time bounds checking, stack protection etc. Most solutions are either not able to protect against indirect buffer overflow or experience excessive system overload for such protection. PointGuard [Cowan *et al.*, 2003] does deal with pointer protection. However it makes weak assumptions about its attack space. We discuss some of the existing solutions in more detail in Section 2.3.

In this chapter we offer an efficient and effective technique to defend against buffer overflow attacks directed at function pointers in stack, global, or heap data spaces. The key idea is to encrypt a function pointer whenever it is stored in the memory. It is decrypted after it is retrieved from memory before any use. Moreover, each function pointer has a distinct encryption/decryption key that is altered for each run. The attacker can overflow into a function pointer only an un-encrypted address. The automatic decryption of an un-encrypted function pointer before its use will generate an invalid address. We discuss the proposed scheme in more detail in Section 2.4.

The encryption algorithm of the protection is vital for security and efficiency. We implement both simple and complex encryption schemes which are discussed in Section 2.5 and 2.6.

We conclude in Section 2.7.

## 2.2 Direct buffer overflow vs. indirect buffer overflow

Buffer overflow occurs due to lack of bounds checking in C. We distinguish between the following two categories of buffer overflow attacks: direct and indirect. The main objective of a buffer overflow attack is to tamper with a program visible value that will eventually be moved to the program counter (PC). We refer to such values as PC-bound values. This is how the control is redirected to the attack program. When such a PC-bound value is moved to PC directly after tampering, we call it a direct attack. Such is the case for return address on the stack. The function return action directly moves the tampered return address into PC. Some of the PC-bound values, however, are moved to PC in a more indeterminate way. For instance, a tampered function pointer is moved to PC only when a function call is made with dereferencing of the function pointer. This is what we consider to be an indirect means of moving a PC-bound value to PC. Direct buffer overflow attacks typically target only the stack area of the memory. Indirect buffer overflow attacks can target all the three areas of memory: stack, heap and global static data.

We show two examples to demonstrate the basic difference between direct and indirect buffer overflow attacks. The shellcode in the two examples comes from [One, 1996]. It generates a system call to spawn a shell. All the examples are tested on RedHat 7.0 with GCC-3.0.

### 2.2.1 Direct buffer overflow example

As shown in Figure 2.1, the array *buf* is declared to be of length 1. Line 12 of *direct.c* puts the address of shellcode into *buf[2]*, which overflows into the return address on the stack. When main returns, the program calls the shellcode which generates a shell for the attacker.

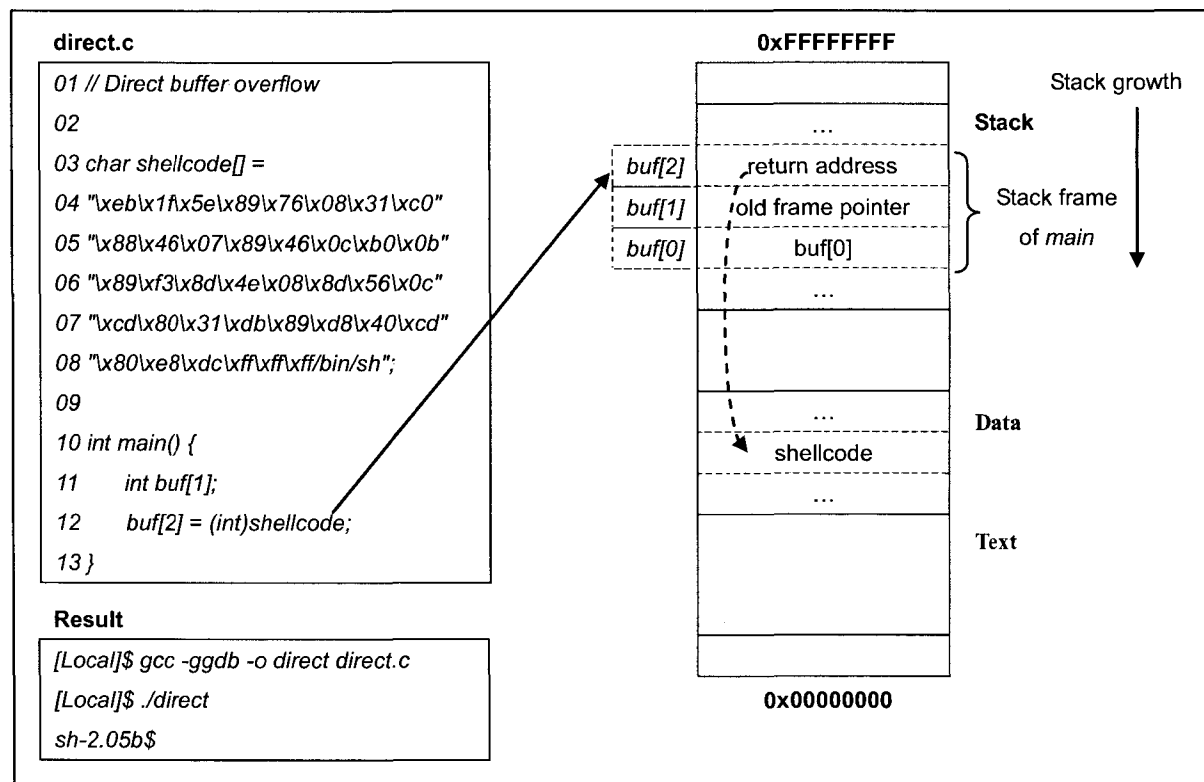


Figure 2.1 Direct Buffer Overflow

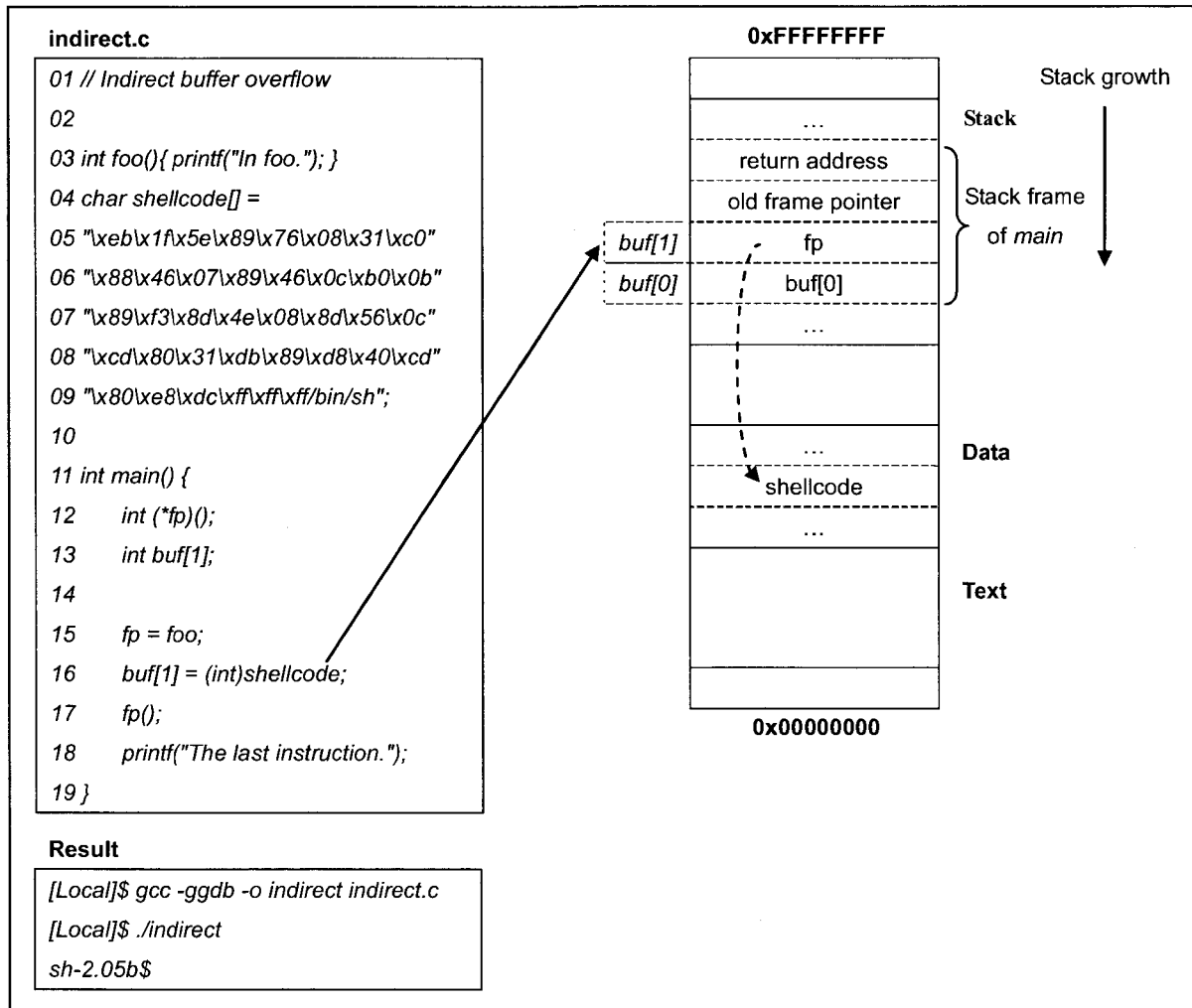
## 2.2.2 Indirect buffer overflow example

As shown in Figure 2.2, the array *buf* is declared to be of length 1. Line 16 of *indirect.c* puts the address of shellcode into *buf[1]*, which overflows into a function pointer *fp*, which is a neighbor of *buf* and which originally points to function *foo*. When *fp* is used to make a function call, the program calls the shellcode instead of the original function *foo*.

In this example the indirect buffer overflow attack occurs in stack. However, it could equally well target stack, heap and global static data space resident function pointers.

Although the two examples here are not real-life attacks, they demonstrate the possibility that indirect buffer overflow vulnerabilities are real and feasible. The majority of the current solutions for buffer overflow attacks focus on direct buffer overflow attacks, which are discussed in Section 2.3. We base our work on the security of function pointers

and defend against indirect buffer overflow, as explained and demonstrated in Section 2.4 and 2.5.



**Figure 2.2 Indirect Buffer Overflow**

## 2.3 Related work on overflow attacks

Significant research has been done against overflow attacks. A complete survey like [Younan *et al.*, 2004] is not presented here due to the limit of space. In this section we list

several popular countermeasures and discuss their shortcomings in the indirect overflow pointer of view.

### 2.3.1 StackShield

StackShield [Vendicator, 2000] is a GCC extension to protect a return address from buffer overflow in stack. It also protects function pointers against overflow. The basic technique is to maintain a fixed depth return address stack similar to the return address stack of a superscalar microarchitecture for jump address prediction. The program stack return address could be validated against the return address given by the return address stack. The weaknesses of this approach are as follows. First, the return address stack itself could be target of an attack. Second, since the return address stack is a finite size ( $k$ ) array, the attack can choose to overflow a return address that is at least  $k$  stack frames away. The return address stack does not maintain the call history that far away, and hence the attack will succeed. Third, the protection for function pointers is not fully functional and stops protecting programs with executable asm code in DATA or STACK segment.

### 2.3.2 StackGuard

StackGuard [Cowan *et al.*, 1998] is another GCC extension to protect against stack smashing attacks. StackGuard is implemented as a small patch to the GCC code generator, specifically the *function\_prolog()* and *function\_epilog()* routines. The procedure *function\_prolog()* has been enhanced to lay down canaries on the stack when functions start, and *function\_epilog()* checks canary integrity when the function exits. Any attempt at corrupting the return address is thus detected before the function returns.

StackGuard only deals with the stack safety with respect to only a smashing attack, which is not enough to protect a function pointer in other parts of memory. The alteration of stack layout by embedding the canary sometimes has other undesirable effects that have led to low acceptability for StackGuard.

### 2.3.3 Efficient Detection

Efficient detection by [Austin *et al.*, 1994] uses an extended representation, safe pointers, to detect all spatial and temporal access errors. Three steps are needed to implement the protection. First, all the pointers in the program are transformed to safe pointer objects. Second, access check code segments are inserted before pointer or array dereferences to detect memory access errors. Third, operator conversion generates and maintains necessary object attributes for safe pointers. This approach leads to execution time overheads, which range from 130% to 540%, and data space overheads, which are below 100%. This approach also changes the representation of a pointer, which makes it incompatible with codes without bounds checking enabled.

### 2.3.4 Automated Detection

The automated detection by [Wagner *et al.*, 2000] formulates detection of buffer overruns as an integer range analysis problem. The basic idea is to model buffers as pairs of integer ranges that identify the size in use and the allocated size. An integer range constraint is generated for each statement in the program. Then each string buffer is checked with the constraints to see whether the size in use is not bigger than the allocated size. Due to imprecision in the range analysis, the approach generates a large number of false alarms.

### 2.3.5 Annotation-assisted Static Checking

Annotation-assisted static checking by [Larochelle *et al.*, 2001] extends the semantics comments of *LCLint* to represent assumptions and constraints of buffers. When a function call is issued, *LCLint* checks if the preconditions required of the called function are satisfied before the call. The technique uses annotations provided by programmers to detect possible buffer overflows. As many buffer overflows use unsafe library functions, those overflow

vulnerabilities could be detected by annotating the standard library as well. This approach produces false warnings and it also misses some vulnerabilities.

### 2.3.6 Cleanness Checking of String Manipulations

Cleanness checking by Dor *et al.* [Dor *et al.*, 2001] is another static analysis to detect all the possible overflow errors at compiling time. The result is promising as it will not report many false positives. However, this approach requires not only access to the source codes but also the annotation of preconditions, post conditions and side effects of functions from the programmer.

### 2.3.7 Backwards-compatible Bounds Checking

“Backwards-compatible bounds checking” by [Jones *et al.*, 1997] avoids incompatibility of “efficient detection” with legacy codes. This approach does not change the pointer representation. It maintains a table of all valid pointers with static bounds information about the data objects bound to the pointer. The function calls to allocate and free memory (like *malloc()/free()*) are modified to add/delete dynamically allocated objects. For stack objects, constructor/destructor mechanism is utilized to track those variables (as GCC is built to handle C++ as well). The pointer and array operations are modified to add bounds checking using the table. The performance overhead of this approach is up to 100 times the original execution time [Jones *et al.*, 1997].

### 2.3.8 Type-assisted Dynamic Buffer Overflow Detection

Type-assisted dynamic buffer overflow detection by [Lhee *et al.*, 2002] is a dynamic bounds checking technique. It uses a structure to remember buffer information. For automatic buffers and static buffers, this technique collects necessary buffer information during compilation. For dynamically allocated (heap) objects, it maintains a table to track those objects and their sizes at runtime. Range checking is done by looking up the table at run time.

This approach has inherent limitations such as its incapability to deal with stack buffers dynamically allocated with *alloca()* and variable-length automatic arrays. The execution time overhead of this approach for single function call is up to 6 times the original execution time.

### 2.3.9 Libsafe and Libverify

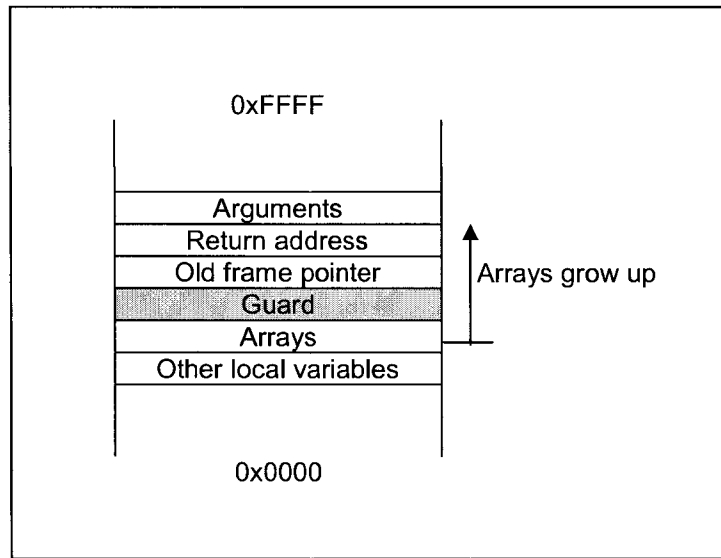
Libsafe and Libverify [Baratloo *et al.*, 2000] protect return address at runtime. Libsafe calculates a safe upper limit on the size of the buffer automatically according the previous frame pointer. All the unsafe library calls (like *strcpy()*) will be monitored such that the result won't overflow the address on top of the previous frame pointer. Libverify saves canary values (return addresses) into canary stack at the beginning of a function call and checks the canary when a function call returns. Libverify is different from StackGuard in that Libverify inserts checking codes at the runtime such that it applies to all legacy codes. Libsafe and Libverify do not protect against indirect overflow attacks.

### 2.3.10 IBM GCC extension

IBM GCC extension [IBM, 2005] protects applications from stack-smashing attack. This approach uses guard/canary similar to the StackGuard. The guard is put between the old frame pointer and the local variable to form a safe frame structure.

The guard will be set when the program enters a function call and will be checked before the program leaves a function call. If the guard is compromised when the program leaves a function call, the program stops and the behavior is logged.

The approach works partially for indirect overflow attacks in the way that it declares all the local variables in the stack are safe. From source codes transformation, it brings the declarations of all the local arrays to the top of the function. In such a way, if a smashing stack happens in one of the arrays, it can only overflow the memories on top of it in the stack. All the other local variables are safe, including function pointers.



**Figure 2.3 Safe Frame Structure**

However, this approach has the following limitations:

1. The function pointers in DATA section are not protected, for example, global function pointers, static function pointers, etc.
2. The function pointers within a data structure are not protected.
3. The function pointers in arguments are not protected.
4. The function pointers before dynamic allocated arrays are not protected.
5. The functions with trampoline codes are not protected.

### 2.3.11 PointGuard

PointGuard [Cowan *et al.*, 2003] is a very recent research which has been developed simultaneously with our work (Our work first appeared in a final project report to DARPA [Tyagi, 2003]).

PointGuard protects pointers from buffer overflow attacks. The basic scheme of PointGuard is to encrypt pointer values in memory and decrypt the pointers when loaded into CPU registers.

The main difference between PointGuard and our work are:

1. PointGuard uses one key per program. We use one key per function pointer. This increases the search space overwhelmingly for the adversary.
2. PointGuard assumes that the encrypted function pointer values are not read-compromised by the attacker. Otherwise, the attacker could take advantage of this assumption to bypass PointGuard. For example, if the attacker could read the encrypted value of a single pointer, the key of the program can be derived trivially since the function pointer value can be duplicated on a similar computing node controlled by the attacker. PointGuard is then compromised for all the pointers. We generate a unique key for each function pointer for each run of the program. Hence, even if the attacker is able to read the encrypted value for one function pointer, that is the only compromised function pointer; and the advantage lasts only for that run.
3. PointGuard utilizes a simple XOR encryption/decryption which is so weak that they have to make strong assumptions about the attacker. We implement RC5 encryption/decryption for function pointer such that the attacker can not derive the key even if it has the access to part of the clear text and encrypted text.
4. C allows integers to be cast into function pointers and vice versa. PointGuard makes it possible to violate the program semantics when such casting occurs. For instance, consider an integer variable assigned a function pointer through casting. PointGuard decrypts a function pointer only on dereferencing. Hence, the integer variable will have a different value than intended, i.e. an encrypted version that can alter the program behavior.
5. The casting in C also causes PointGuard to protect an incomplete set of variables that affect function pointers at runtime. For example, the attacker might overflow an integer which casts into a function pointer in the future. In such way, PointGuard protection is bypassed.

### 2.3.12 Non-executable User Stack

Solar Designer [Designer, 1997] proposes a runtime operating system patch against overflow attacks. The patch will disable any codes running in the stack. The approach covers the situation that the attacker's codes reside in the stack. Even if the indirect overflow attack succeeds, the codes will not execute. The advantage of this approach is that it does not change the source codes, and it applies for all the legacy codes.

However, the disadvantages of this works are

1. The indirect overflow attacks which overflow memories other than stack (heap section, for example) bypasses this protection.
2. Some programs rely on executable stack to generate dynamic codes. Those programs are not protected by this protection.

## 2.4 Function pointer protection

In Section 2.3, we argued that the current protection schemes would not protect against indirect buffer overflow attacks efficiently and with a complete coverage. Static bounds checking produces a large number of false positive detections and dynamic bounds checking schemes incur prohibitive system overhead. PointGuard, which protects pointers, has a weak adversary model and it also modifies the program semantics.

We focus on the protection of a function pointer directly. We propose a technique to protect function pointers allocated in any part of the memory: stack, global or heap data space. The main insight is to store only encrypted function pointer values into the memory and to decrypt them before each use (as an address or as an arithmetic value).

The encryption/decryption functions can either be invertible, simple functions or they could be one-way functions for added security.

The encryption keys are generated dynamically at runtime. Each function pointer will have one unique key. Consequently, it is very hard for an attacker to guess the key.

### 2.4.1 Potential function pointers

In this section, we define the set of function pointers protected by our scheme in a formal way. The C language has a very powerful casting mechanism allowing any pointer, even an integer, to be cast as a function pointer. Consider the following program.

Ex1.c:

```
void foo(int i){}
int main(){
    void (*pt)(int i);
    int i;
    i = foo;
    pt = i; }
```

*Notations:*

*e*: Type environment, *V*: variable, *E*: expression, *fp*: function pointer,  
*pfp*: potential function pointer, *exp*: expression type, *pfpexp*: pfp expression type,  
*op*: binary operations,  
*V<sub>type</sub>*: the type of *V*, stands for that in environment *e*, *V* indicates pfp type.

*Limitations:*

The rules explain pfp propagation only.  
 No validity of operations or assignments is checked.

*Type rules for pfp propagation:*

$$\frac{e \vdash V : (fp, e)}{e \vdash V : (pfp, e)} \quad (R1)$$

$$\frac{e \vdash V : (pfpe, e)}{e \vdash V : (pfpe, e)} \quad (R2)$$

$$\frac{e \vdash E_1 : (pfpe, e_1) \quad e \vdash E_2 : (exp, e_2)}{e \vdash E_1 \text{ op } E_2 : (pfpe, e_2)} \quad (R3)$$

$$\frac{e \vdash V : (V_{type}, e_1) \quad e \vdash E : (pfpe, e_2)}{e \vdash V = E : (pfpe, e_2[V \mapsto pfp])} \quad (R4)$$

$$\frac{e \vdash E_1 : (T_1, e_1) \quad e \vdash E_2 : (T_2, e_2)}{e \vdash E_1, E_2 : (T_2, e_2)} \quad (R5)$$

**Figure 2.4 PFP Propagation**

Although there is a warning during compilation of *Ex1.c*, GCC would still generate binary code for the program. If we limit the potential function pointers set to only the declared function pointers, our technique will not protect the variables assigned a function pointer through casting. An adversary could exploit this to attack the cast function pointer variable, for example, variable *i* in *Ex1.c*. The later assignment of *i* to *pt* with casting would not be encrypted/decrypted leaving it open to an attack.

We define a new type, *pfp*, to characterize the potential function pointer bound variables set. The potential function pointer bound variables set initially contains only the declared function pointers. However, through a static type flow analysis, this type (*pfp*) is propagated.

#### 2.4.2 Function pointer encryption/decryption

The basic mechanism to protect potential function pointers is encryption and decryption. We encrypt a potential function pointer before storing it in the memory and decrypt it before it is dereferenced or used in an expression. The encryption and decryption function should not cost too much computation time or it will not be acceptable despite the protection it offers.

The easiest scheme is to choose a simple encryption and decryption function, such as *XOR* (Boolean exclusive-or) of the function pointer with a key. Hence  $e(fp) = fp \text{ XOR } key$  and  $d(val) = val \text{ XOR } key$ , where  $e$  is the encryption function and  $d$  is the decryption function. We talk about simple scheme in Section 2.5.

However, the simple encryption scheme brings the concern of security vulnerability. If the attacker manages to know the original value and encrypted value, the key is trivial. To solve this problem we also implement complex scheme with a much stronger algorithm to protection function pointers in Section 2.6.

## 2.5 XOR encryption

### 2.5.1 Key selection

The key in XOR encryption needs to be carefully chosen. Some of the desirable characteristics are:

1. Variability: The key should have a different value for each function pointer that is encoded/decoded for the orthogonality based robustness, wherein a compromise of a single value has no impact on the others.
2. Random: The key should be hard to guess by an attacker. For each program run, it would be desirable to have a different key value. Any advantage derived by an attacker by guessing some of the keys in a given program run are completely lost for the future program runs (if each run chooses a different set of keys). This renders “incremental” attacks ineffective. A typical attack focuses on one sub-area of the program, derives some information about it in a given run. The future runs then target different sub-areas of the program. The information/knowledge derived in each program run is additive. This is a very powerful tool available to an attacker. By denying this additive/increment knowledge gain property, we take away one of the most powerful tools available to the adversary.
3. Invariance: The key should not change its value during a given program run so that each encryption and each decryption of the same function pointer sees the same key value (for correctness).

We could generate random keys, one instance per function pointer for a simple scheme. However, it poses a large overhead to store all the keys corresponding to all the protected variables. This will also make it more likely for an attacker to attack the keys themselves. One solution may be to generate keys that are dynamically derived at the run

time from function pointers and program themselves. As long as we guarantee the same key for encryption and decryption, the program correctness can be guaranteed.

For this scheme, we generate keys in two steps:

1. First step, we use the address of the pointer/variable as the encryption/decryption key at the compile time. Hence, encryption XOR instructions such as  $*fp0 = *fp0 \text{ XOR } \text{addr}(fp0)$  are generated at RTL level. This step gives us key variability.
2. Second step, note that the address of a function pointer is a fairly weak attribute. An adversary with access to a similar operating system and compiler can reproduce the addresses of the variables within a reasonable amount of time. That is the reason at each run we generate a random number with the seed of process id and current time. This random number serves as an XOR mask. Each instance of an encryption instruction  $fp \leftarrow fp \text{ XOR } \text{addr}(fp)$  is turned into  $fp \leftarrow fp \text{ XOR } \text{addr}(fp) \text{ XOR } \text{rand}(fp)$ .

An adversary can infer  $\text{addr}(fp)$  easily on a similar computing node which is a weakness of PointGuard. However, with this per run randomization, the adversary would have to guess  $2^{32}$  possible random masks (for a 32-bit architecture) in order to gain any useful information, which severely limits the attack space.

### 2.5.2 Implementation

We implemented the simple encryption/decryption scheme as an extension to GCC-3.0 (GNU Compiler Collection). The idea can be applied to any other compiler equally well.

GCC does not translate directly from high-level language into machine code but uses an intermediate representation called RTL (register transfer language). We patch GCC at the RTL abstraction layer. We change GCC source code to generate encryption and decryption RTL codes for function pointers. During the pass when GCC scans trees to generate RTL codes, we check function pointer type. If encryption or decryption is needed, we add codes to

generate RTL codes to implement encryption or decryption. In the following passes, GCC will generate machine codes for encryption and decryption according to the RTLs we generate.

There are several reasons that we chose RTL level to patch:

1. Easy to debug. GCC could generate a file that contains generated RTL codes, which makes it easy to check encryption and decryption codes we generate.
2. Easy to judge variable types. Tree representations in GCC contain a rich set of information about the variables in the program, which makes it easy to judge if encryption or decryption codes need to be generated.
3. Hard to implement in the passes after RTL generation. There is not much type information available at RTL level. Also after RTL generation, GCC has many passes to optimize the code. It is hard to implement our scheme at one of these passes.

### 2.5.3 Security testing

In this section, we describe the experimental setup used to validate the security robustness of the proposed. Consider *indirect.c* as an example. The original version presented in Figure 2.2 succeeded in overflowing the function pointer. However, after being compiled by our patched GCC, the program ends in segmentation fault. Let us take a look at *indirect.c*:

```

15  fp = foo;
16  buf[1] = (int)shellcode;
17  fp();

```

At Line 15, *fp* is assigned the value *foo*. This value is encrypted by the pointer key of *fp* and stored in the memory. At Line 17, when *fp* is used to make a function call, a decryption is performed before the function call. However, *buf[1]* overflows the attacker

code into *fp* at Line 16. Thus the decrypted value, generated by XORing *fp* with *fp* key, will not be the starting address of function *foo* or attacker's code shellcode. Hence the program ends in Segmentation fault.

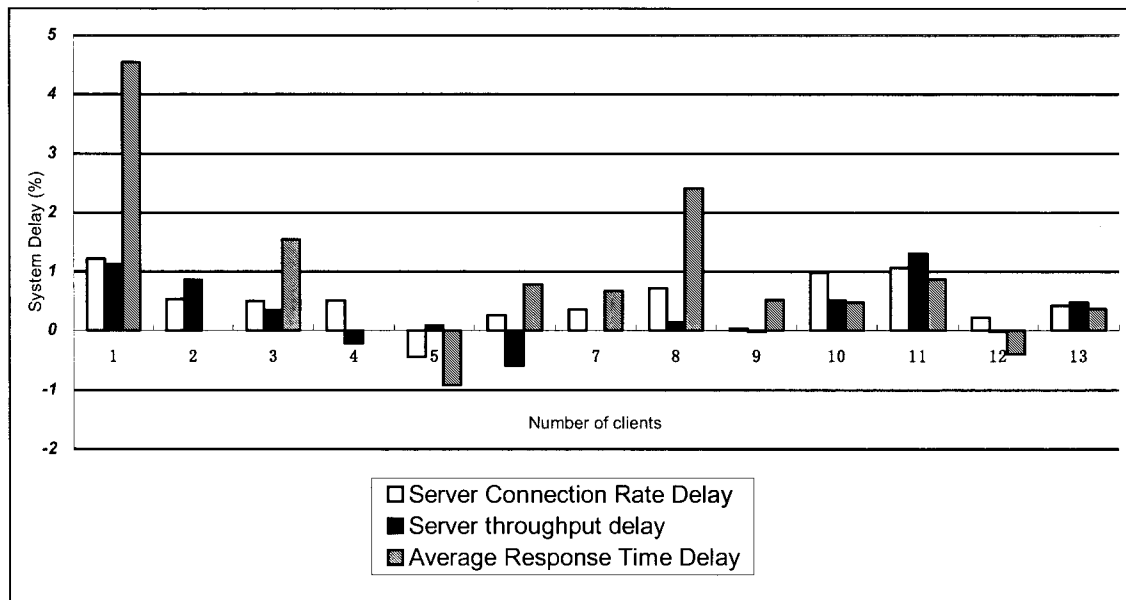
It will be very hard for an attacker to bypass our protection scheme. The reasons are:

1. The keys are generated dynamically at runtime. Even if the attacker reproduces the system on his/her own machine, the dynamic unique key generation forces him/her to guess  $2^{32}$  possible random masks in order to infer the key for each function pointer. Even if the attacker is able to scan the memory through format string attacks to read an encrypted value for a function pointer *fp*, they still need to know the un-encrypted value to be able to infer the key. Even if the key for *fp* is compromised, it does not help in attacking another function pointer. This orthogonality of all function pointer keys leads to a graceful, incremental compromise space; wherein each pointer has to be attacked separately. This makes the scheme very robust.
2. If the attacker tries a brute force attack, it is likely to be too expensive to be useful. The fact that the attacker cannot attack the system incrementally by retaining the partial key information from earlier attacks due to randomized keys makes it impractical to mount a brute force attack within the life span of a program. Besides, the program will generate enough errors to attract the attention of the administrator.

#### 2.5.4 Performance

Efficiency is another important consideration with respect to security schemes. If the performance overhead of a protection scheme is unacceptable, it may not be deployed despite its effectiveness at preventing attacks.

We need to assess the performance overhead of the proposed encryption/decryption scheme. We use Apache-1.3.22 and WebStone2.5 to quantify the scheme overhead. The current implementation of patched GCC-3.0 encrypts/decrypts all the function pointers or variables that eventually will hold a function pointer. The current implementation uses the function pointer address as the key. We are in the process of linker/loader modification to use the Boolean AND of the function pointer address and a unique random mask as the encryption/decryption key. Note, however, that the performance overhead of both schemes would be identical. Both the schemes use one exclusive-or instruction per encryption or decryption. The additional work pertaining to random mask generation and to an operation of the mask with the function pointer address is performed by the linker/loader. Hence it does not modify the execution time of the program directly. The Apache web server was compiled with our version of GCC to encrypt/decrypt function pointers. We use WebStone2.5 to test the web server to compare the results with normal Apache web server to assess and quantify the overhead.



**Figure 2.5 WebStone Performance**

The features that we choose to compare are server connection rate, server throughput, and average response time. From Figure 2.5, we can see that the scheme introduces an overhead of no more than 5 percent. The average appears to be about 2%.

Other benchmarks were also used to test our scheme, specifically SPEC 95 CPU benchmarks. As SPEC 95 CPU benchmarks are designed to measure the processor performance in computer architecture, they do not seem to use too many function pointers. Hence we did not observe any difference in the performance of these benchmarks when compiled with normal GCC versus when compiled with the patched GCC. Hence, we do not report those results.

## 2.6 RC5 encryption

The naïve XOR encryption/decryption is certainly not enough for security purpose. If the countermeasure manages to obtain part of the clear texts and encrypted texts, the key is possible to be reversed. In current literature, many advanced encryption techniques exist to prevent such attacks from happening. To apply one of them here is a natural choice.

We choose RC5 [Rivest, 1995] to encrypt/decrypt the function pointers. The main reason we choose RC5 is that to reach a certain level of security most of the encryption/decryption algorithms work on the block of data more than 64 bits. However, the function pointer is only 32 bits in normal Linux architecture. To pad the function to 32 bits and store the rest of the encrypted text into a link list is one solution for 64 bits encryption for function pointers. However, the solution does not sound straightforward and the link list itself will cost more time to build and maintain.

RC5 has been thoroughly researched on [Kaliski *et al.*, 1995], [Knudsen *et al.*, 1996], [Biryukov *et al.*, 1998], [Selcuk, 1998]. Although it has certain vulnerabilities, we make a tradeoff between the security and implementation here and choose RC5 as a 32 bits encryption algorithm for protection of function pointers.

### 2.6.1 RC5

RC5 is a fast block encryption/decryption algorithm designed by Rivest on 1995. In this section we give a brief introduction of RC5 from [Rivest, 1995].

Input:

1.  $2$   $\omega$ -bit registers  $A$  and  $B$
2.  $1$  key table  $S[0..t-1]$  consisting of  $t = 2(r+1)$   $\omega$ -bit words.

Encryption algorithm:

1.  $A = A + S[0]$
2.  $B = B + S[1]$
3. *for*  $i = 1$  *to*  $r$  *do*
4.  $A = ((A \oplus B) \ll B) + S[2 * i];$
5.  $B = ((B \oplus A) \ll A) + S[2 * i + 1];$

Decryption algorithm:

1. *for*  $i = r$  *downto*  $1$  *do*
2.  $B = ((B - S[2 * i + 1] \gg A) \oplus A)$
3.  $A = ((A - S[2 * i] \gg B) \oplus B)$
4.  $B = B - S[1]$
5.  $A = A - S[0]$

Notice that  $\ll / \gg$  are cyclic left/right rotations. Also, if the size of key table is larger than the user secret key, a routine exists to expand the user secret key into the key table. However, we will mention that we create random key table from the beginning so the expanding routine is not needed.

## 2.6.2 Implementation

To implement RC5 encryption/decryption for function pointers, we modify enhanced GCC-3.0 [Zhu *et al.*, 2004] of the simple function pointer protection. We implemented RC5-16/16/36 where:

1. Word size  $\omega = 16$
2. Number of rounds  $r = 16$
3. Number of bytes in the secret key  $b = 36$

The target is still the potential function pointer set. Every time a potential function pointer is used, the address will be sent to a monitor function for encryption/decryption.

The secret key is generated by random process with the seed derived from the present time and the current pid. Then the program will run a loop to generate 36 bytes secret key.

## 2.6.3 Result

We take the following simple program as one example:

```
#include <stdio.h>
static void foo(int *zzz(int));
static void foo1(register char);
int main(){
    int (*p)(int i);
    int (*q)(int i);
    int j;
    q = foo1;
    p = q;
    foo(p);
}
static void foo(int *zzz(int)) {
    printf("H%x \n", zzz);
    zzz(1);
}
static void foo1(register char in){
    printf("infoo1%c\n", in);
}
```

Compiled with a normal gcc, the program runs as follows:

```
H80483de
info1
```

80483de stands for the address of function *foo1*.

Compiling with a enhanced gcc with rc5 support, we have the following output during compilation:

```
In zPro | main | zOption = 1 | zInMonitorFunction = -1
Propagation count: 0
zPostModifyExpr | Encryption | For lhs | Static PFP Count = 1
zEXPAND_EXPR_Normal_Value | Static PFP Count = 2
zEXPAND_EXPR_Normal_Value | Static PFP Count = 3
zPostModifyExpr | Encryption | For lhs | Static PFP Count = 4
zEXPAND_EXPR_Normal_Value | Static PFP Count = 5
zEXPAND_EXPR_Normal_Value | Static PFP Count = 6
In zPro | foo | zOption = 1 | zInMonitorFunction = -1
Propagation count: 0
zInitialize | Eecrypt PARM_DECL, zzz | Static PFP Count = 7
zEXPAND_EXPR_Normal_Value | Static PFP Count = 8
zEXPAND_EXPR_Normal_Value | Static PFP Count = 9
zEXPAND_EXPR_Normal_Value | Static PFP Count = 10
zEXPAND_EXPR_Normal_Value | Static PFP Count = 11
In zPro | foo1 | zOption = 1 | zInMonitorFunction = -1
Propagation count: 0
```

Notice that the static PFP count reaches 11 which mean that totally 11 encryption/decryption actions are inserted into the program. There exists no propagation in the program such that the propagation count is 0.

The program runs and gives the following result:

```
The key: 4e9e2a87 64e24e85 1a473c22 65f2112a 4f95a04b 101cc1ce 4ecbc5c4
78037497 7a0d0344 3164a179 2b43c6dc 40d97470 2c52f3a9 1beacd41 547f2d35
7d95f765 14205be5
1, (main, q, 10, Encryption)
zPostModifyExpr | Encryption | For lhs
```

*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
 2, (main, q, 11, Decryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Decryption: Old = 1ddf40bf*  
*New = 0804885b*  
 3, (main, q, 11, Encryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
 4, (main, p, 11, Encryption)  
*zPostModifyExpr | Encryption | For lhs*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
 5, (main, p, 12, Decryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Decryption: Old = 1ddf40bf*  
*New = 0804885b*  
 6, (main, p, 12, Encryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
 7, (foo, \*, 17, Encryption)  
*zInitialize | Eecrypt PARM\_DECL, zzz*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
 8, (foo, \*, 17, Decryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Decryption: Old = 1ddf40bf*  
*New = 0804885b*  
 9, (foo, \*, 17, Encryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
*H804885b*  
 10, (foo, \*, 18, Decryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Decryption: Old = 1ddf40bf*  
*New = 0804885b*  
 11, (foo, \*, 18, Encryption)  
*zEXPAND\_EXPR\_Normal\_Value*  
*Encryption: Old = 0804885b*  
*New = 1ddf40bf*  
*infool*

## 2.6.4 Performance

Due to the complexity of GCC compiler and limitation of the time to submit this thesis, we do not compile apache web server using RC5 enhanced GCC compiler. However, we provide an estimated system overhead here. We run the RC5 encryption and XOR encryption program with Unix shell command ‘time’ and get the result that RC5 encryption will bring about 20 times overhead against XOR encryption.

We use the following formula to estimate the overhead for RC5 encryption:

$$Overhead_{RC5} = \frac{20 \times Overhead_{XOR}}{20 \times Overhead_{XOR} + (100 - Overhead_{XOR})} \times 100$$

With this formula, we use the experimental data from Figure 2.5 to calculate the estimated overhead of RC5. From Table 2.1 we can see that the estimated overhead brought by RC5 protection of function pointers is no more than 15%. RC5 protection brings more system overhead than XOR protection. However, 15% is still acceptable for a complete protection of function pointers by RC5 encryption.

Table 2.1 RC5 Encryption Overhead Estimation

Test	Server Connection Rate Delay	Server Throughput Delay	Average Response Time Delay
1	1.218329084	1.125592417	4.545454545
2	0.537125498	0.860730916	0
3	0.500098273	0.347753512	1.538461538
4	0.518699329	-0.206810975	0
5	-0.440874765	0.095264017	-0.917431193
6	0.262061262	-0.589121798	0.78125
7	0.36310358	0	0.675675676
8	0.723162404	0.148228002	2.409638554
9	0.038009164	-0.013491635	0.529100529
10	0.98366157	0.50993022	0.4784689
11	1.068948205	1.300424628	0.873362445
12	0.223163076	-0.013399437	-0.401606426
13	0.422640259	0.480128034	0.373134328
XOR Average	0.493702072	0.311171377	0.837346838
RC5 Average	9.027254351	5.876022063	14.4482754

## 2.7 Conclusion and future work

We describe a scheme to protect against buffer overflow attacks on function pointers. The scheme encrypts any variable that is either a declared function pointer or can be assigned a function pointer value through casting before storing it in memory. Similarly, each use of such a variable is decrypted. The encryption and decryption instructions are inserted by the compiler (GCC in our case). The implemented encryption/decryption scheme is very low cost exclusive-or instruction. The function pointer value is encrypted with a key.

For simple scheme, the keys are designed to be robust in many ways. First, each function pointer has its own unique key. This isolates the damage through compromise of a given function pointer key only to that function pointer. Second, a different, random set of keys are chosen for each program run.

For complex scheme, we implement RC5 encryption protection of function pointers with dynamic key for each run.

The dynamic key limits the damage from a key compromise in a given program run only to that run. Any compromised key values do not help the adversary in any of the future program runs.

In summary, our scheme defends against buffer overflow attacks via pointers. The protection targets pointers instead of stack resident return addresses. We protect all pointers regardless of their location (stack, heap or global static data space) from buffer overflow attacks. The proposed scheme has a performance overload of at most 5% (average of 2%) for simple scheme. Since each pointer key is unique and is randomized for each run through an augmented loader/linker, the attacker needs to be able to mount a  $2^{32}$  iteration attack on each key within a single run of the program, which appears to be extremely infeasible. The future works include the following.

### **2.8.1 Hidden key**

The current scheme uses the addresses related to the program as encryption/decryption key. This could be vulnerable if the hacker is able to see those addresses in the memory during run time.

A hidden system key may be the solution. Design of a system key that is invisible to the software (perhaps architecturally invisible) is a challenging task. This type of program transparent support for a key might have to be a complete micro-architectural scheme. The function calls could be encrypted and decrypted with that hidden key but the attacker does not see the key, since it is not in the memory.

### **2.8.2 Dynamic linking**

The current scheme assumes that all the files with protected function pointers are compiled with the patched GCC and are linked statically. However, dynamic linking creates new challenges, since which of the variables in a linked program are PC-bound may not be determinable until after linking. Consider a procedure where one of the actual parameters is cast as a function pointer. The calling program will never tag any computations leading up to that actual parameter as PC-bound. Similarly, if a function pointer is cast as an integer or other non-pointer value, and then passed to a procedure as an actual parameter, the called procedure does not have any way of knowing it until after linking. Linker-based techniques could be developed to account for these cases.

## CHAPTER 3. CONTROL FLOW CHECKING AUTOMATA

In Chapter 2, we introduce a compiler patch to protect function pointers against overflow attacks at runtime. In this chapter, we take a different path to fight against overflow attacks: program level trust through control flow checking automata. The notion of trust has traditionally been deployed at transaction level in order to bypass expensive security checks. We extend the trust model to individual programs. Moreover, we develop a self assessment/monitoring framework for trust based on control flow integrity that can be incorporated into a compiler. We also extend the concept of Schneider's enforceable security policy into that of an enforceable trust policy. This trust assessment model has been implemented with SUIF and GCC compilers. An architectural modification to support efficient management of control flow integrity based trust model has also been developed. The trust value indicates how well the runtime program behavior match the control flow graph derived from source codes. The overflow attacks within codes are detected with low threshold of trust level.

### 3.1 Introduction

Trust has become a central tenet in the world of network driven computing. There are many definitions of trust highlighting different aspects of a trust model and management [Abdul-Rahman *et al.*, 1997] [Chen *et al.*, 2000] [Khare *et al.*, 1997] [Blaze *et al.*, 1996] [Blaze *et al.*, 1999]. Some strive to build a model of authenticated accesses [Li *et al.*, 2003]. Some develop a model of trust propagation such as the Trusted Third Party model. The Microsoft's trusted zone model to bypass the sandboxing for the JAVA virtual machine is yet another example of trust where user is asked to specify the trust value. One of the shortcomings in these trust frameworks is that the “trust value” is static with respect to evolving security attacks on the constituent nodes. In other words, the trust value is derived solely from the history of the node behavior within the context of the transaction under

consideration. However, the other aspects of the program behavior relating to that node can provide valuable information about its trustworthiness. For instance, if a trusted certificate server has been compromised, its trustworthiness should decline. Even if there is only an indication of a footprint of an attack, it ought to play a role in determining its trustworthiness. There is no current mechanism to incorporate such dynamic information into the trusted transactions. The trust in such a server may decline over time due to a projection of its compromised behavior at the transaction level, but that could be a long latency feedback loop.

In this chapter, we describe a technique to generate a dynamic trust level from the runtime behavior of a program to reflect how well the program runtime states comply with a pre-specified trust policy. The trust policy could initialize the trust level to 1, which will be lowered if the program behavior deviates from the expected (trusted) behavior.

We take the viewpoint that trust is meaningful only when specified within the context of a security policy. If a certain security policy is routinely respected (such as always authenticating before a critical resource allocation step), our trust with respect to that policy ought to be high. Such a high trust value permits us to eschew explicit verification of this security policy each and every time. For each security policy, there is a corresponding trust value. Hence trust is a multidimensional attribute with each security policy corresponding to a dimension. Although, our discussion so far has related trust with a security policy, the concept really holds with respect to any program level correctness policy. For instance, the compliance of a program to a policy that states that each object should be freed after its last use can be captured with a trust value corresponding to this policy. We present a formal definition of trust policy, trust automata and a primitive implementation of program level trust in this chapter.

Schneider [Schneider, 2000] defines a security policy as a set of unacceptable executions with respect to the runtime behavior of a program. He also defines a class of security policies named “Execution Monitoring”, which are the security policies that could

be enforced by execution monitoring. Schneider uses security automata, which are non-deterministic finite-state automata, to describe security policies. When the program execution behavior is not accepted by the security automata, the program is terminated.

Instead of terminating the program, we are more interested in assessing the program execution state with respect to its trustworthiness. Towards this end, we introduce the notion of a trust policy. A trust policy describes the acceptable actions of the program execution. It also associates a trust function with each transition/action. Note that typically we monitor only a small subset of actions within a program. Those actions are the only ones that constitute the input to such a trust policy. The trust policy is defined as follows:

**Definition of trust policy:** Trust policy is specified by a predicate on sets of executions and a trust function associated with program actions.

A runtime monitor will compare program runtime behavior against a trust policy and execute the corresponding updates to the trust value, which represents the trust level of the program. This program level trust could serve many roles. It could be supplied to an upper level control process to formulate a more refined response (a low trust value could lead to the invocation of an intrusion detection system). Alternately, the program level trust values could constitute the primary input into a transaction level trust management framework for traditional trust management.

The program level trust is a dynamic, runtime value to capture the degree of compliance of the program execution with the trust policy. We argue that the program level trust should have the following features to reflect the trust level of a single program:

1. The process to generate program level trust cannot be compromised. Else the trust value itself is not trustworthy.

2. The program level trust should be updated according to the trust policy and runtime state. When the program is compromised, the trust level should be lower than a pre-specified threshold. Whenever the trust level is lower than this threshold, an alert can be issued to the system administrator.
3. The computational overhead of trust value maintenance should be tolerable.

There are many ways to implement a trust monitoring system. The two interesting points are when the monitoring process is implemented as a separate process from the monitored program or when it is embedded into the monitored program. We must guarantee the three features of program level trust mentioned earlier regardless of the implementation choice, however, to ensure the robustness of the trust results. In this chapter, we present a primitive implementation which embeds the trust automata and corresponding monitor into the program. Moreover, the trust policies on an arbitrary program state are translated into trust policies on the program control flow graph.

We discuss trust automata in Section 3.2. Embedded implementation of program level trust will be presented in Section 3.3. We outline related work in Section 3.4 and future work in Section 3.5 respectively. We conclude in Section 3.6.

## 3.2 Trust automata

Schneider [Schneider, 2000] uses “Buchi automata”, which basically has the power of deterministic finite-state automata, to express security policies.

Deterministic finite automata are simple and powerful enough for many security policies, like access control. However, a trust policy to describe acceptable program behaviors can either focus on the space of program actions such as access to shared resources or could be based on the low level program execution attributes (such as return address integrity). A deterministic finite automaton is not powerful enough to capture the execution attributes. Take the trust policies that are related to runtime stack as one example:

**Trust policy of inter-procedural control flow graph:** This trust policy states that only the program specified procedure call and return events can be instantiated at the runtime.

Program function calls consistent with the control flow graph will increase the trust level of the program. Program function calls not from the control flow graph will decrease the trust level.

This trust policy generates a trust value that reflects how well the program runtime behavior complies with the inter-procedural control flow graph. In order to observe and track the program control flow, and to compare against control flow graph generated from the program source code, the trust monitor needs access to a function call stack to track all the function calls and returns. Deterministic finite automata are not suitable for this task.

The trust automata can come in two flavors based on the computational complexity of the policy. A DFA like trust automata can verify trust policies based on security policies verifiable with Schneider's security automata [Schneider, 2000]. In general such policies specify relative sequencing of two events: An event of type P must occur before an event of type Q can occur. A more general version can be a finite precedence graph over a class of events which are easily formulated as a DFA. We choose to present pushdown automata as the basic trust automata due to their expressive power. The reader should keep in mind though that should the security policy be simple enough to be captured by a DFA, we intend the trust automata implementation to be a DFA from efficiency considerations. Pushdown automata are powerful enough to describe program runtime behavior. Yet pushdown automata are simple enough to be verified formally.

Notice that pushdown automata which use infinitely many stacks are not practical for real computer programs. So we restrict the trust automata to a subset of pushdown automata: deterministic pushdown automata.

$P = (t, Q, \Sigma, \Gamma, \delta, f, q_0, Z_0, F)$ , where

$t$ : The trust variable in range  $[0,1]$ , with 1 standing for fully trusted and 0 for totally distrusted.

$Q$ : A finite set of states;

$\Sigma$ : A finite set of input symbols;

$\Gamma$ : A finite stack alphabet;

$\delta$ : The transition function,  $\delta(q, a, X)$ , where  $q \in Q$ ,  $a \in \{\varepsilon\} \cup \Sigma$ ,  $X \in \Gamma$ ; The output of  $\delta$  is a finite set of pairs  $(p, \gamma, f)$  where  $p$  is the new state,  $\gamma$  is the string of stack symbols that replaces  $X$  at the top of the stack,

$f : [0,1] \rightarrow [0,1]$ . For each transition, the automata will update  $t \leftarrow f(t)$ .

$q_0$ : The start state.

$Z_0$ : The start symbol.

$F$ : The set of accepting states.

Also, the trust automata satisfy the following conditions in order to be deterministic pushdown automata:

1.  $\delta(q, a, X)$  has at most one member for any  $q$  in  $Q$ ,  $a$  in  $\Sigma$  or  $a = \varepsilon$ , and  $X$  in  $\Gamma$ .
2. If  $\delta(q, a, X)$  is nonempty, for some  $a$  in  $\Sigma$ , then  $\delta(q, \varepsilon, X)$  must be empty.

When the program runs and the program states trigger the input to the trust automaton, which resides in a monitor or embedded code, the trust value of the program will be updated.

When the trust value of the program is too low, an alert will be issued to raise attention and some of the undesirable program behaviors might be suppressed. A simple alert raising mechanism is to raise a specific exception.

### 3.3 Embedded framework

As we mentioned earlier, there are two possible implementations for program level trust assessment and monitoring frameworks. In this section, we present a primitive embedded monitor framework for program level trust. As trust level should be lowered if the program behavior deviates from the trusted behavior defined by trust policy, a clear expectation as to the power of the adversary is desirable. If the adversary has the full control of the program, the trust process cannot generate a trustable program level trust. We assume that the adversary could access the computer from network and may have the ability to overflow buffers in the global data and stack portions of memory but not within the source code (text) area of memory.

As we implement the current framework with embedded codes, we assume that the program control flow change will happen within the source codes. If an attacker manages to jump outside the codes of the current execution, the trust automata will not execute and the trust value is invalid. We demonstrate one example in Section 3.3.3.

#### 3.3.1 Trust policy of control flow graph

**Trust policy of control flow graph:** The program control flow actions (entering and leaving basic block) consistent with the statically derived control flow graph will increase the trust level of the program. All the other actions will decrease the trust level of the program.

This trust policy arises in the traditional fault tolerance community [Oh *et al.*, 2002]. If a processor or system level hardware fault tampers the program behavior causing unexpected control flow to be instantiated, such a policy will lower our trust in those faulty executions. This may be of value to safety-critical applications such as flight control.

Trust policy of control flow graph has a close relationship with the control flow graph generated from the program source code.

A control flow graph is defined as  $G = (V, E)$ , where  $V = \{v_1, \dots, v_n\}$  is a finite set of nodes and  $E \subset V \times V$  is a set of directed edges between nodes.

We assume that  $G$  is a complete control flow graph which not only represents the intra-procedural control flow but also inter-procedural control flow. A call to another function corresponds to an edge from the block of call instruction in the caller procedure to the first block of the callee procedure, and another edge from the exit block of callee procedure to the successor block of call instruction in the caller procedure.

We define the trust automata of control flow graph as:

$P = (t, Q, \Sigma, \Gamma, \delta, f, q_0, Z_0, F)$ , where

$t: [0, 1]$

$Q: V$

$\Sigma: \{ En(v) \mid v \in V, En(v) \text{ represents program action of entering block } v \}$

$\Gamma: NULL$

$\delta: \{ (v_1, En(v_2)) \rightarrow v_2 \mid (v_1, v_2) \in E \}$

$f: t \leftarrow t, \text{ when } (v_1, En(v_2)) \rightarrow v_2 \in \delta; t \leftarrow 0.9 * t, \text{ otherwise.}$

$q_0: En[v_0]$  where  $v_0$  is the first block in *main()*.

$Z_0$ : The start symbol.

$F$ : Final states

Now we will discuss the detailed implementation of this trust automaton in embedded framework.

### 3.3.1.1 Implementation

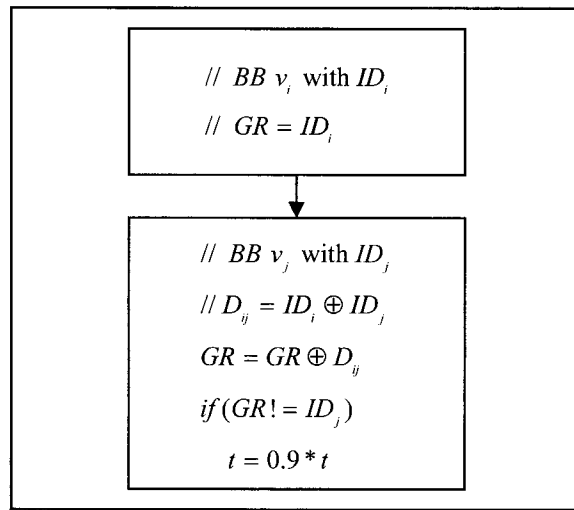
Control flow checking in essence verifies at run time that a monitored control flow edge is indeed instantiated as intended (from its source basic block to its target basic block). Such control flow checking applied to each control flow edge has been used for processor fault tolerance [Oh *et al.*, 2002]. We, however, recognize that security automata [Schneider,

2000] and edit automata [Ligatti *et al.*, 2003] can also be formulated as control flow path monitoring paradigms. This allows us to treat basic program structure violations such as buffer overflow with the same control flow integrity verification technique as a transaction level property. We first outline the control flow signature framework.

*Property (1) :*

Assume  $parent(v_i) = \{v_j \mid v_j \in V \wedge (v_j, v_i) \in E\}$ ,

then  $\forall v_a, v_b \in V (parent(v_a) \cap parent(v_b) = \emptyset)$



**Figure 3.1 Basic Block with One Parent**

[Oh *et al.*, 2002] proposed to use control flow signatures for fault tolerance in a processor. Each basic block  $i$  in the program have a unique  $ID_i$  associated with it. There is a global signature register GR. The invariant is that after the initial signature book-keeping at entry into a basic block, the global signature register GR should contain the ID of this basic block  $ID_i$ .

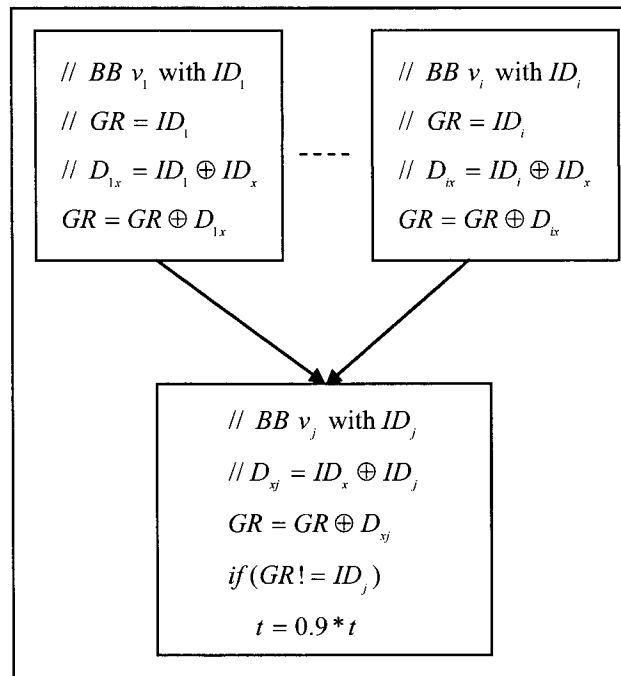


Figure 3.2 Basic Block with Multiple Parents

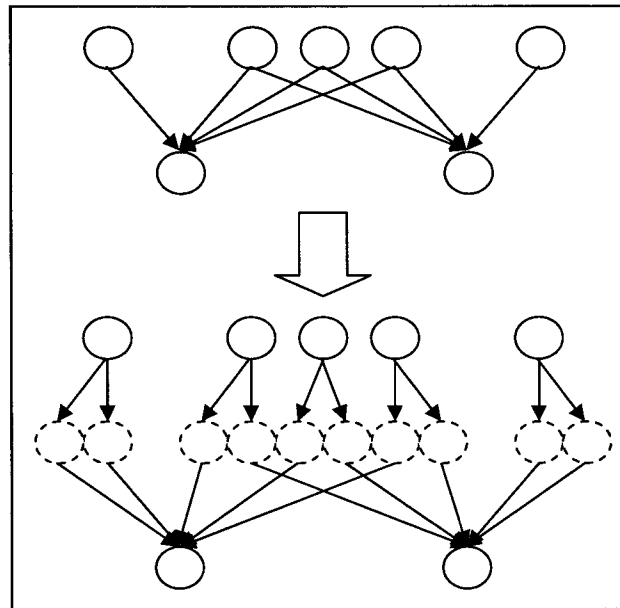


Figure 3.3 CFG Modification

The basic algorithms for signature verification are presented in Figure 3.1 and Figure 3.2. The  $D_{ij}$  and  $D_{xj}$  are embedded into source codes by compiler. When entering *main*, the *GR* is initialized to be  $ID_{v_0}$ . Notice that in the multiple parents' situation,  $ID_x$  comes from  $v_x$  which is a randomly chosen parent from the parent set of  $v_j$ . However, when a block has multiple parents, and those parents have multiple children, we cannot fix uniquely the randomly chosen parent as a distinguished node because one parent set can have only one chosen parent. If a node belongs to two parent sets and they have different distinguished chosen parents, a conflict will arise when it computes. That is the reason we enforce the following restrictions on the control flow graph.

For a trust automaton of control flow graph to be embeddable into program source code, the control flow graph has to satisfy:

Modifications to control flow graph are necessary if the original control flow graph does not satisfy Property (1). As shown in Figure 3.3, when the control flow graph does not satisfy Property (1), we will insert one extra level of dummy nodes to erase the conflict.

With the modification technique of Figure 3.3, we are able to turn a security automaton into an equivalent automaton that could be embedded into program.

### 3.3.1.2 Experimental results

The C compiler gcc-3.3 has been modified to embed CFC automata. During compile time, we add passes to:

1. Generate unique ID for each basic block.
2. Collect information on inter-procedure calls.
3. Embed control flow checking codes into the program.

During runtime, the program will:

1. Initialize global registers for checking algorithm.
2. Initialize the ID set for inter-procedure calls.

3. Check the control flow at the beginning of each block.

### Overhead

The control flow checking algorithm will increase the program size and run time dramatically. We have compiled two of the SPEC2000 benchmark programs: gzip and mcf to evaluate the dynamic run time overhead. The data is presented in Tables 3.1, 3.2. Notice:

1. Block number and instruction number in the table 1 show the increased program size by using patched gcc-3.3.
2. The run 1 in table 3.2 is when the program compiled by un-patched gcc-3.3 and run 2 in table 3.2 with program compiled by patched gcc-3.3 to generate full control flow checking.

It is expected that the system overhead for complete control flow checking will be large since each control flow edge is checked. Note though that this is an upper bound on the trust monitoring overhead. Almost any other realistic trust policy will monitor a much smaller subset of control flow edges. Appropriate hardware support, such as the one proposed by [Zhang *et al.*, 2004], is necessary for bringing down the overhead to a reasonable value.

Note that for each security automaton, one set of signature space needs to be maintained. Not every control edge needs to be verified, however, as was the case for the trust policy of control flow checking. The performance overhead can be significantly reduced if however the architecture manages the trust attributes.

Table 3.1 SPEC 2000 Benchmarks static count

Program	Old Block	New Block	Old Insn	New Insn
gzip	1730	3945	17429	73047
mcf	395	962	4565	17397

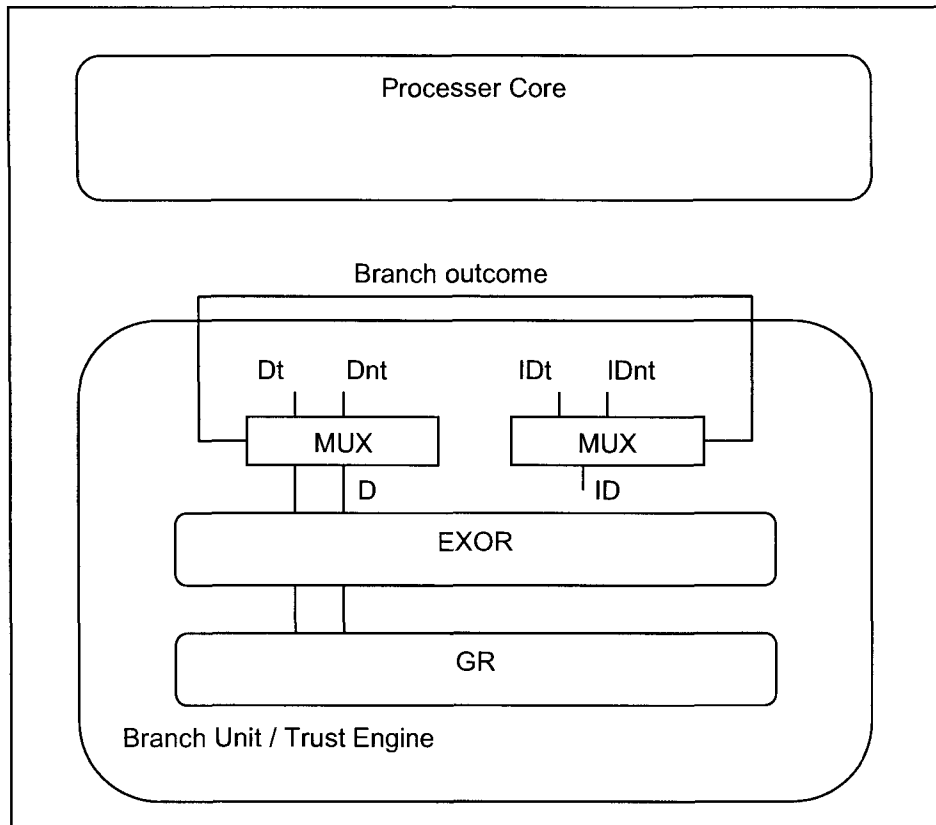
Table 3.2 SPEC 2000 Benchmarks runtime overhead

Program	Ref. Time	Run 1	Run 2	Ration
gzip	1400	202	11969	59.2
mcf	1800	477	1611	3

### Architecture Level Signature Checking

One simple way to accomplish this would be to associate extra attributes with branch instructions. This is under the assumption that each program object that triggers trust evaluation will be a multiple of basic blocks, which appears to be a reasonable assumption. Most trust policies are defined at the granularity of procedures or larger objects. A special branch instruction in addition to performing the branching operation would also specify the difference value  $D$ , and the entry basic block's unique ID. Note that the hardware trust engine maintains a current global signature  $GR$ . The  $D$  part of the branch instruction will be EXORed with  $GR$  to give  $GR = GR \oplus DR$  for a taken branch. The hardwired comparison of the resulting  $GR$  with respect to the basic block ID embedded in the branch instruction can also be efficient. If the check fails, this hardware trust unit can raise an exception.

Consider Figure 3.4 which shows a schematic for the proposed architecture. The modified branch instructions have the form: *beqs*  $R0, R1, target, D_t, ID_t, D_{nt}, ID_{nt}$ . This is a signature checking equivalent of the traditional “branch on equal” (*beq*) of MIPS like instruction set. It branches to the basic block at address given by  $PC + target$  for a taken branch, and to  $PC + 4$  for a not taken branch. In addition to the arguments of the traditional branch, it also specifies the difference value and the ID of the block on a taken branch  $(D_t, ID_t)$  and of the block for a not taken branch  $(D_{nt}, ID_{nt})$ .



**Figure 3.4 Trust Engine Support Architecture**

Note that these values are known to the compiler at compile time, and hence it is feasible to generate instructions embedding these values. One multiplexer selects one of the difference values  $D_t$  or  $D_{nt}$  on the basis of the branch direction. Another one selects the ID from  $ID_t$  and  $ID_{nt}$ . The  $D$  value is *EXORed* with the existing global register value  $GR$ , which replaces the current value of  $GR$ . The multiplexed  $ID$  value is compared against this  $GR$  value. If they are not equal, an exception is raised.

We plan to implement such a trust engine as part of SimpleScalar [Burger *et al.*, 1996] simulator.

### 3.3.2 Electronic commerce examples

Trust policy could come from security policy, though security policy defines the executions that are unacceptable. Trust policy defines trust function for both acceptable and

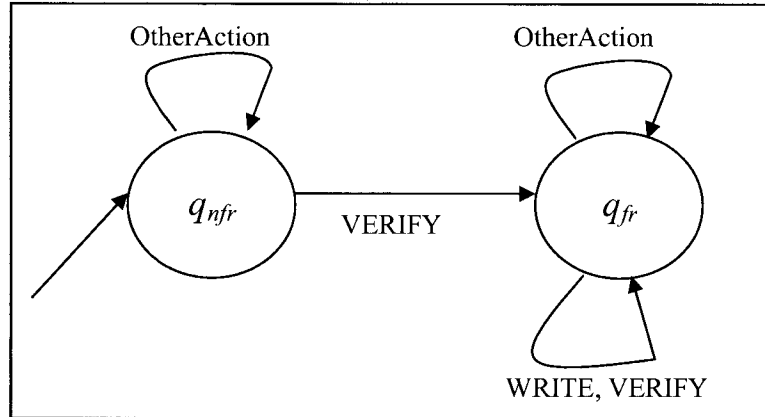
unacceptable actions in order to determine the final program level trust. It is pretty simple to turn a security automaton into a trust automaton.

Note that in order for the trust automata to be embeddable into program source code, they have to satisfy Property (1). The transformation in Figure 3.3 could be used to modify trust automata into embeddable automata.

In this section, we take one simple example and one example from Schneider's security automata [Schneider, 2000] to demonstrate the conversion of normal security automata into trust automata. Note that these examples use DFA like structures for the policy specification instead of push-down automata like structures. In practice, most of the examples in the literature seem to require only a DFA level of expressiveness. We defined trust automata to be PDA like for more expressiveness.

### Example 1

Consider a security policy that the program can not write to the critical data unless it has executed verification function. The security automaton is shown in Figure 3.5.



**Figure 3.5 Critical Data Access after Verification**

We could transform this security policy into a trust policy as follows.

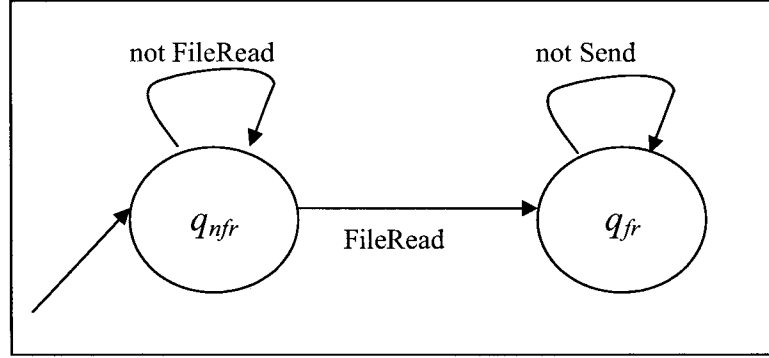
$P = (t, Q, \Sigma, \Gamma, \delta, f, q_0, Z_0, F)$ , where  
 $t : [0,1]$   
 $Q : \{q_0, q_1\}$   
 $\Sigma : \{ \text{VERIFY}, \text{WRITE}, \text{OtherAction} \}$   
 $\Gamma : \text{NULL}$   
 $\delta : \{ (q_0, \text{OtherAction}) \rightarrow q_0, (q_0, \text{VERIFY}) \rightarrow q_1, \\ (q_1, \text{OtherAction}) \rightarrow q_1, (q_1, \text{VERIFY}) \rightarrow q_1, \\ (q_1, \text{WRITE}) \rightarrow q_1 \}$   
 $f : t \leftarrow t$ , for transitions in  $\delta$ ;  
 $t \leftarrow 0.1 * t$ , otherwise.  
 $q_0 : q_0$ .  
 $Z_0$  : The start symbol.  
 $F$  : Final states

If the trust policy is violated, the trust level of the program will drop dramatically. Each violation decreases the trust value to one tenth of its original value. A low enough trust value will alert the administrator or manager program for further action. Also note that these examples start with an initial trust value of 1. This is an appropriate initial state when the program was written by a trusted source, and the host node is safe except for external attacks. In other words, external attacks or programming bugs are the only causes that will lower the trustworthiness of these programs over time. On the other hand, if the program under consideration is a distributed transaction, or if it is acquired from a non-trustworthy source (such as a downloaded Java applet), an initial trust value of 0 will be more appropriate.

## Example 2

The security automaton in Figure 3.6 comes from Schneider's Execution Monitoring paper [Schneider, 2000]. It specifies that there can be no send action after a file read action has been performed. This security policy could prevent information leakage.

When the program starts, the automaton enters state  $q_{nfr}$ . If there is a *FileRead* action, the automaton enters state  $q_{fr}$  and prevents further Send actions.



**Figure 3.6 No Send after FileRead**

We turn this security automaton into a trust automaton as follows.

$P = (t, Q, \Sigma, \Gamma, \delta, f, q_0, Z_0, F)$ , where

$t : [0,1]$

$Q : \{q_0, q_1\}$

$\Sigma : \{FileRead, Send, OtherAction\}$

$\Gamma : NULL$

$\delta : \{ (q_0, OtherAction) \rightarrow q_0, (q_0, FileRead) \rightarrow q_1, \\ (q_0, Send) \rightarrow q_0, (q_1, OtherAction) \rightarrow q_1, \\ (q_1, FileRead) \rightarrow q_1 \}$

$f : t \leftarrow t$ , for transitions in  $\delta$ ;

$t \leftarrow 0.1 * t$ , otherwise.

$q_0 : q_0$ .

$Z_0$  : The start symbol.

$F$  : Final states

In this example as well, if the trust policy is violated, the trust level of the program drops to one tenth its original value. A low enough trust value will alert the administrator or manager program for further action.

### 3.3.3 Program control flow violation examples

In this section we present a simple example of the program control flow violations. The program control flow violation within the program could happen when the function pointer is overflowed. Another case is that a user might modify the program source codes to bypass certain verification checks.

Take a look into the following program:

```

01  #include <stdio.h>
02  void foo(int in){
03      printf("foo: ==> %d\n", in);
04  }
05  void bar(int in){
06      if (in > 2) {
07          printf("koo: ==> in > 2\n");
08      } else {
09          printf("koo: ==> in <= 2\n");
10          goto end;
11      }
12      printf("koo: ==> After comparing.\n");
13  end:  printf("koo: ==> End.\n");
14  }
15  int main(int argc, char **argv){
16      int a;
17      a = argc * 10;
18      if ( a > 5 ) {
19          bar(argc);
20      } else {
21          foo(argc);
22      }
23      printf("main ==> Done.\n");
24  }

```

The running result of our embedded implementation looks like:

In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
-1	0	0	2044897763	-1	0	NULL	main	1
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	4	1967513926	212289701	0	NULL	main	2
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	1	1	1365180540	605741370	0	bar	main	3
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
-1	0	0	1957747793	-1	0	NULL	bar	4
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	5	424238335	1845067950	0	NULL	bar	5
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	1	2	596516649	986137558	0	printf	bar	6
koo: ==> in <= 2								
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
1	0	2	596516649	986137558	0	printf	bar	7
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	8	1189641421	1701209060	2079532151	NULL	bar	8
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	1	4	1350490027	1835372305	0	printf	bar	9
koo: ==> End.								
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
1	0	4	1350490027	1835372305	0	printf	bar	10
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	9	783368690	2127533657	0	NULL	bar	11
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	-1	6	1102520059	1862688521	-1	NULL	bar	12
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
1	0	1	1365180540	605741370	0	bar	main	13
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	6	1540383426	177169086	1240492694	NULL	main	14
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	1	3	1303455736	1603350444	0	printf	main	15
main ==> Done.								
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
1	0	3	1303455736	1603350444	0	printf	main	16
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	7	35005211	1336345827	0	NULL	main	17
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	-1	5	521595368	486591219	-1	NULL	main	18

Congratulations for passing all CFG tests!

If we change the control flow within the current program at line 10 to jump to line 12 instead of 13, the embedded trust automata will generate the following warning:

In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
-1	0	0	2044897763	-1	0	NULL	main	1
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	0	4	1967513926	212289701	0	NULL	main	2
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
0	1	1	1365180540	605741370	0	bar	main	3
In	Out	Index	zID	zd	zD	zFuncName	zCurFuncName	zRuntimeCount
-1	0	0	1957747793	-1	0	NULL	bar	4

```

In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 0 5      424238335      1845067950      0      NULL      bar      5
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 1 2      596516649      986137558      0      printf      bar      6
koo: ==> in <= 2
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
1 0 2      596516649      986137558      0      printf      bar      7
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 0 8      1189641421      1701209060      2079532151      NULL      bar      8
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 1 3      1025202362      1598947414      0      printf      bar      9
zError: zCurID=1649760492, zID=1025202362, zTrust=0.900000. Normal check fails.
koo: ==> After comparing.
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
1 0 3      1025202362      1598947414      0      printf      bar      10
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 1 4      1350490027      1835372305      0      printf      bar      11
koo: ==> End.
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
1 0 4      1350490027      1835372305      0      printf      bar      12
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 0 9      783368690      2127533657      0      NULL      bar      13
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 -1 6      1102520059      1862688521      -1      NULL      bar      14
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
1 0 1      1365180540      605741370      0      bar      main      15
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 0 6      1540383426      177169086      1240492694      NULL      main      16
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 1 3      1303455736      1603350444      0      printf      main      17
main ==> Done.
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
1 0 3      1303455736      1603350444      0      printf      main      18
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 0 7      35005211      1336345827      0      NULL      main      19
In Out Index      zID      zd      zD      zFuncName  zCurFuncName  zRuntimeCount
0 -1 5      521595368      486591219      -1      NULL      main      20
Warning : The trust value is lower than 1!

```

### 3.4 Support for traditional trust management

Blaze et al introduce the term “trust management” in their classical paper [Blaze *et al.*, 1999].

The program level trust could be a fundamental supplement for the study of trust management. Traditional trust management mainly focuses on the trust propagation [Blaze *et al.*, 1999]. The trust levels remain static once the trust relationships have been built, and the network structure remains the same. The program level trust, on the other hand, dynamically

generates the trust estimate for the runtime processes. The leaf level predicates within propositions capturing the system level trust are often the trustworthiness attributes of a program. If the dynamic trust value estimates arising from our implementations of program level trust are continually accounted for in the transaction level trust engine, we could have a more correct dynamic trust estimation framework at system and/or transaction level. Note that system level trust engine could use any algorithm for trust propagation (not necessarily logic based). The main point to note is that a dynamic program level trust will be reflected in a dynamic system level trust relationship if periodic re-evaluations are undertaken. In fact, a system or transaction level trust automata can capture many of the commonly deployed system level trust management algorithms.

We illustrate a real example of combining various program level trust values with a specific trust propagation algorithm from Josang [Josang, 1999]. This demonstrates the power of our model in formal specification and propagation of system level trust. Here certain criteria are explained thus eventually leading to a foregone conclusion.

### 3.4.1 Trust algebra

Josang [Josang, 1999] proposes a formal algebra for assessing trust in certification chain. The trust is represented by a symbol

$$\omega_p^A = \{b_p^A, d_p^A, u_p^A\},$$

where

$\omega$  is trust of  $A$ 's belief about  $p$ ,

$A$  is an agent,

$p$  is a statement,

$b$ ,  $p$ ,  $u$  represent the belief, disbelief and uncertainty components, and  $b + d + u = 1$ .

Two opinions are also defined in the paper, which are the key authenticity of the key used to certify and recommendation trustworthiness of how much one entity trusts another entity into recommending and certifying other keys. Those two opinions will be generated at the start of the network and remain stable for the rest of the time. We want to introduce another opinion about the system level trust, which is derived from the composition of the relevant program level trust values.

The system level trust is a composition of program level trusts of the processes running on  $A_2$  which have relationships to the correct functionality of the network or authentication. The system level trust is a dynamic state representing the healthy state of the system. If part of the system is attacked and the program level trust of some critical process is lowered, the system level trust will be lowered correspondingly. The system level trust will be queried frequently between the entities. One possible update function could be as follows.

$$\omega_{ST(A_2)}^A = \{b_{ST(A_2)}^A, d_{ST(A_2)}^A, u_{ST(A_2)}^A\}$$

where

$$b_{ST(A_2)}^A = (0.9^{time} \times t_{old} + t_{new}) / (t_{new} == 0 ? 1 : 2)$$

$$d_{ST(A_2)}^A = 0$$

$$u_{ST(A_2)}^A = 1 - (0.9^{time} \times t_{old} + t_{new}) / (t_{new} == 0 ? 1 : 2)$$

The time parameter in this function remembers the time interval between the old trust value and the new trust value (the period between trust updates). Every system in the network queries each constituent program (party) on its current value. Its own trust value is updated as a function of the updated values through these queries. These actions are undertaken at a regular period. If a query yields an answer that the trust value is not available, the belief will be lowered and the uncertainty will be increased.

### 3.5 Related work on program level anomaly detection

As we are the first to study the program level trust, we can not list any related work on program level trust in this section. Instead, we talk about two program level anomaly detection works.

#### 3.5.1 Fast Automaton-Based Method

Fast Automated-Based Method [Sekar *et al.*, 2001] is anomaly detection work based on system calls. Finite state automata are trained during runtime to model the sequence of system calls. After the training is finished, the system call sequence and location will be monitored against the finite state machine. The solution has both false positive and false negative on overflow attacks.

#### 3.5.2 Intrusion Detection via Static Analysis

Intrusion Detection via Static Analysis [Wagner *et al.*, 2001] is another anomaly detection work based on system calls. Four models are presented: trivial model, callgraph model, abstract stack model and digraph model. The trivial model monitors the runtime process against a set of possible system calls. The callgraph model generates a non-deterministic finite automaton from the call graph. The abstract stack model derives a non-deterministic pushdown automaton from the call graph. The digraph model computes a list of the possible  $k$ -sequences of consecutive system calls for all the possible executions. The automata derived from source codes describe the process behavior better than the trained automata of Fast Automated-Based Method.

Both Fast Automated-Based Method and Intrusion Detection via Static Analysis use the model based on system calls to describe process runtime behavior. The disadvantages of these approaches are:

1. The system calls model protection could be bypassed by mimicry attacks [Wagner *et al.*, 2001]. The attacker could choose the state to attack the program or mimic the system calls to force the automaton into certain state, such that certain system calls are permitted.
2. The system calls model is not fine-grained enough for program level trust.  
Wagner [Wagner *et al.*, 2001] argued that “A compromised application cannot cause much harm unless it interacts with the underlying operating system, and those interactions may be readily monitored.” This statement does not apply for program level trust because the program output is not longer trusted when the program control flow is changed by attacker at runtime, regardless of system call.

### **3.6 Future work**

#### **3.6.1 Separate monitor**

Another implementation of trust automaton separates the monitor from the monitored program. We need to guarantee several characteristics to generate a trustworthy value in a separated monitor framework.

1. The runtime monitor cannot be compromised.
2. The messages between the monitor and the program cannot be falsified.
3. The compromised program cannot lie about its own state.

To achieve these goals, we propose to add more architecture level and operating system level support based on trusted computing platform published by Trusted Computing Platform Alliance. The trust monitor will be moved into the trusted platform module (TPM) past of the computing engine. Some key aspects of the program state crucial input to the trust engine will be maintained by the TPM like engine in order to ensure the integrity of the monitored state variables.

### **3.6.2 Transaction level trust**

A transaction which involves several programs will receive several program level trust values. How to combine those values into a single value is an open question. We might need more powerful tools to describe the interactions between processes and calculate the trust level based on the component program level trust values.

### **3.6.3 Trust function**

Trust function associated with trust automata will determine the dynamics of the trust value according to the program actions. It is a critical part of the trust policy. How to choose a suitable trust function for a given trust policy is an open question.

## **3.7 Conclusions**

We proposed a control flow integrity based trust model. We argued why it makes sense to maintain a program's self assessment of trust. A compiler driven approach was presented and its performance overhead was reported. We also proposed an architecture level approach that is likely to be more efficient.

## CHAPTER 4. STREAM AUTOMATA

Both function pointer protection and control flow checking are compiler patches which require access to source codes. To cover the overflow attack and enforce security policies regardless of source codes, we implement a runtime monitor enforcing stream automata. Run time monitoring observes the action stream from a program either allowing a token to pass through to the operating system if the action is deemed to be safe or terminating the program on an unsafe action. Schneider [Schneider, 2000] evaluates the types of policies enforceable by security automata. Edit automata [Ligatti *et al.*, 2005a] enhance the security automata by incorporating edit actions on the program action stream. We develop the notion of stream automata that sit between two communicating processes (we refer to one of the processes as the program and to the other as the system). The stream automata can perform simple edit actions on the program stream before forwarding it to the system and vice versa. These edit actions include action suppression, insertion, metamorphosis, forcing and two way forcing. We demonstrate that stream automata are strictly more powerful than either of security automata or edit automata. We show many policies enforceable by stream automata which cannot be enforced with edit automata. Moreover, stream automata allow us to model an open/reactive system, the complete interaction between a program and its environment, which was not possible with the earlier models. We implemented a security policy against buffer overflow attacks and a simple honeywall to demonstrate the power of stream automata.

### 4.1 Introduction

Formal verification of a program to determine its security vulnerabilities with respect to a security policy is an active research area. An alternative to static verification is run-time monitoring. For some policies, run-time monitoring is much more efficient since the instantiated run-time program state is available. In module checking [Kupferman *et al.*, 1996] [Kupferman *et al.*, 2001], all possible values for an environment variable have to be

considered, which often leads to exponential explosion in search space. For instance consider a pointer in a C program. In a run-time monitoring paradigm, we know exactly the value of the pointer when a certain property pertaining to this pointer needs to be verified. However, in module checking, all  $2^{32}$  possibilities (for 32-bit addresses) might have to be considered.

This raises the question of what kind of policies are best monitored at run-time. The answer to a large extent depends on the monitored system. Consider the interactions of a program and the operating system (OS). The monitor screens all the OS service requests originating at the program before forwarding them to the OS. If these service requests violate a predefined policy, the monitor can either suppress such a request or terminate the program altogether. The computation time of the monitor is a relevant parameter since the program/OS transactions will experience this latency. Even a bigger concern might be the resource needs of such a monitor. If the monitor requires more resources than the monitored program, it may cease to be of practical value. Schneider [Schneider, 2000] considers such a monitoring paradigm with respect to monitors based on security automata (with resource usage similar to that of finite state machines). Security automata terminate the program on property violation (program execution truncation). A property compliant program action is passed to the OS unaltered. Ligatti *et al.* [Ligatti *et al.*, 2005a] and Besson *et al.* [Besson *et al.*, 2001] extended the domain of monitor actions from program termination and pass-on to some editing actions on the program actions. For instance, if a program action is deemed harmful to the OS, it could be suppressed or thrown away. The OS never gets to see it. These monitors were named edit automata. Edit automata are shown to be strictly more powerful than security automata in as much as they can enforce policies not enforceable by security automata.

Note, however, that both these classes of monitors observe only the actions of one party (program) destined for another party (OS). How will the policies that entail examining the actions of both the OS and the program be enforced? One could argue that in such cases,

one edit automaton will monitor program actions destined to the system; and another edit automaton  $E_{PS}$  will monitor the system actions with program as the target. What if the policy determines acceptability of an action based on what actions occur in program-to-system stream and system-to-program stream simultaneously? This is the gist of the current chapter. We define and develop *stream automata* to enforce security policies on  $P-S$  (program to system) and  $S-P$  (system to program) action streams. We also formally show that there exist real policies enforceable with stream automata which cannot be handled with edit automata.

Note that the stream automata can be viewed as monitors (intermediaries) between an open/reactive program and the system. This makes them suitable for monitoring security policies of embedded systems. In fact, they can monitor the transactions between any arbitrary pair of processes (which need not to be limited to system and program viewpoint). For instance, a session between two ftp demons could be monitored with stream automata. A further generalization will allow the stream automata to consider more than two parties, each generating a stream of actions for the other parties.

In the following sections, we will call the two parties in the environment of the monitor as system (which can denote as operating system, internet etc.) and a program (which can denote as process, applet etc.). The stream automata have full control over program but only partial control over system.

The main contributions of the chapter are as follows:

1. A new model of run time monitoring of open systems is proposed using stream automata. Syntactically, stream automata are similar to input-output automata [Maraninchi *et al.*, 1996]. The proposed automaton sits between a system and a program and acts as a stream transformer to enforce security policies.
2. Monitoring capabilities of stream automata are formally defined using SOS style operational rules [Plotkin, 1981] that clearly identify a specific monitoring action such as suppression or editing. This approach leads to a clear and formal way of

specifying monitoring actions by using rules for process composition as used in concurrency literature [Hoare *et al.*, 1985] [Milner, 1989]. Our approach, thus, simplifies the underlying automata (unlike Edit automata where the automata definition is more complex and monitoring actions are informally specified).

3. Stream automata are capable of all monitoring actions of edit automata. They can, in addition, perform new monitoring actions called metamorphosis, forcing and two-way forcing which are introduced in this chapter. These new monitoring actions are intended for the monitoring of open systems and are able to enforce additional policies called resource blind access control, ordering consistency, and barrier dependency.
4. We develop a formal Turing machine model of stream automata monitoring framework. In this model, stream automata monitoring is incomparable to security and edit automata. We however informally establish that stream automata are more powerful than security or edit automata, and are capable of monitoring complex open systems such as embedded systems without any noticeable monitoring overhead.

The rest of the chapter is organized as follows. We present related work in Section 4.2. The notion of stream automata is introduced in Section 4.3. This section also presents the formalization of the monitoring capabilities of stream automata using structural operational semantics (SOS) [Plotkin, 1981] style rules. Section 4.4 presents examples of some enforceable security policies. This section also presents the power of stream automata using a new class of policies introduced in this chapter. Some of the experimental results are presented in Section 4.5. We conclude the chapter in Section 4.6.

## 4.2 Related work of formal model on monitoring

Schneider [Schneider, 2000] introduced a formal model for execution monitoring. This was redefined by Viswanathan [Viswanathan, 2000] to account for computational capabilities of the monitor. Ligatti *et al.* [Ligatti *et al.*, 2005a] enhanced the model further through edit automata that generalize the enforcement mechanisms beyond program termination. Fong [Fong, 2004] characterizes monitoring capabilities when the information about the program execution available to the monitor is parameterized (through shallow execution depths). Hamlen *et al.* [Hamlen *et al.*, 2003] generalize the monitor actions to program rewriting. Ligatti *et al.* [Ligatti *et al.*, 2005b] generalize edit automata to infinite execution traces to establish the feasibility of non-safety property monitoring.

## 4.3 Stream automata

Stream automata may be viewed as input-output automata [Lynch *et al.*, 1989] [Maraninchi *et al.*, 1996] that transform streams. Figure 4.1 illustrates it through a schema.

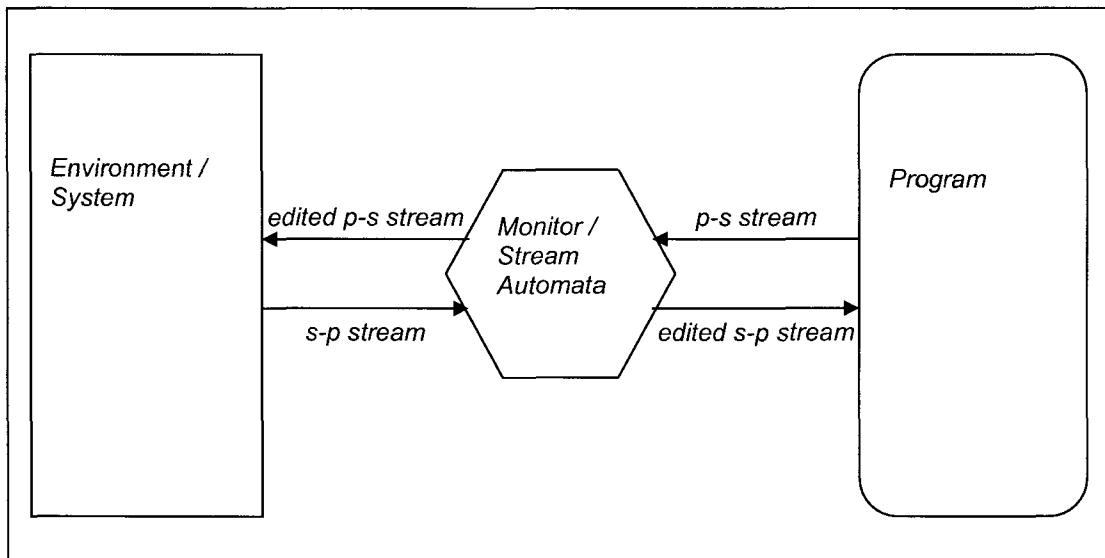


Figure 4.1 Stream Automata Runtime Monitor

One entity, program, generates a stream  $(\Sigma^P)^*$  of actions on its action alphabet  $\Sigma^P$ . Another entity - system - generates another stream  $(\Sigma^S)^*$  on its action alphabet  $\Sigma^S$ . A stream automaton performs transformation of program streams (similar to edit actions of edit automata), system streams (edit actions applied to the system side) and can perform edit actions on both sides (program and system) at the same time. Inputs received either from the program or the system are used to trigger transitions in the automaton and outputs are used for editing actions. Type of editing actions depends on the type of output generated. Association of outputs on transitions is standard in reactive systems [Harel *et al.*, 1996] [Harel *et al.*, 1996] [Maraninchi *et al.*, 1996] [Maraninchi *et al.*, 2001]. This approach leads to simplification of the monitor transition system compared to edit automata that require multiple types of transition functions on the same automata (one for state transition and another for the specification of edit actions).

The state space of a stream automata is partitioned into external (or system) states and internal states similar to standard open system models proposed by Kupferman *et al.* [Kupferman *et al.*, 1996] [Kupferman *et al.*, 2001]. The monitor receives external inputs either from the program or the system in an external state. In an internal state, the monitor engages in some internal computation (such as operation on internal variables or synchronization with other monitors).

A transition in stream automaton is based on current state and input (an external input when in an external state or internal input from an internal state). The transition specifies a next state, and also an output action (which is a set).

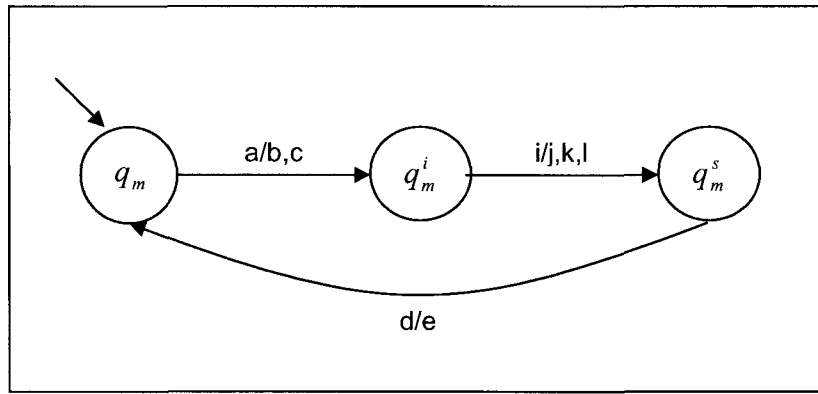
A definition of stream automata follows.

**Definition 4.1.** A stream automaton is a 5-tuple  $M = (Q_m, q_m^0, \Sigma_m, \Gamma_m, \rightarrow_m)$ , where

1.  $Q_m$  is a set of states that is partitioned into  $Q_m^e, Q_m^i$  that are either external or internal states. External states interact with external processes (both program and system) and internal states perform internal computation or synchronization with other concurrent stream automata, respectively. Let  $Q_m = Q_m^e \cup Q_m^i$ .
2.  $q_m^0$  is the initial state.
3.  $\Sigma_m$  is the input alphabet that is partitioned into  $\Sigma_m^p, \Sigma_m^s, \Sigma_m^i$  that are program, system, and internal input alphabet respectively. Let  $\Sigma_m^e = \Sigma_m^p \cup \Sigma_m^s$  be the set of external inputs. Then  $\Sigma_m = \Sigma_m^e \cup \Sigma_m^i$ .
4.  $\Gamma_m$  is the output alphabet that is partitioned into  $\Gamma_m^p, \Gamma_m^s, \Gamma_m^i$  that are programs, system and internal output alphabet respectively. Let  $\Gamma_m^e = \Gamma_m^p \cup \Gamma_m^s$ . Then  $\Gamma_m = \Gamma_m^e \cup \Gamma_m^i$ .
5.  $\rightarrow_m: Q_m \times \Sigma_m \times 2^{\Gamma_m} \times Q_m$  is a transition relation satisfying the following constraints:
  - (a) If  $(q_m, \sigma, \gamma, q_m') \in \rightarrow_m$  and  $q_m \in Q_m^e$ , then  $\sigma \in \Sigma_m^e$
  - (b) If  $(q_m, \sigma, \gamma, q_m') \in \rightarrow_m$  and  $q_m \in Q_m^i$ , then  $\sigma \in \Sigma_m^i$

In a stream automaton  $M$ , a transition is of the form  $(q_m, \sigma, \hat{\gamma}, q_m')$  where  $q_m$  is the source state,  $\sigma$  is an input action,  $\hat{\gamma}$  is a set of output actions, and  $q_m'$  is a destination state. The transition triggers when the action  $\sigma$  occurs (either in the environment or internally) and the triggering of the transition produces a set of outputs and results in the automaton changing state. We use superscript  $e$  or  $i$  to indicate if the monitor state is an external or internal state ( $q_m^e$ , for example, denotes an external state of the monitor). Given a pair of states  $q_m, q_m'$  we will use standard infix notation  $q_m \xrightarrow{\sigma, \hat{\gamma}} q_m'$  rather than  $(q_m, \sigma, \hat{\gamma}, q_m') \in \rightarrow_m$ .  $\lambda \in \Gamma_m^i$  is a special output of the monitor that indicates that the monitor will suppress some actions occurring in its system.

The additional conditions on the transition relation (conditions (a), (b) in Definition 1) constrain the input part of the transition based on the source state of the transition. When the source state of a transition is an external state, the input  $\sigma$  has to be an external input ( $\sigma \in \Sigma_m^e$ ), when the source state is an internal state then the input has to be an internal input. An example monitor of a stream automaton is shown in the Figure 4.2.



**Figure 4.2 Example of Stream Automaton**

In this example, there are three states. State 0, which is an external state, is also the start state of the automata. State 1, is an internal state and State 2 is another external state. Input  $a$  is a program input,  $i$  an internal input and  $d$  a system input. From State 0, when an input  $a$  occurs in the program, the stream automaton, produces two different outputs  $b, c$ , where  $b$  is a system output and  $c$  is a program output. Thus, in this example, the monitor upon receiving input  $a$  from the program, produces two different outputs, one for the program and the other for the system.

### 4.3.1 Enforcement actions

This section presents the enforcement actions of stream automata. We formalize the enforcement actions by defining composition of the monitor with the program or the system (depending on the current state of the monitor). For composition, we follow standard approach used in concurrency and process algebras for process composition that uses SOS style operational rules [Plotkin, 1981] for process composition. We assume that the monitor sits between two processes, the system and the program and can monitor the state of both these processes. Using this model, we are able to formalize the existing enforcement mechanisms such as truncation, suppression, and insertion. In addition, we define new kinds of enforcement actions called *metamorphosis*, *forcing* and *two-way forcing*.

In the following, we assume that the system is an automaton of the form  $S = (Q_s, q_s^0, \Sigma_s, \rightarrow_s)$  and the program is  $P = (Q_p, q_p^0, \Sigma_p, \rightarrow_p)$ . Note that  $\Sigma_s = \Sigma_m^s$  and  $\Sigma_p = \Sigma_m^p$ . Also, the transition relation is defined in the standard way:  $\rightarrow_s \subseteq Q_s \times \Sigma_s \times Q_s$  and  $\rightarrow_p \subseteq Q_p \times \Sigma_p \times Q_p$ . We will use CCS [Milner, 1989] unobservable action  $\tau$  to represent all internal input actions in the composition. When an input action is  $\tau$  in the composite system, it is unobservable in the system (and hence will be hidden for both parties in the environment of the monitor, i.e. the system and the program). Greek symbols will be used in the following rules for input and output actions. A set of actions  $\sigma$  will be indicated by  $\hat{\sigma}$ . Special internal action  $\lambda$  will be used for hiding or suppression. While  $\lambda$  indicates an output action,  $\hat{\lambda}$  is used for the set containing the action  $\lambda$ . Set representation is needed as a stream automaton generates sets of actions in the output of a transition.

In SOS style, rule for process composition is defined as follows:

$$\frac{\text{premises}}{\text{conclusion}}(\text{side condition}).$$

For the sake of enforcement rules, the premise part is used to specify the transitions of the monitor and that of either the system or the program. The conclusion part specifies the transition of the composite system if the premises are true and also the side condition holds (the side condition is optional). We use  $q_m, q_m'$  (with appropriate superscript) to denote the states of the monitor,  $q_s, q_s'$  for the states of the system and  $q_p, q_p'$  to denote the states of the program.

1. **Truncation:** This rule specifies that if the actions of the program conform to the actions of the monitor then it is allowed to proceed (by allowing the composite system to proceed). Implicitly, this rule causes termination, whenever there is an action in the program that is outside the monitored actions. In that case, the composite system has a null transition (i.e., the program is terminated). Note that this rule is identical to the CSP rule for parallel composition of concurrent

processes [Hoare et al., 1985]. Note that truncation can be only performed over programs but not over systems.

$$\frac{q_m^e \xrightarrow{\sigma} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\sigma} (q_m', q_p')} (\sigma \in \Sigma_m^p)$$

2. **Suppression:** This rule specifies that violating actions can be suppressed for either the program or the system (there are two rules, one for the program and the other for the system). Suppression is performed whenever the program (the system) has a specific action that the monitor wants to suppress. The monitor then generates a special output  $\hat{\lambda}$  to suppress the program action. In the composite system, there is a  $\tau$  (an internal or unobservable [Milner, 1989]) transition.

- a. Program action suppression

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\lambda}} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\tau} (q_m', q_p')} (\sigma \in \Sigma_m^p, \hat{\lambda} \in 2^{\Gamma_i})$$

- b. System action suppression

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\lambda}} q_m', q_s \xrightarrow{\sigma} q_s'}{(q_m^e, q_s) \xrightarrow{\tau} (q_m', q_s')} (\sigma \in \Sigma_m^s, \hat{\lambda} \in 2^{\Gamma_i})$$

3. **Insertion:** This rule specifies that program or system actions can be edited such that in addition to the received action, additional actions may be inserted into the action sequence. The monitor performs editing by outputting the observed action  $\sigma$  from the program (or the system) together with additional actions  $\gamma$ . In the composite system, the input actions are internalized (become  $\tau$ ) and the output actions ( $\{\sigma\}, \gamma$ ) are visible. Note that  $\gamma$  is a set while  $\sigma$  is an individual action (which has to be made into a set since outputs from the monitor are sets).

- c. Program action insertion

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\sigma}\hat{\gamma}} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\tau/\hat{\sigma}\hat{\gamma}} (q_m', q_p')} (\sigma \in \Sigma_m^p, \hat{\sigma}, \hat{\gamma} \in 2^{\Gamma_m^s})$$

- d. System action insertion

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\sigma}\cup\hat{\gamma}} q_m', q_s \xrightarrow{\sigma} q_s'}{(q_m^e, q_s) \xrightarrow{\tau/\hat{\sigma}\cup\hat{\gamma}} (q_m', q_s')} (\sigma \in \Sigma_m^s, \hat{\sigma}, \hat{\gamma} \in 2^{\Gamma_m^p})$$

4. **Metamorphosis:** This rule specifies that program or system actions can be suitably altered (they undergo metamorphosis) before reaching the other party. During metamorphosis, the monitor transforms a program action to a different action and then sends it to the system. A dual is possible, where the monitor performs a metamorphosis of a system action before it is sent to the program.

e. Program action metamorphosis

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\beta}} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\tau/\hat{\beta}} (q_m', q_p')} (\sigma \in \Sigma_m^p, \hat{\beta} \in 2^{\Gamma_m^s})$$

f. System action metamorphosis

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\beta}} q_m', q_s \xrightarrow{\sigma} q_s'}{(q_m^e, q_s) \xrightarrow{\tau/\hat{\beta}} (q_m', q_s')} (\sigma \in \Sigma_m^s, \hat{\beta} \in 2^{\Gamma_m^p})$$

5. **Forcing:** This rule specifies that program or system actions can be altered by force. If the program requests an action  $\sigma$ , the monitor performs a different action  $\alpha$  (for example, an applet requests the opening for the password file to the OS and in response the monitor actually opens a fake file). Note that unlike metamorphosis, the altered action of the monitor is visible only to the program. A dual forcing rule can similarly be applied for system actions. In the composite system, the input is internalized ( $\tau$ ) and the output is  $\hat{\alpha}$ .

g. Program action forcing

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\alpha}} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\tau/\hat{\alpha}} (q_m', q_p')} (\sigma \in \Sigma_m^p, \hat{\alpha} \in 2^{\Gamma_m^p})$$

h. System action forcing

$$\frac{q_m^e \xrightarrow{\sigma/\hat{\alpha}} q_m', q_s \xrightarrow{\sigma} q_s'}{(q_m^e, q_s) \xrightarrow{\tau/\hat{\alpha}} (q_m', q_s')} (\sigma \in \Sigma_m^s, \hat{\alpha} \in 2^{\Gamma_m^s})$$

6. **Two Way Forcing:** This rule generalizes forcing to both sides. This is possible either after receiving a program action or a system action. The monitor upon

receiving action  $\sigma$  from the program, generates two sets of outputs  $\hat{\beta}$  which is meant for the system and  $\hat{\gamma}$  which is meant for the program. In the dual rule, the monitor upon receiving action  $\sigma$  from the system, generates outputs  $\hat{\beta}$  which is meant for the program and  $\hat{\gamma}$  which is meant for the system.

i. Two way forcing upon receipt of program action

$$\frac{q_m^e \xrightarrow{\sigma / \hat{\beta} \cup \hat{\gamma}} q_m', q_p \xrightarrow{\sigma} q_p'}{(q_m^e, q_p) \xrightarrow{\tau / \hat{\beta} \cup \hat{\gamma}} (q_m', q_p')} (\sigma \in \Sigma_m^p, \hat{\beta} \in 2^{\Gamma_m^s}, \hat{\gamma} \in 2^{\Gamma_m^p})$$

j. Two way forcing upon receipt of system action

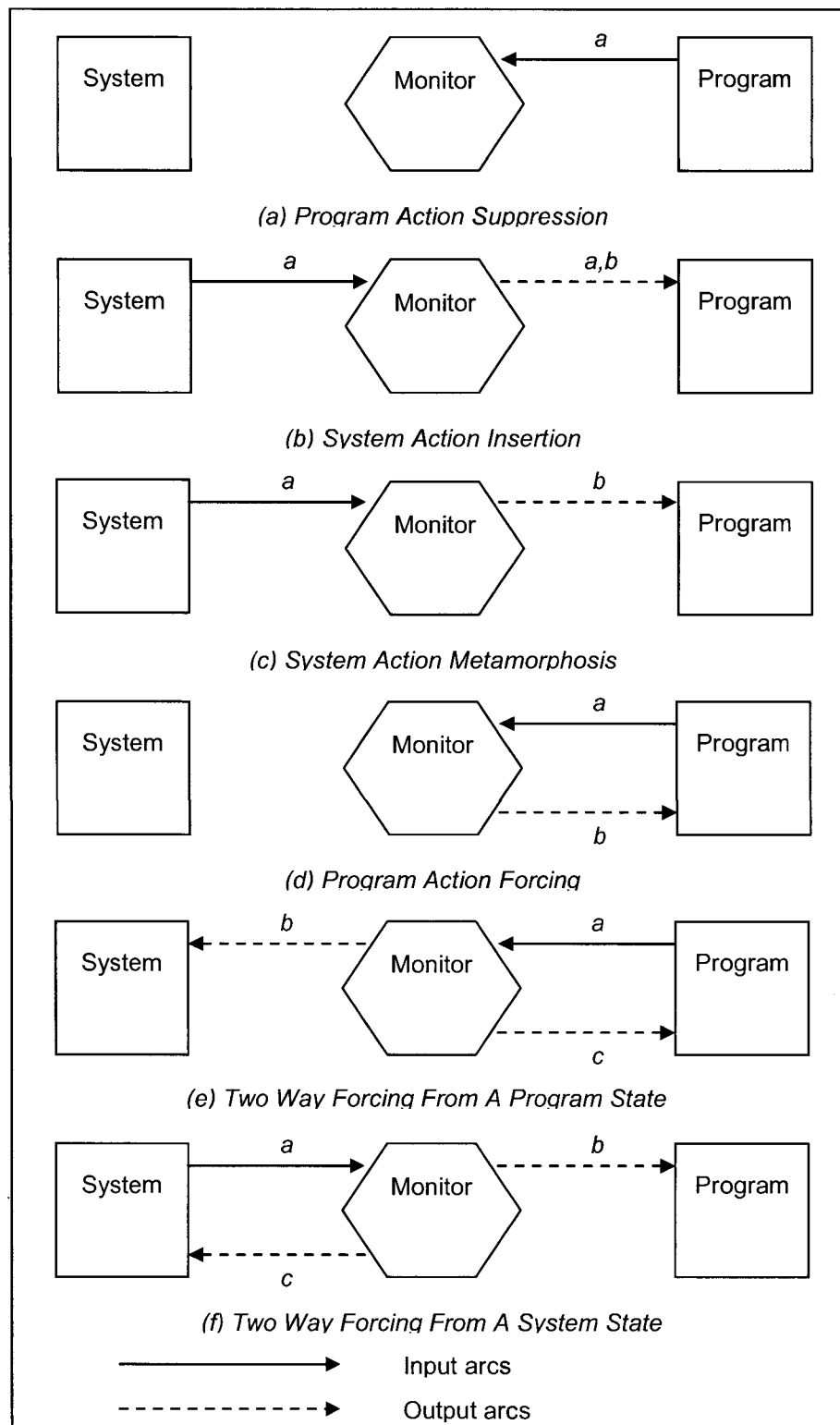
$$\frac{q_m^e \xrightarrow{\sigma / \hat{\beta} \cup \hat{\gamma}} q_m', q_s \xrightarrow{\sigma} q_s'}{(q_m^e, q_s) \xrightarrow{\tau / \hat{\beta} \cup \hat{\gamma}} (q_m', q_s')} (\sigma \in \Sigma_m^s, \hat{\beta} \in 2^{\Gamma_m^p}, \hat{\gamma} \in 2^{\Gamma_m^s})$$

Note that during an insert, it is not possible to make the input part empty in the output part. This is how the edit automata defines insert. Hence, there is a need to define the new operation that we callas metamorphosis. The difference between these rules is visually depicted using the Figure 4.3.

## 4.4 Enforceable policy examples and the power of stream automata

In this section, we present many examples monitors using stream automata.

Subsequently, we discuss the power of stream automata compared to existing techniques for run-time monitoring. In the following Figures, we use the following notation on the transitions: *input-part / (system-outputs; program-outputs)*. If any of the output parts are not specified, it indicates empty output for that part.

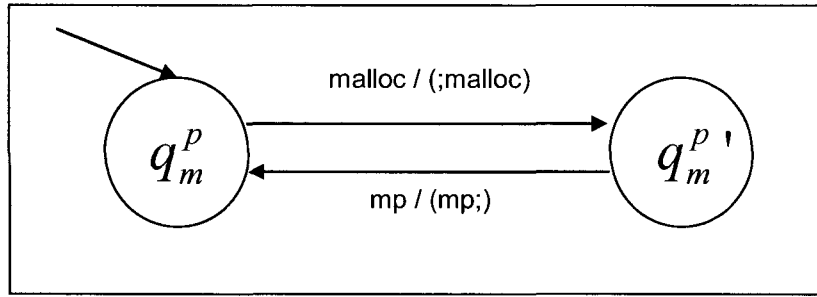


**Figure 4.3 Enforcement Operation Examples of Stream Automata**

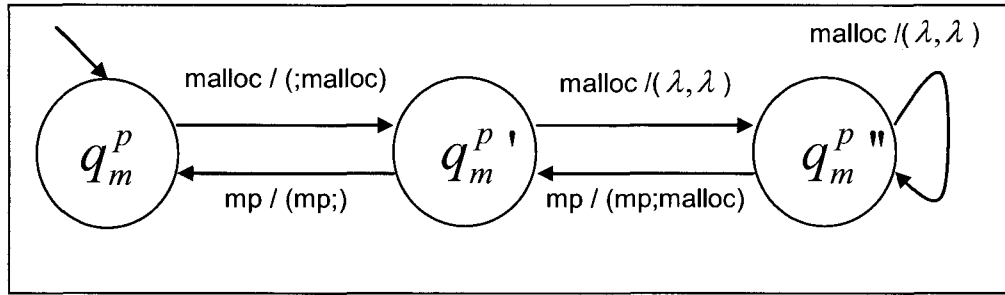
### Paired Action Policy

Consider a policy that insists on ensuring that before a new service request is entered, all the preceding request for that service ought to be completed. Such a policy makes sense for interrupt handling (only one pending interrupt per group of devices), or for cache miss handling from Level 2 cache to main memory (at most one or some small number of pending memory misses), or at most one http connection per *httpd demon*, or at most one or less than five *malloc* requests (until corresponding allocated *memory pointer*, *mp* actions).

Figure 4.4 illustrates such a policy.  $q_m^p$  is a program state which allows a *malloc* request from the program to pass through (since the memory pointers for all the preceding *malloc* requests have already been delivered). Hence, if a *malloc* request comes from the program, it is passed through unchanged to the system (OS). Similarly, in system state  $q_m^s$  the monitor is waiting for the allocated memory pointer for a preceding *malloc* request from the operating system. On such an input, the memory pointer *mp* is passed to the program unchanged, and the next state is  $q_m^p$ . What should happen if a *malloc* comes along in state  $q_m^s$ ? If we assume persistent programs and systems that continue to assert their actions until services (like an interrupt), it stalls the program at that *malloc*, which it services only when the monitor transitions to  $q_m^p$  after a *mp* from the system. We could build a monitor with some finite buffering for efficiency considerations. Let us say we would remember up to 3 pending *malloc* actions. Then on a *malloc* in state  $q_m^s$ , we can transition to an internal state  $q_{m1}^i$ , queue the *malloc* parameters into an internal data structure, and output  $\lambda$  for the operating system ( $\lambda$ , ). On a memory pointer from OS in this state, we can transition to  $q_m^s$  with output *malloc* (with appropriate parameters) to OS and output *mp* to the program (*mp*, *malloc*). Figure 4.5 shows this scenario. In this example, we have demonstrated both suppression and insertion actions.



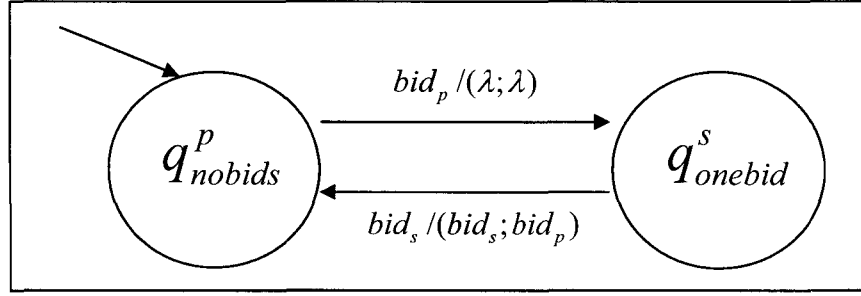
**Figure 4.4 Paired Action Monitor**



**Figure 4.5 Modified Paired Action Monitor**

### Online Bidding/Game Move Policy

Consider another policy wherein two (or more) bidders submit a bid. The bidding protocol requires that all the bids be broadcast to all the bidders. However, the protocol must ensure that each bidder has already placed a bid before they are allowed to see all the other bids. A similar situation exists in a multi-player online game where each game proceeds in rounds. Each player makes a move in each round simultaneously. In these situations, the monitor should collect the tokens from both the parties before it sends them out to either of them. Figure 4.6 illustrates this policy. In state  $q_{nobid}^p$ , a bid from program(Party 1) is expected. However, when that bid  $bid_p$  does come in, it is not revealed to any party through empty output action  $(\lambda, \lambda)$ . The monitor transitions to state  $q_{onebid}^s$ . In  $q_{onebid}^s$ , when the other bid  $bid_s$  is received, the monitor broadcasts both the bids to each other through output action  $(bid_s, bid_p)$ . This example illustrates both suppression and two-way forcing.



**Figure 4.6 Online Bidding Monitor**

### Monitor with a Transformed Program Semantics

This example brings out another interesting and more powerful model for monitoring. Figure 4.7 illustrates the paradigm. The monitor sits between a system (network in this example) and program (the entire operating system or a specific demon in a server in this case). The monitor has an embedded firewall in this example. A basic firewall would interpret the firewall rules and classify each packet as suspicious (s) and drop it or would classify it as normal and let it pass. A monitoring firewall is shown on the RHS of Figure 4.7. It is a parallel composition of several automata. On the left, many automata are deployed for many different firewall rules. A packet  $p$  arriving in state  $q_0$  is processed for all firewall rules (each rule corresponding to an automaton starting in state  $q_1$  or  $q_2$  or  $q_3$ ). If the packet is deemed suspicious, the output destined for the program contains an annotated packet, the original packet  $p$  along with a flag  $s$  to indicate suspicion, and an empty output for the system  $((p, s; \lambda))$ . The second automaton shown in the RHS of the monitor classifies several levels of suspicion. If  $k$  consecutive suspicious packets (tagged as s by other firewall automata) are found, suspicion level  $s_k$  is indicated to the system. In Figure 4.7, we only show two suspicion levels  $s_1$  and  $s_2$ .

The program (demons or the entire OS kernel) associates semantics with these annotations (suspicion flags) inserted by the monitor (firewall). In the example semantics, level  $s_1$  (only one suspicious packet) results in loss of file access privileges. Level  $s_2$  also loses internet access privileges.

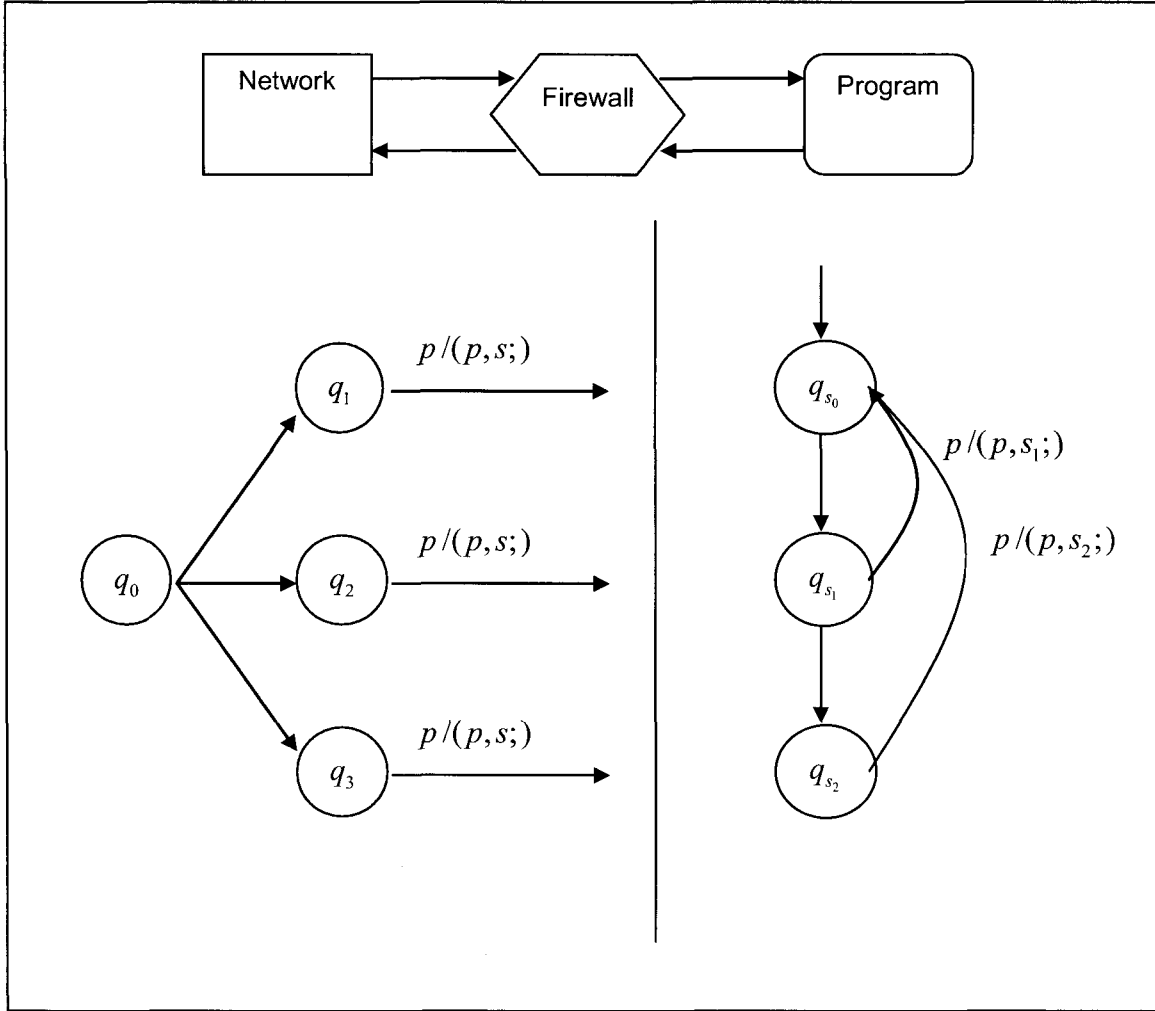


Figure 4.7 Monitor Based on Transformed Program Semantics

#### 4.4.1 Power of stream automata: TM based model

In this section, we show that stream automata (SA) are incomparable to security automata [Schneider, 2000], RW-enforceable class through program rewriting [Hamlen *et al.*, 2003], edit automata [Ligatti *et al.*, 2005a], or infinite executions edit automata [Ligatti *et al.*, 2005b]. This is in a formal Turing machine based model. We show in Section 4.4.2 later however through informal reasoning that stream automata are inherently more powerful than both Schneider's security automata [Schneider, 2000] (which will be referred to as EM

denoting enforcement monitor) and Ligatti *et al.*'s edit automata [Ligatti *et al.*, 2005a] (which will be referred to as EEM denoting edit enforcement monitor).

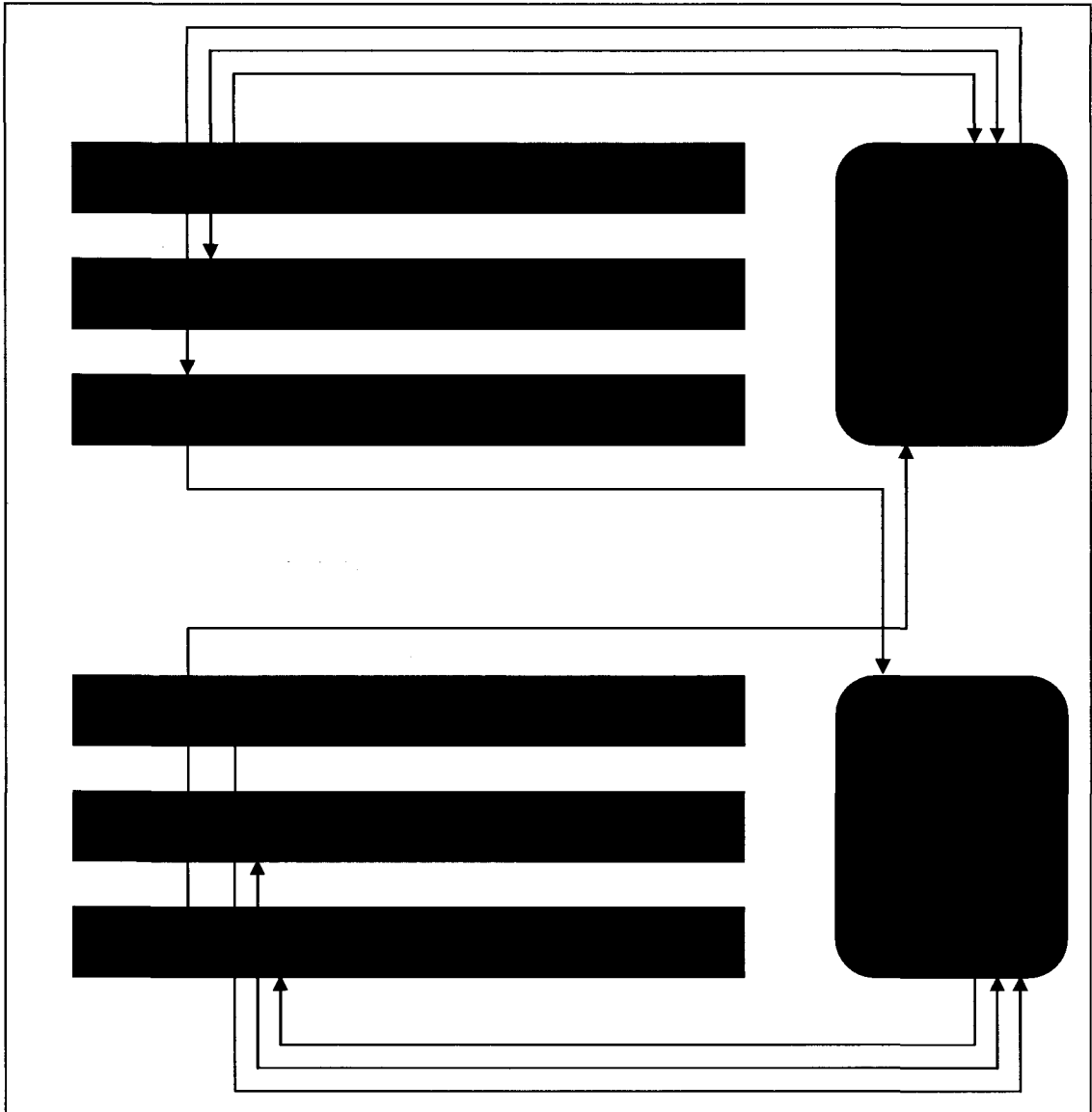
**Definition 4.2.** A program-system pair can be modeled as a pair of Turing machines  $(T_p, T_s)$ . The program Turing machine  $T_p$  is like *PM* of [Hamlen *et al.*, 2003]. It has an input tape, a work tape, a trace tape recording infinite sequence of actions, and a finite control. Similarly, the system Turing machine  $T_s$  is also a *PM* with an input tape, a work tape, and a trace tape. A key difference between a cross-product of two machines  $(T_p \times T_s)$  and the composition  $(T_p, T_s)$  assumed by us is that finite control of  $T_p$  bases its decision solely on its internal state  $q_p$ , its input tape and work tape symbols  $(\sigma_p \in \Sigma_p, \sigma_p^w \in \Sigma_p^w)$ , and the contents of the trace tape  $T_s$  given by  $e_s$ . Thus we have

$$\delta_p(q_p, (\sigma_p, \sigma_p^w), e_s) \rightarrow (q_p', (H_i, (\sigma_p^w', H_w)), (e_p, H_s)).$$

This means that on a transition, the heads of input tape, work tape, and trace tape for  $T_p$  move independently along with unique write symbols for the work tape and trace tape of  $T_p$ .  $H_i$ ,  $H_w$  and  $H_s$  specify head movements on the input tape and work tape of  $T_p$  and  $T_p$ 's head on the trace tape of  $T_s$ . Note that we also assume that the head on the trace tape of  $T_p$  can only keep moving right, and hence no head direction is specified. Also note that  $T_p$ 's head on  $T_s$ 's trace tape can only read. The transition function of  $T_s$  is defined similarly. Figure 4.8 shows such a Turing machine pictorially.

#### 4.4.1.1 Model

A monitoring paradigm strives to project the state of the monitored program  $P$  through some projection function into an infinite stream of symbols  $A_p^\infty$  [Ligatti *et al.*, 2005b]. The state projection function  $F_s(p)$  need not be static for the entire program execution.



**Figure 4.8 Coupled Turing Machines for Program and System**

Hamlen and Schneider [Hamlen *et al.*, 2003] model this a Turing machine, called a program machine PM, such that it has a private state captured on an internal working tape, and a publicly observable trace tape where monitor observable sequence  $A_p^\infty$  is written. No assumption is made regarding how  $A_p^\infty$  is derived from  $P$  allowing for an arbitrary, potentially non-uniform [Ruzzo, 1981], family of state projection functions  $F_s(p)^k$ .

The stream automata, however, model two concurrent processes that are composed. This machinery looks more similar to software protection models of oblivious Turing machines or RAMs as in [Goldreich *et al.*, 1996]. We define the model formally now.

#### 4.4.1.2 Some observations

We make some observations about incomparability of PM and PSM (program, system TM pair) in the following.

1. For concurrent processes, PSM exposes the internal partitioning of a composite PM that would otherwise model a (system, program) pair. Within a PM model, the enforcement mechanism will have to argue overall possible partitions of the PM state space in order to satisfy the property predicate  $\hat{P}$  in Schneider's framework [Hamlen *et al.*, 2003] [Schneider, 2000], a significantly harder task than determining applicability of  $\hat{P}$  to the two streams given by  $A_p^\infty$  and  $A_s^\infty$ .
2. When an environment in an embedded system is captured as one of the processes (system or program), the relative power of SA becomes even more interesting. An environment in an embedded system could very well be a natural, non-computational process (such as a sensor capturing density of pollen in the air). Such environments give rise to true input streams for data getting us into the murky and contested discipline of coupled Turing machines [Copeland, 1997] [Viswanathan, 2000]. The fact remains, however, that SAs are better prepared to handle such embedded systems than either of security or edit automata.
3. We are more similar to edit automata than to program rewriting paradigms of Schneider [Hamlen *et al.*, 2003] since the monitoring action forces an action on the program solely through its input stream rewriting. The program itself is not rewritable.

4. We will develop a class hierarchy within this monitoring model in our future work.

#### 4.4.2 Power of stream automata: informal reasoning

In this section, we reason about the relative power of SAs with respect to security (EM) and edit (EEM) automata informally with respect to policy classes. Before formalizing the power of stream automata, a new classification of security policies will be proposed. This classification extends Schneider's [Schneider, 2000] classification. The new class of policies is arrived at by weakening the access control policy and also by including policies that deal with multiple processes (rather than just talking about the program) as follows:

##### Schneider's classification:

1. Access Control - No execution of a program may operate on certain resources or invoke certain critical system operations.
2. Availability - If a resource is acquired by a program then it is eventually released.
3. Bounded Availability - If a program accesses a resource then it must release it within a specified bound (say within some fixed  $n$  steps).
4. Information flow - Let  $s_1, s_2$  denote some input output pair such that  $s_2 = f(s_1)$  for some function  $f$  that performs the transformation operation. This policy specifies that if there exists a trace in the execution of the program where  $s_2 = f(s_1)$ , then there must exist another trace where  $s_2 \neq f(s_1)$ .

##### New Policies:

1. Resource Blind Access Control - This requires that when a program tries to access some critical resource, it does not have capability to determine the quality of allocated resource, or does not care to determine this quality. It suffices for the resource requester to know that a resource has been allocated. This includes the

scenario where a fake resource (such as a fake-password file) is allocated by the resource manager (such as OS). In some sense, all that is required is for the resource requester is to know that a resource has been allocated (in the context of theory of knowledge).

2. Ordering Consistency - When multiple concurrent processes execute, it is important to specify acceptable models for shared resource accesses in order to maintain an unambiguous semantics. The multiprocessing environments with shared memory have memory consistency models for this purpose. For concurrent processes composed through their input/output streams, ordering of events/actions at their input/output streams gives rise to a consistency model. This policy tries to ensure some ordering between the actions of the two parties involved (the system and the program). It requires that the program (the system)) be allowed to perform an action only after the system (the program) satisfies some condition.
3. Barrier Dependency - This requires that the program and the system are allowed some information about the other party only after both of them satisfy some condition (have arrived at their barrier).

These policies may be thought of as either safety properties (no bad states exist in the program which is defined by Lamport [Lamport, 1977] as nothing bad ever happens) or liveness properties (which assert that the program eventually enters a good state or in Lamport's terms - something good eventually happens). While access control, ordering consistency, barrier dependency and resource blind access control are safety properties, availability is a liveness property. Some policies like bounded availability combine safety and liveness. It has been shown in [Ligatti *et al.*, 2005a] that EEM is strictly more powerful than EM. While EM only guarantees safety properties, EEM is able to guarantee safety and some safety-liveness properties (such as bounded availability). SA is capable of all operations performed by EEM (it has capability of truncation, insertion and suppression by

composition rules defined in Section 3.1). Hence it can emulate all monitoring actions of EEM. We illustrate the additional power of SA through the following theorems.

**Theorem 4.1.** SA can enforce blind resource access control. This is not feasible under EEM.

*Proof.* Blind resource access control demands that if some restricted resource is accessed, then it suffices for the program to receive a resource (whether genuine or a fake resource, or of any quality on a bigger continuum). It is easy to see that this policy is implementable under SA using the *forcing rule*. The proof is constructive. Given this policy, a monitor may be constructed such that whenever it sees the program performing access to the restricted resource (say action  $\alpha$ ), it will immediately output a fake (some) action for the program (say action  $\beta$ ). Thus, program action  $\alpha$  is transformed to another program action  $\beta$  that conforms to the requirements of the policy.

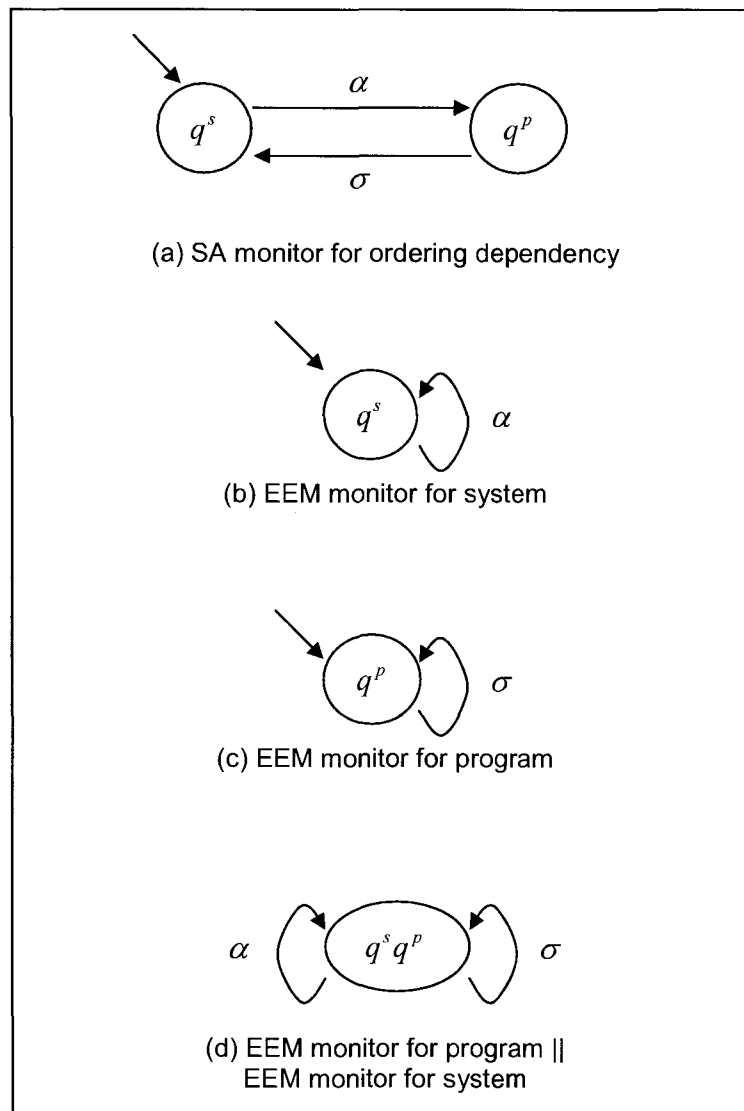
This is not feasible under EEM as none of the rules (truncation, suppression or insertion) can transform one program action into another.

**Theorem 4.2.** SA can enforce ordering consistency. This is not feasible under EEM.

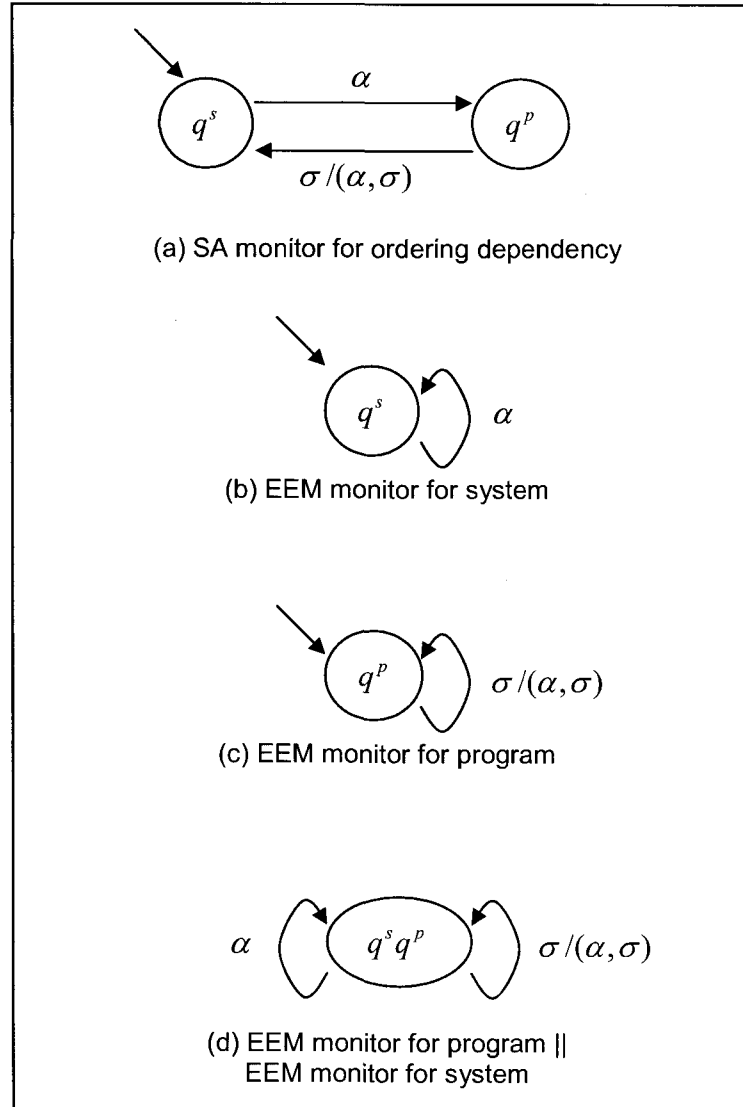
*Proof.* Ordering consistency requires some ordering relationship between the actions of program and system (or system and program). The proof of this theorem is also constructive. Given an ordering consistency such that the program perform action  $\sigma$  only after the system action  $\alpha$ , we can create a monitor with two constructive transitions, one triggered by  $\alpha$  from a system state and the next triggered  $\sigma$  by from a program state. This monitor will make sure that the program action ( $\sigma$ ) will follow only after the system action ( $\alpha$ ) is completed. This monitor is shown in Figure 4.9(a).

To show that it is not feasible under EM or EEM to enforce this policy, we use the following counter example. Obviously, there is no single monitor of EM or EEM category to achieve this enforcement. Hence, let us use two different monitors, one for the system side (shown in Figure 4.9 (b)) and the other for the program side (shown in Figure 4.9 (c)). The

system side monitor will have a transition triggered by  $\alpha$  and the program side monitor will have a transition triggered by  $\sigma$ . When these two monitors are executed in parallel (free running synchronous parallel, denoted  $\parallel$ , like CCS [Milner, 1989]), the composite system will have a state that will have both transitions enabled (either the program or the system can take its transition, independent of the other). This is depicted in Figure 4.9 (d). This composite monitor could not enforce this policy.



**Figure 4.9 Constructive Example**



**Figure 4.10 Constructive Example**

**Theorem 4.3.** SA can enforce barrier dependency. This is not feasible under EEM.

*Proof.* SA enforces barrier dependency using two-way forcing. The proof is constructive (given a barrier dependency, the corresponding SA is easy to construct and the enforcement is performed using two way forcing rule).

To prove that it is not feasible under EEM to enforce barrier dependency, we use the following counter example. Consider a barrier dependency that demands that only after the program has performed action  $\sigma$  and the system has performed action  $\alpha$  will both be allowed to view the action of the other party (this is similar to the online bidding policy discussed in the previous section). A monitor in SA for this policy is shown in Figure 4.10 (a).

Now consider EEM monitoring. Obviously, there is no single EEM monitor that can perform this. Hence, let us consider two EEM monitors, one for the system and the other for the program. The two monitors and their parallel composition are shown in Figure 4.10 (b), (c), (d) respectively. Obviously, the composite monitor is not able to enforce the barrier dependency policy.

## 4.5 Buffer overflow & honeynet examples

In this section, we demonstrate stream automata for more substantial system examples for a buffer overflow scenario, firewall and honeywall.

### 4.5.1 Program monitor example

Take the program *01.c* as one simple example. It tries to call *verify()* followed by a call to *critical()*. However, due to the deliberate/intended over flow coding at Line 17, 3 bytes of the function pointer *f* will be overflowed at runtime.

Among the many possible runtime results, Table 4.2 shows one expected normal execution. Table 4.3 shows that the malicious user input could change the program behavior to execute *critical()* without executing *verify()* first.

Notice that the function addresses are as follows:

*verify* = 0x08048458

*critical* = 0x08048470

Line 17 of *01.c* will overflow *ch* as *scanf("n%x",&ch)* will get 4 bytes and *ch* is only 1 byte. The result will then overflow into 3 consecutive bytes following *ch*, which are the 3 lower bytes of the function pointer *f*. As the highest byte of *f*'s address is already 0x08, the user could change the lower 3 bytes of *f* to point to *verify()* or *critical()* or a random invalid address to cause a core dump.

Table 4.1 01.c

01	<i>#include &lt;stdio.h&gt;</i>
02	<i>int verify(){ printf("--- VERIFY ---\n"); }</i>
03	<i>int critical(){</i>
04	<i>FILE *fp;</i>
05	<i>char A[1024];</i>
06	<i>printf("--- CRITICAL ---\n");</i>
07	<i>fp = fopen("password", "r");</i>
08	<i>fscan(fp, "%s", A);</i>
09	<i>fprintf("%s\n", A);</i>
10	<i>fclose(fp);</i>
11	<i>}</i>
12	<i>int main(){</i>
13	<i>int (*f)();</i>
14	<i>char ch;</i>
15	<i>f = verify;</i>
16	<i>printf("Input : ");</i>
17	<i>scanf("%x", &amp;ch);</i>
18	<i>printf("ch : %x\n", ch);</i>
19	<i>f();</i>
20	<i>f = critical;</i>
21	<i>f();</i>
22	<i>}</i>

Table 4.2 overflows *f* with 3 lower bytes of *verify* and Table 4.3 overflows *f* with 3 lower bytes of *critical*.

For this program, we want to enforce a simple security policy: the program will access the real password file *password* only if *verify()* function has been called for proper verification. If a buffer overflow or similar attack bypasses *verify()* then only a fake password file *fakepass* is presented. This security policy could be used for privacy issues. This simple

policy also sets up a simple honeypot fake file (the attacker is not denied a password file, but is given access to a “*honeypot*” fake password file).

Table 4.2 Expected Execution

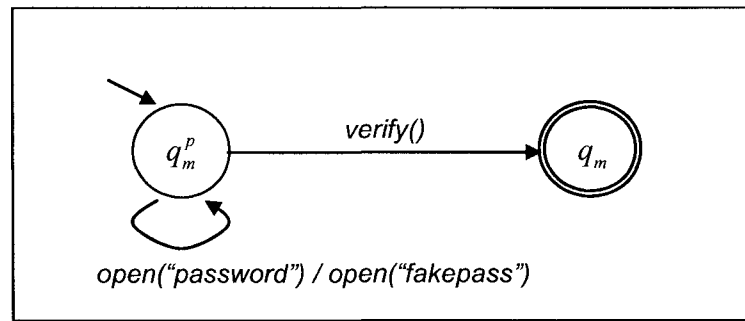
```
$ ls
01 01.c fakepass password
$ cat password
This.is.password.file.
$ cat fakepass
This.is.fakepass.file.
$ 01
Input : 04845800
ch : 0
--- VERIFY ---
--- CRITICAL ---
This.is.password.file.
```

Table 4.3 Exploited Execution

```
$ 01
Input : 0484700
ch : 0
--- CRITICAL ---
This.is.password.file.
--- CRITICAL ---
This.is.password.file.
```

Schneider's security automaton could also prevent the access to the real password file if the program calls `critical()` without first calling `verify()`. Bauer's edit automaton might be able to insert `verify()` action if possible or suppress the `critical()` action. Stream automaton, on the other hand, can force-change the program behavior totally. As shown in Figure 4.11, our automaton can force-change the `open("password")` to `open("fakepass")`.

We implemented the monitor for this security policy. Table 3.4 shows the runtime result of the program with the monitor for this security policy.



**Figure 4.11 Resource Access Control Monitor**

Table 4.4 Security Policy Enforced

```

$ zmonitor 01
Input : 04845800
ch : 0
Monitor Output : Verified
--- VERIFY ---
Monitor Output : Verified accessing critical area.
--- CRITICAL ---
This.is.password.file.
$ zmonitor 01
Input : 04847000
ch : 0
Monitor Output : Unverified accessing critical area.
--- CRITICAL ---
This.is.fakepass.file.
Monitor Output : Unverified accessing critical area.
--- CRITICAL ---
This.is.fakepass.file.

```

This example illustrates resource blind access control. It demonstrates a very general scenario where a malicious input changes the program behavior due to (unintended) programming errors and bad programming habits. In such a case, we could still enforce some very powerful and flexible security policies based on stream automata instead of simply terminating the program. The advantage is that we not only prevent the attacker from getting valuable information, but we also generate false information to delude the attacker.

### 4.5.2 Honey wall policy scenarios

Stream automata have access to a very powerful mechanism in program semantics transformation (Figure 4.7) to change the interaction between program and system fundamentally. We use a real application as an example to demonstrate its power: firewall.

Firewall is a popular security technique. Sitting between Internet and LAN, the normal firewall rejects unwanted traffic and allows desirable traffic. The real firewall policies differ from case to case.

We assume that the Internet is the system and LAN is the program. In this case, the firewall has full control of the program and traffic originating at LAN, but only partial control over the traffic from Internet. This is an ideal match to stream automata.

Normal firewall accomplishes the following functions:

1. Block the undesirable incoming traffic except for the LAN network service.
2. Screen the outgoing traffic, allowing most of it.

To formalize these two properties, we could use one automata to monitor the stream from LAN to Internet, outgoing traffic, and another one for Internet to LAN, incoming traffic. Both security automata and edit automata can handle these cases since computation involves accepting or rejecting an action stream. However, some coupled policies over two streams are not implementable by two independent automata.

Take botnet as an example. We want to set up a policy that prevents sensitive information leakage. “.win” is one command of certain botnet which retrieves victim's Windows serial number. We want to stop the traffic from the potential victim once the firewall detects an IRC packet which contains “.win” in the incoming stream. Note that as to which token (such as “.win”) actually serves as a trigger for the property action is an orthogonal issue. We are only concerned with the feasibility of such an enforcement action.

We want to show that two independent automata are not able to implement this policy. The trigger for the policy is part of the Internet to LAN stream. The action required of this

policy operates on the symbols LAN to Internet stream. Unless, a coupled automata to couple the two streams is feasible, which is not the case in security and edit automata, this policy is not implementable with two independent automata.

We consider several honeywall policies to further demonstrate the power of stream automata. Honeywall is a more complex firewall in front of a honeynet which is designed to trap/observe/monitor botnet behavior. Unlike a simple firewall which drops the malicious packets, a honeywall has a much more complex task. A honeywall needs to follow several basic principles:

1. Allow malicious traffic on incoming stream.
2. Allow the bot (LAN) connections to the server (Internet).
3. Disallow the bot (LAN) exploits of other machines.

For simplicity, we consider a countermeasure policy to a script kid using PhatBot downloaded from Internet. The reason we choose PhatBot is that its source code is publicly available. We embed a stream automaton at the firewall to defend/detect/monitor the script kid's actions. Notice that PhatBot has the ability to dynamically update its exploit codes. We do not intend to generate a complete set of firewall rules targeting PhatBot. That would be a too complex task for this chapter. We will take a few exploits from PhatBot as example for writing honeywall policies.

We consider bagle exploit in PhatBot for stream automata. The PhatBot zombies might send an exploit containing string A to the target. We also define a harmless string B, a string C with alert semantics for the LAN, and a string D for a fake serial number.

- |   |                       |
|---|-----------------------|
| A | 43FFFFFF303030010A... |
| B | 000000000000000000... |
| C | 'ALERT'               |
| D | '1234567890'          |

We will explore several firewall rules for this bagle exploit in order to compare the power of the three types of monitor automata: security automata, edit automata and stream automata. We will show that security automata and edit automata suffice for a limited set of firewall rules, while stream automata are capable of implementing a complicated honeywall.

### Truncation

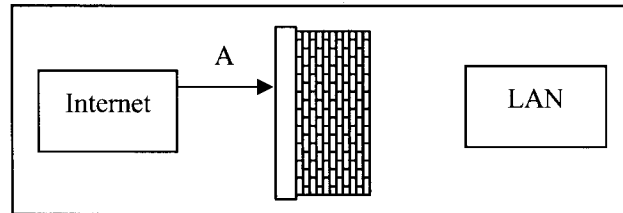
The simplest policy drops the connection if network traffic violates the honeywall policy. All three kinds of automata can satisfy this kind of policy. We skip this example.

### Suppression

The difference between suppression and truncation is that truncation will drop the whole connection but suppression only suppresses certain packets in the connection. Edit automata have this capability through suppression. Security automata's only enforcement mechanism is to terminate the offending program.

The stream automata to suppress the bagle exploit from PhatBot looks like:

$$q_m^s \xrightarrow{A!} q_m^s'$$



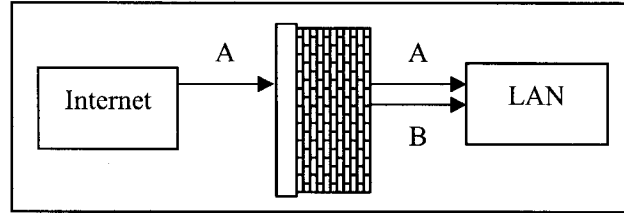
**Figure 4.12 Suppression**

### Insertion

Let us assume that the honeywall allows some of the malicious traffic to pass from Internet to LAN. It can, however, generate an alert token for the LAN, which can be interpreted with a different, benign semantics by the receiving LAN process. Security automata cannot generate an extra output, and thus are not capable of implementing this policy. Stream automata and edit automata can both satisfy this policy.

The stream automata to insert alert for the bagle exploit from PhatBot looks like:

$$q_m^s \xrightarrow{A/BA} q_m^s$$



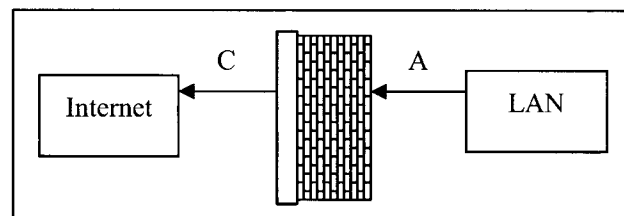
**Figure 4.13 Insertion**

### Metamorphosis

Now let us assume that there is one PhatBot bot in the honey net which is located in the LAN. The firewall has to permit the traffic between the bot and the server. It must however immunize or sanitize the exploit traffic from the bot. Immunization/sanitization here means that the traffic is modified to be harmless (no information leak or similar weakness). This policy cannot be implemented by edit automata only. The combination of suppression action and edit action of edit automata might be able to express similar properties, but the structural framework to couple the incoming and output streams is missing.

The stream automata look like:

$$q_m^s \xrightarrow{A/C} q_m^s$$



**Figure 4.14 Metamorphosis**

### Forcing

To further develop the preceding example, let us assume that the type of return packet expected by the bot can be predicted on the basis of outgoing token from the bot. The bot can be made to believe that it is operating in its desired context by the stream automata forcing the expected type of packet back to the LAN on receiving a malicious token in the outgoing stream. This behavior is benign with respect to the LAN honey net. Edit automata does not have the mechanism of reacting back to the program through a forced action. This limits its ability to implement such a policy.

The stream automata look like:  $q_m^s \xrightarrow{A/D} q_m^s$

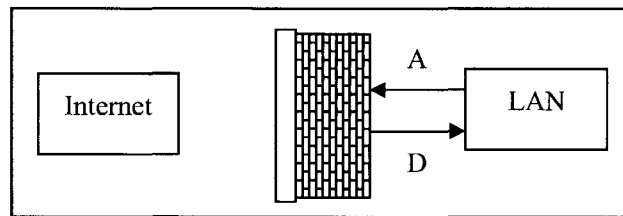


Figure 4.15 Forcing

### Two way forcing

This is a combination of insertion and metamorphosis. We assume that the firewall will send an alert packet to the LAN monitor, and it will also alter the outgoing packet.

The stream automata look like:  $q_m^s \xrightarrow{A/B,C} q_m^s$

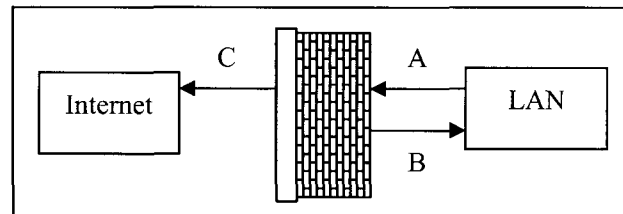
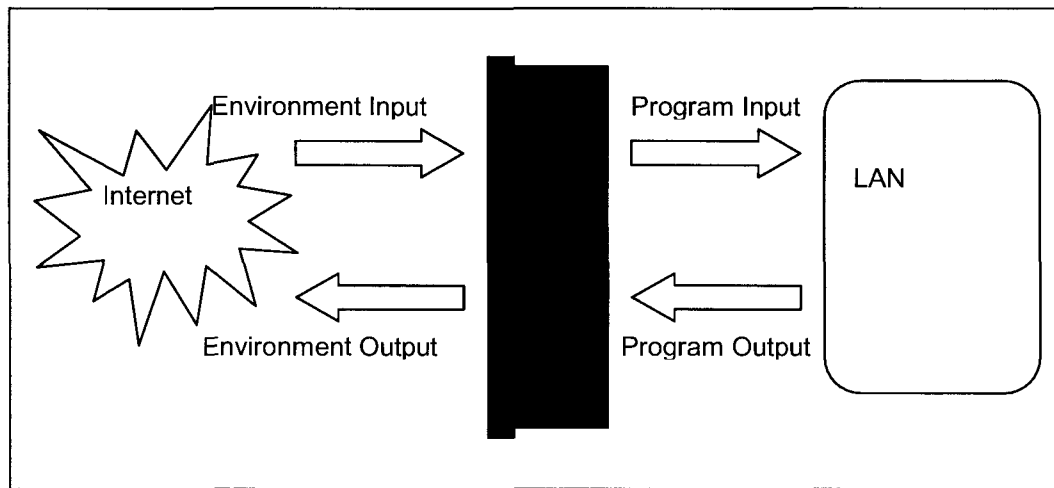


Figure 4.16 Two-way Forcing

### 4.5.3 Project SAFE

Project SAFE (Stream Automata Firewall Engine) implements a firewall/honeywall engine using formal semantics of security automata.

A transition is defined as  $State_1 \rightarrow State_2(EnvIn, EvnOut, ProIn, ProOut)$ .



**Figure 4.17 Traffic flow of SAFE**

#### 4.5.2.1 Honeywall case

In this session, we will demonstrate using a stream automaton to build a honey specially crafted for W32/Spybot.worm.gen.b.

W32/Spybot.wrom.gen.b is a special breed of W32/Spybot.worm.gen. W32/Spybot.worm.gen is a botnet worm which mainly spread through 2 ways [McAfee, 2003]:

1. Scanning subnets for systems already infected by sub7 or kuang2.
2. Camouflage itself with names like “porn.exe”, “Matrix Screensaver 1.5.scr” etc. and try to spread through p2p.

```

%{
#include <math.h>
#include "calc.h"
%}
NAME                \["\0-9A-Za-z_"]*\"
%%
{NAME}              {
                    //printf( "An identifier: %s\n", yytext );
                    return ID;
                    }
DEFINITION          return(DEF);
STATE               return(STATE);
INITIAL_STATE       return(INI_STATE);
ENVIRONMENT_INPUT   return(ENV_INPUT);
ENVIRONMENT_OUTPUT  return(ENV_OUTPUT);
PROGRAM_INPUT       return(PRO_INPUT);
PROGRAM_OUTPUT      return(PRO_OUTPUT);
TRANSITION          return(TRANSITION);
:                  return(COLON);
\{                 return(LEFT_BRACE);
\}                 return(RIGHT_BRACE);
,                 return(COMMA);
\(                 return(LEFT_PAREN);
\)                 return(RIGHT_PAREN);
->                return(ARROW);
[ \t]+            /* eat up whitespace */
\n                {
                    return RETURN;
                    }
.
%%

```

**Figure 4.18 Lexical**

After infecting the target system, the bot tries to connect to an external irc server on port 7000. Assume that the design goals of the honeywall are:

1. Allow the outgoing traffic of irc such that we could observe the botnet activities.
2. Disallow the sensitive information leaking out of the honeywall.
3. Disallow the attacks get out of the honeywall.
4. Falsify the output of bot such that the botnet master will not notice that the bot is after a honeywall.

```

%%
Input:
    /* Empty */
    | Input Line      {}
    ;
Line:
    RETURN
    | DEF RETURN
    | STATE Definition
    | INI_STATE Definition
    | ENV_INPUT Definition
    | ENV_OUTPUT Definition
    | PRO_INPUT Definition
    | PRO_OUTPUT Definition
    | TRANSITION RETURN
    | Name ARROW Name LEFT_PAREN Name COMMA Name COMMA Name
    COMMA Name RIGHT_PAREN RETURN
    ;
Definition:
    COLON LEFT_BRACE List RIGHT_BRACE RETURN
    ;
Name:
    ID
List:
    ID
    | List COMMA ID
    ;
%%

```

**Figure 4.19 Semantic**

The normal irc traffic of the worm is listed in Figure 4.21. Take a look at the following botnet commands (a sample of similar commands could be found at [Mich, 2003]):

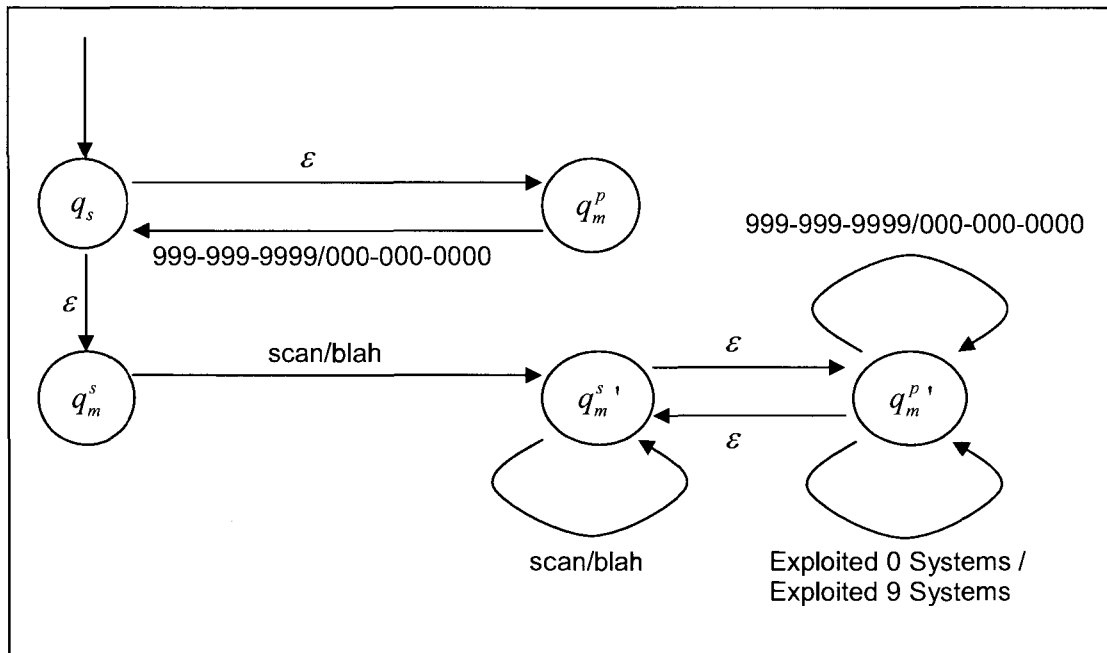
1. “.win”: Retrieve the windows key from the bot.
2. “.scan 123.1.1.1 80 1 portscan.txt”: Scan the network with ip address starting from 123.1.1.1 on port 80. The interval of the scan is 1 second and the result is stored in portscan.txt.
3. “.ntstats”: Get the information of the result of the scanning from the bot. The botnet expects a reply of “.ntstats :[NTScan]: Stats - Exploited 0 Systems.” Where 0 is the number of the systems that the bot successfully exploits.

```

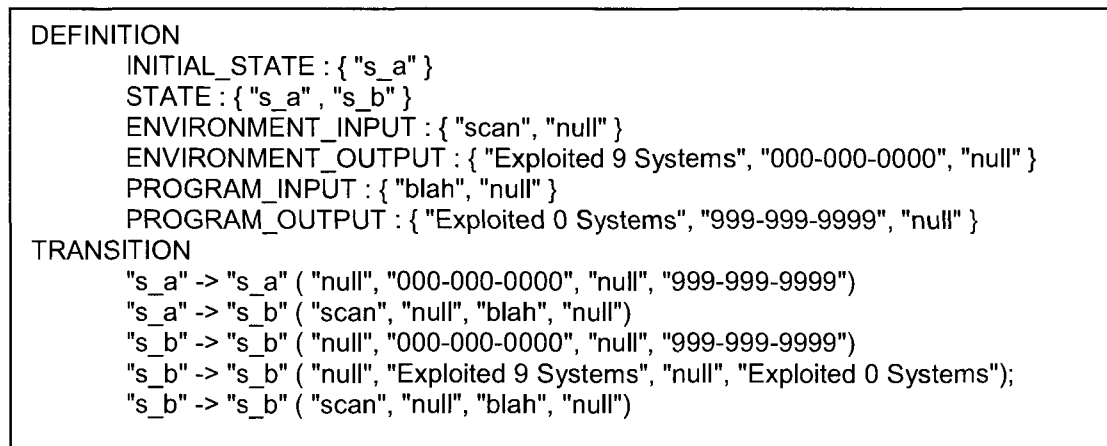
NICK leechbot-idrk
USER idrk "hotmail.com" "newshitz.dynu.com" :idrk
JOIN #h4x0r h4x
MODE uqzph +iR
:irc.m00.gov 332 leechbot-idrk :admin
:irc.m00.gov 332 leechbot-idrk :info
PRIVMSG :info :Version:. Leechbot r1.5b Private .cpu:. 2791MHz. .Ram:. 255MB total, 160MB
free 37% in use. .OS:. Windows 2000 (5.0, build 2195). .Uptime:. 0d 0h 45m. .Date:.
20:Oct:2005 .Time:. 09:22:36 .Current user:. Administrator .IP:. 192.168.2.2 .Hostname:.
Win2000Pro .Windir:. C:\WINNT\ .Systemdir:. C:\WINNT\System32\
:irc.m00.gov 332 leechbot-idrk :startlogger
PRIVMSG :startlogger :Send: :startlogger
:irc.m00.gov 332 leechbot-idrk :listprocces
PRIVMSG :listprocces :[System Process]
PRIVMSG :listprocces :System
PRIVMSG :listprocces :smss.exe
PRIVMSG :listprocces :csrss.exe
PRIVMSG :listprocces :winlogon.exe
PRIVMSG :listprocces :services.exe
PRIVMSG :listprocces :lsass.exe
PRIVMSG :listprocces :svchost.exe
PRIVMSG :listprocces :SPOOLSV.EXE
PRIVMSG :listprocces :msdtc.exe
PRIVMSG :listprocces :svchost.exe
PRIVMSG :listprocces :llssrv.exe
PRIVMSG :listprocces :regsvc.exe
PRIVMSG :listprocces :mstask.exe
PRIVMSG :listprocces :VMwareService.e
PRIVMSG :listprocces :inetinfo.exe
PRIVMSG :listprocces :dfssvc.exe
PRIVMSG :listprocces :explorer.exe
PRIVMSG :listprocces :VMwareTray.exe
PRIVMSG :listprocces :VMwareUser.exe
PRIVMSG :listprocces :mdm.exe
PRIVMSG :listprocces :idag.exe
PRIVMSG :listprocces :rundll.exe
PRIVMSG :listprocces :...3.Operation Completed!
() .10 [CTRL] (Changed window)
:irc.m00.gov 332 leechbot-idrk :scan 123.1.1.1 80 1 portscan.txt
:irc.m00.gov 332 leechbot-idrk :about
PRIVMSG :about :.Bot Versie:. Leechbot r1.5b Private, Leechbot r1.5b
:irc.m00.gov 332 leechbot-idrk :win
PRIVMSG :win :Windows key: 888-888-8888
:irc.m00.gov 332 leechbot-idrk :ntstats
PRIVMSG :ntstats :[NTScan]: Stats - Exploited 0 Systems.

```

**Figure 4.20 W32/Spybot.worm.gen.b Traffic**



**Figure 4.21 Stream Automaton for Worm Honeywall**



**Figure 4.22 SAFE Definition**

We want to implement the following stream automaton in figure 4.22 to enforce the honeywall design goals. Simply put, the stream automaton enforces that:

1. Change outgoing sensitive information (Windows key, represented by 999-999-9999, into a fake key, represented by 111-111-1111).

2. After receiving a “.scan” command, the honeywall will fake exploit statistics reply to the query “.ntstats”.

Finally, by using the stream automaton definition for SAFE in Figure 4.23, we finally implement the stream automaton to enforce honeywall policies for W32/Spybot.worm.gen.b.

## 4.6 Conclusions

This chapter proposes stream automata as a powerful tool for formal modeling of runtime monitors of *open systems* such as embedded systems. Open systems are reactive and hence interaction with the system is a key component of such systems. Kupferman *et al.* [Kupferman *et al.*, 1996] [Kupferman *et al.*, 2001] have proposed an approach for the verification of open systems using automata models that have both internal and external states. Our approach to run-time monitoring is inspired by this model. We view the monitor to be a process sitting between two processes, the system (say the operating system) and a program. The job of the monitor is to constrain both parties so that certain security policies are enforced (unlike earlier enforcement monitors that only constrain the program). We have developed a formal automata model of such monitors (which we call stream automata). A stream automaton has both internal states (where it performs internal computation or synchronization with another automaton) and system states (where it mediates between the system and the program). We have formalized the monitoring capabilities using SOS style operational rules for composing the monitor with its system. Using these rules we demonstrate how stream automata may be used for the formal modeling of existing enforcement monitors (such as Schneider's enforcement monitors and Ligatti *et al.*'s edit automata). We also have new rules for additional enforcement actions called metamorphosis, forcing, and two way forcing. We develop a Turing machine based model of stream automata based execution monitoring. This model seems to indicate that the power of stream automata is incomparable to that of security and edit automata. We can informally establish though

that stream automata are strictly more powerful than existing enforcement monitors. Stream automata can perform all actions of edit automata. It can, in addition, enforce new policies called resource blind access control, ordering consistency and barrier dependency. Many practical security monitoring examples are given to illustrate the power of stream automata. We also have implementation results to show that the overhead of run-time of such automata is minimal. In the future, we will explore composition and verification of run-time monitors using stream automata. We will also develop computability class hierarchy given by stream automata with the proposed TM model.

## CHAPTER 5. CONCLUSION AND FUTURE WORK

### 5.1 Conclusions

This dissertation explores methods to prevent overflow attacks. Most of the current research, both in compiling phase and execution phase, focuses on the protection of return address in the stack. However, advanced overflow attack which targets function pointer, integer, or other data structures could easily bypass the current protection.

Starting from this problem, we implement three different approaches:

1. Protect the potential function pointer set by encryption/decryption;
2. Enforce the program runtime behavior according to its design;
3. Enforce security policy on the interaction between program and system;

The potential function pointer protection work is one of the first research efforts to target the set of function pointers. We implemented a compiler patch to get the set of potential function pointers according to the source code analysis. At runtime the variable in the set will be protected in a way so that its value will be encrypted before going to the memory and decrypted before being used in the program. The key for the encryption/decryption is generated by a random function with seed of the current time and process id. A remote hacker will not succeed as the key is dynamic for each run. Both weak and strong encryptions are implemented for this approach. The weak encryption function is XOR and the strong one is RC5. The strong encryption/decryption guarantees that even if the hacker gets both clear text and encrypted text, the key is not derived easily. The system overhead is acceptable and is lower than 5% for XOR encryption scheme.

The protection of function pointers does not provide complete coverage for overflow attack targets. For example, the attacker could change the value of a critical variable to bypass security checks and run privileged codes. To conquer this problem we came up with control flow checking automata from anomaly protection point of view.

Control flow checking automata are generated according to the control flow graph of high level source codes. The automata are enforced during runtime by codes inserted during compiling phase. As the control flow checking automata work at a basic block level, the attacks that change the program control flow within the program will be detected immediately. The integer overflow attacks and part of the function pointer overflow attacks fall into this category. Control flow checking automata also generate a program level trust value indicating how well the runtime program behavior matches the design.

The preceding two implemented works are both at compiling phase and not suitable for legacy codes. To extend the formal monitoring work of control flow checking, we implemented stream automata monitor which monitors the program execution at the runtime.

Stream automata monitor screens the interactions between system and program. The system represents the runtime environment for the program. We assume the monitor has full control of the program and partial control of the system. The stream automata monitor could enforce security policies on different virtual levels. We implement the program level monitor to enforce control flow checking policy and detect overflow attacks. We also present a network level monitor to enforce firewall/honeywall policies and build honeypot to trap Internet attackers.

## **5.2 Future work**

The future work of the thesis includes:

1. The architecture support for encryption/decryption. The strong encryption/decryption will bring obvious system overhead. Specially designed hardware chips will decrease system overhead dramatically.
2. The architecture support for security monitor. As the runtime monitor has the privilege to monitor and control other programs, the monitor itself could easily be the target. Thus a separated runtime environment is necessary for the success of

monitoring. A similar framework in Microsoft's Next Generation Security Computing Base is one of the candidates which provide support for separated running.

3. Formal verification of complex stream automata. NuSMV [Cavada, 2006] is a tool to verify the property of general automata. A similar tool working for stream automata could help to verify complex program monitoring policies and firewall/honeywall policies formally.

## BIBLIOGRAPHY

- [Abdul-Rahman *et al.*, 1997] A. Abdul-Rahman and S. Hailes. A Distributed Trust Model. In *Proceedings of ACM New Security Paradigms Workshop*, 1997.
- [Austin *et al.*, 1994] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.
- [Baratloo *et al.*, 2000] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.
- [Bauer *et al.*, 2002] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.
- [Besson *et al.*, 2001] F. Besson, T. Jensen, D. L. Metayer, and T. Thorn. Model checking security properties of control flow graphs. In *Journal of Computer Security*, pages 9(3):217--250, 2001.
- [Biondi *et al.*, 2006] Philippe Biondi and Fabrice Desclaux. Silver Needle in the Skype. In *Black Hat Europe*, 2006. See <http://www.blackhat.com/html/bh-media-archives/bh-archives-2006.html#eu-06> (retrieved April 3 2006)
- [Biryukov *et al.*, 1998] A. Biryukov and E. Kushilevitz. Improved cryptanalysis of RC5. In *Advances in Cryptology - Eurocrypt '98*, pages 16-228, Springer Verlag, 1998.
- [Blaze *et al.*, 1996] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164-173, May 1996.
- [Blaze *et al.*, 1999] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed system security. In *Secure Internet Programming*:

- Security Issues for Distributed and Mobile Objects*, J. Vitek and C. Jensen (eds.), *Lecture Notes in Computer Science*, volume 1603, pages 183-210. Springer Verlag, 1999.
- [Burger *et al.*, 1996] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. *Technical Report CS-TR-96-1308*, University of Wisconsin, Madison, 1996.
- [Cavada, 2006] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, Marco Roveri, Simone Semprini and Andrey Tchaltsev. NuSMV. See <http://nusmv.iit.it/> (retrieved April 3 2006)
- [CERT, 2005] CERT/CC. Statistics 1988-2005. See [http://www.cert.org/stats/cert\\_stats.html](http://www.cert.org/stats/cert_stats.html) (retrieved April 3 2006)
- [Chen *et al.*, 2000] R. Chen and W. Yeager. Poblano: A Distributed Trust Model for Peer-to-Peer Networks. *Technical Report*, Sun Microsystems, 2000.
- [Conover *et al.*, 1999] Matt Conover and w00w00 Security Team. w00w00 on Heap Overflows. 1999. See <http://www.w00w00.org/articles.html> (retrieved April 3 2006)
- [Copeland, 1997] B. Copeland. The broad conception of computation. In *American Behavior Scientist*, 40:690-716, 1997.
- [Cowan *et al.*, 1998] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, PerryWagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.
- [Cowan *et al.*, 2000] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000.
- [Cowan *et al.*, 2003] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *12th USENIX Security Symposium*, pages 91-104, 2003.

- [Designer, 1997] Solar Designer. Non-executable user stack. 1997. See <http://www.ussg.iu.edu/hypermail/linux/kernel/9706.0/0341.html>, (retrieved April 3 2006)
- [Dor *et al.*, 2001] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In *Static Analysis Symposium, volume 2126 of Lecture Notes in Computer Science*, Springer Verlag, June 2001.
- [Fong, 2004] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2004.
- [Goldreich *et al.*, 1996] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. In *Journal of the ACM*, pages 43(3):431-473, 1996.
- [Hamlen *et al.*, 2003] K. Hamlen, G. Morrisett, and F. Schneider. Computability classes for enforcement mechanisms. *Technical Report Technical Report 2003-1908*, Department of Computer Science, Cornell University, 2003.
- [Harel *et al.*, 1987] D. Harel. Statecharts. a visual formalism for complex systems. In *Science of Computer Programming*, pages 8:231-274, 1987.
- [Harel *et al.*, 1996] D. Harel and A. Naamad. The statemate semantics of statecharts. In *ACM Transactions on Software Engineering and Methodology*, October 1996.
- [Hoare *et al.*, 1985] C. A. R. Hoare. Communicating Sequential Processes. *Prentice-Hall International*, 1985.
- [IBM, 2005] IBM. GCC extension for protecting applications from stack-smashing attacks. 2005. See <http://www.trl.ibm.com/projects/security/ssp/> (retrieved April 3 2006)
- [Jones *et al.*, 1997] Richard W.M. Jones and Paul H.J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG'97)*, May 1997.
- [Jsang, 1999] A. Jsang. An algebra for assessing trust in certification chains. In *Proceedings of the Network and Distributed Systems Security (NDSS'99) Symposium*, 1999.

- [Kaliski *et al.*, 1995] B.S. Kaliski Jr. and Y.L. Yin. On differential and linear cryptanalysis of the RC5 encryption algorithm. In *Advances in Cryptology - Crypto '95*, Springer-Verlag, pages 171-183, 1995.
- [Khare *et al.*, 1997] R. Khare and A. Rifkin. Trust Management on the World Wide Web. In *Proceedings of DIMACS Workshop on Trust Management in Networks*, 1997.
- [Knudsen *et al.*, 1996] L.R. Knudsen and W. Meier. Improved differential attacks on RC5. In *Advances in Cryptology - Crypto '96*, Springer-Verlag, 1996.
- [Kupferman *et al.*, 1996] O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. In *Computer Aided Verification - CAV*, 1996.
- [Kupferman *et al.*, 2001] O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. In *Information and Computation*, pages 164:322-344, 2001.
- [Lamport, 1977] L. Lamport. Proving the correctness of multiprocess programs. In *IEEE Transactions on Software Engineering*, pages 3:125-143, 1977.
- [Larochelle *et al.*, 2001] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.
- [Lemos, 2003] Robert Lemos. Counting the cost of Slammer. 2003. See [http://news.com.com/Counting+the+cost+of+Slammer/2100-1001\\_3-982955.html](http://news.com.com/Counting+the+cost+of+Slammer/2100-1001_3-982955.html) (retrieved April 3 2006)
- [Lhee *et al.*, 2002] Kyung-suk Lhee and Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [Li *et al.*, 2003] Ninghui Li, Benjamin N. Grosz, and Joan Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. In *ACM Transaction on Information and System Security (TISSEC)*, pages 1-42. 2003.

- [Ligatti *et al.*, 2003] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. In *International Journal of Information Security*, 2003.
- [Ligatti *et al.*, 2005a] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. In *International Journal of Information Security*, pages 4(1-2):2-16, February 2005.
- [Ligatti *et al.*, 2005b] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, September 2005.
- [Lynch *et al.*, 1989] N. A. Lynch and A. Tuttle. An introduction to input/output automata. In *CWI Quarterly*, pages 2(3):219-246, 1989.
- [Maraninchi *et al.*, 1996] F. Maraninchi and N. Halbwachs. Compositional semantics of nondeterministic synchronous languages. In *Proceedings of the European Symposium on Programming (ESOP)*, LNCS Vol. 1058, pages 235-249, 1996.
- [Maraninchi *et al.*, 2001] F. Maraninchi and Y. Remond. Argos: An automaton-based synchronous language. In *Computer Languages*, pages 27:61-91, 2001.
- [McAfee, 2003] McAfee©. W32/Spybot.worm.gen. 2003. See [http://vil.nai.com/vil/content/v\\_100282.htm](http://vil.nai.com/vil/content/v_100282.htm) (retrieved April 3 2006)
- [Milner, 1989] R. Milner. Communication and Concurrency. *Prentice Hall International*, 1989.
- [Mich, 2003] Mich. Spybot 1.2c. 2003. See <http://www.megasecurity.org/trojans/s/spybot/Spybot1.2c.html> (retrieved April 3 2006)
- [Moore *et al.*, 2001] David Moore and Colleen Shannon. The Spread of the Code-Red Worm (CRv2). 2001. See [http://www.caida.org/analysis/security/code-red/coderedv2\\_analysis.xml](http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml) (retrieved April 3 2006)

- [Muchnick, 1997] S. Muchnick. Advanced Compiler Design Implementation. *Morgan Kaufmann*, 1997.
- [Oh *et al.*, 2002] N. Oh, P.P. Shirvani, and E.J McCluskey. Control-flow checking by software signatures. In *Reliability, IEEE Transactions on Computers*, March 2002.
- [One, 1996] Aleph One. Smashing The Stack For Fun And Profit. 1996. See <http://www.phrack.org/show.php?p=49&a=14> (retrieved April 3 2006)
- [Plotkin, 1981] G. D. Plotkin. A structural approach to operational semantics. *Technical Report DAIMI FN-19*, Computer Science Department, Aarhus University, 1981.
- [Rivest, 1995] R.L. Rivest. The RC5 encryption algorithm. In *CryptoBytes*, Spring 1995.
- [Ruzzo, 1981] Walter L. Ruzzo. On Uniform Circuit Complexity. In *Journal of Computer and System Sciences*, pages 22:365-383, June 1981.
- [Satoh *et al.*, 2003] A. Satoh and S. Morioka. Hardware-Focused Performance Comparison for the Standard Block Ciphers AES, Camellia, and Triple DES. In *Proceedings of ISC 2003*, LNCS 2851, pages 252-265, Springer Verlag, 2003.
- [Schneider, 2000] Fred B. Schneider. Enforceable security policies. In *Information and System Security*, pages 3(1):30-50, 2000.
- [Schneider *et al.*, 1999] F. Schneider, S. Bellovin, M. Branstad, J. Catoe, S. Crocker, C. Kaufman, S. Kent, J. Knight, S. McGeady, R. Nelson, A. Schiffman, G. Spix, and D. Tygar. Trust in Cyberspace. *National Academy Press*. 1999.
- [Selcuk, 1998] A. A. Selcuk. New results in linear cryptanalysis of RC5. In *Proceedings of 5th International Workshop on Fast Software Encryption*, Springer Verlag, 1998.
- [Sekar *et al.*, 2001] R. Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144-155, Oakland, California, U.S.A., May 2001.

- [Spafford, 1988] Eugene H. Spafford. The Internet Worm Program: An Analysis. 1988. See <http://protovision.textfiles.com/100/tr823.txt> (retrieved April 3 2006)
- [Spafford, 1989] Eugene H. Spafford. The Internet Worm Incident. In *Proceedings of the 1989 European Software Engineering Conference (ESEC 89)*, 1989.
- [Statecharts, 1987] D. Harel. Statecharts : a visual formalism for complex systems. In *Science of Computer Programming*, pages 8:231-274, 1987.
- [Stats, 2006] Internet World Stats. Internet Usage Statistics - The Big Picture. See <http://www.internetworldstats.com/stats.htm> (retrieved April 3 2006)
- [Teuscher *et al.*, 2002] Christof Teuscher and Moshe Sipper. Hypercomputation: hype or computation. In *Communications of the ACM*, pages 45(8):23-24, 2002.
- [Tyagi *et al.*, 2000] A. Tyagi and G. Lee. Encoded Program Counter: Self Protection from Buffer Overflow Attacks. In *Proceedings of International Conference on Internet Computing (IC '2000)*, pages 387-394, June 2000.
- [Tyagi, 2003] A. Tyagi. Encoded Program Counter: Self protection from Buffer Overflow Attacks. *Final Project Report for DARPA/AFRL ATIAS BAA00-06-SNK*, 2003.
- [Vendicator, 2000] Vendicator. Stack Shield. 2000. See <http://www.angelfire.com/sk/stackshield/> (retrieved April 3 2006)
- [Viswanathan, 2000] Mahesh Viswanathan. Foundations for the Run-time Analysis of Software Systems. *Ph.D. thesis*, University of Pennsylvania, December 2000.
- [Wagner *et al.*, 2000] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities, In *Network and Distributed System Security Symposium*, February 2000.
- [Wagner *et al.*, 2001] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, Oakland, California, U.S.A., May 2001.

- [Whitehouse, 1996] Whitehouse. Background on Clinton-Gore Administration's Next-Generation Internet Initiative. See <http://clinton6.nara.gov/1996/10/1996-10-10-background-on-next-generation-internet-initiative.html> (retrieved April 3 2006)
- [Wiki, 2006] See <http://en.wikipedia.org/> (retrieved April 3 2006)
- [Younan *et al.*, 2004] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. *Technical Report CW386*, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.
- [Zhang *et al.*, 2004] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Hardware Supported Anomaly Detection: down to the Control Flow Level. In *CERCS Technical Reports*, 2004. See <http://www.cercs.gatech.edu/tech-reports/tr2004/git-cercs-04-11.pdf> (retrieved April 3 2006)
- [Zhu *et al.*, 2004] G. Zhu and A. Tyagi. Protection against indirect overflow attacks on pointers. In *Information Assurance Workshop, 2004. Proceedings. Second IEEE International*, April 2004.

## ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me for all these years.

First, I want to thank my mentor – Dr. Akhilesh Tyagi. Dr. Tyagi is a great professor who is not only knowledgeable but also patient for his students. Whenever I have ideas that I want to discuss, he is always there for me. Without his guidance and help over last five years, there is no way that this thesis could ever be completed.

Second, I want to thank the professors who contributes to this paper in various ways: Dr. Partha Roop for the discussion on Stream Automata; Dr. Soma Chaudhuri for the discussion on Input/Output Automata; Dr. Gary Leavens for the discussion on Propagation; Dr. Johnny S. Wong, Dr. Wallapak Tavanapong and Dr. James A. Davis for suggestions on my final paper.

Third, I want to thank my friends, my family for supporting me all these years. To my parents, Zhichao and Congfu, my sister, Hua: thanks for your support for me. Also to Yulong Li, thanks for always believing in me and encouraging me for almost twenty years.

Last but not the least: I want to thank my wife, Weiyi. In short, I am lucky to have you with me. Thanks.