2010

Efficient Validation of Control Flow Integrity for Enhancing Computer System Security

Yong-joon Park
Iowa State University

Efficient validation of control flow integrity for enhancing computer system security

by

Yong-Joon Park

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

Securing today's computer system is an urgent and difficult problem. Recent trends show that today's computer system attackers have economic incentives, making attacks likely to continue. For example, attackers are selling computing assets on large networks of compromised machines, called botnets. Spammers and organized crime syndicates use botnets for activities such as on-line extortion, delivering unsolicited email and advertisements, and identity theft through fraudulent web sites. As attackers can make money by compromising vulnerable hosts and renting them out, skilled attackers will continue to find new and ingenious methods to break into computer systems. According to the annual CSI/FBI computer crime and security survey of U.S. businesses, computer security compromises continue despite the best efforts of current practices of security [32]. For instance, in the year 2006, although most companies used and promptly updated computer security products, more than 72% of the companies surveyed experienced at least one computer security incident. For example, 98% of the companies used firewalls, 97% used Anti-virus software, and 78% used Anti-spyware software. This survey clearly shows that the current state-of-the-art computer security products could not prevent all computer security incidents.

Among various types of computer security intrusions, executing malware (MALicious softWARE) or malicious code on a victim system is the most favorable and widely spread attack method used by attackers [32]. The unauthorized code execution is the most critical attack; attackers can achieve arbitrary goals with arbitrary code executions. There are largely two ways for attackers to execute malware on a victim system [66]: computer viruses and Internet worms. One way to execute malicious code is through Internet worms which exploit vulnerabilities of programs to execute malware and propagate by themselves. By exploiting program bugs, attackers can alter the execution or control flow of program so that malicious code or program is executed instead of the original code. The other way of executing malicious code is through viruses or Trojan horses. Attackers may deceive users into downloading and executing malware such as virus and Trojan horse by injecting malicious code on the Internet web site or attaching malware in e-mails. In such a way, malware can be executed with or without the users' consent.

In order to secure computing system from malware execution, there are a number of different types of security products commercially available. Largely, they can be categorized into two types of defense mechanisms on the host side: misuse-based and model-based [66]. Each of them uses different system information to mitigate computer security attacks. The most popular misuse-based security product is anti-virus software, which generates signatures from the manual analysis of malware. The binary of malware is scanned structurally and reverse engineered to find an invariant to be used as a signature. Malwares are detected by matching existing signatures with the accessed binary stream. Although the method has been used effectively on production systems, it only prevents known malwares and has difficulty detecting newly created malware, including polymorphic malware such as encrypted code and run-time packed malware. In general, the execution of malware through the social engineering can be prevented by the combination of the anti-virus software and the user's cautions. However, recent malwares use evading techniques to bypass the anti-virus software as well as users' cautions.

Model-based systems operate by comparing the observed behavior of an application to models of normal behavior, which may be derived automatically via static analysis or learned by analyzing the run-time behavior of programs. Attacks are detected when the dynamic behavior of the monitored program deviates from the normal behavior captured by the model. In contrast to the misuse-based approaches, model-based schemes have the advantage of being able to detect ingenious attacks. Most model-based intrusion detection systems monitor the sequence of system calls issued by an application, mostly taking into account some execution state which may include a current program counter(PC). However, it may be still vulnerable to ingenious attacks that craft the program run-time states, since the states information is obtained from the memory subsystem. Additionally, it may still be vulnerable to mimicry attacks due to its limited precision; some of the model-based systems construct a state for the system calls or function calls.

1.1 Overview of Thesis Work

In order to secure computing systems from the execution of malware, This thesis study both misuse- and model-based systems that utilize and validate fine-grain control flow information at program run-time. Since the dynamic control flow information is a good source for representing states of a running program, it can be used for detecting anomalies in program behavior. Hence, for the model-based approach, we have built prototype systems to validated the run-time control flow information with pre-defined control flow information to detect anomalies in the systems. Also, control flow information is an unique characteristic of a program that can be considered as DNA of the program. It is hard, if not impossible, for two programs to exhibit the identical control flow behavior. Therefore, for the misuse-based approach, we constructed malware signature from its control flow information and detected malware by matching the control flow behavior of programs with that of malware at run-time.

1.1.1 Control Flow Validation against Control Flow Attacks (CFAs)

One of the most critical threats to buggy code is a control flow attack (CFA). CFAs have been frequently exploited to make the first breach to computer system security. Many service programs have unintended vulnerabilities in their dynamic execution environments, which are explored by attackers to redirect their control flow to a harmful code. Once the adversary has successfully obtained the control flow of the victim system, he or she may be able to perform various malicious actions such as stealing and/or corrupting important data in the victim machine. Control flow attacks do not only cause breaches to individual systems. More often, the adversary utilizes the victim as a zombie or a bot to launch large-scale attacks such as DDOS (distributed denial of service) attacks. Unfortunately, vulnerabilities leading to CFAs are inherent and pervasive in software as reported by the National Vulnerability Database (NVD).

There are two fundamental assumptions that make the current processor architecture vulnerable for CFAs:(1) memory is trusted and (2) control flow tracking is blindfolded without validity a check. Recent studies, XOM and AEGIS, try to encrypt all memory bound data to enforce a secure running environment, but such an approach has been shown to be ineffective for sophisticated CFAs. Blanket encryption of all data is not able to protect control data corruption by legitimately encrypted control data set by compromised normal data. Also generic protection mechanisms are proposed with hardware modifications. Dynamic Information Flow Tracking [65] and Minos [25] modifies hardware to efficiently track down spurious data to prohibit using this data as control data. However, these solutions require major hardware modifications that may not be adopted quickly in practice.

Control flow validation, another generic defense mechanism against CFAs, is an effective approach to prevent CFAs. In this approach, each control flow transfer is validated at run-time with the intended control flow graph of the program. Rather than finding a solution from a high level of application source code or even behavior specifications, control flow validation focus on each individual instruction at run-time. In the current processors, control flow transfer at the machine instruction level is blindfolded without validity check. Processors blindly follow the program counter to fetch and execute instructions. We believe this a fundamental deficiency in the hardware that causes the endless chase of software vulnerability, its exploitation, and its patch. Hence one of our goals is to mitigate software bugs that leads to control flow interception from critical security issue to simple software bugs by dynamically validating the control flow of an application program to detect anomalies.

Program Shepherding [44] has been proposed to run vulnerable programs on a dynamic code optimization system, which is called DynamoRIO. The system monitors every indirect branches to enforce security policies on program control transfers. However, it incurs large performance overhead up to 7.5 times slowdown due to the inefficient monitoring method. Abadi et al. [8]. proposed the IRM (inline reference monitoring) defense mechanism that instruments the program binary code in a manner that validates every indirect control transfer by inserting validation stubs before the control instruction or replacing control instructions with security code. It requires a sophisticated binary instrumentation tool to identify legitimate targets and insert the security code. The binary instrumentation technique has been used in various academic area of study, but changing the original code may not be a favorable approach to protect the system in practice. The binary instrumentation may raise robustness and compatibility issues as well as legal concerns in practice.

Therefore we propose efficient and non-invasive dynamic control flow validation system implemented with existing hardware features in commercial micro-processors. In contrast to the previous approaches, middleware and IRM, the proposed control flow validation does not interpret the instruction stream nor modify the source image. By leveraging the existing hardware features in commercial processors, the control flow validation mechanism can be implemented transparently to the target application. Furthermore, since the state of hardware facility is transparent to applications and cannot be crafted by other entities in the computer system; the implementation provides strong guarantees that a successful CFA is extremely difficult, if not impossible, under the control flow validation system. We have built the prototype systems called, IBMON (Indirect Branch MONitor), on three different micro-processors. The prototype systems effectively detect various CFAs and exhibit the performance overhead which ranges from 0% to 33.5% an average of 8.3% for SPECINT2000 benchmarks. For other server benchmarks, the performance degradation of IBMON is negligible.

1.1.2 IBF-Cache: Hardware supports for Control Flow Validation

Control Flow Validation is an effective approach to detecting and preventing CFAs. Although IBMON is the most efficient control flow validation system among existing control flow validation systems, it still incurs non-negligible performance overhead, up to 33.5% for one of the SPECINT2000 benchmarks. Therefore we also examine effective yet minimal hardware support which can provide seamless control flow validation environment. IBF-Cache (Indirect Branch Filter Cache) is the cache design that effectively reduces the frequency of control flow validations by exploring the temporal locality of control flow transfers in programs. In various performance tests, based on both trace-based and cycle accurate simulations, IBMON with IBF-Cache shows negligible performance overhead on all SPECINT2000 benchmarks and other server benchmarks.

1.1.3 Control Flow Inspection for Detecting Polymorphic Malware

The conventional approach to detecting malwares is based on static scanning of malware signatures in the computer system files and memory. It is effective for many existing malwares, but it is very limited for malwares that disguise the malicious code with run-time packing and encryption.

Recently, packed malwares, which use one of the obfuscation techniques, impose a significant problem in malware analysis and detection. Such programs consist of a decompression or decryption routine that extracts the transformed payload from the memory and then executes it. Hence a malware detector has to scan malware after the decryption or decompression has been done. Commercial anti-virus software has a limited capability to detect such malwares. If the decryption routine or unpacking routine is known, it tries to decrypt or unpack the malware and scans the result. However, it is impossible or hard to apply all known compression or encryption algorithms. Some generic unpacking through emulation techniques can detect packed malwares. However emulation places a time limit on the execution of packed program and is restricted by the emulation environment. Hence some malware which uses significant amount of time to unpack its payload, can bypass the emulation-based detection method.

Figure 1.1 shows the statistical distribution of packers used by malwares in the year 2006 according to Panda security [15]. The new generation of malwares are increasingly using these run-time packers. According to a recent study [15], 78% of malwares used run-time packing method. Currently, new run-time packers are created from existing

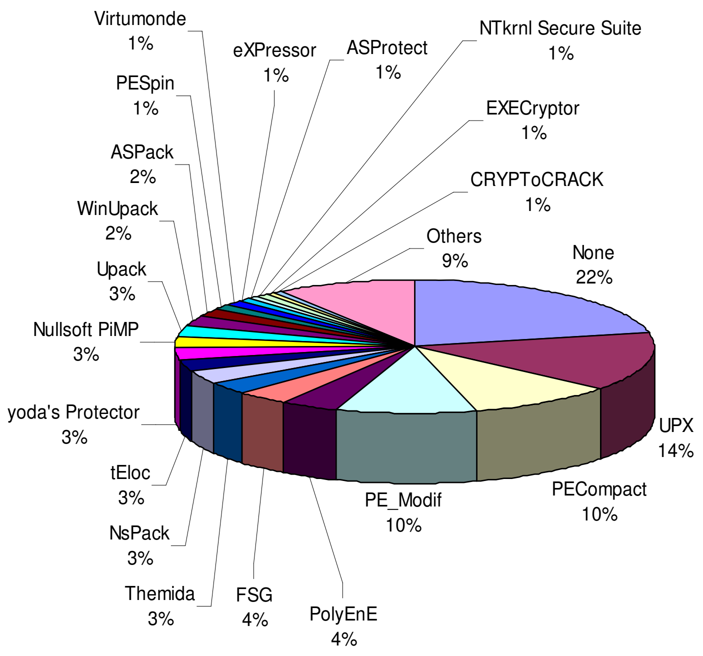


Figure 1.1 Statistic of techniques that malware used in year 2006

ones at a rate of 10 to 15 per month. Hence it may not be effective to simply disallow the execution of run-time packed program due to the lack of ability to identify run-time packers. As a result, malware writers have a large selection of tools to pack their malware. Consequently, an old malware may appear to be a new malware even though the malware signature is known for traditional detection mechanism.

Unfortunately, according to [49], most of anti-virus software cannot deal with run-time packers. There are three main issues: detection rates, false alarms and crashes/speed problem. The detection rate varies from 10% to 80% for commercial anti-virus software. Moreover, run-time packed malwares also increase scanning time of anti-virus software by a factor of 1.5 and 10 times and raise false alarms or crash during scanning.

In order to detect polymorphic malware, a number of different generic unpackers are proposed. Some of the generic unpackers use the emulation environment or single step execution to identify the starting of the plain malware. However, these approaches have a few drawbacks. The emulation or single step execution imposes high performance overhead to detect the execution of the plain malware so that it may be difficult to use them in practice. Since the heuristic used for identifying the stating location of the plain malware is specific to a subset of run-time packers, some of the them cannot analyze the plain malware.

Another approach to detect polymorphic malware is to detect modified code execution. It enforces the program execution policy that raises alarms upon code execution from a written page. Since polymorphic malware generates plain malware dynamically and stores it into writable page, the plain malware execution violates the policy. The mechanism can identify and mark suspicious memory pages that contains the plain malware. Later, when a dangerous system call is invoked, anti-virus scanner is activated to scanning the marked memory pages. However, malwares can try to bypass the scanning by using other code obfuscating techniques, such as hiding entry point of malware, mixing code and data, and so on, in addition to run-time packing and encryption. For those malwares, anti-virus software still has to perform advanced malware detection and expensive operations such as disassembling or emulation to detect those malwares [67]. These operations will incur high performance penalty.

Therefore this thesis also explores a possible solution to effectively detect polymorphic malware. Since control flow information can be considered a unique characteristic of a program, utilizing control flow information has a potential benefit to existing misuse-based defense mechanisms. Control flow inspection is an effective mechanism to detect polymorphic malware that evades the static scanning of anti-virus software by hiding its binary through polymorphic methods. Since such a malware reveals its original code during run-time, the traditional system-call monitoring system with dynamic control flow information is able to detect such a malware during run-time.

The fine-grain control flow validation system can also inspect and record the recent control flow of a target application. The hardware debugging features can be extended to be used for inspection of recent control flow transfers. Hence, this thesis also proposes RCFI (Recent Control Flow Inspection) system that constructs malware signatures from their control flow information and detect ingenious malwares at run-time by matching the control flow signature. RCFI system can detect polymorphic malware but shows limitations on detecting metamorphic malware that changes instructions instead of obfuscating image.

1.2 Thesis Structure

This thesis proposes to use existing hardware feature to implement efficient control flow validation systems. The thesis demonstrates the generality and practicality of the approach by implementing and testing prototype systems on various system platforms. The thesis also propose a hardware support to provide seamless control flow validations. It also discusses control flow inspection method to detect polymorphic malware.

Chapter 2 describes more details on CFAs and the detection mechanisms. First, details of CFAs are examined and related works against CFAs are described. Chapter 3 presents prototype system, IBMON (Indirect Branch MONitor), and detailed analysis on its effectiveness as well as performance impact. Chapter 4 describes a hardware support, IBF-Cache, that provides seamless dynamic control flow validation. The thesis presents details of IBF-Cache design and implementation. It also demonstrates the efficacy of the hardware support by presenting the result of both cycle accurate simulation and trace-based simulation. Chapter 5 describes a run-time malware detection system, RCFI (Recent Control Flow Validation), that uses control flow information to detect polymorphic malware at run-time. It presents the recent malware trend and other approaches to detect polymorphic malware. Chapter 5 also gives an insight into RCFI system and offers detailed analysis of its effectiveness and performance. Lastly, chapter 6 concludes the dissertation by emphasizing the efficiency and the effectiveness of IBMON and the novelty of IBF-Cache as well as the benefit of the RCFI, which is dynamic control flow inspection system.

2 Background

In the classical view of computing system security, there are four kinds of threats to the security of computing system in general: interruption, interception, modification, and fabrication. In an interruption, an asset of the system becomes lost, unavailable, or unusable. For example, attackers may destroy a hardware device, erase a program or a data file. Attackers also may launch the Distributed Denail of Service (DDoS) to interrupt the network services. An interception means that some unauthorized party has gained access to an asset. Examples of these types of attacks are illicit copying of program or data files and wiretapping to obtain data in network. In a modification, attackers not only access but also tamper with an asset. For example, someone might change the values in a database, or alter program so that it performs an additional computation, or modify data being transmitted electronically. Finally, an unauthorized party might fabricate counterfeit objects on a computing system. The intruder may insert spurious transactions to a network communication system or add records to an existing database.

There have been numerous countermeasures to remove each of these threats individually on the part of hosts. These include applying strict access control and encryption of data or network transaction; using of intrusion detection system, intrusion analysis and system monitoring tools. These countermeasures are effective on some types of security attacks. However, recent trend of security attacks can neutralize all the countermeasures by taking total control of a victim system. These attacks exploit software bugs or vulnerability to achieve their malicious goals. Software bugs or vulnerabilities cannot be only simple glitches in computing systems but also critical security problems, surrendering total control of a system. Therefore, in order to circumvent security countermeasures, the ultimate goal of most security attacks is to gain full control of the victim system by taking control of vulnerable privileged programs.

2.1 Control Flow Attack and Computer System Security

One of the most critical computer security threats is an attempt of gaining unauthorized access to a computer system. It allows attackers to achieve malicious goals including objectives from four classical threats. A process or a running instance of a program is the sequential execution of machine instructions. The process is made up of multiple basic blocks that have one entry point and only one exit point. The dynamic sequence of these blocks is arranged by control instruction such as branch instruction, which is located at the end of the basic block. The transition between basic blocks or reentering the same block is called control flow. Control flow attack changes intended control flow and redirect it to malicious code or existing code in memory to execute arbitrary code to achieve the attackers' malicious goal. Control flow attack or execution flow hijacking is widely used and a powerful method for gaining unauthorized access to a computer system.

2.1.1 Control Data

There are several different types of control data in a process. In order to hijack the control flow of a process, attackers change the control data in the process' writable address space. In this section, we examine control data that can be overwritten by attackers to achieve CFA.

2.1.1.1 Return Address in Run-time Stack

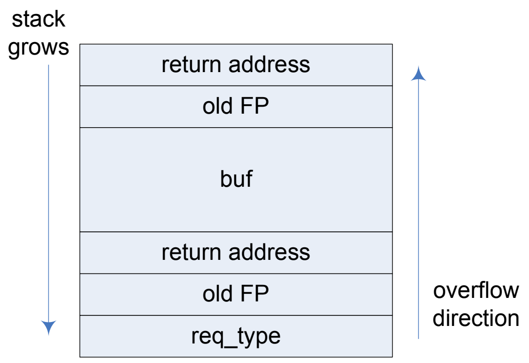
Run-time stack is a stack data structure that stores information about the active procedure or the function of a program. It contains procedure linking information such as return address and previous stack base frame information. Also, it contains parameters for the called function and local variables. Return address is one of the control data in a process. When a function is called, prior to executing the first instruction of the called function, a processor pushes the return address, which is the address of the next instruction of the function call in the caller, into the run-time stack. When the called function returns, a processor pops the top of the run-time stack to transfer back the control to the caller. Although a processor tracks all the call-return sequences with its branch predictor, called a return address stack, or a return address buffer, the processor relies on the information in the run-time stack due to precision of the branch predictor. Since the run-time stack is writable memory area of a process, control data in run-time stack is often the victim of CFA. The classical stack smashing buffer overflow attack is a well-known CFA that alters the return address to redirect the control flow of the victim process to a malicious code.

Figure 2.1(b) shows the memory layout of the run-time stack. Traditionally, run-time stack grows downward in the process memory for the virtual memory management purpose. In other words, when data is pushed into the stack, the address of the top of stack is reduced by the size of the data. On the other hand, when data is popped from the stack, the address of the top of stack is increased by the size of the data. However the buffer or array in the stack grows from lower address to higher address. Therefore, overflowing the buffer in the run-time stack may result in overwriting the return address.

Figure 2.1 uses an example to demonstrate the mechanism of a typical buffer overflow attack. In the program shown in Figure 2.1(a), function the `process_input` accepts a request from an Internet connection and calls the `parse_input` to interpret the request. In a typical system, the run-time stack stores the local variables and *return address* for functions, as shown in Figure 2.1(b). This program runs correctly for a normal request whose request type is less than 32 bytes plus one null character. An over-length `type` in a benevolent request usually crashes the program but does not cause a security breach. However, an adversary can construct a request such that the buffer is filled with malicious code and the return address is overwritten with a pointer to the malicious code. At the function return point, the changed return address is fetched and used to direct the control flow. The control flow executes the malicious code on the victim

```
void process_input(FILE* socket) {
  char buf[4096];
  fgets(buf, 4096, socket);
  parse_input(buf);
  ...
}
void parse_input(char *buf) {
  char req_type[33];
  sscanf(buffer, "%s", req_type);
  ...
}
```

(a) A program that contains buffer overflow vulnerability.



(b) Part of the run-time stack frame of the program.

Figure 2.1 Typical Buffer Overflow Error.

system Then, the attack code can carry out any of the threats desired by the adversary with the privilege of the victim program.

2.1.1.2 Frame Pointer in Run-time Stack

Stack smashing attacks directly modify the control data, return address in run-time stack. Frame pointer overwriting attacks are another type of control flow attack that occurs in the run-time stack. In this section, the stack frame in the x86 architecture is examined to describe frame pointer overwriting attacks.

The stack is typically divided into frames. Each stack frame can contain local variables, parameters to be passed to another procedure, and procedure linking information. The stack frame base pointer, EBP register, identifies a fixed reference point within the stack frame for the called procedure. To use the stack frame base pointer, the called procedure stores the previous EBP on the stack. The stack frame base pointer then permits easy access to data structures passed on the stack, to the return address, and to local variables added to the stack by the called procedure.

Prior to branching to the first instruction of the called procedure, the *CALL* instruction pushes the address in the EIP register into the current stack. Upon returning from a called procedure, the RET instruction moves the return address from the stack back into

```
pushl %ebp                movl %ebp, %esp
movl  %esp, %ebp          popl %ebp
subl  $n,   %esp          ret
       (a)                        (b)
```

Figure 2.2 (a) function prologue (b) function epilogue

the EIP register. The execution of the calling procedure then resumes. The processor does not keep track of the location of the return address. It is up to the programmer to insure that the stack pointer is pointing to the return address on the stack before issuing a *RET* instruction. Therefore the stack frame base pointer is used to help keep track of the previous stack frame.

The segment of the code in figure 2.2 shows the function prologue and epilogue. The function prologue is the first code segment after *CALL* instruction is executed. The function epilogue is the code segment executed right before the execution of *RET* instruction. The first instruction in the function prologue pushes the old frame pointer into the top of the run-time stack to store the previous stack frame base pointer. After that, current top of the stack is stored into EBP register to track current stack frame. *$n* is the size of the current stack frame which includes the size of the local variables. ESP is adjusted by subtracting the size of stack from current ESP value. During the function epilogue, the stack pointer, ESP, is restored by moving current stack base frame pointer, EBP, to the ESP. After that previous base frame pointer value is restored by popping the top of the stack. Consequently ESP points return address before the execution of *RET* instruction.

Since the processor does not require that the return address to return back to the calling procedure, prior to executing the *RET* instruction, the return address can be manipulated in the software to point to any address in the current procedure. By altering stack frame base pointers in the run-time stack, attacker can manipulate the location of the return address. Consequently, attacker can change the return address of the calling procedure without directly modifying the return address in the run-time

```
1.    void fun1(const char *str){...}
2.    void main(int argc, char **argv){
3.       static char buff[128];
4.       static void (*fptr)(const char *str);
5.       fptr = &fun1;
6.       strncpy(buff, argv[1], strlen(argv[1]));
7.       (void)(*fptr)(argv[2]);
         ...
      }
```

Figure 2.3 Example of Function Pointer

stack.

2.1.1.3 Function Pointer and longjmp() Buffer in Heap/Data section

Function pointers are used to implement the late-binding of functions or implement callbacks. Figure 2.3 shows an example of the function pointer. The static character array, *buff*, and the function pointer, *fptr*, are both uninitialized and stored in the BSS segment. The *fun1* is called via the function pointer *fptr*. Note that the function pointer and character buffer also can be stored in the heap area by allocating it dynamically via *malloc()*.

Function pointer overwriting modifies a function pointer to point to the malicious code supplied by attackers. For instance, in the line number 6, prior to calling the *fun1* via the function pointer *fptr*, the function pointer can be overwritten with the string copy function *strncpy()*, supplied with exceedingly large size of characters via *argv[1]*. When the program executes a call via the function pointer in the line number 7, the attacker's code is executed instead of the original intended code.

The buffer, used by *longjmp()* function, contains control data. C99 defines the *setjmp()* macro, the *longjmp()* function, and the *jmp_buf* to bypass the normal function call and return discipline. The *setjmp()* macro saves its calling environment which includes the program counter, the stack pointer, and the base frame pointer into the *jmp_buf*, for later use by the *longjmp()* function. The *longjmp()* function restores the environment saved by the most recent invocation of the *setjmp()* macro into appropriate registers and jump to the location of the saved program counter. Since the *longjmp()* function blindly fetches the saved program counter and jumps to the location, attackers can change the saved program counter value to hijack the control flow of the program.

2.1.1.4 Global Offset Table(GOT)

The Global Offset Table(GOT) is another place where control data is stored in the process address space. The position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold the absolute addresses in private data. Addresses are therefore available without compromising the position-independence and shareability of a program's text. A program references its GOT using position-independent addressing and extracts absolute values. This technique redirects position-independent references to absolute locations.

Initially, the GOT holds the information, required by its relocation entries. After the system creates memory segments for a loadable object file, the run-time linker processes the relocation entries. The run-time linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link-editor creates an object file, the run-time linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a GOT entry. Because the executable file and shared objects have a separate GOT, a symbol's address can appear in several tables. The run-time linker processes all the GOT reloaction entries before giving control to any code in the process image. This processing ensures that absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This symbol enables a program, such as the run-time linker, to find its own dynamic structure without having yet processed its relocation entries. This method is especially important for the run-time linker, because it must initialize itself without relying on other programs to relocate its memory image.

The system can choose different memory segment addresses for the same shared object in different programs. The system can even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

Therefore GOT entry can be the target of control flow attacks. For instance, when a program requests to use *printf()*, the run-time linker locates the symbol and the location is then loaded into the GOT. After that the function is accessed via the Procedure Linkage Table(PLT). GOT overwriting attacks modify a GOT entry to redirect it to other code segment. For instance, an attacker can change the entry for *printf()* to *system()* address to execute *system()* instead of *printf()* when *printf()* is invoked in the process.

2.1.1.5 VPTR in VTable

In the object oriented program, virtual functions are used to intelligently change the called function via late-binding. In the C++ program language, a virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. The virtual function is declared as virtual in the base class using the virtual keyword. The virtual nature is inherited in the subsequent derived classes and the virtual keyword need not be re-stated there. The whole function body can be replaced with a new set of implementation in the derived class.

Whenever a program has a virtual function declared, a VTable is constructed for the class. The VTable consists of addresses to the virtual functions for classes that contain one or more virtual functions. The object of the class containing the virtual function contains a virtual pointer that points to the base address of the virtual table in the memory. Whenever there is a virtual function call, the VTable is used to resolve to the function address. An object of the class that contains one or more virtual functions contains a virtual pointer called the VPTR at the very beginning or the end of the object

```
1     #include <iostream>
2     class A{
3           private:
4              char str[11];
5           public:
6              void setBuffer(char * temp){strcpy (str, temp);}
7              virtual void printBuffer(){cout << str << endl ;}
8     };
9     void main (void){
10          A *a;
11          a = new A;
12          a->setBuffer("coucou");
13          a->printBuffer();
14    }
```

Figure 2.4 Example of vunerable virtual function

in the memory. Hence the size of the object, in this case, increases by the size of the pointer. This object contains the base address of the virtual table in memory. Note that virtual tables are class specific, i.e., there is only one virtual table for a class irrespective of the number of virtual functions it contains. This virtual table in turn contains the base addresses of one or more virtual functions of the class. At the time when a virtual function is called by an object, the VPTR of that object provides the base address of the virtual table for that class in the memory. This table is used to resolve the function call as it contains the addresses of all the virtual functions of that class. This is how dynamic binding is resolved during a virtual function call.

Overwriting the VPTR works on the same basis as overwriting the function pointer. An attacker can change VPTR to redirect control flow to crafted VTable to take the control flow of the program. Figure 2.4 shows the code segment that is vulnerable to VPTR overwriting. Class A contains a buffer, *str* in line number 4, and the string copy function, in line number 6, to feed the buffer. An attacker can overflow the buffer with three three types of information: the address of injected code, the injected code, and the address that the VPTR will point to. After successful overwriting with the VTable, the injected code by the attacker can be executed when the virtual function is invoked.

2.1.1.6 Function Destructors in .dtors and __exit_funcs Sections

In the Linux environment, there are other places which hold the control data. The GNU C compiler (gcc), and GNU tools in general, keep track of the initialization and termination functions of a program by maintaining lists of pointers to them: *.ctors (constructors)* and *.dtors (destructors)*. When a program is executed, the functions in *.ctors* are called, then *main()* is called and finally the destructors in *.dtors* are called. The constructors and destructors are specified by the programmer as following:

```
Static void start(void) __attribute__((constrouctor));
Static void stop(void) __attribute__((destructor));
```

In the produced Executable and Linkable Format (ELF) executable images, this will be represented as two different sections, .ctors and .dtros with the following layout:

```
0xffffffff <function address1> <function address2> ... 0x00000000
```

Both sections start with *0xffffffff* and end with *0x00000000*. Theses sections are mapped in the memory in the process' address space and writable by default. Also they are allocated in the memory even though programmers do not set up any constructor and destructor. Since constructors in *.ctors* are executed prior to the execution of program *main()*, attackers are mainly interested in changing addresses of destructors in *.dtors* section. Once attackers find programs that have been compiled and linked with GNU tools, and place to store the shellcode until the program exits, it is relatively easy to make successful CFA by overwriting the entry in the *.dtors*. By analyzing ELF binary image, it is easy to determine the exact position in the memory.

Similarly, C99 defines a general utility function *atexit()* to registers a function to be called without arguments at normal program termination. The registered function address is presented in *__exit_funcs* structure. Hence it is possible to transfer control to arbitrary code with an arbitrary memory write into the *__exit_funcs* structures. Even when the vulnerable program does not call the *atexit()* function, other destructors such as the *_dl_fini()* and *_libc_csu_fini()* are present in the *__exit_funcs*.

2.1.2 Attack Payload

So far, this thesis has examined different types of control data that can be the target of attackers. Once attackers obtain control flow of a process by modifying the control data, Attackers executes the malicious code called a payload to achieve the attackers' desired goal. In this section, Two types of payloads are mainly examed mainly: injected code and existing code.

2.1.2.1 Injected Code

Injecting malicious code is one of the favorite methods for attackers to take control of the victim system after obtaining control flow of a program. The injected code is also called *shellcode* in general because it typically starts a command shell from which the attacker can control the compromised system. A shellcode can either be *local* or *remote*. A local shellcode is used by an attacker to escalate its privilege on the local victim system. When the local shellcode is successfully executed, it provides the attacker with access right to the machine with the same privileges as the victim process.

A remote shellcode is used when the victim machine is connected with an attacker via network connection. Upon the successful execution of the shell code, the shellcode provides the attacker for accessing to the victim machine across the network. Typically, the remote shellcode uses three types of connections: *connect-back, bindshell* and *socket-reuse*. The connect-back shellcode connects back to the attacker's machine. However, because it leaves the trace of the attacker, it is less favorable to attackers. The bindshell opens a certain port on which the attacker can connect to control it. A firewall can be used to detect the outgoing connections made by the connect-back shellcode and the attempt to accept incoming connections made by the bindshells. The third type is the socket-reuse shellcode. it is used when the network port of the victim process is not closed before the shellcode is run. The socket-reuse shellcode can reuse the connection

```
#include <stdio.h>
void main(){
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
   exit(0);
 }
```

(a)

```
\xeb\xlf\x5e\x89\x76\x80
\x31\xc0\x88\x46\x07\x89
\x46\x0c\xb0\x0b\x89\xf3
\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\x31\xdb\x89\xd8
\x40\xcd\x80\xe8\xdc\xff
\xff/bin/sh;
```

(b)

Figure 2.5 (a) An example C code for spawning a shell and its binary code (Linux/ x86)

to communicate with the attacker. Since it reuses the opened port, it may bypass the firewall in the system. However it may harder for attackers to create the shellcode, since the shellcode requires prior knowledge of the suitable port. The remote shellcode also can download and executes malicious code in the victim system. This type of shellcode does not spawn a shell, but it downloads a certain executable file from other machines on the network.

In order to understand the shellcode mechanism, we further examined the details of the local shellcode that has a basic feature and the smallest code size. Figure 2.5 (a) shows the C code for spawning a *shell* code in X86 Linux platform and Figure 2.5 (b) is the its equivalent binary code for working local shellcode. The shellcode basically starts shell command via the *execv()* system call. In case of failing the execution of *execv()*, the code exits normally with the *exit(0)* system call. An attacker generates the optimized shellcode by performing reverse engineering on the C shellcode. We followed the attacker's logic for generating customized shellcode for the successful local shellcode. The fist step for generating the shellcode is to identify the basic requirement for launching the command shell. Attackers usually analyze the code in the assembly level to extract the minimum requirement of the shell code to generate a working payload that has the minimum size.

The low level operations for the *execv()* in the code Figure 2.5 (a) are:

1. Copying 0xb(the system call number for *execv()*) into the EAX register

2. Copying the address of the string "/bin/sh" into EBX

3. Copying the address of the *name*[] into ECX

4. Copying the address of the null pointer into EDX

5. Executing the software interrupt (int $0x80)

Similarly, the low level operations for the *exit(0)* can be identified and attackers can put them together to make the body of the shellcode. The second step is to craft the shellcode to make it as an injected payload. In order to make it as a working payload, attackers need to solve a couple of problems. The first problem is to identify the location of the string "/bin/sh" dynamically. The attackers do not know the exact address of the string during the attack. The second problem is to eliminate *0x00* or *NULL* in the shellcode. Since attackers might use problematic functions such as the *strcpy()* to inject code into the address of the victim process, the function might terminate copying process when it meets the *NULL* byte in the binary sequence.

In order to identify the location of the string "/bin/sh", attackers use indirect jumps and indirect calls. As we described before, when a call is issued, the processor pushes the return address, $PC +$ *size of a call instruction*, into the top of the stack. Therefore, if the string is located right after a call instruction, the address of the string can be obtained by popping the top of the run-time stack. There is another constraint for the string: the string has to be located at the end of the shellcode so that the shellcode runs correctly. Therefore, since the attacker knows the relative address of the call instruction from the starting of the shellcode, attackers use indirect jump, which can use the relative address to make a jump, to execute the call instruction which is located immediately before the string. Since the indirect call instruction also can use relative addresses, the attacker can call a *pop* instruction to obtain the location of the string. Therefore the overall flow of the local shell code is:

1. Jumping to the *CALL* instruction which is located before the string "/bin/sh".

2. Calling the instruction that pops the top of the stack.

3. Executing the body of the shellcode with obtained the address of the string.

Another requirement for the attack code is avoiding using NULL byte. Since the *strcpy()* function stops copying the input when the function detects NULL byte. One of the solutions for eliminating NULL byte is NULL byte substitution. Attackers make a register value to be NULL without using NULL byte and substitute NULL byte with the register. For instance, an attacker inserts the *xor reg reg* instruction, which makes *reg* value to be NULL, right before the instruction that contains NULL byte and substitute the NULL with the register *reg*. Figure 2.5(b) shows the final binary form of the working local shellcode that does not contain NULL byte and uses both indirect jumps to obtain the address of the sting. Similarly, attackers can generate other types of shellcode by following the similar procedure given above and substituting the body of the shellcode with the appropriate code.

2.1.2.2 Return-into-Lib(c) Exploits

Other types of the attack payload can be the existing library code in the system. Using the existing code is commonly used to evade the protection offered by the non-executable stack. Instead of returning into the code located within the stack, the vulnerable function returns into a memory area occupied by a shared library in the victim system. The return-into-libc attack is achieved by crafting a stack frame and replacing the origianl return address with a function address in the library. When the vulnerable function returns, the execution will resume at the function in the library. This thesis examines the three types of the return-into-libc attack by summarizing the methods presented in [52]

Figure 2.6 shows the stack frame before and after an attacker generates the fake stack frame for the return-into-libc attack. Figure 2.6(a) is the run-time stack layout before the return-into-libc attack for a vulnerable function. Figure 2.6(b) is the run-time

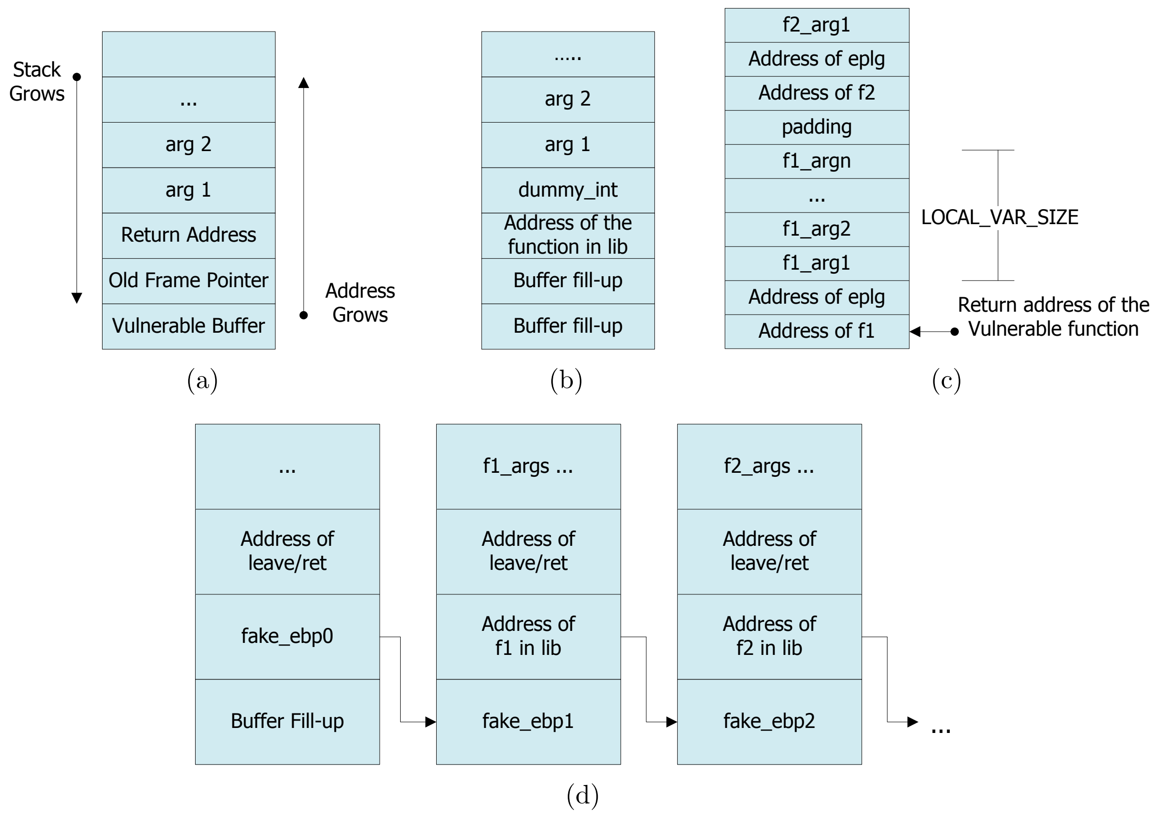


Figure 2.6 Stack Memory Layouts: (a) Before the Attack. (b) Single Return-into-Lib(c) Attack. (c) ESP Lifting Return-into-Lib(c) Attack. (d) Frame Faking Return-into-Lib(c)Attack.

```
eplg:                                              eplg:
       addl $LOCAL_VARS_SIZE, %esp                          popl  reg
       ret                                                  ret
                (a)                                             (b)
```

Figure 2.7 Function epilogue for programs with -fomit-frame-pointer flag

stack layout for the single return-into-libc attack. When the vulnerable function returns into the entry point of the library function, the *dummy_int* becomes the return address of the library function and *arg1* and *arg2* become arguments for the library function. For example, an attacker can launch a command shell by using the single return-into-libc attack. The attacker changes the return address of the vulnerable function with the address of the *system()* function and also changes the *arg1* with the address of the string */bin/sh*. In addtion to the changes, the attacker also properly sets the value of *dummy_int*. If the *dummy_int* is the arbitrary value, it might generate a segmentation fault error and leave the foot print in the system log file. Hence the *dummy_int* is usually set to the address of the *exit()* function to exit without remaining the foot print.

Attackers can also make chained return-into-libc attack by replacing *dummy_int* with the address of the appropriate code segment instead of the address of the *exit()*function . In figure 2.6(c), when the library function returns, the instruction pointed by the *dummy_int* is executed. When attackers excavate the appropriate code, such as the function epilogue, and write the address as a *dummy_int*, the following faked stack frame can be used to make another return-into-libc attack which is called ESP lifting return-into-libc attack. Figure 2.7 shows the typical function epilogue when a program is compiled with a -fomit-frame-pointer flag. When the returning function has multiple arguments, the function epilogue shown in figure 2.7 (a) can be used. When one argument is used for the returning function only one *pop* instruction is sufficient to adjust the stack pointer and return into the next function for the chained return-into-libc attack. Similarly, when the frame pointer is enabled for the compiler, it is also possible to make appropriate faked stack frames for the chained return-into-libc attack. Figure 2.6(c) shows the stack layout for the ESP lifting return-into-libc attack. The faked stack frames are used with the library function that has more than one arguments. When the attacker is unable to find the function epilogue whose the size of *$LOCAL_VARS_SIZE* is the same as the size of the returning function, the attacker may choose a code segment that has the larger size of *$LOAL_VAR_SIZE* and insert pads into the fake stack frame. When the first function,*f1* in the library, returns to the designated function epillogue, the epilogue adjusts the esp pointer to point the address of the function, *f2*, and jumps to the *f2* function. Similarly the attacker can generate the subsequence stack frames for futher chained return-into-libc attacks.

Figure 2.6(d) shows the run-time stack frame for the chained return-into-libc attack via the stack frame faking method. In this attack method, the return-into-libc attack uses the *leave, ret* code sequence in the function epilogue. In the first faked frame, the return address is set to return to the code segment,*leave, ret*. The *leave* instruction in the first stack frame sets the EBP to the first crafted stack frame. The *ret* instruction jumps to the function *f1*. When the function *f1* returns, the *leave, ret* code sequence in the second stack frame is excuted. Similarly, the second faked stack frame is used for executing other existing code. Therefore attackers can generate several calls to the existing code segments.

2.1.3 Attack Examples

Both the local and the remote CFA can be constructed with the different combinations of control data types and payloads. In this section, the thesis discusses the three different types of attack: the classical stack smashing attack, the remote CFA, and the *DLmalloc* exploit. The first type of the attack, the classical stack smashing attack, takes advantage of the buffer overflow vulnerability in a victim program to change the return address of the vulnerable function. This type of the attack can also be used for the heap based buffer overflow attack to escalate attackers' privilege in the victim system. The second type of the attack, addressed in [25], changes the entry of GOT entry to remotely launch a malicious program via the format string exploit. The remote CFA also can be exploited locally. The third type of the attack is a *DLmalloc* exploit. In contrast to the first two examples, the third example shows the way of modifying the target address indirectly via the system routine which manages the heap area. The other two examples overwrite the control data directly via the buffer overflow and the format string exploits.

2.1.3.1 Classical Buffer Overflow Attack

The classical stack smashing attack exploits buffer overflow vulnerabilities to overwrite the control data, the return address and the frame base pointer, in the run-time stack. Buffer overflow occurs when the size of the data to copy exceeds the capability of the buffer to store the data. From the viewpoint of security, it is a software flaw that may lead to a severe system breach. The buffer overflow attack consists of three steps: (1) injecting a piece of shellcode, (2) transferring the control flow of the machine to the shellcode, and (3) opening the system to the adversary by the shellcode. In this type of the attack, the adversary chooses a local data buffer of a procedure as the target. Since the buffer resides in the program stack, overflowing the buffer can overwrite the return address of the vulnerable function and the frame base pointer of the previous stack frame which is also stored in the stack. This makes the second step happen when the system transfers the control flow of the victim procedure to the previous procedure. The attack payload is a usual part of the data copied into the buffer, but the attacker can reuse the existing library function as the payload that are explained in the *return-into-libc attack*.

Figure 2.8 and figure 2.9 show the code used for exploiting buffer overflow vulnerability in the root privileged program. In this example, an attacker uses two programs to escalate its privilege from *user* to *root*. The user program called *exe* sets the environment variable and executes the vulnerable root privileged program *vul* containing a buffer that will be overflowed when the environment variable is copied into it.

The user program *exe.c* requires two arguments that are paths to the vulnerable program to modify the address of the buffer and to smash the buffer in the vulnerable program. An attacker may identify the location of the vulnerable buffer with a few tries. With the address of the vulnerable buffer, the *exe* fills the address of the vulnerable

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
   char buffer[96];
   strcpy(buffer,getenv("ENV"));
   return 0;
}
```

Figure 2.8 Vulnerable Root Privileged Program (*vul.c*)

```
#include <stdio.h>
#include <unistd.h>

extern char **environ;

int main(int argc, char **argv){
   char lg_string[128];
   long *lg_ptr = (long *)lg_string;
   int i;
   char shellcode[] =
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/sh";

   for (i = 0; i < 32; i++)
      *(lg_ptr + i) = (int) strtoul(argv[2], NULL, 16);
   for (i = 0; i < (int) strlen(shellcode); i++)
   lg_string[i] = shellcode[i];
   setenv("ENV", lg_string, 1);
   execle(argv[1], argv[1], NULL, environ);
   return 0;
}
```

Figure 2.9 Malicious User Privileged Program (*exe.c*)

buffer into the *lg_string*. Then the shellcode is copied at its beginning of the *lg_string*. Finally, *execle()* is called, so that the called program will have the corrupted environment variable. Note that the shell code including the address of the vulnerable buffer is already resided in the environment string section in the run-time stack of the vulnerable program before executing the *main()*.

2.1.3.2 Hannibal Attack

The next example is the *hannibal* attack, addressed in [25], which hijacks the control flow of the ftp server, *wu-ftpd*, running on Red Hat Linux 6.2 in the Intel processor. The hannibal attack uses a return-into-libc attacks to thwart the non-executable pages protection. The *hannibal* attack takes advantage of the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) in the Linux system that provides calls to dynamically linked library functions. As we examined GOT in previous section, GOT contains the absolute address of the dynamically linked library's functions. In order to resolve the absolute address during run-time, ELF binary executable contains PLT. Figure 2.10, the popular C program, exhibits the use of a PLT and GOT.

In the *.plt* section of ELF binary, the PLT entry for *printf()* is statically bounded. Figure 2.11 shows the disassembled binary which is generated from *gcc-3.4.6* for the PLT entry for the *printf()*. The location of *0X08482b0* is statically bounded to *printf()* PLT entry. When the *printf()* is called from the *main()*, the program jumps to the PLT entry of the *printf()*. The first instruction of the PLT entry is indirectly jumping to the target address which is loaded from corresponding GOT entry stored at *0X08049574*. If the absolute address of the *printf()* is not resolved yet, the GOT entry contains the next PLT instruction address *0X080482b6*. It pushes the identifier of the *pritnf()* and jumps to the function to resolve the symbol address and update the GOT entry for *printf()* in the memory location at *0x08049674*. Once the entry is updated, the first instruction of the PLT jumps to the *printf()*.

The main technique of the *hannibal* attack replaces GOT entry of the publicly available function *rename()* in the daemon with the *execv()* function. The vulnerable *wu-ftpd*

```
  #include <stdio.h>
  int main(){
   printf("Hello World\n");
   return0;
}
```

Figure 2.10 An example of C code for printing ”Hello World”

```
080482b0 <printf@plt>
80482b0:    jmp *8049574
80482a6:    push $0x0
80482ab:    jmp 8048290 <_init+0x18>
     (b)
```

Figure 2.11 Disassembled PLT entry for the *printf()*

daemon accepts commands *RNFR* and *RNTO* that accepts the source and the target file names to be changed accordingly. The first step of the attack uploads the malicious program into an anonymous FTP server and writes the GOT entry of the *rename()* function's address with the address of the PLT entry of the *execv()* function. When *RNFR* and *RNTO* commands are issued, the victim *wu-ftpd* daemon stores the file names and executes the *rename()* to change the file name. Since the GOT entry of the *reanme()* is replaced with the address of the PLT entry of the *execv()* via the format string exploit, it executes the *execv()* function instead of the *rename()* function. By crafting the heap memory area of the *wu-ftpd* program, the attacker can successfully execute the uploaded malicious program. Hence the attacker can executes arbitrary program with the privilege of the *wu-ftpd* daemon.

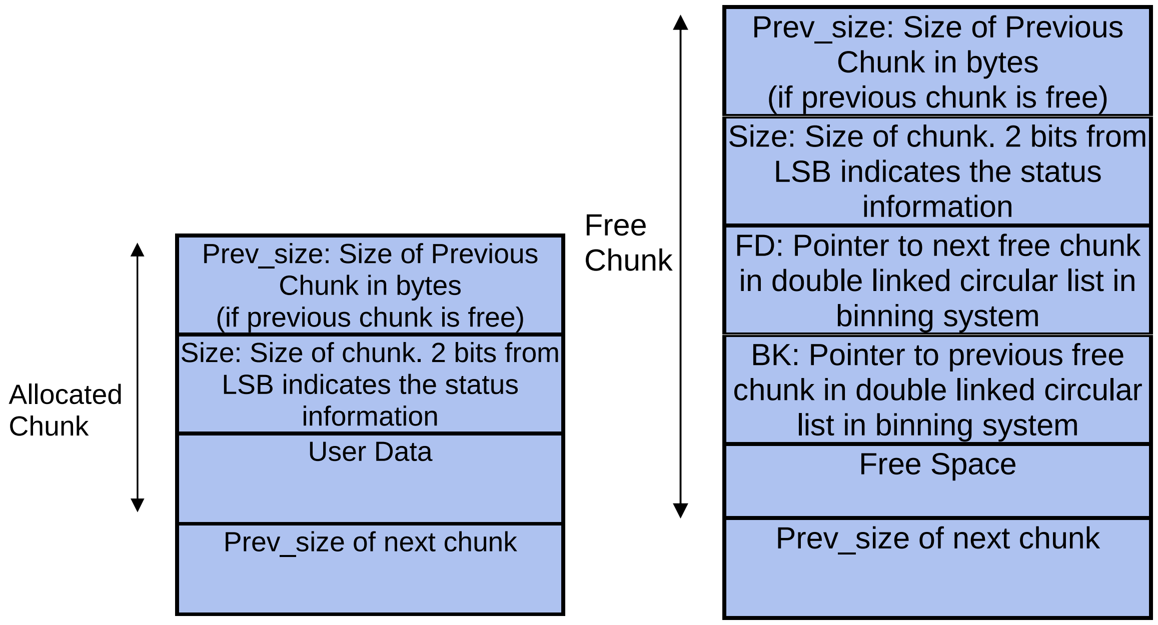
2.1.3.3 DLmalloc Exploit

In the Linux environment, GNU Libc adopts Doug Lea's memory allocator algorithm for the heap memory area management. The goals of the algorithm is to maximize the portability and the compatibility. It also tries to maximize the locality of the allocated memory chunks so that it can minimize the fragmentation and the cache misses during the program execution. It also considers minimizing the memory space used by the allocator and minimize the time for allocation and deallocation of the memory chunks.

In order to achieve the before-mentioned goals, the allocation algorithm keeps track of the free chunks of the memory by using a binning system. It stores free chunks in 128 bins. In each bin, free chunks of memory locations are stored as doubly linked list structure. In the first 62 bins, called small bins, chunks of the exact size are stored. The remaining bins store chunks in a particular range. DLmalloc uses macros to manage the binning system. Since these macros do not check the validity of the information stored in the memory chunk structure, CFAs exploit the vulnerability to write the arbitrary memory location. Figure 2.12 shows the memory layout for an allocate chunk and a free chunk. The *prev_size* is the 4-byte field to store the size of the previous chunk if the previous chunk is free. If the previous chunk is not free, this field is overwritten by the user data of the previous chunk. Each chunk size is multiple of 8-byte and the 2 bits from the least significant bit in the *size* field are used as flags. One of the flag, *PREV_INUSE* is used to identify if the previous chunk is allocated.

When the deallocation function, *free()*, is called, the algorithm frees up an allocated chunk and adding it into the binning system. If the neighboring chunks are free, the free chunks are coalesced into a larger free chunk. When the binning system coalesces the free chunks, the neighboring free chunks are unlinked from the binning system. Figure 2.13 presents the macro code for unlink a chunk of memory from the binning system. In the line 4 and 5, the value in the memory locations are modified.

Attackers can specify FD as the arbitrary location in the memory and the BK as the arbitrary value. For example, an attacker can specify the FD as the location of the GOT entry, and BK as the address of the malicious code to be executed when the function in the GOT entry is called. Figure 2.14 shows the example code that overflows heap data structure to execute the shellcode. There are two memory chunks, **vul* and **p*, are allocated next to each other. When an attacker overflows the **vul* to overwrites the *pre_size* field of the **vul* structure, the memory chunk **p* is set to be a free


Allocated Chunk
Prev_size: Size of Previous Chunk in bytes (if previous chunk is free)
Size: Size of chunk. 2 bits from LSB indicates the status information
User Data
Prev_size of next chunk
Free Chunk
Prev_size: Size of Previous Chunk in bytes (if previous chunk is free)
Size: Size of chunk. 2 bits from LSB indicates the status information
FD: Pointer to next free chunk in double linked circular list in binning system
BK: Pointer to previous free chunk in double linked circular list in binning system
Free Space
Prev_size of next chunk

(a) Memory Layout for Allocated Chunk in Heap (b) Memory Layout for Free Chunk in Heap

Figure 2.12 Memory Layout for Chunk in Heap

```
#define unlink( P, BK, FD) { \
   BK = P->bk;               \
   FD = P->fd;               \
   FD->bk = BK;              \
   BK->fd = FD;              \
}
```

Figure 2.13 Unlink() macro in DLmalloc

```
#include <stdio.h>

int main(void){
   char* vul = malloc(768);
   char* p   = malloc(64);
   scanf ("%s", vul);
   free(vul);
   free(p);
   return 0;
}
```

Figure 2.14 Sample Code for Demonstrating DLmalloc Explot

chunk. Therefore when the *free(vul)* is called, the *unlink()* macro is called to coalesce the chunks. Consequently, in order to achieve the successful CFA, attackers need to generate overflowing data with following forms:

- overwrite *size* field of **vul* so that the memory chunk **p* is designated as free chunk
- overwrite BK field of **p* with the start address of malicious code. (i.e. shellcode)
- overwrite FD field of **p* with function pointer (i.e. GOT entry of *free()*
- inject shellcode

Similarly, since the *frontlink()* macro, which inserts a free chunk into the binning system, also modifies a pointer and its value without checking the validity, CFAs can exploit the vulnerability to write arbitrary value into arbitrary memory location. Figure 2.15 shows the *frontlink()* macro.

The *frontlink()* macro inserts P, whose size is S and pointers FD and BK, into bin IDX. The else statement, in the line 1, is executed when the free chunk P has larger size than the small bin size. The while loop, in the line 2, the largest FD that is smaller than the size S. In the line 4, BK is assigned FD->bk. And in the line 5, fd fields of BK is set. Consequently, it assigned P into (FD->bk)->fd. Therefore, if attackers create a FD with

```
  #define frontlink( A, P, S, IDX, BK, FD ) {            \
      if ( S < MAX_SMALLBIN_SIZE ) {                     \
          IDDX = smallbin_index( S );                    \
          mark_binblock( A, IDX );                       \
          BK = bin_at( A, IDX );                         \
          FD = BK->fd;                                   \
          P->bk = BK;                                    \
          P->fd = FD;                                    \
          FD->bk = BK->fd = P;                           \
1     } else {                                           \
          IDX = bin_index( S );                          \
          BK = bin_at( A, IDX );                         \
          FD = BK->fd;                                   \
          if ( FD == BK ) {                              \
              mark_binblock(A, IDX);                     \
          } else {                                       \
2             while ( FD != BK && S < chunksize(FD) ) {  \
3                 FD = FD->fd;                           \
              }                                          \
4             BK = FD->bk;                               \
          }                                              \
          P->bk = BK;                                    \
          P->fd = FD;                                    \
5         FD->bk = BK->fd = P;                           \
      }                                                  \
  }
```

Figure 2.15 Frontlink() macro in DLmalloc

bk being a function pointer location, then BK->fd would allow attacker to overwrite the function pointer with arbitrary value which can be the location of malicious code.

2.2 Related Works

After the Morris worm incident in 1988, the security in computing system was warned to be enhanced. However, until the *Code Red Worms* had breached system security successfully, the computing system security was not favorable subject to be thoroughly studied by computing researchers. More than 359,000 computers were infected with the *Code Red Worm* in less than 14 hours on July 2001. The economic costs resulting from the *Code Red Worms'* attack were estimated to be over $2.4 billion. This incident immediately brought the attention of the security researchers and encouraged to develop various defense mechanisms to protect computing systems. However the proposed security solutions provide limited protections which are dedicated to certain attack methods. Some of them also have difficulties on implementing in the real world.

As before-mentioned, the control flow attack is composed of three stages: overwriting control data, transferring program control to the malicious code and executing the malicious code. One of the solution removes the software vulnerabilities and the other prevents one of the stages in the CFAs. Hence the solutions which prevent control flow attack can fall into one of the following categories:

1. Removing software vulnerability
2. Detecting illegal modification of control data
3. Preventing execution of malicious code.
4. Preventing illegal control transfer to malicious code

The first type of the defense mechanisms are the compiler patches and the middleware that remove the software vulnerabilities such as lack of bounds checking of the memory object or using dangerous functions. However these type of approaches provide the limited security protections and have the software incompatibility problems.

The second type of the defense mechanism prevents or detects the illegal modification of the certain types of the control data. Due to the limited protection of control data type and the existence of the various types of control data, hackers immediately published various ways to defeat the propoesed protection methods.

The third type of the defense mechanism prevents the execution of the malicious code in the victim system. Since, the payload was mainly injected in the program memory space during the modifying control data, the control flow attack can be folded by prohibiting the execution of the injected code. In order to inject the malicious code in the victim program memory space, the memory space must be writable. However the normal program code resides in the read only memory area. Consequently, by prohibiting the code execution from the writable memory space, some of the control flow attacks can be prevented. The solution seemed to be promising and the major processor vendors adopted the mechanism and provided a hardware support for the defense mechanism. However, attackers developed a new type of attack called *return-into-lib(c)*. As we examined before, the attack reuses existing code in the system library, which resids in the read only memory area, to achieve the malicious goals.

Although the third type of defense mechanism limits the large portion of control flow attacks, attackers can still bypass the defense mechanism. Hence security researchers took more efforts to find the root cause of the control flow attack. Since the root cause of the control flow attack is that attacker can make control transfer to malicious code by crafting control transfer data, the last type of defense mechanism is to detect illegal control transfer to malicious code. By monitoring problematic control transfer in the vulnerable program, control flow attack can be detected and halted. This type of defense mechanism is more generic and its effectiveness does not bound to certain type of attack. However monitoring all the problematic control transfer is challenging and imposes high performance or implementation overheads. In this section, we further introduce more details on the four types of defense methods for protecting a system from control flow attack.

2.2.1 Removing Software Vulnerability

A number of different projects have addressed the buffer overflow problem with different approaches. One approach eliminates vulnerable code from a source program and helps the application to be safe from the software vunlerabilities. There is an auditing tool [69] that provides the automate source code review for security. It helps to eliminate the use of dangerous functions such as strcpy, gets, and etc., but the tool has the limited vulnerability detection; it is not able to check the boundaries of a pointer variable.

Software bounds check [40] verifies the bounds of each pointer object to prevent buffer overflow attack. It is the modification of front-end of GCC compiler to add code to check pointer arithmetics and to maintain a table of known allocated storage region. The performance overhead depends on the application type but it can incur performance overhead up to 30 times slowdown for the matrix applications. Recently, Weihaw et al [20] proposed checking limited type of pointer object, that receives data from network, to reduce performance overhead. However other overwriting method such as the *format string attack* cannot be prevented.

Libsafe [12] is based on a middleware software layer that intercepts all function calls made to the vulnerable library functions. Libsafe redirects the dangerous function calls to the substitute version of the corresponding function which implements the original function. For instance, when a program calls unsafe call *gets(string)*, Libsafe intercepts the call and executes its own version of *gets*. This new version of *gets()* computes the length of the buffer, *string* and calls *fgets(string, size, stdin)*. The *fget()* is much safer than *gets()* since it only allows the stated number of characters to be read into the buffer to prevent the buffer overflows. Similarly calls to the other unsafe functions are redirected to the safe version of functions. However, Libsafe has a few drawbacks; it is not compatible with applications that have been linked with the certain version of the shared library, libc5. It cannot catch overflows in statically compiled programs since Libsafe works by intercepting calls to shared libraries. Similarly, when a function is included as the inline function, it cannot catch the overflows. The other drawback is that Libsafe may not work with the *setuid* programs since the *setuid* programs ignore

LD_PRELOAD that is required to load Libsafe to intercept function calls.

2.2.2 Detecting Illegal Modification of Control Data

The first attempt to prevent control flow attack is StackGuard [24]. The compiler technique inserts protection tags, called *canary word*, before the control data in the run-time stack, and verifies those tags before the control data are used during function prologue and epilogue. The control data may also be encoded to thwart the attempts to change them. StackGuard only protects the stack smashing attack, that modifies the return address in the run-time stack by overflowing the vulnerable buffer in the victim program stack. Since the buffer overflow attack, also called stack smashing attack, overflows a vulnerable buffer to overwrite a return address in the run-time stack. When stack smashing occurs, the data between the return address and the vulnerable buffer is also overwritten. Therefore, StackGuard inserts a *canary word* next to every return address in the run-time stack, and verifies the canary value before the return address is used. When the *canary word* is changed, it detects the stack smashing attack. However hackers immediately pointed out the weakness of the defense mechanism by demonstrating a number of different methods to bypass the protection mechanism. Hence PC Encoding [57] were proposed to compensate the deficiency by encrypting the return address during the function prologue, and recovering the original value before using it during the function epilogue. It uses the simple symmetric encryption routine that simply XORing the return address with an encryption key. Therefore, without knowinga the key of the encryption, attackers cannot make control transfer to the malicious code. Consequently these mechanisms can successfully prevent stack smashing attack. However, this defense mechanism has a few limitations. It only provides limited protections; it is only capable of protecting the control flow attack that modifies the certain control data, *return address* and*base frame pointer*. The compiler patches also requires source code recompilation to insert security routines for the manipulations of the return addresses and the run-time stack structure.

StackShiled [4] and RAD [16] are other types of compiler techniques that monitor the illegal modification of the return address. They create a separate stack to store return addresses during the function prologue and check the integrity of the return addresses in the run-time stack before using it. The redundant copy of the return addresses are kept in a secure place in the memory, and is compared with the return address before the address is refered from the run-time stack. Again they can only protect a system from a certain control flow attack that modifies return address in the run-time stack. Propolis [28] protects not only return addresses but also function pointers in the run-time stack by rearranging the stack to place a vulnerable buffer to the highest part of the stack frame. It also creates copies of arguments of the function and relocates them together with local variable to protect the arguments. It was implemented as a patch to GCC 3.x and included in the GCC 4.1.

There are also hardware defense mechanism for preventing the control flow attack via the return address corruption. Secure Return Address Stack(SRAS) [55] is the security enhancement for the existing hardware return address buffer, which predicts the target of a *return* instruction. In most programs, function calls and returns are always paired. When a *call* instruction is issued, the return address, which is the *program counter(PC) + size of call instruction*, is pushed into the return address buffer. When a *return* instruction is issued, the top of return address buffer is popped to predict the next instruction to be fetched. The hardware can identify the correctness of the prediction when the return instruction is retired. Consequently, by enhancing the return address buffer to predict a return address accurately, the hardware can detect the illegal modification of the return address.

2.2.3 Preventing Execution of Malicious Code

Another type of control flow attack defense mechanism is to prevent executing the malicious code. Since the large portion of CFAs uses injected code as its payload, preventing execution of data, or code in the writable memory can effectively prevent CFAs using the injected payload. Solar Designer [26] has developed a linux kernel patch that makes the run-time stack non-executable. The patch is a part of openwal project that address the stack smashing problem. This patch makes the run-time stack portion of a user process non-executable, so that the injected code into the stack cannot be executed. The patch offers the advantage of zero performance penalty without the recompilation of the target program. However it fails to address the problem of buffer overflow attacks that do not place the attack code on the stack. The attacker may inject the attack code into a heap-allocated or statically allocated buffer, and redirect the function address or function pointer to point to the attack code. Moreover, the non-executable stack cannot detect the *return-into-libc* attack.

PaX [3] is a patch for the Linux Kernel that implements least privilege protections for memory pages. It flags data pages as non-executable and code pages as non-writable. It also randomly arranges the program memory space. PaX basically employees two mechanisms to prevent malicious code execution in a victim system: non-executable pages and address space layout randomization(ASLR). Non-executable pages are enforced by PAGEEXEC and SEGMEXEC mechanisms with the restricted *mprotect()* function.

PAGEEXEC uses or emulates the no-execute(NX) bit. On processors that supports the hardware NX bit, PaX utilizes the NX bit without incurring the performance overhead. If a hardware NX bit is not available, NX bit emulation is done by changing the permission level of non-executable pages. It uses the semantics of the memory management unit and the page attributes to enforce non-executable pages. The supervisor bit in the page table entry is overloaded to represent the non-executable page. SEGMEXEC emulates the functionality of an NX bit on x86 processors by splitting the address space in half and mirroring the code mappings across the address space. PaX guarantees that the physical page cannot have both writable and executable page attributes at the same time. It does not allow a memory page to have permission PROT_WRITE and PROT_EXEC both enabled by restricting the use of *mprotec()*. Also, it prohibits a memory page to be masked with PROT_EXEC after the page attributes are initialized. This prevents a memory pages, which may contain injected code, from being changed to both writable and executable page. The ASLR is a mechanism that prevents the arbitrary execution of code, or return-into-libc attacks.

PaX randomizes the stack base and the heap base in the virtual memory when the ASLR is enabled so that attackers cannot guess the addresses of functions in the shared library. It also optionally randomizes the *mmap()* base and the executable base of a program. PaX leaves a portion of the addresses out of the randomization calculations for avoiding the address space collision of the stack, the heap and the shared library. However, PaX is still vulnerable to the format string exploit that reveals the contents of the random memory location. Hence attackers can obtain the base address of libc for a successful return-into-libc attack. PaX tries to prevent a borad range of the CFAs that results in making the system more complex. Later, a critical vulnerability, which may lead privilege escalation, was also found in PaX itself.

2.2.4 Preventing Illegal control transfer to Malicious code

So far, this thesis has examined the various control flow mitigation mechanisms that focus on thwarting an individual attack methodology. Therefore these mechanisms usually provide the limited security protections. Security researchers have made great effort to find more generic approach to detecting CFAs. There are two basic mechanisms to prevent illegal control transfer to a malicious code: validating control transfers and tracking the dynamic information flow in a program. The control transfer validation checks the source and the target of a control instruction. One of the validation approach applies general security policies to verify the legitimacy of the control transfers. Another validation approach checks the source and target pair with pre-defined source and target pairs to attest thier integrity.

The dynamic information flow tracking mechanism tracks the origin or the integrity of the control data. This mechanisms scrutinize the semantics ofa the control flows in programs to find anomalies in the use of the control data. For instance, since the control data is originated from the I/O during the control flow attack, prohibiting the illegal control transfer, which refers the control data from I/O, prevents the control flow attack. The following subsections introduces the details of the related works.

2.2.4.1 Validating Control Transfer

Program Shepherding [44], derived from DynamoRIO run-time optimizer, detects the unauthorized control transfers. DynamoRIO translates executable binaries and generates code blocks to apply dynamic optimization techniques. In order to achieve low performance overhead, DynamoRIO uses a software code cache to support the native execution of the newly constructed fragments. When the control transfer occurs, the part of DynamoRIO code obtains the program control and dispatches the subsequence instructions from the code cache. Since the address of instructions in the code cache differs from the address in the original binaries, it requires the address translations to identify the target location. Therefore DynamoRIO can identify the control instructions and verify the control transfers.

It establishes general security policies: the restricted code origin, the restricted control transfer, and the un-circumventable sandboxing. The code origin check is done at the point when the system copies a basic block into the code cache. The check needs to be executed only once for each basic block. It identifies whether the code has been modified from its original image on disk or dynamically generated. In order to validate control transfers, the direct branches are checked at the time of linking the basic blocks. For indirect branches, the hash table lookup routine translates the address of the target code segment into a basic block entry address. At this point, it enforces the general policies for the indirect branches. The targets of indirect branches are matched against the entry points of PLT-defined and dynamically resolved symbols to enforce restrictions on inter-segment transitions, and target of returns are checked to ensure that the target is an instruction after a call site. In order to prevent jumping into middle of a block to bypass the checks, It only allows the control flow transfer to the top of basic blocks or traces in the code cache. Program Shepherding is implemented on a dynamic optimization infrastructure, which is an additional software layer between a processor and an application. As a result, Program Shepherding may have high performance overheads. The space overhead is reported to be 16.2% on average and 94.6% in the worst case. It also incurs up to 7.6 times performance slowdown for a benchmark program.

Inline Reference monitor(IRM) is also able to validate the execution of every branch instruction with the valid control flow information of the program. CFI [8] is software fine-grained control flow integrity checking mechanism. It instruments the program executable to insert a label immediately before each function or a code block. Every indirect branch instruction is also rewritten as a small piece of code that checks the label before the control transfer. The labels are constructed from static analysis of the control flow graph (CFG) of the program. It has a few drawbacks, in certain cases, the use of label cannot preserve the precise CFG information; for example, if two indirect branches share one target (and the associated label) but not other targets. It may weaken the security strength. Another drawback is that CFI changes the source code by using binary instrumentation tools. The source code modification may affects the trustworthiness of the program execution and may lead a complication of the adoptation in the practice. The overhead is non-trivial; up to 50% and on average 21% for SPECCPU2000 benchmarks.

A hardware version of CFI with ISA extension [13] is also proposed to reduce the performance overhead. New instructions are introduced to replace the guard code with a single instruction. In the initial performance evaluation, five integer benchmarks from SPEC CPU2000int are observed with test input set. They reported that the results showed maximum 7% performance overhead and around 2% on average. Nevertheless, the aforementioned issue of label sharing still exists.

2.2.4.2 Dynamic Information Tracking

The software based information flow tracking systems usually incur high performance overhead. LIFT [58] is a software information flow tracking system which utilizes binary instrumentation tools. It tags each byte in data memory to identify the unsafe data that comes from other source rather than the one program generates and tracks the data. When the unsafe data is used as control data, the system raises an alarm. Although it achieves best performance among software based information flow tracking systems, it still incurs 3.6 times slowdown on average.

Another form of hardware protection can prevent attacks to all memory regions but requires the extensive changes of the CPU internals. Suh et al. [65] proposed dynamic flow information tracking that tracks every data in the memory to detect and track the spurious data that comes from external I/O. It tracks the information flow inside modified hardware and checks the authenticity of the control data by tracking four types of dependencies: copy, computation, load-address, and store-address dependencies. The modified processors dynamically tracks spurious information flows by tagging the result of an operation as spurious if it has a dependency on spurious data. As for the hardware changes, an one-bit tag is required for every memory data, CPU registers, and CPU data paths changes to propagate the tags during the address calculation. When suspicious data is used as control data, it raises an alarm.

Crandall and Chong [25] proposed *Minos* that also tracks the information flow in program execution using one-bit tags. In order to track the integrity bit, it also requires hardware modification similar to the one in [65]. However, instead of tagging spurious data rooted from I/O, Minos decides the high integrity data, which can be used as control data. An alarm will be raised if a low-integrity data is used as a target address. In the Minos implementation, there is a set of rules to decide high integrity data. For instances, high integrity data is created before which all libraries and trusted file were established. Everything created after the establishment time is marked as low integrity. The static binaries can be trusted and their control flow and that of their children are marked as the high integrity data. Any process communication is marked as the low integrity data except the communication using the shared memory. Also the *read()* system call forces the data read by the process to be the low integrity data unless both the *ctime* that is the time when changes were made to *inode* and *mtime* that is the time when the actual contents of a file was last modified, are set before the establishment time. Additionally, the argument variables of the *execv()* system calls are forced to be the low integrity data. Minos successfully prevent various control flow attacks on emulated environment.

3 IBMON: System-level Approach for Run-time Control Flow Validation

As beforementioned in section 2.1, it is hard to protect a system against all types of control flow attacks by protecting each type of control data, since there are various types of control data and various system entities that manage the control data. Also combining various protection mechanisms to protect the different types of control data makes security system more complex. Unfortunately a complex system tends to contain bugs that lead to attacks on the system itself [3].

Control flow validation is an effective approach to mitigating control flow attacks. This thesis proposes an efficient control flow validation mechanisms to detect and prevent malicious actions. The control flow graph information is pre-collected through the static analysis of a program or the dynamic program profiling. A recent work implements such a mechanism by instrumenting the program binary code: machine instructions doing the validation are inserted before branch instructions [8]. The target of an indirect branch is allocated dynamically in the data area of the program's address space, and an attacker can manage to overwrite it by exploiting the memory corruption vulnerabilities such as the stack smashing, the format string, and the heap overflow vulnerablities. Later the execution of the compromised indirect branch gives the attacker great flexibility to re-point the control flow to any any code segment that the attacker desires. Hence, any control flow transfer by an indirect jump should be validated before the architecture actually uses it. It is sufficient to validate only indirect branches that use register contents as branch targets to thwart the CFAs, because only those branches use the control data stored in the memory. The approach can prevent most control flow attacks, but there are several concerns regarding the implementation. First, in order to validate efficiently at run-time, an indirect branch has to be validated with a control flow graph of simplified structure, which may reduce the strength of validation. Second, the inserted instructions increase the program execution time and affect instruction cache performance, which are non-trivial overhead.

There are two notable related works that are monitors the control transfers in target applications: Program Shepherding [44] and CFI [8]. The Program Shepherding uses dynamic optimization tools to identify the problematic control transfers. Due to the performance overhead of the validation mechanism it performs the dynamic optimizations. However, it still incurs up to 7.6 times performance slowdown in a certain application. It only protects malicious control transfer to injected code or middle of the program routine. This may not prevent mimicry attacks, which jump to legitimate entry point of routine such as return-to-libc attack, due to lack of the location sensitivity or precision. CFI checks the label embedded immediately before the targets of indirect branches. Since each indirect branch has unique label due to limitation of the binary instrumentation technique, the use of the label cannot attest indirect branches precisely.

This thesis explores the possibility of implementing simple and effective control flow validation mechanism by using emerging features in commercial processors. Based on the commercial hardware features, we have implemented a prototype, called *Indirect Branch MONitor* (IBMON), with Linux-2.6.27 on Intel Core 2 Duo, Core i7 processors and Linux-2.4.20 on Intel Pentium 4 processor. The experiments show that the system can prevent the various types of the control flow attacks. IBMON takes advantage of that the corrupted indirect branch target always causes the indirect branch mis-prediction. IBMON detects the hardware branch mis-prediction events by using the hardware monitoring features. IBMON works efficiently because the indirect branch instruction ratio and the indirect branch mis-prediction rate are both low for almost all real-world programs. The branch prediction unit of modern processors is well designed to minimize the branch mis-prediction rate. Compared with the binary instrumentation, the frequency of validation in IBMON is dramatically reduced and it is simpler and more efficient. However there are a few issues to implement efficient IBMON on commercial processors.

The key issue in this method is to limit the performance overhead of interrupt-based validations. The following sub-sections describe details of these hardware features and discuss the collection strategy of control flow information.

3.1 Hardware Components and Control Flow Information

IBMON monitors the dynamic control flow of a system to prevent control flow attack without modifying the code of the target applications. IBMON relies on two hardware features available in modern processors: hardware branch prediction and hardware performance monitoring. We observe that it is sufficient to detect most control flow attacks by validating only mis-predicted indirect branches. Furthermore, the hardware performance monitor on most processors can be configured to raise an interrupt upon branch mis-predictions events. Therefore, it is promising to build an efficient control flow monitoring system on the top of the two hardware features.

3.1.1 Hardware Performance Monitoring

IBMON utilizes hardware features in order to inspect indirect control transfers without recompiling program binary or special execution environment. Recent Intel processors provide hardware performance monitoring and debug features. The hardware performance monitor provides a number of different architectural and non-architectural hardware events.

In the NetBurst microarchitectures, Pentium 4, retired mis-predicted indirect branch events can be counted by setting a set of the appropriate machine specific registers(MSRs) [37]. Upon a counter overflow event, the processor is capable of generating an interrupt to reset the counter value. Hence, upon an indirect branch mis-prediction event by setting the counter value to *max-1* or using a feature that forces to generate an interrupt on single event, IBMON is able to obtain control of a system and verifies the source and the destination addresses of the indirect branches.

However, since the interrupt is imprecise, the interrupt is usually delivered a few cycles later; the program counter value may not be the address of the indirect branch instruction which generates the mis-prediction event. Therefore, IBMON uses another hardware debugging features, called Last Branch Recording(LBR). This feature provides recording the source and destination addresses of the last taken branch, exception and interrupt in the hardware stack. Since all indirect branch events are taken branches, IBMON accurately obtains the source and the destination addresses of mis-predicted indirect branch. Hence, the control flow information is collected from the LBR stack to be validated when indirect branch mis-prediction events occur.

Similarly, Core and Nehalem micro architectures (Core 2 Duo, Core i7) also provides those features, but they only can count executed mis-predicted indirect branch. [1] The difference between *retired* and *executed* is that *executed* means the mis-predicted instructions are not necessarily retired i.e the instructions may be bogus(mis-speculated). It results counting more mis-predicted indirect branch instructions than actual retired mis-predicted indirect branch instructions. However this does not waken the security strength of detecting CFAs, but increases the validation frequency.

3.1.2 Hardware Branch Predictors

IBMON is an interrupt-based control flow validation system. Hence the validation frequency is one of the major component that affects the performance. Hardware branch predictors help reducing the validation frequency without degrading the security strength. Recent commercial processors employ different types of branch predictors to enhance the performance of the processor.

Return Stack Buffer(RSB) is used to predict the destination of a return instruction in x86 architecture. When a call instruction is issued, the address of the after the call instruction is pushed into the RSB. When a return instruction is issued, the top of the RSB is popped to predict the next instruction to be fetched. The processor compares the predicted destination address and the one stored in the run-time stack in the memory to validate the prediction result. If the prediction is wrong, the processor squashes the

[1] we also explored other hardware features such as precise event sampling with LBR mechanism [37] to efficiently collect the source and target addresses. It turns out that this method is the most efficient in the Intel processors

pipe line and re-fetches instructions from the destination address in the run-time stack. Hence if the destination address in the run-time stack is tempered, it always results in the return address mis-prediction. Thus this guarantees that IBMON inspects the tempered destination address in the run-time stack. Note that an untempered destination address may not result in the return address mis-prediction and does not need to be validated by IBMON.

Currently, most microprocessors also have other type of branch prediction facility, branch target buffer (BTB). Some processors have the enhanced support for indirect branch prediction. We notice that branch predictors many processors can be used for doing a portion of validation in the form of indirect branch prediction. The branch predictors store the previously executed branch to predict the target of a branch instruction for the future execution. Hence, branch prediction can be used for the part of the validation of indirect branches. Note that all the branches stored in the branch predictor have been validated before due to the cold misses. Since the validation unit checks every target before it is loaded into the program counter, the targets presented in the branch prediction units have passed the validation in the first place. This implies a control flow transfer from a correct branch prediction is guaranteed to be safe. On the other hand, during an attack, the target address in the memory is corrupted and will not match the validated one in the branch prediction units, resulting in a mis-prediction. Notice that while an attacker is able to overwrite a value in memory due to all kinds of vulnerabilities, it cannot directly compromise the content in the software-transparent prediction units at the same time. Consequently, a mis-prediction event of an indirect branch becomes a symptom of an attack and the validation can be activated only on that event, rather than every instance of indirect branches.

Utilizing branch prediction results in reducing the frequency of the control flow validation by checking only when the branch predictor predicts the target incorrectly for indirect branch. Consider a simple example: A program has a simple loop that calls a function via indirect function calls over million times. Then binary instrumented approach may need over twice as much as the number of function calls; every function calls and returns. However, our design requires less number of validations, since branch predictor can accurately predict the targets most of the times. Modern high-performance processors have accurate branch predictors, whose prediction accuracy is usually over 95%, and branch mis-predictions are exceptions. Because our system only do validations when indirect branch mis-predictions happens, unnecessary validations are avoided.

3.1.3 Indirect Branch Pair (IBP)

In order to attest control flow integrity of a program, an indirect branch and its target address pair(IBP) are checked with pre-generated valid set of address pairs. Including both the branch and the target address is necessary not only for preventing control flow from transferring to an unintended destination but also intercepting a jump heading to a legitimate target but from an illegal source site.

There are basically two ways to fill up the IBP table with legitimate IBPs. The one way to obtain IBPs is via the static analysis of the control flow graph of the program. One may extract the legitimate IBPs from the existing execution trace of legacy code offline [33, 63] too. For the dynamically loaded library and the shared libraries, the linker and loader can help finding the legitimate targets of branches when they patch the program with absolute addresses. The second way to initialize the IBP table is to perform the program*profiling* as many model-based solutions have done [29, 31, 73]. By running the application either in a particular time interval or until the unique IBPs converges in a secure environment, the processor can regard all seen IBPs as legitimate ones. Also, during the software development and testing phase, the test cases being used should cover most, if not all, possible execution paths for each branch; therefore an IBP table can be generated as a side product of the testing phase in the software development. We also tested IBP convergence of an Apache server on Red Hat Linux 7.3 over Simics, an IA-32 emulator. We generated both static and dynamic loads from a remote machine while collecting the addresses of the indirect branches and targets on the simulated machine. Figure 3.1 shows that the number of IBPs does converge quickly.

However it may be difficult to collect all legitimate IBPs during in-house training

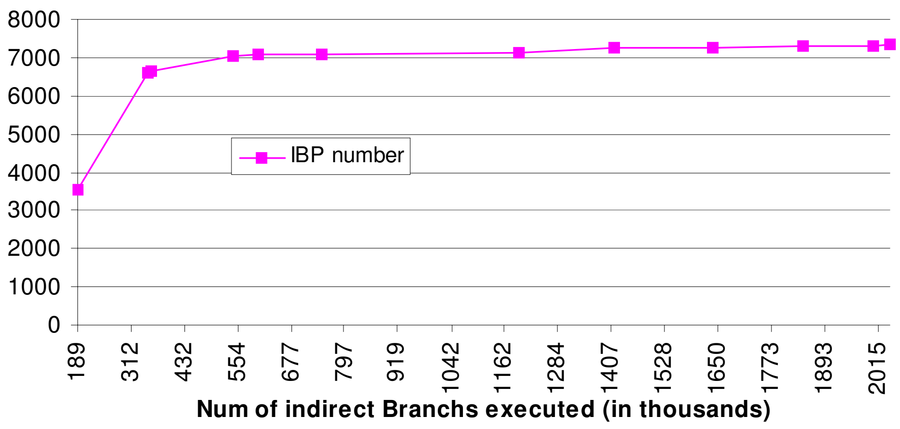


Figure 3.1 The number of unique IBPs against indirect branches that have been executed. Each data point is an additional workload.

Restricting	Policy
Returns	allows only if the destination is after a call instruction
Indirect Jumps	allows only if the destination address is within read-only code section or a function entry point
Indirect Calls	allows only if the destination address is a function entry point

Table 3.1 Sample Security Policies

without understanding internals of the application. Hence IBMON also employ the generic security policies with information provided by hardware features during the in-house training. The least restrictive policy is allowing any indirect branch instruction during training. This policy is used during software testing phase of software development or testing applications in the secure execution environment. In this phase, IBMON is set to monitor all return instruction and mis-predicted indirect branches to collect new IBPs. Note that a newly executed return instruction may not generate branch mis-prediction due to the semantics of return address prediction [54].

Once most of legitimate IBPs are collected, the next restrictive policy can be employed. Table 3.1 lists the sample policy for different types of indirect branches, similar to the one used in Program Shepherding [44], used in the prototype.

A return instruction is used to return from functions. Since it refers destination address stored in the run-time stack, it often becomes a target of CFAs. When a return mis-prediction occurs, IBMON inspects the destination of the return instruction. In the sample security policy, IBMON only allows control transfer if the previous instruction of the destination is a call instruction in the read-only code section. It also prevent function return in application code to the start of the shared library routine to reduce the possibility of successful mimicry attacks.

The indirect jumps are used in the implementation of the jump table of switch statements and the dynamically shared libraries. When an indirect jumps mis-prediction occurs, IBMON inspects weather the destination is a function entry point in the case of dynamically shared libraries or within the read-only code section in the case of switch statements. Otherwise IBMON prohibits control transfers to the destination. For indirect calls, IBMON only allows control transfers to a function entry points.

The sample security policy is somewhat weaker than policy provided by other security tools for mimicry attacks. The security policy for restricting return instruction is weaker than validating call-return program semantics provided by previous works [44, 24, 16, 57] due to the lack of hardware supports. [2] Although this policy may miss highly sophisticate mimicry attacks such as the return-into-libc from the shared libraries, it is good enough to thwart the execution of injected code and the return-into-libc attack from application code. However, the policy for indirect calls and indirect jumps provides the same level of security strength with Program Shepherding.

In order to reduce the validation latency and the risk of the mimicry attacks, IBMON can be set to only allow the pre-defined intended indirect control transfers. The most restrictive security policy in IBMON is to allow control transfers with pre-collected indirect branch pairs(IBPs) only.

[2]Currently,hardware can only efficiently provides information of a branch instance not branch history.

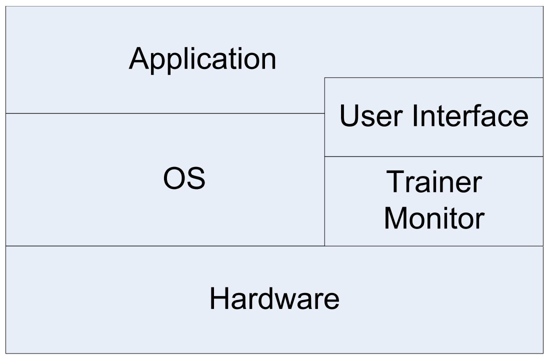
3.2 System Architecture

IBMON operates in two modes: training, and monitoring. In the training mode, it collects IBPs with the security policies. IBMON sets the hardware performance monitor to monitor every mis-predicted indirect branch instruction and every return instruction, since a return instruction may not generate return address mis-prediction. During the monitoring mode, IBMON sets hardware performance monitor to generate interrupts on indirect branch mis-prediction events. When an indirect control transfer violates the security policy or it is not the intended control transfer, IBMON terminates the process.

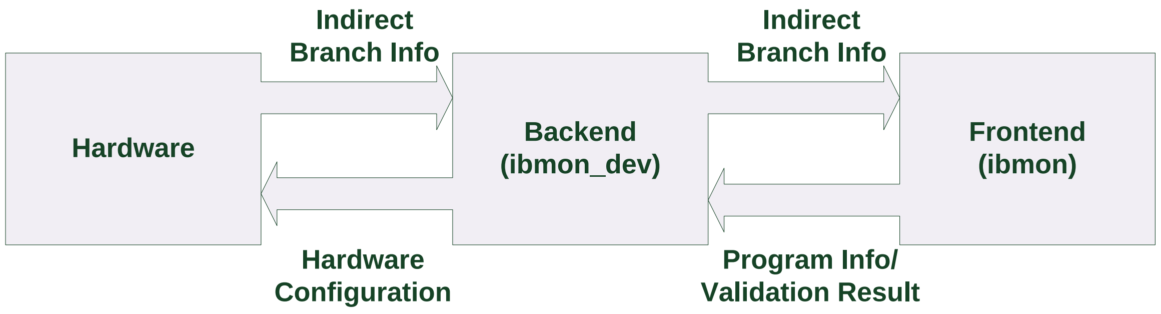
Figure 3.2(a) describes the IBMON system structure. In order to perform the operations, The system is composed of the front-end, called *ibmon_dev* and the back-end components, called *ibmon*. The linux kernel is modified to install an interrupt service routine and call back functions to detect the context switch. We also add monitor bit into ptrace flag in the process structure in the kernel. The monitor bit propagates when child process is created. Hence IBMON can selectively monitor programs.

Figure 3.2(b) describes the information flow of the IBMON system. The back-end, *ibmon_dev*, is the kernel module. It sets appropriate MSRs to enable the performance monitoring features and the debug feature. It also provides an interrupt service routine that retrieves indirect branch information from the LBR stack and forwards the IBPs to the front-end *ibmon*. To minimize the communication frequency between *ibmon_dev* and *ibmon*, *ibmon_dev* maintains a software cache per the monitored application for the recently collected IBPs. If the collected IBP is in the cache, *ibmon_dev* skips forwarding the information to *ibmon*.

The user interface,*ibmon*, is provided to wrap the target application. *ibmon* also sets the operation mode and informs the target application to *ibmon_dev*. Depends on the operation mode, *ibmon* performs either the validation with the forwarded IBP or stores them into user space. During the training mode, *ibmon* constructs the hash table from the forwarded IBPs. During the monitoring mode, it validates IBPs with the constructed hash table. When the validation is failed, it raises an alarm and then the target application is suspended. An analyzer may be called to confirm the validation



(a) IBMON System Stack



(b) Information Flow in IBMON

Figure 3.2 IBMON System Structure

result optionally.

3.3 Experiment Result

We first analyzed the effectiveness of control flow monitoring for control flow attack and the limitations of using the performance monitoring feature for control flow monitoring. Then we present the detailed results of the performance overhead incured by IBMON. We have verified our system using several vulnerable programs, including the proof of concept and the real world exploits.

3.3.1 Effectiveness

The functional verifications of IBMON are done in the machine, running Linux-2.4.26 on Intel Pentium 4 630 processor with 2 GB memory, and Bochs x86 emulator [2]. We conducted a set of benchmarks with various vulnerabilities such as the buffer overflows and the format strings including the test programs in the Libsafe package [12]. We also conducted functional the experiment with vulnerable applications running on RedHat 6.2 on Bochs emulator.

- Stack buffer overflows: Stack smashing attacks overflow a buffer in the run-timestack to overwrite the return address. We conducted the experiments based on the classic examples in [53, 12]
- Heap buffer overflows: Similar to stack smashing attack, attackers overflow a buffer in heap area to overwrite a function pointer. We conducted the experiment on modified sample attack code in [23]
- Format String Attacks: Attackers can modify the arbitrary memory locations with the arbitrary values via the format string vulnerability. We conducted experiment on sample attack code in [59].
- vudo: Attackers can corrupt the heap structure to modify the GOT entry to call the malicious code when *free()* is called. We conducted the experiment based on vudo [41]
- *wu-ftpd*: Attackers can execute an arbitrary code on the FTP server with the privileges of the FTP daemon by exploiting an error in the file globbing that leads the heap corruption [10].
- *tracerout*: Attackers can locally exploit *traceroute* vulnerability. The *free()* is called twice with a pointer when multiple command line arguments are given with the same flag [27].

Exploit Name	Control Data Type	Payload	Vulnerability Type	Prevention?
Stack Smashing	Return Address	Injected Code	Buffer Overflow	Yes
Heap Buffer Overflow	Function Pointer	Injected Code	Buffer Overflow	Yes
Format String	Function Pointer	Injected Code	Format String	Yes
vudo	GOT entry	Injected Code	Heap Corruption	Yes
wu-ftpd	GOT entry	Injected Code	Heap Corruption	Yes
tracerout	GOT entry	Injected Code	Heap Corruption	Yes
su-dtors	.dtors Section	Injected Code	Format String	Yes

Table 3.2 Experiment Result

- glibc executing /bin/su: Attacker can exploit the format string vulnerability in glibc's locale functionality to overwrite *.dtors* section to execute an arbitrary code [11].

Table 3.2 shows the result of the effectiveness test. IBMON successfully detected all CFAs with the moderate security policy that allows control transfers even though the IBPs are not in the pre-collected database. We also extended our experiments to test the return-into-libc attack. Buffer overflow attacks with injected code execution are detected since the injected code is in data area. However the moderate policy misses the return-into-libc attack when the both the source and the destination addresses are belong to the library routine. That is the same security level as the one provided by Program Shepherding. However the attack is detected with the most restrictive security policy that only allows the control transfers if they are found in the pre-defined legitimate IBPs.

3.3.2 Performance Analysis

The performance data is collected by Intel VTune performance Analyzers [38] and IBMON. Table 3.3 shows the summarized tested platforms. The IBMON systems are implemented across three different commercial microprocessors: Pentium 4, Core 2 Duo and Xeon 5520. In order to support the IBMON systems, two versions of the linux kernels have been modified to support the context switching and the target applications tracking.

Processor	Cock Speed	Architecture	Memory	OS	gcc
Pentium 4	3 GHz	NetBurst(32-bit)	2GB	Linux 2.4.26	3.1.4
Core2 Duo	2.13 GHz	Core(32-bit)	2GB	Linux 2.6.27	4.1.2
Xeon 5520	2.27 GHz	Nehalem(32-bit)	2GB	Linux 2.6.27	4.1.2

Table 3.3 Tested Platforms

In order to evaluate the performance overhead of IBMON, we obtained performance data from the multiple runs of several benchmarks. The SPEC2000 benchmarks [5] and the number of different server applications, including FTP, web and database servers, are compiled with *gcc* compilers. SPEC2000 benchmarks are compiled with the -O2 optimization flag.

Figure 3.3, figure 3.4 and figure 3.5, show the results of the branch instruction profiling of SPEC2000 benchmarks. Since CFP2000 benchmarks, floating point programs, exhibit a negligible number of executed indirect branches, the performance impact is negligible for the floating point benchmarks. Hence we only focused on analyzing the CINT2000, integer program, benchmarks. The x-axis represents the names of the benchmarks and the y-axis is the ratio of the retired indirect branches for the Pentium 4 processor and the executed indirect branches for Core 2 duo and Xeon processors. Pentium 4 can count the retired indirect branches but the Core 2 Duo and the Xeon only support counting the executed indirect branches. Therefore the ratio of the indirect branches of Pentium 4 is slightly less than the one in other processors. However the the Xeon processor does the better indirect branch predictions so that its mis-prediction per instruction is similar or less than the one in Pentium 4 processor. The indirect branch prediction of Core2 Duo processor is comparable to the one in Pentium 4 processor, but the accuracy of the return address prediction is lower than the one in Pentium 4 and the Xeon processor. Averagely, the ratio of indirect branch instruction, including the return instructions, to the retired all instructions are 1.4% for the Pentium 4 processor, 1.55% for the Core 2 Duo and 1.73% for the Xeon processor. The averages of indirect branch mis-prediction ratio to the retired instructions are 3.4% for the Pentium 4 processor,

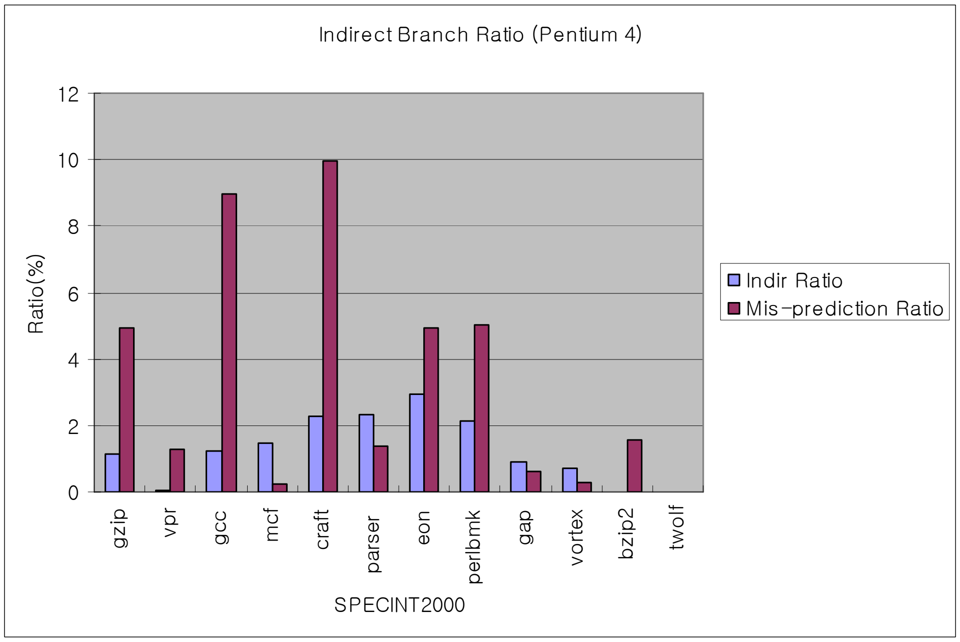


Figure 3.3 Indirect Branch Behavior of SPECINT2000 on Pentium 4 processor

10.94% for the Core 2 Duo and 1.76% for the Xeon processor.

The frequency of the indirect branch mis-predictions has the direct relationship with the performance overhead. Figure 3.6, figure 3.7 and figure 3.8 show the number of mis-predicted indirect branch instructions per the retired instruction. The number of indirect branch instructions over 1000 instructions of the integer programs range from 0.00 to 1.01 with an average of 0.44 for the Pentium 4 processor, from 0.12 to 2.89 with an average of 1.3 for the Core 2 Duo processor, and from 0.00 to 1.3 with an average of 0.38 for the Xeon processor. Since validation is performed upon mis-prediction events, the small values indicate that the performance overhead is likely to be acceptable.

Equation 3.1 gives the close estimation of the performance overhead as the percentage of execution time increase. Where the average checking overhead is the time of a validation represented in the processor cycles. For example, a quick estimation could be given as follows. The *parser* program, on average, exhibits only about 0.03 mis-prediction per one thousand instructions on the Xeon processor. The average CPI for the *parser* is 1.00

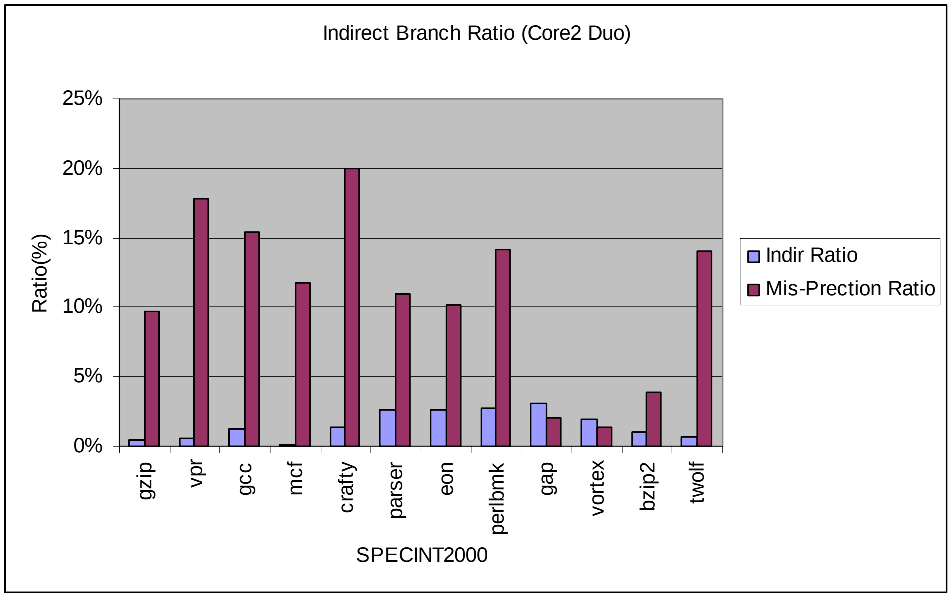


Figure 3.4 Indirect Branch Behavior of SPECINT2000 on Core2 Duo processor

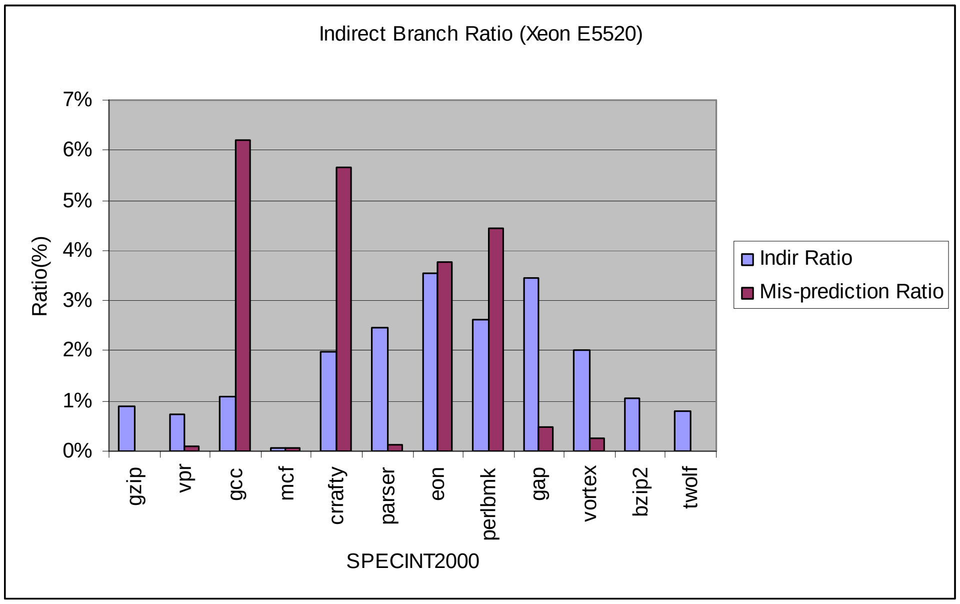


Figure 3.5 Indirect Branch Behavior of SPECINT2000 on Xeon processor

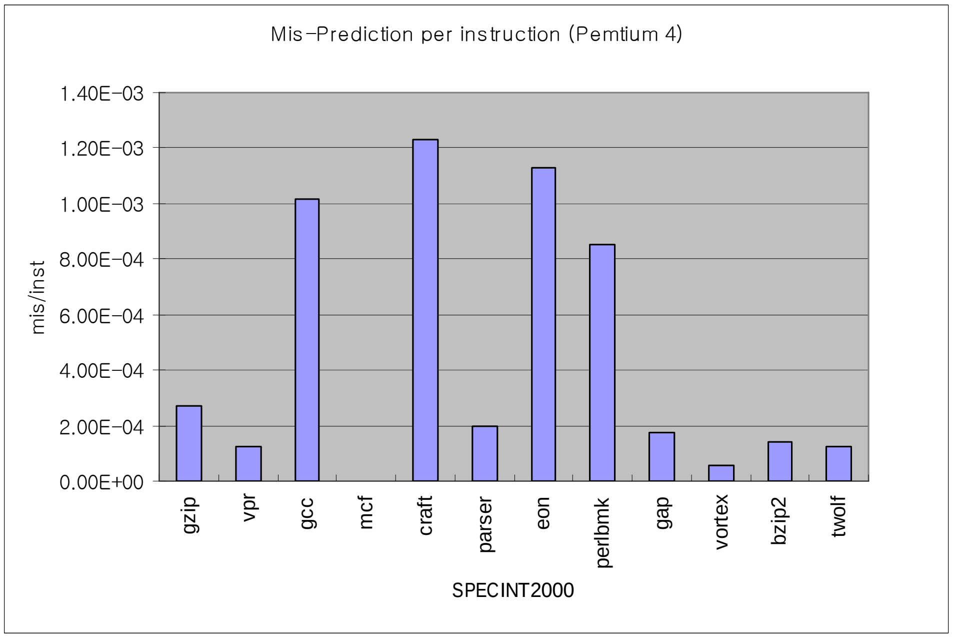


Figure 3.6 Indirect Branch Instructions per Instruction on Pentium 4 processor

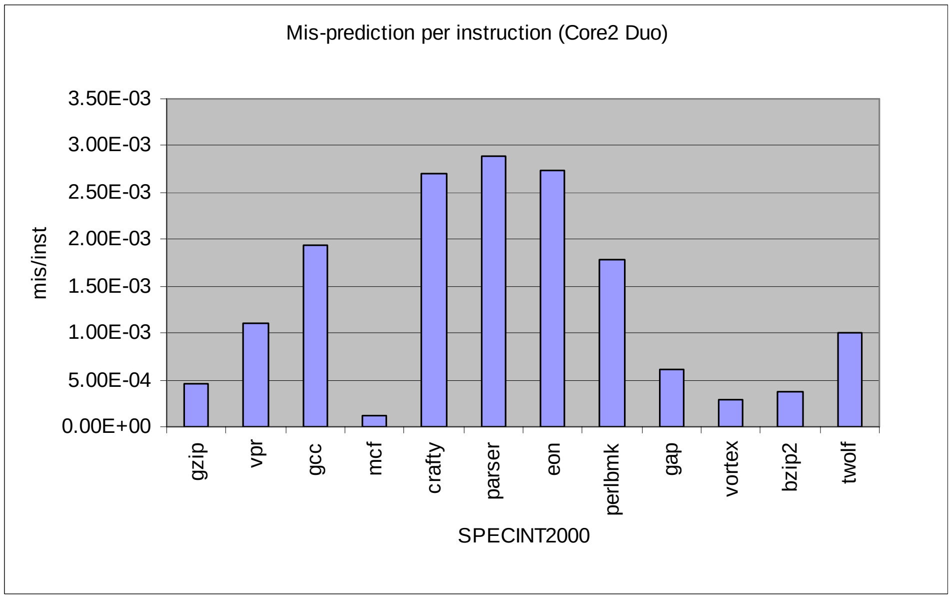


Figure 3.7 Indirect Branch Instructions per Instruction on Core2 Duo processor

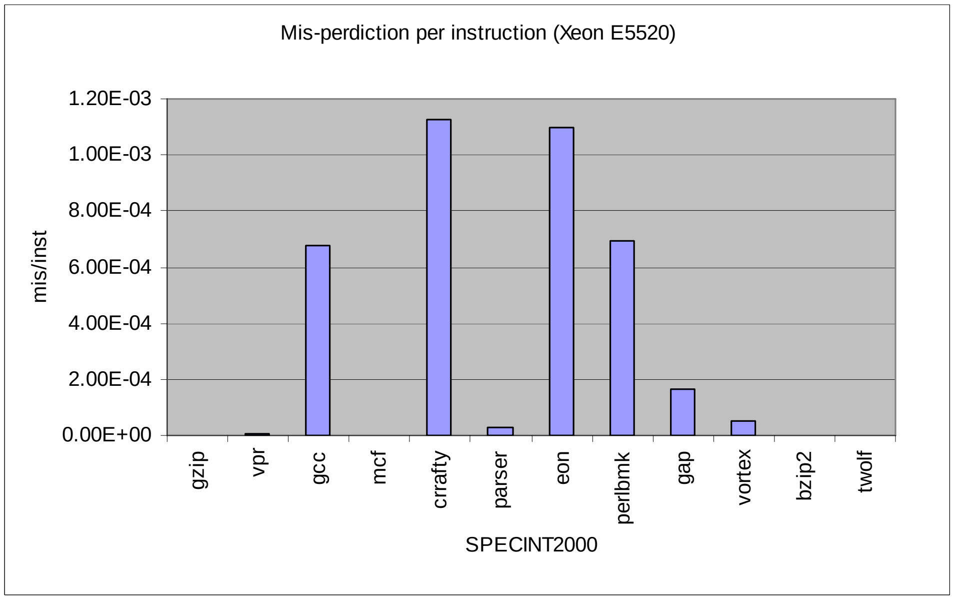


Figure 3.8 Indirect Branch Instructions per Instruction on Xeon processor

$$\text{Performance overhead} = \frac{\text{Average checking overhead} \times \text{Mis-prediction per instructions}}{\text{Cycle per instructions}} \tag{3.1}$$

Processor	*rdmsr*	*wrmsr*	*Overall*
Pentium 4	308	1523	1600/ 3600
Core2 Duo	299	303	1229
Xeon 5520	146	154	1074

Table 3.4 Instruction and Interrupt Handler Latency

and the checking overhead is 1021 cycles. Hence the estimated performance overhead is 3.10%. The major performance overhead is incurred from the interrupt handler that reads IBPs and resets the performance monitors. The performance monitoring features consists machine specific registers(MSRs) which are on the virtual data path [37]. In order to read and write from/to the MSRs, the machine specific instructions, *rdmsr* and *wrmsr*, are used. Note that *wrmsr* instruction is the serializing instruction that flushes pipeline when it is issued. Table 3.4 shows the average latency of machine specific instructions and the average latency of the IBMON interrupt handler routine.

Each microarchitecture has a different implemenation for the performance monitoring features and the monitored events. For example, as briefly mentioned before, the Pentium 4 supports the sampling of the retired mis-predicted and/or indirect branch events, however the Xeon and the Core 2 Duo only supports for the sampling of the executed event whose instruction may not be retiring. Consequently IBMON in the Xeon and the Core 2 Duo may count more mis-predicted indirect branch events. The performance monitoring facility generates the performance monitoring interrupts(PMIs) based on the performance monitoring counter(PMC) overflow event. When a specific event is set to be monitored, the corresponding PMC is initialized. The PMCs can be read and written via the machine specific instructions,*rdmsr* and *wrmsr*. Also, when a PMI is generated, the interrupt handler must reset the control register to monitor the target application continuously. For the Xeon and the Core 2 Duo, the PMC must be set to PMC_MAX - 1 to generates a PMI on every event. In contrast, the Pentium 4 supports the forced PMI that generates a PMI without setting the value of the corresponding PMC. Furthermore, the number of PMCs used for monitoring mis-predicted indirect branch is different. In the Pentium 4 and the Core 2 Duo processors, indirect branches are classified into two categories: *indirect branch* and *return*. However, in the Xeon processor, indirect branches are further classified into *non-call indirect branch*, *call indirect branch* and *return*. Therefore there is an extra step to determine which PMC generate PMI to reset the corresponding PMC.

In Figure 3.9, figure 3.10 and figure 3.11, the first serie presents the performance overhead of IBMON for SPEC2000 integer benchmarks when every indirect mis-prediction is validated. Although validating the integrity of the IBPs upon every event of mis-prediction reduces the validation frequency, the IBMON systems show the large performance overhead on some of the benchmarks. Since the checking overhead is larger than in-line reference checking [8] due to the high interrupt latency, they show slightly worse performance overhead compare with CFI; on the Pentium 4 processor, the performance overhead is ranged from 0% to 138% with an average of 21.26%. The result of performance measurement on the Core 2 Duo is somewhat pessimistic. Although the latency of the serializing instruction of Core 2 Duo, due to the inaccuracy of the indirect branch prediction, the Core 2 Duo exhibits the worst performance overhead that ranges from 2.2% to 348% with an average of 161%. On the Xeon processor, the system shows the better performance on most benchmarks. However a few of them exhibit the large performance overhead: *gcc*(60.87%), *crafty*(167.07%), *eon*(117.59%) and *perlbmk*(92.35%). The latency of serializing instruction and the performance of the indirect branch predictors are dramatically improved for most benchmarks as the average overall CPI is also improved on the system. Therefore the performance overhead ranges from 0.08% to 167.07% with a mean average of 40.36%.

Hence we further scrutinized the system and CFAs to reduce the performance overhead of the IBMON systems. Since we were not able to reduce the overhead of the interrupt handler of the IBMON, we closely examined the payload of the control flow attacks. In order to make a successful attack with injected payload, the series of indirect branches are used in the payload. Since these indirect branches are in the injected code, all of them occur indirect branch mis-predictions. Similarly, in order to achieve

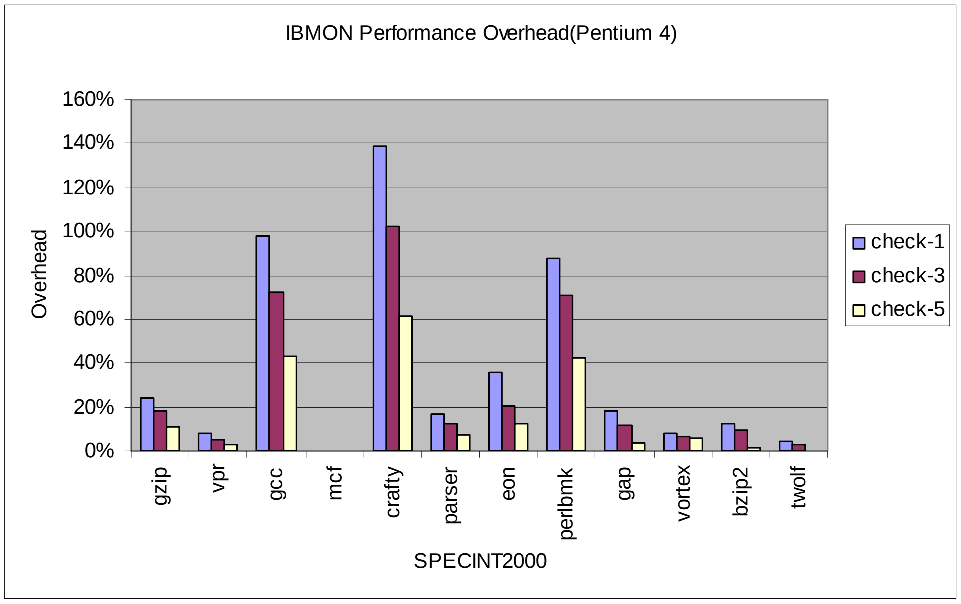


Figure 3.9 IBMON Performance Overhead on the Pentium 4 processor

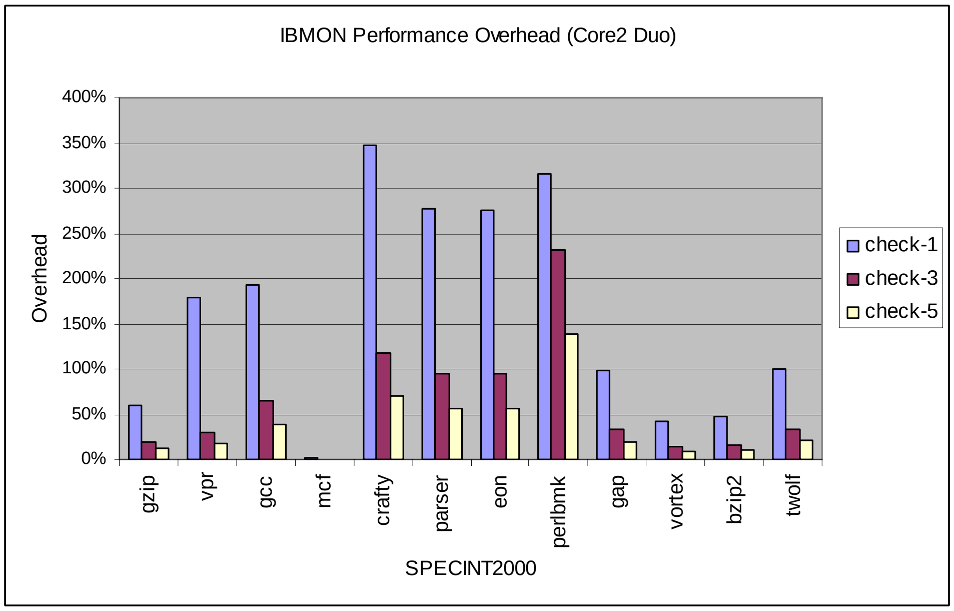


Figure 3.10 IBMON Performance Overhead on the Core 2 Duo processor

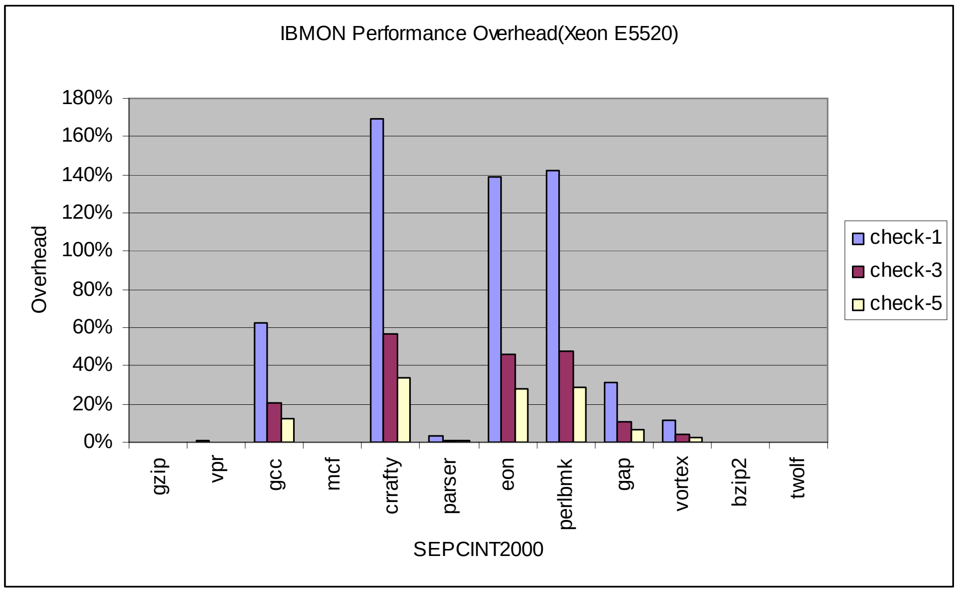


Figure 3.11 IBMON Performance Overhead on the Xeon processor

successful return-into-libc attack, attackers also need to craft multiple run-time stack frames [52]. Whenever the control transfers are made with the crafted control data in the run-time stack, they also generate events of mis-prediction on return addresses. Furthermore, when the injected code executes code segments that are not the part of the vulnerable program, the IBPs of the code segments, highly likely, are not in the IBP database of the vulnerable program. Most injected code attacks invoke *execv()* with */bin/sh* parameters. In order to find the location of the parameter dynamically, first, the vulnerable program returns to the attack code. Secondly, the attack code makes relative jump, indirect jump, to the *call* instruction that is placed immediately before the parameter. Thirdly, the attack code makes the relative call, also indirect call, to the first instruction of the preparation code for calling *execv()*. Hence the attack code requires at least three indirect branch mis-prediction events to invoke the *execv()* system call. Therefore it is safe to validate IBPs on every three indirect branch mis-prediction events without compromising security level for the injected code attack. Furthermore, after the *execv()* system call, the *shell* code has different IBPs from the vulnerable program, any event of mis-predict indirect branch will raise alarm. Similarly, it is also safe to validate every three or five mis-predicted IBPs without compromising security strength for the advanced return-into-libc attacks. With the different validation intervals, we also conducted effectiveness experiment. With the validation on every indirect branch mis-prediction event, IBMON detects the CFA at the time of control flow hijacking. With the validation on every three indirect branch mis-prediction events, the attack has been detected during ithe preparation routine for the main attack body. With the validation on every five indirect branch mis-prediction events, the attack has been detected in the early stage of the main attack in the worst case. Hence with the most restrictive security policy of IBMON, IBMON can successfully detects all the attacks in table 3.2.

The second and the third column in figure 3.9, figure 3.10 and figure 3.11 present the performance overhead of IBMON for validation IBPs on every three and five indirect branch mis-prediction events accordingly. On the Pentium 4 processor, this validation requires additional reset routines for the PMCs. Since the average validation latency becomes 1600 cycles to 3600 cycles, the performance of the larger validation interval is not dramatically improved. The performance overhead for validating every three events rages from 0% to 102% with an average of 12.57%. On the Xeon and the Core 2 Duo processors, configuring the larger validation interval does not require additional routine, the performance improvement is roughly proportional to the number of validations. On the Core 2 Duo processor, the performance overhead ranges from 0.7% to 232% with an average of 62.79%. On the Xeon processor, when the IBOM system, on the Xeon processor, validates the IBPs integrity on every three indirect mis-prediction, the performance overhead of IBMON is comparable to the one presented in CFI. The performance overhead ranges from 0% to 56% with an average of 13.81%. For the largest validation intervals, validating every five events, the results of performance overhead measurement are optimistic for the Pentium 4 and the Xeon processor. On the Pentium 4 processors, the performance overhead ranges from 0% to 61% with an average of 7.5%. Similarly, on the Core 2 Duo processor, the performance overhead ranges from 0.4% to 139% with an average of 37.67%. The IBMON system, on the Xeon processor, out performs CFIs when it validates IBPs on every five indirect branch mis-prediction events. The performance overhead ranges from 0% to 33.51% with an average of 8.31%.

We also conducted server program benchmarks. In this performance test, we measured performance overhead on the Core 2 Duo processor with 1Gbps Ethernet connection. IBMON is configured to validate IBPs on every mis-predicted indirect branch event. For web server benchmarks, we used httpd-2.2.11-2, nginx-0.7.59, and lighthttpd-1.4.22 as web servers. We perform the benchmarking with various size of static web pages as well as the various number of the concurrent web page requests to the servers. The benchmarking tool *ab* is used for generating and measuring web server performance. Figure 3.12, figure 3.13 and figure 3.14 show the static web benchmarks for three different servers. We measured the throughput, web page request per second, of the web servers. The average performance overhead is negligible for nginx (1%)and lighthttpd(0.23%) server. X-axies shows the number of concurrent web page request and Y-axis show the performance degradation ratio in terms of request per second. Each series shows the different size of web page in kilo-byte. Note that the average size of web pages on the Internet is approximately 12KB.

Similarly, we also conducted other server programs which are FTP servers *vsftpd-2.0.7, wu-ftpd-2.6.2* and database server *PostgreSQL-8.3*. We used *dkftpd-0.45* to measure the performance overhead for FTP servers. Various size of files, ranging from 1KB to 100MB, are requested to FTP server and the throughputs are measured. The result shows that the overall performance degradation is negligible. Hence we do not report the details. The performance of *PostgreSQL-8.3* under IBMON is measured with TPC-C specification. The TPC-C specification is the on-line transaction model for a database server to measure the several metrics. The performance degradation for the database server is around 3%.

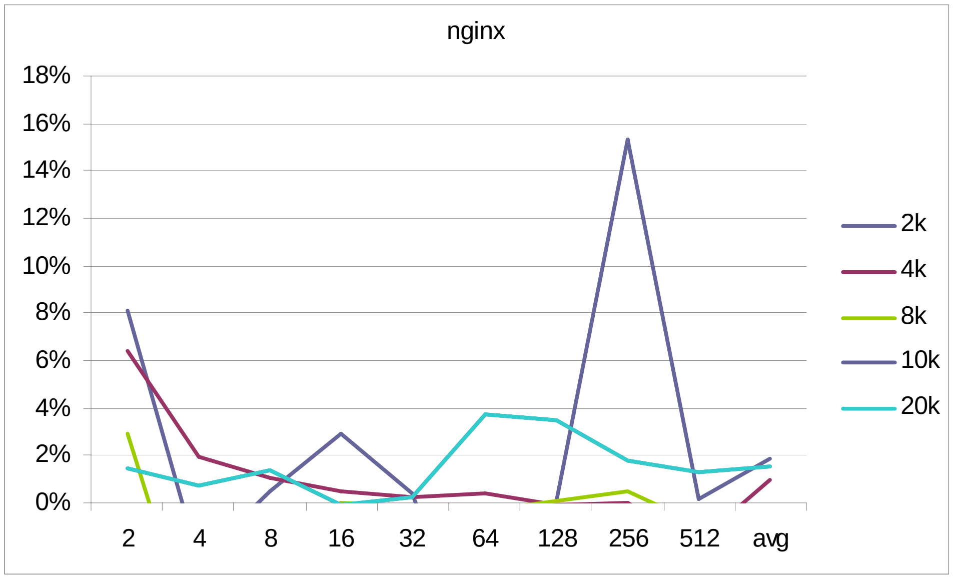


Figure 3.12 *nginx* Web server Performance under IBMON Monitoring

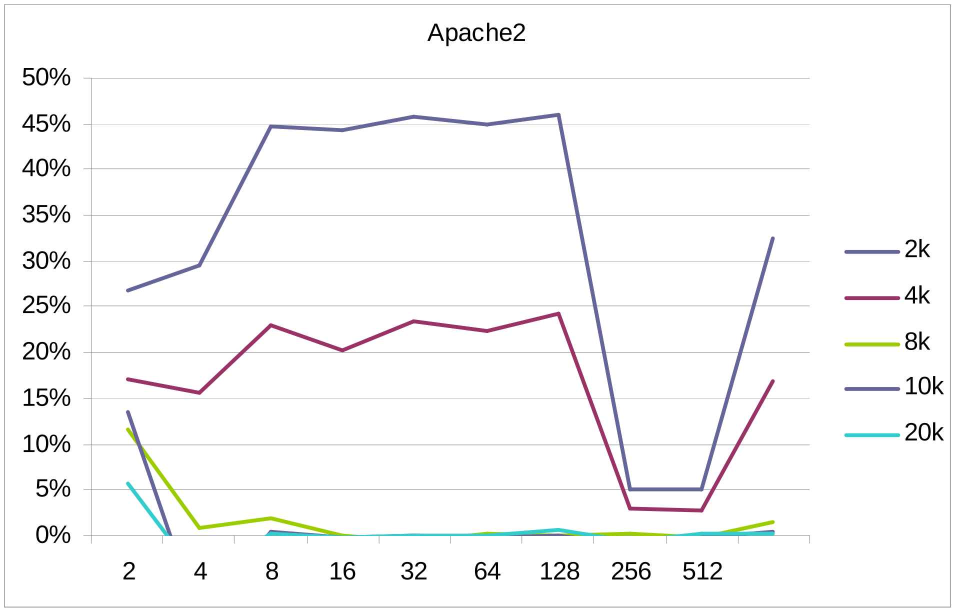


Figure 3.13 *Apach2* Web server Performance under IBMON Monitoring

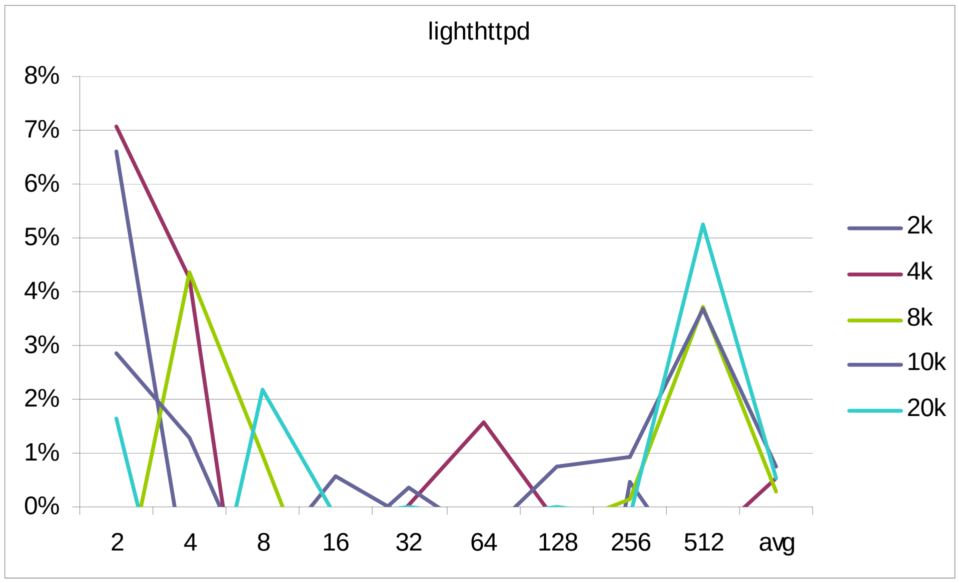


Figure 3.14 *light-httpd* Web server Performance under IBMON Monitoring

4 IBF-Cache: Hardware Support for Efficient Control Flow Validation

Although the processor achieves low branch mis-prediction rate, the IBMON system-still incurs the non-negligible performance overhead. Therefore, this thesis also proposes an efficient hardware mechanism to reduce the validation frequency. The Indirect Branch Filter(IBF)-cache mainly takes advantages of the temporal locality of the behavior of indirect branches in programs. The IBMON system maintains the software cache to reduce the communication between the kernel module and the user mode application. Since the number of IBPs are relatively small and indirect branches exhibit the high temporal locality, the IBMON system reduces the large number of the user level validations. Therefore, when the cache is implemented in the hardware, the frequency of the most expensive component, interrupt handling, is reduced. Hence, implementing the IBF-cache is the straight-forward approach to reducing the number of interrupts that result in reducing the large portion of the performance overhea. The subsequent sections present more details of the IBF-cache.

4.1 Indirect Branch Filter(IBF)-Cache

The IBF-cache records the indirect branch address pairs that have been validated recently. It is a small component and is as fast as L1 caches. For a mis-predicted indirect branch, the validation starts at the execute/write-back stage where the branch and target addresses are known. This address pair is sent to the IBF-cache. If the pair resides in the cache, the pair has been previously validated. This validation can be done within branch mis-prediction penalty cycle, hence, there will be no overhead for accessing IBF-cache.

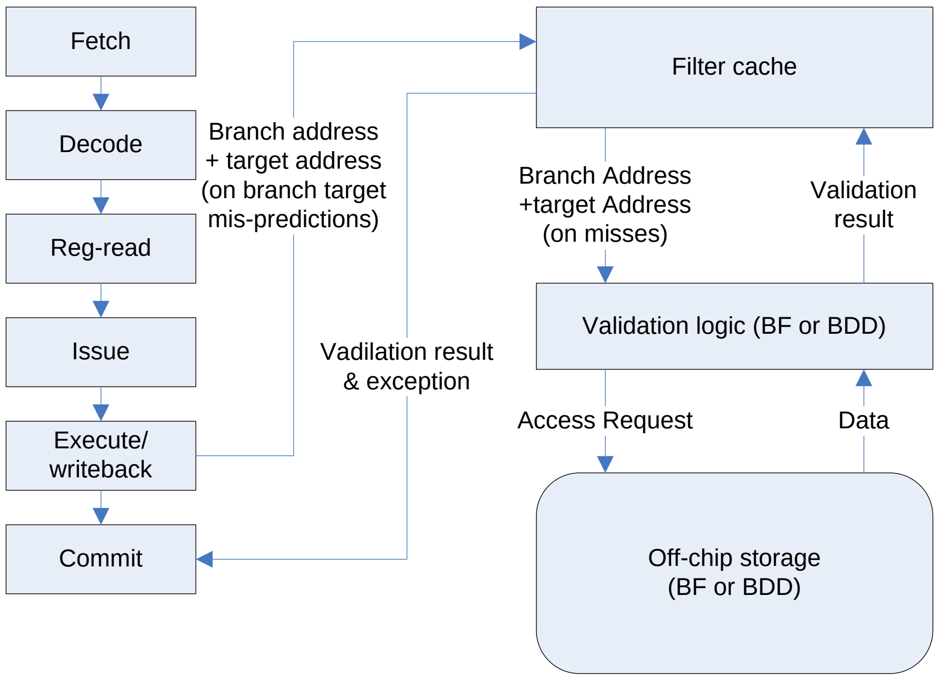

Fetch
Decode
Reg-read
Issue
Execute/ writeback
Commit
Branch address + target address (on branch target mis-predictions)
Vadilation result & exception
Filter cache
Branch Address +target Address (on misses)
Validation result
Validation logic (BF or BDD)
Access Request
Data
Off-chip storage (BF or BDD)

Figure 4.1 The overview of the IBF cache design with data path connections to a 6-stage out-of-order pipeline.

Upon IBF-cache miss, the hardware generates an interrupt through the performance monitoring features for the IBMON system. The system validates the control flow with the information stored in the off-chip main memory, expensive validation. If the outcome is negative (no alarm), the monitoring tool returns from the interrupt routine. Otherwise, the IBMON system raises an exception and let an analyzer further examines the program.

Figure 4.1 shows the overall structure of the hardware design with the data path connections to an out-of-order pipeline using issue queue with conventional BTB (branch target buffer) designs. We presents a 6-stage pipeline for simplicity, while our simulation uses a 9-stage pipeline.

The validation is done only for an indirect branch that causes a branch mis-prediction. For a correctly predicted indirect branch, the BTB (branch target buffer) must have predicted the target address correctly, which also means the BTB has seen the branch address pair before. The first time the branch address pair is brought into the BTB, it must have caused a branch mis-prediction and have been validated.

The IBF cache uses an XOR-based indexing scheme, which are from both the branch address and the target address, to avoid misses from indirect branches which have multiple targets during the program execution. If an indirect branch uses multiple targets, the XORing of the target address will distribute all branch address pairs of this branch over the cache address space. Otherwise, those branch address pairs will be mapped onto the same cache entry, causing severe conflicts. XOR-based indexing scheme has been used in both cache indexing and branch target address prediction (XORing branch address with branch history). Nevertheless, the branch target address prediction cannot use the XORing with branch target address because it is unknown at the time of branch prediction. This is the reason the IBF cache can not be replaced by an enlarged BTB.

4.1.1 Nonblocking IBF Cache Design

There are several reasonable designs for the control and data path between the pipeline logic and the IBF cache. A simple design is *blocking IBF cache*: The pipeline stalls when an indirect branch miss in the IBF cache. Since the IBF cache is small and fast, a hit will cause little delay. An IBF cache miss, nevertheless, may cause the pipeline to stall for hundreds of cycles. We found in our experiments that even the blocking IBF cache works well for many programs because the IBF cache has very low miss rate.

However, some programs with very large instruction footprint and fair temporal locality in the control flow may favor a non-blocking IBF cache design. For those programs, the amount of active control path may exceed the size of the IBF cache. A non-blocking design may reduce the pipeline stalls for two reasons. First, there can be a overlap between the pipeline execution and the BF validation logic. On a branch target misprediction, it will take dozens of cycles for the pipeline to recover for the mis-prediction. The pipeline may not stall until the ROB, the issue queue, or any other resources are full. More importantly, a nonblocking design allows concurrency in the BF validation logic, which reduces the average penalty of IBF cache misses. This is particularly true for today's high-bandwidth DRAM memory systems. For example, the BF validation logic may need to access the DRAM four times for a validation request. While the first access may take 70ns to finish, the four accesses may be fully pipelined on a FB-DIMM memory system [39], and each other access only takes 5ns extra. In other words, a first validation may finish within 85ns, and the following ones only take 20ns each, if the validation logic can see all pending requests. After all, a non-blocking design is also very simple.

Figure 4.2 illustrates the design for non-blocking IBF cache. The commit logic is revised to include a V-bit (Validation Bit) vector. The function of V-bit is to tell if an indirect branch it is validated successfully or not when the instruction is to be committed. The V-bit vector has as many entries as the ROB, but is made separate to avoid conflicts in ROB accesses. When an instruction is decoded, it is allocated with a V-bit in additional to a ROB entry and any other resources the instruction may require. The V-bit is set to 1 by default. If the instruction is an indirect branch and causes a branch target mis-prediction, the pipeline is flushed and at this time its V-bit is reset to 0. The completion bit, which is already in the ROB entry, is reset and will be set when the validation is done. Three information fields are then sent to the IBF cache controller: the ROB index, the branch address, and the target address. The latter two are used to access the IBF cache. If hit, then the validation request is done; otherwise, the IBF cache sends a request to BF validation logic. In either case, if the validation is negative, the IBF cache will assert the "Validated" signal and send back the ROB index; the corresponding V-bit is set and so is the completion bit of the instruction. The commit logic will raise a special exception when it finds at the ROB head an indirect branch that is ready to be committed and its V-bit is zero. The operating system will handle this special exception and examine the program states in detail.

4.2 Experimental Methodology

We have verified and evaluated our design by cycle-accurate simulation and trace simulation. The majority of the performance evaluation is on the IBF cache design. The overall performance overhead is virtually nonexistent because of the use of IBF

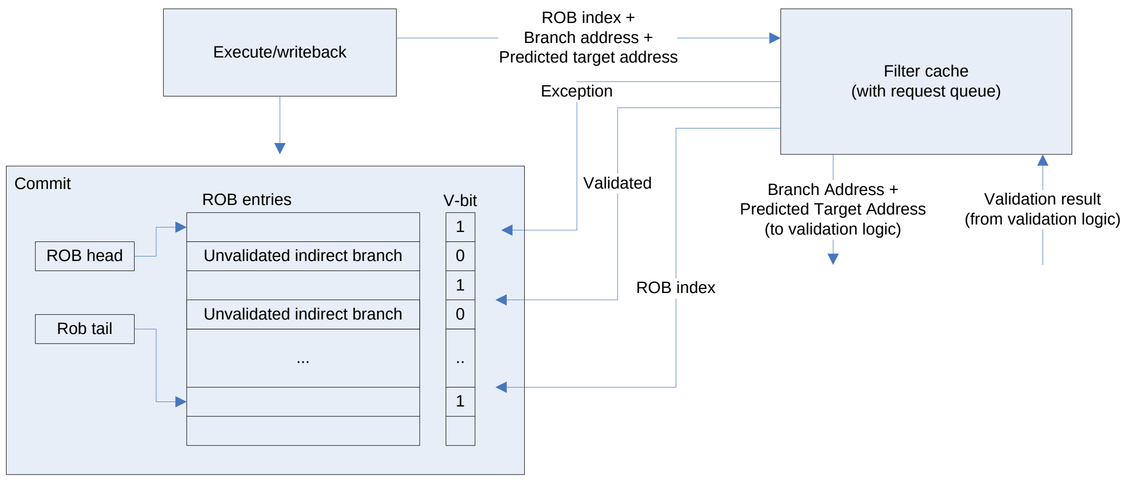


Figure 4.2 The design of a non-blocking IBF cache the an out-of-order pipeline.

cache.

4.2.1 Cycle-Accurate Simulation

We have extended the SimpleScalar simulation suit [14] to simulate the IBF cache and the BF validation logic. The pipeline simulation is revised to access the IBF cache upon branch mis-predictions on indirect branches as discussed in Section 4.1. The index of IBF-cache is selected from a simple XOR-based hash value of the branch and target addresses. Note that the IBF cache only have tags, which are the concatenation of the branch address and the target address. In the simulation, we use the SPEC CPU2000int benchmarks and run them to the completion by using the MinneSPEC input sets. We do not use the SPEC CPU2000fp benchmarks because they have negligible fractions of indirect branches. Table 4.1 shows the most important simulation parameters.

4.2.2 Trace-based IBF Cache Simulation

Because of the limit of the run-time execution environment of SimpleScalar, we cannot run many real-world workloads in the simulation. We use trace-based simulation to evaluate the IBF cache with a wider range of workloads than SPEC CPU2000int. The trace-based simulation does not report the overall performance overhead, but we

Parameters	Values
Processor speed	3.2 GHz
Pipeline	8-issue, 9-stage
Functional units	8 IntALU, 2 IntMult, 8 FPALU, 2 FPMult
Issue queue size	32
Reorder buffer size	128
Load/store queue size	32 LQ, 32 SQ
Branch predictor	gshare predictor of 8192 counters, 8-entry RAS
Misprediction penalty	7 cycles
L1 caches	64KB Inst/64KB Data, 4-way, 2-cycle hit latency
L2 cache	Unified 2MB, 8-way, 12-cycle hit latency
Memory	500-cycle latency

Table 4.1 Simulator parameters.

can estimate the overhead from the IBF cache miss rate and the latency of the BF validation logic. We utilize the performance monitoring features in Intel Pentium 4 to monitor the events of indirect branch mis-predictions and to generate the trace of address pairs, and then use them as input to a trace-based simulation. We compile the programs with gcc 3.3.2 and run them on Redhat Linux with kernel 2.4.26. We have used the following workloads:

We collected the trace of indirect branch mis-prediction on Intel Pentium 4 processor using IBMON. The trace is then fed into a trace-based simulation of the IBF-cache. We also use the validation latency of Pentium4 processor to evaluate the performance overhead. We have used various workloads in our performance evaluation: TPC-C workload with *Postgres* 7.4.13 database system, *WebStone 2.5* benchmark with *Apache 2.0.47*, and *dkftpd* benchmark, FTP benchmark, with *vsftpd* demon. We also evaluate *SPECINT 2000* and *SPEC 2006* integer benchmarks. All programs are compiled with gcc 3.3.2 and run on Redhat Linux with kernel 2.4.26.

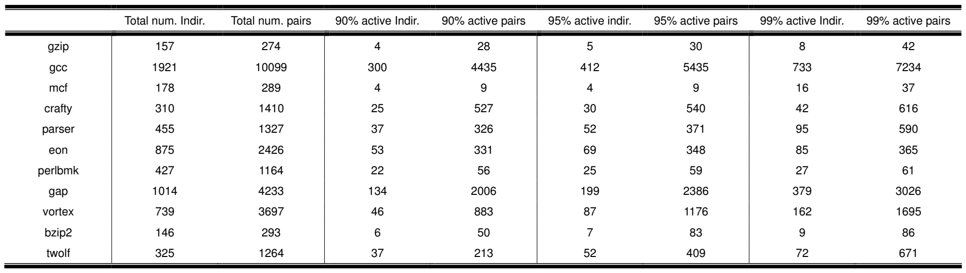
- SPEC CPU2000int and SPEC CPU2006int benchmarks: We use the reference input sets and run all programs to completion. The CPU2006int programs have generally more complex source code than the CPU2000int programs.

- TPC-C workload: TPC-C is an OLTP (on-line transaction processing) workload that emulates warehouse transactions using a database system. We use Postgres 7.4.13 as the supporting database system.
- WebStone 2.5 benchmark: The WebStone benchmark creates a load on a Web Server by simulating the activities of multiple clients. We configure WebStone to use two different types of access methods, HTML and CGI, with 10 to 100 simultaneous clients. The underlying web server is Apache 2.0.47.
- FTP Workload: We use an FTP demon called *vsftpd*, version 1.2.0-5, and an FTP benchmark called *dkftpbench*.

4.3 Experiment Results

4.3.1 Cycle-Accurate Simulation Results

This experiment uses the SPEC2000int benchmarks with the MinneSPEC input sets. Since the IBF cache utilizes the temporal locality in the program control flow, it is important that good locality does exist in the workloads. We found that the indirect branch makes up about 1.5% of all instructions. The mis-prediction ratio for the SPEC integer benchmarks is no more than 10% and on average 3.1%. Table 4.3 shows the indirect branch profiling for SPECINT2000 benchmarks. We do not include the floating point benchmarks because those floating-point programs are much less branch-intensive and therefore the performance overhead for them is not a concern. The table shows that a small subset of static indirect branches make up a large portion of dynamic indirect branches. The second and third columns of the table are the total number of static indirect branches and the total number of unique targets, respectively. The fourth column is the number of static indirect branch instructions that are responsible for 90% dynamic indirect branch instructions. The fifth column is the number of unique indirect branch address pairs that are observed in the execution of 90% indirect branch instructions. The sixth to ninth columns are similar except that ratios 95% and 99%



	Total num. Indir.	Total num. pairs	90% active Indir.	90% active pairs	95% active indir.	95% active pairs	99% active Indir.	99% active pairs
gzip	157	274	4	28	5	30	8	42
gcc	1921	10099	300	4435	412	5435	733	7234
mcf	178	289	4	9	4	9	16	37
crafty	310	1410	25	527	30	540	42	616
parser	455	1327	37	326	52	371	95	590
eon	875	2426	53	331	69	348	85	365
perlbmk	427	1164	22	56	25	59	27	61
gap	1014	4233	134	2006	199	2386	379	3026
vortex	739	3697	46	883	87	1176	162	1695
bzip2	146	293	6	50	7	83	9	86
twolf	325	1264	37	213	52	409	72	671

Figure 4.3 Indirect branch profiling for SPEC CPU2000int.

are used. The profiling results show that it is very promising to use a small cache to capture the locality existing in the indirect branch address pairs observed in the program execution; and the frequency of validation may be significantly reduced by the IBF cache.

As discussed before, a key point of the IBF cache design is that the index bits are selected from the XOR of the branch address and target address. A BTB only uses the branch address to form the index, and therefore multiple targets of a single indirect branch will be mapped to the same entry in a direct mapped BTB. The XOR-based indexing eliminates this source of conflicts. A point worth noting is that, even if a set associative BTB is used, the number of targets is usually higher than the number of ways; and only one entry for each branch.

The overall performance overhead is negligible for all programs, therefore we do not report the performance detail. For example, the largest overhead we observed on the configuration with 2-way set associative IBF- cache of 2k entries is 0.38% (for eon), and the average is 0.02%. For the same reason, we do not show the results of non-blocking IBF cache design.

4.3.2 Trace-based Simulation Results

The trace-based simulation allows us to evaluate the IBF cache design with a wide range of applications. Although we cannot to measure the overall execution time, we can obtain the IBF cache miss rates through trace-based simulation and may estimate the overall performance overhead.

Program	256	512	1K	2K	4K	8K
gzip	0.002%	0.002%	0.002%	0.002%	0.002%	0.002%
vpr	0.003%	0.002%	0.002%	0.002%	0.002%	0.002%
gcc	1.325%	0.592%	0.415%	0.297%	0.202%	0.161%
mcf	0.591%	0.591%	0.591%	**0.591%**	**0.591%**	**0.591%**
crafy	2.217%	0.598%	0.172%	0.110%	0.106%	0.102%
parser	2.467%	0.956%	0.271%	0.085%	0.030%	0.028%
eon	0.004%	0.001%	0.001%	0.001%	0.001%	0.001%
perlbmk	**12.245%**	**3.787%**	**0.718%**	0.046%	0.024%	0.001%
gap	0.399%	0.162%	0.042%	0.016%	0.003%	0.001%
vortex	0.045%	0.028%	0.015%	0.012%	0.012%	0.012%
bzip2	0.003%	0.003%	0.003%	0.003%	0.003%	0.003%
twolf	0.016%	0.002%	0.002%	0.002%	0.002%	0.002%
Average	1.610%	0.560%	0.186%	0.097%	0.081%	0.075%

Table 4.2 The IBF cache miss rates for SPEC *CPU2000int* benchmarks with the reference inputs for cache sizes of 256 to 8K entries. The cache set associativity is fixed at four. The numbers in bold type are the maximum number for a given cache size.

IBF cache Miss Rate. Table 4.2 shows the IBF cache miss rates for all SPEC CPU2000int programs with the reference inputs. For programs with multiple reference inputs, we use the average of all inputs. The IBF cache set associativity is fixed at four and we change the size from 256 to 8K entries. As the figure shows, the IBF cache miss rate becomes very small when the cache size increases beyond 1K; the maximum is 0.718% on *perlbmk* for the cache size of 1K. The miss rate of most programs with 8K entries is close to zero.

Table 4.3 shows the IBF cache miss rates for all SPEC CPU2006int programs with the reference input sets. The CPU2006int programs have generally more complex source code than CPU2000int programs. Additionally, there are three more C++ programs, omnetpp, astar, and xalancbmk, which may have high frequency of indirect branch instructions. Again for programs with multiple reference inputs, we use the average of all inputs, and the IBF cache set associativity is fixed at four and we change the size from 256 to 8K entries. As in results for the CPU2000int programs, the IBF cache miss rate becomes very small when the cache size increases beyond 1K; the maximum is 0.898% on *perlbmk* for the cache size of 1K. The miss rate of most programs with 8K entries is close to zero. The three new C++ programs are not very different from

Program	**256**	**512**	**1K**	**2K**	**4K**	**8K**
perlbmk	1.124%	0.220%	0.071%	0.012%	0.005%	0.004%
bzip2	0.054%	0.010%	0.009%	0.009%	0.009%	0.009%
gcc	0.800%	0.297%	0.148%	0.089%	0.055%	0.030%
mcf	0.021%	0.020%	0.020%	0.020%	0.020%	0.020%
gobmk	**5.271%**	**2.193%**	**0.898%**	0.351%	0.117%	0.055%
hmmer	0.038%	0.028%	0.023%	0.022%	0.022%	0.022%
sjeng	0.540%	0.052%	0.006%	0.004%	0.003%	0.003%
libquantum	1.876%	0.706%	0.695%	**0.695%**	**0.695%**	**0.695%**
omnetpp	0.285%	0.052%	0.007%	0.004%	0.003%	0.003%
astar	0.617%	0.581%	0.568%	0.563%	0.562%	0.562%
xalancbmk	1.986%	0.988%	0.320%	0.053%	0.007%	0.004%
Average	1.147%	0.468%	0.251%	0.166%	0.136%	0.128%

Table 4.3 The IBF cache miss rates for SPEC *CPU2006int* benchmarks with the reference inputs for cache sizes of 256 to 8K entries. The cache set associativity is fixed at four. The numbers in bold type are the maximum number for a given cache size.

the other programs. When compared with CPU2000int, the average miss rate increases slightly for all sizes except 256-entry, for which the average miss rate drops slightly.

Number of Misses Per 10,000 Instructions. Table 4.4 shows the number of IBF cache misses per 10,000 instructions for SPEC2000Int. This number is closely related to the overall performance overhead. It is determined by three factors of a program: the ratio of indirect branch instructions, the branch mis-prediction rate and the IBF cache miss rate. Program *gcc* has the largest number for all cache sizes mainly because it has relatively high frequency of indirect branch instructions. With 2K IBF cache entries, there is only 0.091 miss per 10,000 instructions or less than one instruction per 100,000 instructions. We can give a ballpark estimate of the overall performance overhead as follows: Assume that the off-chip validation takes 469ns or 1500 cycles on a 3.2 GHz processor, and assume that the program CPI is 1.81 [19]. The performance overhead is about $1500 * 0.091/(10,000 * 1.81) = 0.65\%$. Similarly, we obtain an average performance overhead of 0.0059% for CPU2000int programs (Miss rate is one miss per million instructions and CPI is 2.545 on average). In fact, this estimate is pessimistic because Intel processors has relatively high latency in accessing the performance

Program	256	512	1K	2K	4K	8K
gzip	0.000	0.000	0.000	0.000	0.000	0.000
vpr	0.000	0.000	0.000	0.000	0.000	0.000
gcc	**0.366**	**0.183**	**0.131**	**0.091**	**0.060**	**0.046**
mcf	0.000	0.000	0.000	0.000	0.000	0.000
crafty	0.301	0.081	0.023	0.015	0.014	0.014
parser	0.040	0.015	0.004	0.001	0.000	0.000
eon	0.001	0.000	0.000	0.000	0.000	0.000
perlbmk	0.049	0.014	0.003	0.003	0.003	0.003
gap	0.199	0.081	0.021	0.008	0.001	0.000
vortex	0.000	0.000	0.000	0.000	0.000	0.000
bzip2	0.000	0.000	0.000	0.000	0.000	0.000
twolf	0.000	0.000	0.000	0.000	0.000	0.000
Average	0.080	0.031	0.015	0.010	0.007	0.005

Table 4.4 The number of IBF cache misses per 10,000 instructions for SPEC *CPU2000int* benchmarks with the reference inputs for cache sizes of 256 to 8K entries. The cache set associativity is fixed at four. The numbers in bold type are the maximum number for a given cache size.

monitoring counters. Even so, the performance is very good when compared with the previous work [8, 13], which has up to 50% overhead and on average 21% overhead. We found that a two-way set associative IBF cache improves significantly over the direct mapped one with the same size; and that the improvement diminishes when the degree of associativity increases beyond four.

Table 4.5 shows the number of IBF cache misses per 10,000 instructions for SPEC2006Int. Again program *gcc* has the largest number for all cache sizes mainly. With 2K IBF cache entries, there is only 0.147 miss per 10,000 instructions or about one and a half instructions per 100,000 instructions, a slight increase from the gcc program in SPEC CPU2000int. A ballpark estimate like the previous one gives an overall performance overhead of 0.08%. Again it shows that the overall performance overhead is negligible for all SPEC CPU2006Int programs. Overall the average number of misses per 10,000 instructions increases only slightly from SPEC CPU2000int for all configurations except 256-entry, for which the number drops.

Program	**256**	**512**	**1K**	**2K**	**4K**	**8K**
perlbmk	0.690	0.135	0.044	0.008	0.003	0.002
bzip2	0.000	0.000	0.000	0.000	0.000	0.000
gcc	**2.201**	**0.781**	**0.313**	**0.147**	**0.088**	**0.058**
mcf	0.000	0.000	0.000	0.000	0.000	0.000
gobmk	0.042	0.017	0.007	0.003	0.001	0.000
hmmer	0.000	0.000	0.000	0.000	0.000	0.000
sjeng	0.185	0.018	0.002	0.001	0.001	0.001
libquantum	0.000	0.000	0.000	0.000	0.000	0.000
omnetpp	0.032	0.006	0.001	0.000	0.000	0.000
astar	0.000	0.000	0.000	0.000	0.000	0.000
xalancbmk	0.108	0.054	0.017	0.003	0.000	0.000
Average	0.296	0.092	0.035	0.015	0.009	0.006

Table 4.5 The number of IBF cache misses per 10,000 instructions for SPEC *CPU2006int* benchmarks with the reference inputs for cache sizes of 256 to 8K entries. The cache set associativity is fixed at four. The numbers in bold type are the maximum number for a given cache size.

Sensitivity to IBF cache Associativity. Table 4.6 shows sensitivity of the IBF cache miss rates to the set associativity of the IBF cache for SPEC CPU2000int programs, and Table 4.7 for SPEC CPU2006int programs. We found that the direct mapped IBF cache has high conflict miss rate when compared with the 2-way or 4-way IBF cache of the same size. As the set associativity increases, the miss rates drop significantly. The sharp drop of miss rates is surprising; and we have double checked our simulation code and used two independent implementations of simulation to verify the result. This scenario is possibly related to some special property of the conflict patterns of the branch address pairs, which may be very different from conventional instruction or data caches.

4.3.3 Trace-based Simulation Results for Other Workloads

The other workloads include TPC-C, WebStone and an FTP server benchmark. Table 4.8 shows the IBF cache miss rate with varying cache size from 256 to 8K with the set associativity fixed at four. TPC-C has relatively high miss rates with IBF cache of 256 and 512 entries, but the miss rate becomes very small after the size increases beyond 2K. The TPC-C workload is known to have large instruction footprint, and

Program	1-way	2-way	4-way
gzip	0.067%	0.002%	0.002%
vpr	0.007%	0.003%	0.002%
gcc	2.841%	0.843%	0.297%
mcf	0.591%	0.591%	0.591%
crafy	2.813%	0.307%	0.110%
parser	2.283%	0.351%	0.085%
eon	2.701%	0.001%	0.001%
perlbmk	4.733%	0.566%	0.046%
gap	0.530%	0.054%	0.016%
vortex	0.044%	0.014%	0.012%
bzip2	0.004%	0.004%	0.003%
twolf	0.433%	0.025%	0.002%
Average	1.421%	0.230%	0.097%

Table 4.6 IBF cache cache miss rates for *SPEC CPU2000int* benchmarks with the reference inputs with varying set associativity. The cache set associativity increases from 1-way (direct mapped) to 4-way with cache size being fixed at 2K entries.

therefore the result indicates that the control flow in TPC-C has good locality. The WebStone benchmark has very high miss rates with IBF cache of 1K entries or less, and drops to less than 1% when the size increases beyond 2K. The FTP server benchmark incurs few IBF cache misses for all sizes. Table 4.9 shows the number of misses per 10,000 instructions, which is more related to the overall performance overhead. The number is negligible for all the workloads for cache size of 4K or more entries.

Table 4.10 shows the sensitivity of the IBF cache miss rates over the set associativity. The cache size is fixed at 2K entries. Similar to the SPEC benchmarks, set associative caches reduce the cache conflicts significantly.

Program	**1-way**	**2-way**	**4-way**
perlbmk	1.829%	0.096%	0.012%
bzip2	2.675%	1.079%	0.009%
gcc	1.142%	0.186%	0.089%
mcf	0.492%	0.020%	0.020%
gobmk	2.172%	0.680%	0.351%
hmmer	0.103%	0.025%	0.022%
sjeng	4.902%	0.034%	0.004%
libquantum	1.028%	0.701%	0.695%
omnetpp	0.328%	0.006%	0.004%
astar	1.132%	0.569%	0.563%
xalancbmk	1.250%	0.240%	0.053%
Average	1.550%	0.331%	0.166%

Table 4.7 IBF cache cache miss rates for *SPEC CPU2006int* benchmarks with the reference inputs with varying set associativity. The cache set associativity increases from 1-way (direct mapped) to 4-way with cache size being fixed at 2K entries.

Workload	**256**	**512**	**1K**	**2K**	**4K**	**8K**
TPC-C	10.662%	3.226%	0.431%	0.081%	0.001%	0.000%
WebStone	58.523%	43.737%	24.496%	3.389%	0.463%	0.180%
WebStone CGI	59.484%	46.892%	29.596%	6.369%	0.643%	0.194%
FTP server	0.241%	0.146%	0.125%	0.117%	0.116%	0.116%

Table 4.8 The IBF cache miss rates for TPC-C, WebStone and FTP server benchmarks with cache size of 256 to 8K entries. The cache set associativity is fixed at four.

Workload	**256**	**512**	**1K**	**2K**	**4K**	**8K**
TPC-C	1.754	0.531	0.071	0.013	0.000	0.000
WebStone	13.061	9.761	5.467	0.756	0.103	0.040
WebStone CGI	13.361	10.533	6.648	1.431	0.144	0.044
FTP server	0.019	0.012	0.010	0.009	0.009	0.009

Table 4.9 The number of IBF cache misses per 10,000 instructions TPC-C, WebStone and FTP server with cache sizes of 256 to 8K entries. The cache set associativity is fixed at four.

Workload	**1-way**	**2-way**	**4-way**
TPC-C	3.353%	0.919%	0.001%
Webstone	18.149%	11.357%	3.389%
Webstone CGI	21.140%	14.153%	6.369%
FTP server	0.209%	0.151%	0.116%

Table 4.10 IBF cache cache miss rates for TPC-C, WebStone and FTP server benchmarks with with varying set associativities. The cache set associativity increases from 1-way (direct mapped) to 4-way with cache size being fixed at 2K entries.

5 Run-Time Detection of Malwares via Dynamic Control Flow Inspection

Anti-virus software is designed to detect the malicious program by statically analyzing the binary of programs. The scanning is usually done in a manner that matches the specific binary signature from the program binary with the malicious program database by string matching or applying heuristics on the program binary. However, attackers have started using the obfusfication techniques to change the static binary forms to evade the security program by hiding the binary signatures. It has been done by in three ways: encrypting the binary, applying polymorphism, and utilizing metamorphic engines.

The encryption method encrypts the body of the malicious program so that the static binary is made different from the plain binary of malicious program. When the program starts, first it decrypts the encrypted body and executes the malicious program. In order to efficiently encrypt and decrypt the program during run-time, a malicious program uses the simple symmetric encryption scheme such as XORing the binary with a key. Hence ainti-virus software tries to identify the decryption routine and uses the decryption routine as the signature of malicious program. Therefore malicious programs, that simply changes the key of encryption and decryption procdure, can be easily identified.

Another method to hide the static binary signature is using the metamorphic engines. The metamorphism rewrites the malicious program in a manner that performs same functions with the origianl malware. From the straight forward implementation as the skelton of the program, metamorphic engines rewrite the program by substituting instructions or subroutines to equivalent instructions or subroutines. It is very difficult to check the equivalance of two different code segments. Threfore most anti-virus programs generate new version of signature from newly generated malicious program. Fortunately, generating automative metamorphic engine is hard and it consumes a lot of efforts to generate correct metamorphic engine. Since the nature of anti-virus software's detection mechanism, anslysis of binray stream, anti-virus software simply treats the methamorphic malious program as a new malicious program. Therefore anti-virus software has little difficulty on detecting metamorphic malware. However, when the number of malcious programs increases abruptly, most of anti-virus software encounters issues on the performance overhead. Hence the attackers' motivation of using the evading methods is shifted from hiding its binary signatures to quickly generating new malicious programs from existing ones with little effort.

For abovementioned reasons, attackers increasingly use polymorphic method to change the decyption routine. The polymorphic method is to change the run-time decryption routine to hide the signature of the decryption routine. By using a number of different decryption routines, which perform equivalent decryption, attackers can populate new version of malicious program. From the anti-virus program's perspective, it is completely new malicious program and needs to generate a new signature of the decryption routine. Consequently, attackers can generate new malicious program quickly to cause a scalability issue on anti-virus programs. However, since the the polymorphic decryption routine is usually simple, anti-virus programs is able to emulate the decryption routine to identify the body of the plain malicious program. Therefore, a malicious program wirter uses more sophisticate polymorphic method, called run-time packing. Run-time packing compresses programs to reduce the static binary size. When the run-time packed program starts, it starts decompressing the program and execute the program. There are numerous run-time packers that use different types compression algorithms and implementaion of the algorithms. By combining compression methods and the encryption methods, attackers can quickly populate new types of run-time packers. This run-time packing method is more sophisticate than the simple encryption, it causes a number of issues on anti-virus software. Since attckers have large pool of polymorphic tools, they can easily populate the new malware from the old malware even though it has been kown. Also, the emulation techniques to identify the plain malware may no longer be effective due to the performance overhead of the emulation of malware.

Hence the thesis also propose a hardware-assisted, run-time malware detection system to detect malwares that use packing and encryption. It has a control-flow based mechanism called RCFI (Recent Control Flow Inspection) that performs fast and speculative malware detection. RCFI is a system component. We designed a special hardware component, called Enhanced Last Branch Recording (ELBR) stack, which records the address (PC value) of most recently executed branches. Essentially, the ELBR records the most recent control flow of program execution. Upon a certain system call, RCFI searches the content of ELBR for any signature of malware. A special database of malware signatures is pre-built using the control flow information of known malwares after unpacking and is part of RCFI. RCFI can be composed with the existing malware detection methods. In our system, RCFI can be used with behavior-based malware detection [51, 18], and they may further trigger expensive scanning [51, 18].

The novelty of this study is the use of RCFI and ELBR, as well as the use of malware signature built from the control flow information of the plain malwares. RCFI is a fast and reliable method of malware detection. The experimental results show that the system can successfully distinguish 30 malware variants and benign programs that we randomly put together. With RCFI, it is possible to search for the malware signature very frequently. New processors, including Intel and AMD processors [36], have included the Last Branch Recording(LBR) stack to help with the program debugging. We further enhance the LBR stack by introducing supplementary logic to filter the redundant control flow information, i.e. repetitive branches from loops. There is virtually no run-time overhead from ELBR, and the chip overhead of ELBR is negligible for today's processors. We have implemented a prototype system on the Bochs x86 emulator [2]. The estimated overall performance overhead is negligible.

The existing commercial anti-virus software has limited capability of detecting such malwares. Emulation-based unpacking techniques have been used [35], but they have to place a time limit on the emulated execution. Therefore, malwares that use or wait for a long time in unpacking may not be detected. A recent study [50] confirms that commercial anti-virus software cannot detect malwares using unpopular packers or variants of known packers, even though they can detect those using well known packers. Currently, new packers are created from existing ones at a rate of ten to fifteen per month [64]. As a result, malware writers have a large selection of tools to pack their malware, and can easily generate new malwares from old ones.

5.1 Related Work

Detecting polymorphic and metamorphic malwares has been one of the major challenges for the security community. Currently, the best solution against such malware is emulating the malwares and matching signatures or applying dynamic heuristic to detect them. Some recent work has focused on the problem of generic unpacking or decryption of polymorphic malwares to extract plain malware code.

Universal PE Unpacker [68] is a plug-in available for IDA Pro disassembler and debugger [61]. It uses behavior heuristics, single step execution and memory protection mechanism to identify plain malware execution. The generic unpacker can successfully extract plain malwares from mosts of polymorphic malware. However it has a few weaknesses that assumptions used in heuristic to detect the starting point of malware execution, cannot be applied to all polymorphic malware. Hence it may not be able to extract plain malware [60, 42].

Portable Executable Identifier(PEiD) [56] is a widely used tool for detecting binaries that exhibit unpack-execute behavior. It uses a signature database to determine if a binary contains packed code. If a match is found, an unpacking routine is provided to extract the hidden-code. The limitation is that it fails to detect even minor variations. PolyUnpack [60] performs pre-analysis of packed malware by disassembling the malware and partition it into code and data section, and then performs single-step instruction execution and identifies where the instruction is executed from. It has been demonstrated to successfully identify and extract hidden code in the malware. However, it incurs high performance overhead that makes it difficult to be used in real time. OmniUnpack [48] is another type of generic unpacker. It dynamically monitors the execution of a program to detect when the program has been unpacked. It only detects the time of unpacking and invokes virus scanner to scan unpacked executable. It uses the NX bit or emulation of NX bit to detect modified code execution. When a dangerous system call is executed, it invokes virus scanner to scan modified pages where instruction is executed before the system call. It incurs a small overhead to identify the problematic pages but the pages still need to be scanned by anti-virus software. Renovo [42] monitors program execution and memory writes at run-time, determines modified code execution and extracts the hidden code. It is built on top of TEMU [1] and emulates execution of malware.

Chritodorecu et al. [18] proposed semantics-aware malware detection mechanism. Obfuscated malwares are transformed to be formal templates statically. With the templates, the mechanism generates reference semantic and malware detection algorithm that can handle some obfuscation techniques used by malware writer. MetaAware [72] is another semantic characterization and matching tool that uses static control flow and data flow analysis to generate patterns based on the system calls or library. It tries to detect metamorphic malwares by matching the pattern of use of system calls and library.

Kruegel et al. [47] proposed the use of structural analysis to detect polymorphic malwares by comparing the structure. The structure of an executable is described by its control flow graph. Detecting isomorphic subgraphs and coloring each node based on class of instructions improves the result of comparison. However, the proposed static analysis methods have to be applied to plain malwares, otherwise they only identify the encryption or compression routine. Our control flow matching and instruction stream scan share some commonality with this approach.

Another approach to deal with poly/metamorphic malwares is behavior-based detection [17, 43]. High-level description of malicious behavior can detect polymorphic and metamorphic malwares without considering binary code itself. Christodorescu et al.[17] proposed automatic generation of specifications of malicious behavior. It specifies malicious behavior in terms of the dependencies between system calls. The specifications are qualitatively equivalent to those used by commercial anti-virus software and can be used to identify subsequent malware variants.

5.2 Recent Control Flow Inspection (RCFI)

Although syntactic malware detection has limitations in detecting polymorphic or metamorphic malwares, it is efficient for detecting known malwares. On the other hand, semantic malware detection is able to detect advanced malware but it requires more extensive plain malware code analysis by using disassemble or emulation. In this study, we propose Recent Control Flow Inspection (RCFI) that utilizes a special hardware component called Enhanced Last Branch Recording (ELBR) stack. It assists both syntactic and semantic malware detection by recording relevant control flow information near dangerous system calls. We also demonstrate its efficiency by designing hybrid malware detection system. Since the system monitors recently executed control flow, it effectively inspects the execution of code in unpacked form. Therefore, it can bypass the packing or encryption method that a malware may employ. Additionally, sandbox mechanism can be used as another layer of guard against malwares. In that case, our mechanism can effectively detect the presence of malwares and stop their execution.

Figure 5.1 shows an event flow of the proposed RCFI. The system consists of syntactic and semantic detection modules. The left-hand side of flow in the schematic shows the syntactic detection path. RCFI records and matches the executed block sequence of a program. Malware detection is done in two stages in this syntactic detection. In the first stage, it searches the current control flow of a program from malware signature, when a dangerous system call is invoked. At this stage, the monitor lightly checks the sequence of executed block sizes instead of matching sequence of instructions. Since the size of instructions is varying in x86 ISA, block size matching can quickly filter out benign execution flow. In the first stage, the syntactic signature consists of control flow information: instance of system call and size of last n executed blocks. The second stage is instruction stream matching. The monitor generates executed instruction stream from information stored in ELBR stack. In this stage, the reconstructed instruction stream can be analyzed with existing anti-virus software to verify the result of block size matching.

It is possible that advanced malware bypass syntactic detection by using advanced

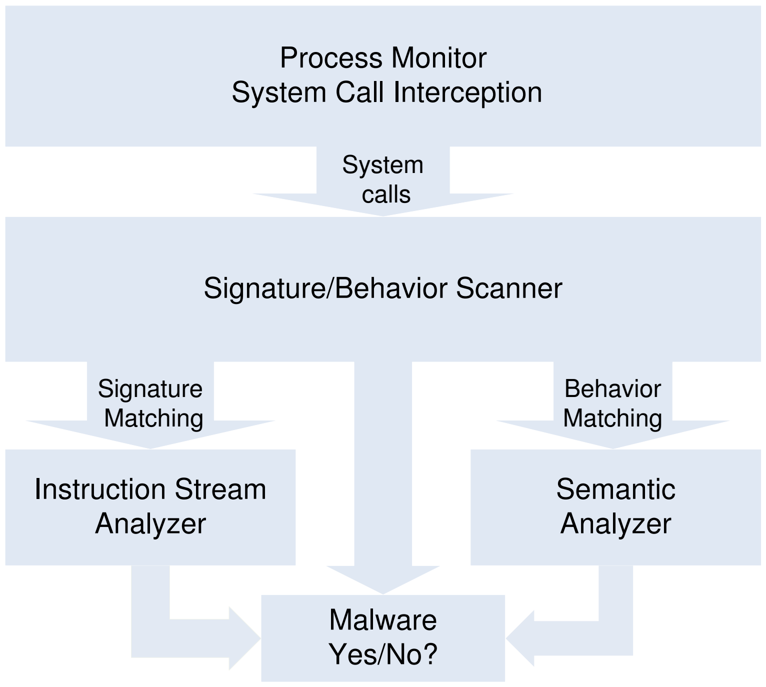


Figure 5.1 Recent Control Flow Inspection (RCFI) System Event flow.

polymorphic techniques such as equivalent instruction substitution, block reordering, and so on. To detect such cases, at the time of control flow monitoring, behavior monitor is being performed. The right-hand side flow in figure 5.1 shows the semantic detection module path.

There are a number of studies which monitor system call sequence to detect anomaly by system call interception and dynamic binary instrumentation. However, system call interception techniques usually impose high performance overhead for analyzing the system call sequence and generating high false alarms due to insufficient process information [30, 21, 34, 62]. In contrast to these approachs, RCFI searches known behaviors of the malware that is similar to [17]. RCFI also monitors the behavior of a program to detect advanced poly/metamorphic malware. When it detects the known behaviors of malware in the system, it analyzes semantics of malware with reconstructed instruction stream to reduce the false positive.

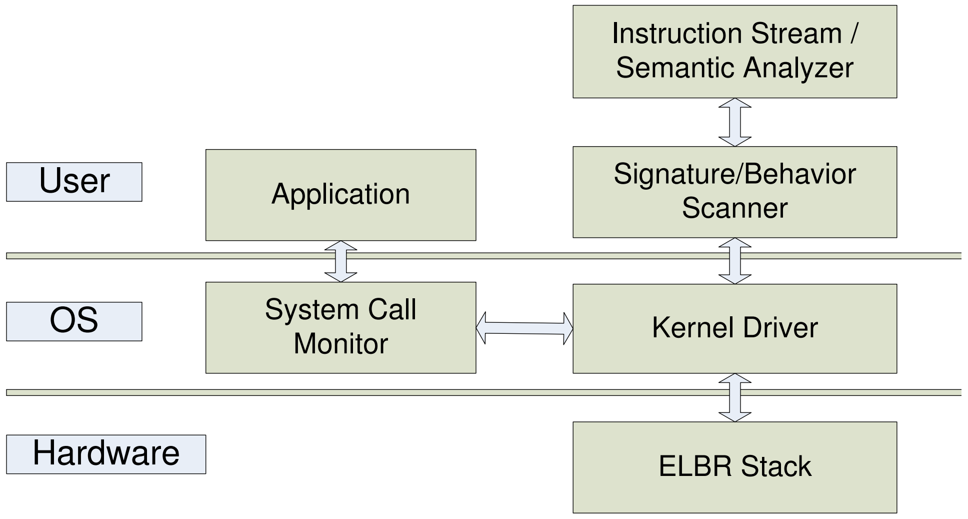


Figure 5.2 RCFI System Architecture.

5.3 Implementation

Figure 5.2 shows the RCFI system architecture. RCFI is composed of hardware ELBR stack, kernel modules, user-space signature/behavior scanner and instruction stream/semantic analyzer. The kernel modules, system call monitor and kernel driver, are responsible for retrieving control flow information from ELBR stack upon a system call event. The signature/behaivor matching componet in RCFI tries to match the control flow and behavior with signature database. Another RCFI user-space components, instruction stream/semantic analyzer, are activated by signature/behavior scanner upon a outcome of the scanner. The instruction stream/semantic analyzer is responsible for confirming the malware instance by instruction stream or semantic analysis.

We first built an initial prototype on Intel P4 processor, which has limited LBR entries. The detection works in some cases, but the LBR is not large enough to hold sufficient control flow information for generating a reliable signature. We further extended the prototype on the Bochs emulator and ran Windows 2000 on top of it. We extend the Bochs emulator [2] to simulate ELBR stack and efficient helper structures for extracting control flow information.

Enhanced Last Branch Recording(ELBR) Stack

Recent commercial processors have provided *Last Branch Recording* mechanism for debugging purpose [36]. Intel Pentium 4 Processor have 16-entry LBR stack and other processors, e.g. AMD have different number of entries. It records taken branch, interrupt, and exception information as source address and target address. The information can be read by issuing machine specific instruction when information is needed. We use the mechanism for tracking the control flow of programs. Since the hardware LBR stack has fixed size, only most recently executed branches can be recorded.

Due to the fixed size of the hardware LBR stack, we revised the LBR stack design to filter out some of the control flow information to make the stored contents in LBR more relevant to malware detection. Our design goal is to minimize the hardware modification for maintaining the simplicity of LBR stack. We consider three common cases that cause scalability issue in LBR stack: irrelevant control flow information, simple loops, and immediate jumps.

The first modification is adding address range checker. The Intel processor LBR design provides an ability to selectively record branch information from User or/and OS; however, it does not distinguish between DLL(library) and program main body. When system call is intercepted without the separation, the LBR stack may have many entries storing the control flow information of DLL or library routine which does not help malware detection. Figure 5.3 shows the number of branches between system calls in the library routine. The data is collected during initial behavior of Win32.Mytob. The x-axis shows the sequence of dangerous system calls and the y-axis shows the number of branches before a dangerous system calls in the library routine. For instance, the dangerous system call sequence one through ten is called from *"LoadLibraryA"* library function. The first bar in the graph shows the number of branches between the entry of the library routine and system call. The second bar shows the number of branches between the first dangerous system call and second dangerous system call.

The results show that simply increasing the size of LBR stack may not guarantee the capturing control flow of malware body. Moreover, since the content of LBR stack needs

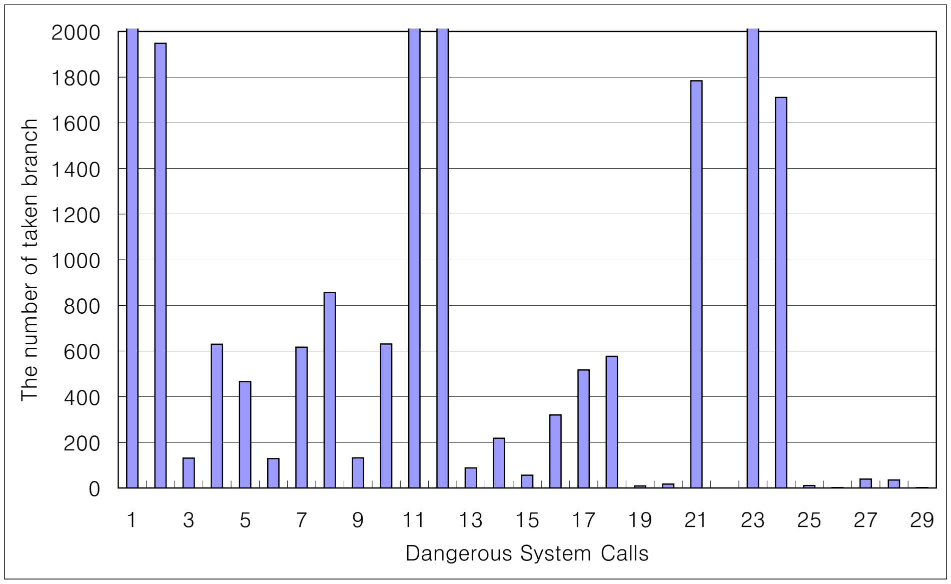


Figure 5.3 The number of branches before a dangerous system calls in library routine for Win32.Mytob.

to be saved for each process at every context switch, excessive size of LBR stack imposes extra context switch overhead. Although the system can filter out library information by inserting routine that disables LBR stack before entering library routine, it would occur performance overhead. Therefore, it is useful to add address range checker to filter out unnecessary information [1].

The second addition is a simple single block loop detector to filter the redundant control flow information from execution trace. The single block loop such as string copy routine usually appears in most malwares. Figure 5.4 shows the partial execution trace for the malware. The first hex decimal represents source address of a branch instruction and the second hex decimal represents target address of the branch. The third column shows the name of WIN32 API. In the trace, the library routine is already filtered out for clarification. The ELBR stack will eliminate the loops for example, the branch number 4,10,18,23,and 24 from the first 28 branches.

[1]It easily identifies corresponding dangerous API calls when dangerous system calls are invoked, so we do not distinguish between dangerous API calls and dangerous system calls in this paper.

Another modification is the elimination of immediate jumps. The immediate jumps are branch instructions that immediately forward control flow to other code block. These jumps are widely used in a program that uses position-independent code. For instance, in figure 5.4(a), jump target in branch 27 and jump source in branch 28 are identical. Hence the jump source in branch 28 is immediately forwarding the control flow to its target. Therefore the information may not useful for detecting malwares. Moreover, malware writer can easily overflow LBR stack with immediate jump insertions, so that the third modification is immediate jump elimination. When a jump instruction is jumping to very next instruction or its target is immediate jumps, it eliminates the jumps .

```
1  00402ed7 00403720
2  0040372a 00403740
3  00403756 00403740
4  00403756 00403740
5  0040375d 00403791
6  0040379a 00402edc
7  00402ef3 00403640
8  0040364b 0040365c
9  00403672 0040365c
10 00403672 0040365c
11 00403679 0040369e
12 004036ab 004036c6
13 004036e6 0040370f
14 00403717 00402ef8
15 00402f06 00403640
16 0040364b 0040365c
17 00403672 0040365c
18 00403672 0040365c
19 0040367d 00403699
20 0040369c 004036a1
21 004036ab 004036c6
22 004036de 004036c1
23 004036de 004036c1
24 004036de 004036c1
25 004036e2 00403718
26 0040371f 00402f0b
27 00402f1c 0041a05b
28 0041a05b 7c4f46c0 CopyFileA
```

(a)

```
1  00402ed7 00403720
2  0040372a 00403740
3  00403756 00403740
5  0040375d 00403791
6  0040379a 00402edc
7  00402ef3 00403640
8  0040364b 0040365c
9  00403672 0040365c
11 00403679 0040369e
12 004036ab 004036c6
13 004036e6 0040370f
14 00403717 00402ef8
15 00402f06 00403640
16 0040364b 0040365c
17 00403672 0040365c
19 0040367d 00403699
20 0040369c 004036a1
21 004036ab 004036c6
22 004036de 004036c1
25 004036e2 00403718
26 0040371f 00402f0b
27 00402f1c 7c4f46c0 CopyFileA
```

(b)

Figure 5.4 Recorded Control flow example for Win32.Netsky.c: (a) without ELBR (b) with ELBR.

Signature/Behavior Scanner

As previously mentioned, the straightforward dynamic method to detect known malware is to scan every new or modified page dynamically. However, blindly scanning those pages would incur unacceptable performance overhead. It is also effective if executed instruction stream is dynamically compared with signature which can be done in emulation environment. However, it is hard to implement such mechanism with acceptable performance penalty without hardware supports.

We use a hardware support to efficiently inspect instruction stream. Our detection mechanism is to dynamically inspect instruction stream when problematic behavior (event) is occurred. The signature scanner projects problematic events by matching control flow of a program at dangerous system calls[2]. However, our control flow matching is somewhat different from the conventional control flow matching: we neither pre-process the executable nor identify basic blocks. The control flow matching simply compares the sequence of executed block size obtained from ELBR stack. Although matching sequence of block size is a coarse-grain inspection, it filters out large portion of benign system calls due to different size of instruction in x86 architecture and uniqueness of malware code.

Control flow monitoring is effective for detecting malwares that use polymorphic decryptors or/and variants of packers to evade signature-based detection mechanism, since the execution of malware body is identical. The signatures consist of system calls (API) and sequence of block size. The signature is extracted from plain malwares . The signature also needs to capture the control flow near monitored system calls or API calls. Since the ELBR stack only records user-level control flow, the most recent entry is an instruction calling library routine or a system call directly from user code. If one instance of control flow is insufficient to determine the malware, the next control flow before the next dangerous system call is also monitored continuously. Note that we assume the existence of the recovery mechanism, which tracks the system behaviors.

[2]We define a system call as a dangerous system call if it can result in changing the status of system such as changing file system and registry. In our experiment, we define 33 system calls as dangerous system calls.

For example, in malware Win32.Netsky.c, a sequence of three API calls from the first dangerous system (API) calls are CopyFileA, RegOpenKeyA, and RegSetValueA. The system call monitor intercepts the system call and tries to match the control flow at the dangerous API calls such as CopyFileA and RegSetValueA. If the control flow prior to the CopyFileA is matched, the information of control flow is passed to be analyzed. If the result of analysis is negative, following control flow at RegSetvalueA is analyzed.

The instruction stream analyzer checks the executed instruction stream when a sequence of block size is matched with malware signature. It is only effective for simple polymorphic and packed malware which do not change the malware body. However, it is also effective for simple metamorphic techniques [67] such as using different registers in new generations, reordering modules (blocks). However, signature scanner in RCFI cannot handle complex metamorphic malwares that changes the size of blocks in malware code, such as equivalent code replacement, garbage code insertion in blocks, spurious jump insertion and so on. In order to detect minor variation of malware, we employed behavior monitor. It helps detecting known malicious behavior of program. Moreover, by using control flow information, algorithmic malware analysis through disassembly, called semantic detection [18, 72], can be easily performed. Even if signature scanner misses complex metamorphic malware, the whole system still has a chance to perform various analysis on malware.

Name	Behavior	Version
Win32.Bagle	Email Worm	an,b,c,i,k
Win32.Bronktok	Email Worm	b,q
Win32.Doonbot	Email Worm	b,g,k
Win32.Mydoom	Email Worm	a,m
Win32.Mytob	Net-Worm	r,t,x
Win32.Netsky	Email Worm	aa,b,c,d,e,m,q,r,t,x
Win32.Nyxem	Email Worm	a,e
Win32.Scano	Email Worm	s, h

Table 5.1 Tested malware samples.

5.4 Evaluation

We obtained 30 live malwares from Offensive Computing [22] and tested them on our revised Bochs emulator [2] running Windows 2000 to demonstrate the effectiveness of our approach. Those malwares include the top 20 malware for May 2008 in [6] and are listed in table 5.1. The first column gives the name of malware. The second column shows the category of the malware and the third column shows the tested versions. Initially, we generated the entire execution trace of malwares to identify the plain malware code by applying similar heuristic used in universal PE Unpacker [68]. Universal PE Unpacker assumes that *GetProcAddress* is always called to set up the import table after the malware body is decrypted or unpacked and before it reaches the original entry point of malware. With the trace of the malwares, we manually generate their signatures near the dangerous system calls.

5.4.1 Experiment

In this section, we illustrate the malware detection process step by step with a sample malware. One of the malwares tested in the experiment, *Win32.Netsky.c*, is packed with PEtite V2.2. The initial behaviors of the malware are that it copies itself into Windows system directory and the edits the registry to auto-run itself from the next booting. It also deletes some of the registry keys.

Execution Flow Monitor The signature scanner generates information from the most recent branch when dangerous system call is intercepted. In this example, CopyFileA, RegSetValueA, and RegDeleteValueA call dangerous system calls. For example, when the first dangerous system call is intercepted, the information generated by the monitor is {CopyFileA, 17, 7, ... }. The first entry is a caller of dangerous system call or system call number if it is called directly from the user code. In this case, library routine CopyFileA invokes a dangerous system call. The second entry is the size of the block prior to the jump. Note that it is not a basic block; it is the executed block that may contain basic blocks. The signature scanner looks up to n branches where n is determined by the length of signature and hardware ELBR stack.

The first stage detection, control flow signature scanning, tries to find the matching control flow with information in ELBR stack and signature database. In our experiment, we generate signature with the last 5 branches for CopyFileA to detect the malware for demonstration purpose. Although we have not seen false alarm in our emulation environment, the signature, which has a small number of branches, may raise false alarm in real system since CopyFileA API is widely used in many application. Therefore, the signature depth should be properly set to effectively filter out benign system calls. The signature scanner also records the control flow information and register values for last m dangerous system calls. The recorded information will be used when the behavior monitor raises an alarm. If a dangerous system call turns to be a benign system call, signature scanner hashes the information and caches it for avoiding instruction stream match when the same dangerous call is encountered.

Note that more complex malware may exploit the cache mechanism to bypass the control flow matching. For example, the first malware may generate random instructions in each block and then the benign code is examined. Once the random instructions pass the instruction stream analyzer, the malware overwrites the random instructions with malicious instruction and performs malicious behavior. At this time execution monitoring misses the malware since the control flow signature is stored in the cache. However, it is hard to generate such malware. The random instructions in the block should not generate any error or random effect that accidentally terminates malware. The random instructions also need to be polymorphic so that it cannot become a signature. Even if such a malware is generated successfully, it might raise an alarm at the behavior monitor and the code will be re-examined. At this point, the overwritten code will be examined.

Instruction Stream/Semantic Analyzer When the control flow of a program is matched with malware signature, instruction stream analyzer reconstructs the executed instructions from the control flow information. At this point, since the instruction stream analyzer already knows the possible candidate signatures, simple string match algorithm [51] is used for detecting malwares.

Semantic analyzer is activated when behavior monitor raises an alarm. It uses the last m dangerous system call information stored in the signature scanner. In the semantic analyzer, with the control flow information, more complex malware analysis can be done that is chosen by anti-virus system designer. In this paper, we choose some of the program static heuristics [51] and semantic based [18, 72] malware detection methods. First, it re-examined the constructed instruction streams for the cases that malwares exploit the weaknesses of signature scanner(See 5.4.3 for more details). If the examined instructions pass the instruction stream analyzer, the system applies dynamic heuristic by inspecting history of system status at dangerous system calls. When the result raises an alarm, it applies semantic or algorithmic detection method to the reconstructed instruction streams to detect malwares that use garbage insertion, and equivalent code replacement techniques.

5.4.2 Effectiveness

In order to demonstrate its effectiveness, we tested 30 live malwares from Offensive Computing, and encrypted and packed a toy program with 6 packers, teLock, ACProtect, UPX, ASpack, ASProtect, and MEW in the Bochs emulator. The toy program is composed of series of dangerous system calls and series of operations between the system calls. During the malware test, 30 live malwares are successfully identified through execution flow monitoring. Also, in the toy program packed with various options in packers, the instruction stream near the dangerous system call is also successfully identified. We further test the obfuscated toy program with metamorphic techniques [67].

Using Different Registers The most simple metamorphic technique is using different registers for different instance of malware. In such a case, the program block size is not changed. The signature scanner generates an alarm and the instruction stream analyzer can simply use static heuristic to identify the toy program.

Instruction Stream Reordering In this test, the toy program blocks are reordered statically. First we divided the toy program into different size of blocks and reorder them but the control flow of blocks is not changed. In this case, signature scanner is able to identify the original execution of blocks order and instruction stream analyzer can identify the program. In the second test, we reordered sequence of block execution if blocks are independent. In the third test, we inserted jumps between the instruction to reordered instructions. Although signature scanner missed it due to different sequence of block size or different size of block, the system call monitor eventually generated alarm and the semantic analyzer successfully identified the toy program by removing jumps and reconstruct the instruction stream.

Garbage Insertion and Equivalent Code Substitution If garbage or semantic no-op instructions are inserted between blocks where the control flow would not reach, the signature scanner raises an alarm and the instruction stream scanner can successfully identify them. However, when the semantic no-op instructions are inserted within blocks, they would change size of blocks. Due to the change in block size, the signature scanner misses the dangerous system calls. When the system call monitor detects malicious behavior, the obfuscated instructions are examined by semantic analyzer.

5.4.3 Limitation

With near native speed, the system performs most of the functions of malware detection that can be done in emulation environment: original entry point discovery, dynamic disassembly, control flow monitoring, and behavior monitoring. However, the system has a few weaknesses. It uses dangerous system calls as a break point of dynamic analysis, so signatures must be generated from instruction stream of near dangerous system calls (API). In our limited experiments, all malwares are identified by instruction stream of near dangerous system calls without false alarm. However, our experiments do not ensure that signatures of all malwares can be generated from instruction stream near the dangerous system calls since anti-virus software uses various data in malwares as signatures.

Another weakness is that the system relies on the hardware LBR stack for retrieving control flow information. Due to the size limitation, the system can verify at most n last branching information. Consequently, in the worst case, the system can verify the last n instructions. If a signature contains more than n instructions, the instruction stream analyzer will fail to match the signature. Even the semantic analyzer cannot identify the signature with the provided control flow information.

The other weakness is that before the dangerous system call, malware can overwrite or encrypt the executed instructions. In this case, reconstructed instruction stream is encryption routine and is considered as semantic no-ops since malware must save system call(API) parameter before encrypting the code and encryption routine does not affect the system calls (API). Therefore, the system cannot make a decision immediately.

These weaknesses do not reduce the security strength of the system; nevertheless, they incur extra performance overhead. In these cases, the system call monitor raises alarm and the semantic analyzer fails to detect malware, but it can identify the reason for the failure of analysis. The reason can be either insufficient number of instructions (partial match), or semantic no-op insertion before the dangerous system calls (API) that does not affect the call. For the first case, it may be possible to disassemble the malware code with the recorded control flow information. However, in the latter case, the semantic no-op operations can be an encryption routine. If the instruction stream cannot be reconstructed with the control flow information, the program has to be examined in slow emulation environment. Thus, the system still has a chance to detect more complex metamorphic malwares at extra cost.

5.4.4 Performance Analysis

Since our method has been implemented in Bochs emulator, we are not able to provide accurate performance overhead. We estimate performance overhead by presenting the number of dangerous calls. The most widely used performance benchmark suite, the SPEC CPU benchmarks, shows negligible performance overhead. We further chose some real applications in Windows. Table 5.2 shows profile data of the selected benign programs. The second column shows the total number of executed instructions on average, and the third column shows the total number of invoked dangerous calls. The following is the description of those benchmarks and the profiling method:

- bzip2: bzip2 is a console mode compression application. It uses bzip2 algorithm to compress and decompress a file. Files of various sizes are compressed and decompressed during the profiling phase. The sizes of input files are 50KB, 100KB, 500KB, and 1MB. The profiled data is the average values of each compression or decompression of files.

- calculator: The test are performed by testing different combinations of functions in the application. The profiled data are the average values of each combination of functions.

- dBpowerAMP Musinc Converter: it converts various music file formats to other formats. We profiled the application when it converts files of various format and size. The profiled data are the average values of each conversion.

- Internet Exploer 6: We profiled it by visiting 10 websites. The profiled data are the average values of opening a single web page.

Name	Total Inst.	Dan. Syscalls
bzip2	61409624	49
calculator	48298230	40
dBpoweramp	545930652	8140
IE6	64121944	481
notepad	1394336920	1343
Winzip11	2920840250	33450

Table 5.2 Profiled Data of Benign Programs.

- Notepad: The text editor is profiled by generating 5 pages of random text with using basic functions. The profiled datum is the average value of 3 different text file generations.
- WinZip11: The GUI based file compression application is profiled with same bzip2 input files. It also performed compression and decompression. The profiled datum is the average of each operation.

We estimate the performance overhead indirectly from the profiled data. The performance overhead comes from system call monitoring, control flow matching, instruction stream matching, and applying heuristic. The system call monitoring can be performed by hooking interrupt description table and systener MSR. Recording and identifying the dangerous system calls can be done in constant time. For each dangerous system call, the control flow matching and instruction stream matching are performed. We used AhoCorasick string match algorithm [9] for the control flow matching and instruction stream matching. The algorithm complexity is $O(m)$ where m is the length of a signature. The maximum value of m is fixed by hardware for control flow monitoring. For instruction stream matching, m is the length of instruction stream signature. The average length of string signature in Clam-AV [46, 74] is from 46 to 124, therefore the execution flow monitoring would incur negligible performance overhead[3].

[3]Our analysis shows that retrieving control flow information in current Intel Pentium4 processor requires from 248 to 355 cycles per entry. Therefore, retrieving 10 branch information would take around up to 3550 cycles. However, we believe that MSR can be optimized and performance overhead can be reduced.

The semantic analyzer is activated when the system call monitor raises an alarm. It mainly reduces false alarm and detects malwares that bypass the control flow monitoring. It performs a series of heuristic scans including instruction stream scan and algorithmic detection. When the semantic analyzer cannot determine malware instance (see the discussions of limitation), it has two possible options: halting the program, or further analyzing the program with complex metamorphic virus detection scheme. The overhead of the heuristic scanner is indefinite in this case. However, the use of control flow information and the failure reasoning of semantic analyzer give more confidence to system call monitor to halt a program.

6 Conclusion

The dissertation supports that the non-invasive dynamic control flow validation system can be implemented by using the existing hardware features in commercial microprocessors. In order to provide a seamless control flow validation, it suggests a hardware support that validates indirect control transfers in programs to augment computer system security. The thesis also explores how the hardware component effectively utilizes the control flow information to enhance computer system secuciry.

Since the Morris worm incident, there have been many proposed defense mechanisms against CFAs. Some of the solutions provide limited protection against a specific CFA method such as protecting return addresses in program stack [24, 57, 16], preventing injected code execution [3], and checking the bounds of the buffers [40]. Also generic protection mechanisms are proposed with hardware modifications in order to prevent CFAs. Dynamic Information Flow Tracking [65], and Minos [25] modify the hardware to efficiently track down spurious data to prohibit using the data as the control data. However these defense mechanisms may have some drawbacks due to the limited protection scope, complicated system implementation, or extensive hardware modifications.

A natural solution to prevent the control flow attack could be monitoring the program execution to ensure that it conforms to a pre-defined specification of its intended behavior [45]. Another solution, a model-based solution, monitors other indirect events such as system call sequence [29, 31, 70]. However these system monotoring methods usually suffer from false alarms due to imprecision of its attack detection and the mimicry attacks due to corase-grain monitoring intervals.

Program Shepherding [44] has been proposed to run vulnerable programs on the dynamic code optimization system, DynamoRIO. The system monitors every indirect branch to enforce security policies on program control transfers. However, it incurs large performance overhead due to the inefficient monitoring method. Abadi et al. [8] proposed the defense mechanism by instrumenting the program binary code in a manner that validates every indirect control transfer by inserting the validation stubs before the control instruction or replacing control instructions with the security code. It requires a sophisticated binary instrumentation tool to identify legitimate targets and insert the security code.

Control flow validation is an effective mechanism to prevent CFAs that change the control data, such as target addresses of jumps, calls and returns to redirect the program flow to the choices of attackers. The traditional control flow validation mechanism has been implemented in two methods: the first method is to use dynamic binary translation used in the various middleware such as dynamic optimization, binary instrumentation, and so on. The second method is the inlined reference monitor(IRM) that use a trusted binary re-writer to insert security code into a target application in a manner that validates the target address of the control instruction. Another possible method is to use the existing hardware features in commercial microprocessors. The hardware features are designed for supporting application debugging and performance monitoring. These facilities also can help with dynamically observing problematic control transfers in a non-intrusive manner. The hardware performance monitoring feature detects and counts various hardware events. When the number of events exceeds the maximum threshold value, the facility generates a hardware interrupt so that the hardware state can be observed. The hardware debugging feature is used in conjunction with the performance monitoring feature to identify the program counter value that triggeers the event. By leveraging the hardware capability, the control flow validation mechanism can be implemented transparently to the target application. Since the states of the hardware facility is transparent to the applciations and the hardware events, monitored by the performance monitoring feature, cannot be crafted by other entities in the computer system, the implementation provides strong guarantees that the successful CFA is extremely difficult, if not impossible, under the control flow validation system.

IBMON is a fine-grain control flow validation system that utilizes the hardware features in commercial processors. Using the hardware monitoring feature has numerous benefits: IBMON guarantees the uncircumvent complete validation of the problematic control flow transfers. The hardware facility and its events are transparent to the target application and the user. Hence CFAs cannot circumvent the control flow validation. Modern processors provide mechanisms to transfer control of a system to IBMON. Hence, the monitor can achieve fine-grained security checks without modifying the source or the binary code of the target program. This facility also gives flexibility to IBMON so that IBMON can dynamically adjust the target applications and the validation intervals. The prototype systems effectively detect various CFAs and exhibit the performance overhead which ranges from 0% to 33.5% an average of 8.3% for SPECINT2000 benchmarks. For other server benchmarks, the performance degradation of IBMON is negligible.

Although IBMON is the most efficient control flow validation system among existing control flow validation systems, it still incurs non-negligible performance overhead upto 33.5% for one of the SPECINT2000 benchmarks. Therefore, the thesis also proposes effective yet minimal hardware suuport which can provide a seamless control flow validation environment. IBF-Cache (Indirect Branch Filter Cache) is the cache design that effectively reduce the frequency of control flow validation by exploring the temporal locality of control flow transfers in programs. In various performance tests, based on both the trace-based and the cycle accurate simulations, IBMON with IBF-Cache shows negligible performance overhead on all SPECINT2000 benchmarks and other server benchmarks.

The thesis also addresses utilizing the control flow information to detect unwanted programs running in a system. The conventional approach to detecting malwares is based on static scanning of malware signature in system files and the memory. This static scanning is effective for detecting many existing malwares, but is very limited for malwares that use packing and encryption methods to hide their contents. New generations of malwares are increasingly using these methods. According to a recent study [15], 77% of malwares seen during January 2007 used run-time packing methods. A straight-forward approach to detecting those malwares is to scan system memory, including that used by all applications, for malware signature from time to time. The scanning may be triggered by timer interrupts, system calls, or other system events. To ensure system security, the scanning has to be done frequently. If the scanning happens too late, a malware may have already taken control of the system or may have done significant damages or caused leakage in the application data. Furthermore, there is little hint available to the scanning process, so a complete scan of the system is necessary. Frequent and complete scanning of system memory, however, will incur so a large system overhead that eventually makes the system unusable.

Therefore, utilizing the control flow information has a potential benefit to the existing misuse-based defense mechanisms. As mentioned before, anti-virus software suffers from polymorphic malwares. However we notice that the most popular polymorphic mechanisms, designed to avoid static scanning, reveals the original control flow of the malware at run-time. Hence we also propose to construct malware signatures from their control flow information to detect ingenious malwares at run-time. It is also potentially beneficial for the system call sequence monitoring approach to add the history of control flow information into the states. Since the mimicry attack takes advantage of the imprecision of the defense mechanism, additional control flow information can make the states more precise.

The RCFI system demonstrates the usefulness of hardware support with a sample dynamic malware detection system. It inspects the control flow and monitors the behavior of a program at dangerous system calls. The existing LBR stack is modified to provide the control flow information with virtually no performance penalty. The initial testing results from a prototype show that the RCFI system can successfully detect signatures from malwares with negligible performance overhead on benign programs. The system stops malware execution when it makes system calls, but it may also be combined with system protection mechanisms including sandbox [71] or snapshot mechanism in the virtual machine [7] so that the system can be safely recovered after malware detection. Our future work is to integrate the RCFI system into sandbox or the virtual machine environment and to test benign programs and malwares more extensively than current experiments.

Bibliography

[1] The bitblaze dynamic analysis component. http://bitblaze.cs.berkely.edu/temu.html.

[2] The cross platform ia-32 emulator bochs. http://bochs.sourceforge.net.

[3] . Pax homepage. http://pax.grsecurity.net/.

[4] . Stackshield. http://www.angelfire.com/sk/stackshield.

[5] . Standard performance evaluation corporation. http://www.spec.org/.

[6] virus top 20 for may 2008. http://www.viruslist.com/en/analysis?pubid=2047920027.

[7] Vmware: Virtualization via hypervisor, virtual manchine and server consolidation. http://www.vmware.com.

[8] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity:principles, implementation and application. In *12Th ACM Symposium on Computer and Communication Security*, Alexandria,VA, 2005.

[9] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333:340, Mar/Apr 1975.

[10] J. Allen. Wu-ftpd heap corruption vulnerability. http://www.sans.org/reading_room/whitepapers/honors/wu-ftpd-heap-corruption-vulnerability_831, 2001.

[11] Anonymous. Working exploit for glibc executing /bin/su. http://www.milw0rm.com/exploits/209.

[12] A. Baratloo, N. Singh, and T. Tsai. Transparent run-tim defense against stack smashing attacks. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, San Jose, CA, 2000.

[13] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, San Jose, CA, 2006.

[14] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[15] P. Bustamante. Packing a punch. `http://research.pandasecurity.com/archive/Packing-a-punch.aspx`.

[16] T. Chiueh and F.-H. Hsu. Rad: A compile-time solution to buffer overflow attacks. In *In proceedings of the International Conference on Distributed Computing System*, Phoenix, AZ, 2001.

[17] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *In Proceedings of the 6th ESEC/FSE*, 2007.

[18] M. Christodorescu, S. Jha, S. A. Seshia, D.Song, and R. E. Bryant. Semantics-aware malware detection. In *In Proceedings of IEEE Symposium on Security and Privacy*, page 32:46, 2005.

[19] M. Christoper Martinez and E. B. John. Multimedia workloads versus spec cpu 2000. In *Spec Benchmarking Workshop*, 2006.

[20] W. Chuang, S. Narayanasamy, and B. Calder. Bounds checking with taint-based analysis. In *Proceedings of International Conference on High performance Embedded Architecture and Compilers*, Goteborg,Sweden, 2007.

[21] C.Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of Eigth European Symposium on Research in Computer Security(ESORICS'03)*, page 326:343, 2003.

[22] O. Computing. `http://www.offensivecomputing.net`.

[23] M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/file/articles/heaptut.txt`.

[24] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, P. A. Grier, Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, page 63?8, San Antonio, TX, 1998.

[25] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, Portland, OR, USA, 2004.

[26] S. Designer. Non-executable stack path. `http://www.openwall.com/linux`.

[27] S. N. Dvorak. Lbl traceroute exploit. http://downloads.securityfocus.com/vulnerabilities/exploits/traceroute-exp.txt.

[28] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/main.html.

[29] S. Forrest, S.Hofmeyr, A. Somayajo, and T. Longstaff. A sense of self for unix processes. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 1996.

[30] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of Network and Distributed System Security Symposium*, 2004.

[31] J. Giffin, S.Jha, and B.Miller. Effiecient context-sensitive intrusion detection. In *Proceedings of 11th Annual Network and Distributed Systems Security Symposium*, 2004.

[32] L. A. Gordon, M. P. Loeb, W. Lucyshyn, and R. Richardson. Csi / fbi computer crime and security survey. http://www.gocsi.comi/survey.

[33] M. Hind and A. Pioli. Which pointer analysis should i use? In *In proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[34] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. In *Journal of Computer Security vol. 6, no. 3*, pages 151–180, 1998.

[35] M. Inc. Advanced virus detection scan engine and dat. http://www.mcafee.com/us/local_content/white_papers/wp_scan_engine.pdf.

[36] Intel Corp. IA-32 Intel Architectures Software Developer's Manual. January, 2006.

[37] Intel Corp. Intel 64 and IA-32 Architectures Optimization Reference Mannual. November, 2006.

[38] Intel Corp. Intel VTune performance analyzers. http://www.intel.com/cd/software/products/asmo-na/eng/Vtune/239144.htm.

[39] JEDEC. Fully Buffered DIMM draft specification. http://www.jedec.org.

[40] R. W. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for array and pinters in c program. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[41] M. M. Kaempf. Vudo malloc tricks. *Pharack Magazine*, 57:File 8, 2001.

[42] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *In Procedings of 5th ACM Workshop on Recurring Malcode(WORM'07)*, 2007.

[43] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *In Proceedings of USENIX Security'06*, 2006.

[44] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *In Proceedings of 11th USENIX Security Security Symposium*, San Francisco, CA, 2002.

[45] C. Ko, C. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of 10th Computer Security Application Conference*, 1994.

[46] T. Kojm. Clam-av. `http://www.clamav.net`, 2004.

[47] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *In Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection*, page 53:64, 2005.

[48] L. Martignoni, M. Cristodorescu, and S. Jha. Omniunpack: Fast, generic and safe unpacking of malware. In *In Procedings of 23rd Annual Computer Security Applications Conference(ACSAC'07)*, 2007.

[49] M. Morgenstern and T. Brosch. Runtime packers: The hidden problem? `http://bakcjhatnetworks.com/html/bh-usa-06/bh-usa-06-speakers.html`.

[50] M. Morgenstern and T. Brosch. Runtime packers: The hidden problem? `http:/www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf/`.

[51] C. Nachenberg. Understanding heuristics: Symantec's bloodhound technology. `http://www.symantec.com/business/security_response/whitepaperarchive.jsp`, May 1998.

[52] nergal. The advanced return-into-lib(c) exploits (pax case study). *Pharack Magazine*, 11:File 4, 2001.

[53] A. One. Smashing the stack for fun and profit. *Pharack Magazine*, 7(49):File 14, 1996.

[54] Y.-J. Park and G. Lee. Repairing return address stack for buffer overflow protection. In *Proceedings of the First Conference on Computing Frontiers*, pages 335–342, Ischia, Italy, 2004.

[55] Y.-J. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE Micro*, 26(4):62–71, July/August 2006.

[56] PEiD. `http://www.peid.info`.

[57] C. Pyo and G. Lee. Encoding function pointers and memory arrangement checking against buffer overflowattack. In *In Proceedings of the 4th International Conference In Information and Communications Security*, 2002.

[58] F. Quin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture, 2006*, San Francisco, CA, 2006.

[59] M. F. Ringenburg and D. Grossman. Preventing format-string attacks via automatic and efficient dynamic checking. In *Conference on Computer and Communication Security*, pages 354–363, 2005.

[60] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *In Proceedings of 22th Annual Computer Security Applications Conference(ACSAC'06)*, 2006.

[61] H.-R. SA. Ida pro disassembler and debugger. `http://www.www.hex-rays.com/idapro/`.

[62] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of IEEE Symposium on Security and Privacy*, page 144:155, 2001.

[63] B. Steensgaard. Points-to analysis in almost linear time. In *In proceedings of Symposium on Principles of Progamming Language*, 1996.

[64] A. Stepan. Improving proactive detection of packed malware. In *In Virus Bulletin*, pages 11–13, 2006.

[65] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, 2004.

[66] P. Szor. *The Art of Computer Virus Research and Defense*. Addision-Wesley Professional;illustrated edtion, 2005.

[67] P. Szor and P. Ferrie. Hunting for metamorphic. In *In Virus Bulletin Conference*, page 123:144, 2001.

[68] P. Vandevenne. Using the universal pe plug-in in ida pro 4.9 to unpack compressed executables. `http://www.hex-rays.com/iapro/unpack_pe/unpacking.pdf`, 2005.

[69] J. Viega, J. Bloch, T. Khono, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *In proceedings of the 168th Annual Computer Security Applications Conference*, New Orleans, Louisiana, 2000.

[70] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, 2001.

[71] C. Willems, T. Holz, and F. Freilling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32:39, Mar/Apr 2007.

[72] Q. Zhang and D. S. Reeves. Metaaware: Identifying metamorphic malware. In *In Proceedings of 23rd Annual Computer Security Applications Conference(ACSAC'07)*, 2007.

[73] T. Zhang, X. Zhuang, S. pande, and W. Lee. Anomalous path detection with hardware support. In *Proceedings of International Conference on Compilers, Architectures and Synthiesis for Embedded Processors*, San Francisco, CA, 2005.

[74] X. Zhou, B. XU, Y. Qi, and J. Li. Mrsi: A fast pattern matching algorithm for anti-virus applications. In *In Proceedings of Seventh International Conference on Networking*, page 256:261, 2003.