Hardware Supported Anomaly Detection: down to the Control Flow Level

Tao Zhang Xiaotong Zhuang Santosh Pande Wenke Lee
Georgia Institute of Technology
College of Computing
801 Atlantic Drive
Atlanta, GA, 30332-0280
{zhangtao, xt2000, santosh, wenke}@cc.gatech.edu

Abstract

Modern computer systems are plagued with security flaws, making them vulnerable to various malicious attacks. Intrusion detection systems have been proposed to protect computer systems from unauthorized penetration. Detecting an attack early on pays off since further damage is avoided and resilient recovery could be adopted. An intrusion detection system monitors dynamic program behavior against normal program behavior and raises an alert when anomaly is detected. The normal behaviour is learnt by the system through training and profiling.

However, all current intrusion detection systems are purely software based and thus suffer from huge performance degradation due to constant monitoring operations inserted in the application code. Due to the potential performance overhead, software based solutions cannot monitor the program behavior at a very fine level of granularity, thus leaving potential security holes as shown in [5]. In this paper, we propose a hardware-based approach to verify the control flow of target applications dynamically and to detect anomalous executions. With hardware support, our approach offers multiple advantages over software based solutions including near zero performance degradation, much stronger detection capability (a larger variety of attacks get detected) and zero-latency reaction upon anomaly and thus much better security.

Categories and Subject Descriptors:
D.3.4 [Programming Languages]: Processors—Run-time environments, Optimization, Code generation.

General Terms: Performance.

Keywords: Anomaly Detection, Control Flow Graph, System Call

1. INTRODUCTION

Modern computers are plagued with security flaws. Potential holes like software bugs, misconfigurations, misuses etc., render computer systems vulnerable to malicious attacks in a networked environment. In recent years, CERT has observed more than double vulnerabilities being discovered each year accompanying a dramatic increase of intrusion activities [1].

Intrusion detection

To protect computer systems from unauthorized penetration, intrusion detection system is one of the techniques to identify such attempts. Due to the following reasons, intrusion detection has become an indispensable means to help secure a networked computer system. 1) A completely secure system is impossible to build. 2) Protecting the software itself through cryptographic mechanisms, such as in the XOM [8] architecture, cannot prevent software's internal flaws, which might be exploited by deliberate intruders. 3) Other operational mistakes, like misuses, misconfigurations etc., may jeopardize the systems as well.

Misuse detection and anomaly detection

Traditionally, intrusion detection can be classified into misuse detection and anomaly detection. Misuse detection tries to identify known patterns of intrusions with pre-identified intrusion signatures, while anomaly detection assumes the nature of the intrusion is unknown, but will somehow deviate the program's normal behavior. Misuse detection is more accurate, but suffers from its inability to identify novel attacks. Anomaly detection can be applied to a wide variety of unknown (new) attacks, however the distinction between normal behavior and anomaly must be properly defined to reduce the number of false positives (or false alarms). This paper will focus on anomaly detection.

A number of anomaly detection techniques have been proposed in the security domain. Most early anomaly detection approaches analyze audit records against profiles of normal user behavior. In their groundbreaking paper [2], Forrest et al. found out that system call trace is a good way to depict a program's normal behavior, and anomalous program execution tends to produce distinguishable system call traces. They record all normal N-grams, i.e. N consecutive system calls, during the learning phase, and use them to detect anomalous system call sequences in runtime. Later, [6] and [7] aim to represent the normal N-grams compactly with finite-state automata (FSA). Recent advances [3][4][5] suggest to include other program information to achieve faster and more accurate anomaly detection. R. Sekar et al. [4] combines the program counter with system call names, so that the state machine becomes deterministic and more accurate. D. Wagner et al. [3] proposed several anomaly detection models based on static analysis of the program code. Their call graph model builds up a non-deterministic finite-state machine (NDFA) extracted from system calls on the call graph. Call graph model guarantees no false positives, but may contain impossible paths that might be exploited by the attacker. Thus, they further proposed the abstract stack model, which involves a pushdown NDFA. Recently, H.H. Feng, et al. [5] put forward a new system-call based approach, which constructs a return address table and a virtual path (procedure entry/exit points traversed between two system calls) table during the training phase and detect anomalies afterwards.

Although anomaly detection through system call monitoring has been shown to perform moderately well, several aforementioned papers [2][5] suggest that to detect more subtle attacks, approaches with finer granularity should be taken with consideration to more control flow information. However, current software anomaly detection systems already suffer from huge performance degradation due to inserted monitoring code, even operating at system call granularity. It is infeasible to extend granularity further in a software based solution. Moreover, the anomaly detection software can be attacked itself like any other software, leading to security weakness.

In this paper, we propose a hardware-based approach to monitor the control flow graph of the target programs and to detect anomalous executions. Our approach offers multiple advantages over software based solutions including near zero performance degradation, much stronger detection capability and zero-latency reaction upon anomaly thus much better security.

The rest of the paper is organized as follows: section 2 gives background knowledge and summarizes related work; section 3 introduces the XOM machine model on which our work is based; section 4 elaborates our hardware dynamic control flow monitoring scheme; section 5 discusses subtle issues in our scheme; section 6 presents experiments and results; and section 7 concludes the paper.

2. Motivation and Related Work

The goal of anomaly detection is to maximally distinguish between normal and abnormal behavior. In other words, it should seldom trigger *false positives* (false alarms) for normal behavior and it should detect the anomaly maximally or reduce *false negatives* (undetected attacks) as much as possible.

It has been well acknowledged that monitoring program behavior dynamically is a good means to detect anomaly. Naturally, program behavior includes function calls generated at runtime, data accessed during the execution, etc. Researchers have shown that a great number of attacks can be thereby detected by analyzing program trail during its execution.

To illustrate the pros and cons of existing approaches, we first introduce the buffer overflow attack, and then evaluate the detection capability of a variety of techniques.

```
int read_input_file (FILE *fp) {
    char buffer[256];

    while(fgets(buf, 400, fp)){
        ......
    }
}
int main(){
    FILE *fp=fopen("inputfile","rt");
    read_input_file(fp);
    fclose(fp);
}
```

(a)

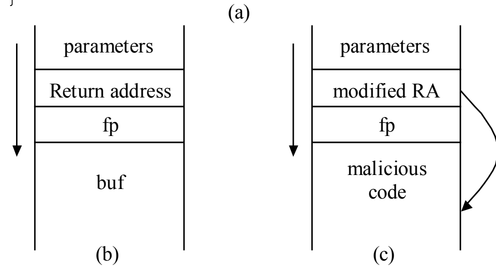


Figure 1. Buffer overflow attack.

Buffer Overflow Attack

Figure 1.a shows a code segment, which illustrates buffer overflow attack. The *main* function opens a file, reads data from the file and finally closes the file. In function *read_input_file(FILE* fp)*, the *fgets* function reads data from the file to a local buffer for further processing. However, due to a programmer mistake, the size of the read-in data may exceed the boundary of the buffer (such a thing can also happen if one uses *gets()* which keeps reading until end of line or file and can result in overflow) . Thus, the attacker manipulated input can overflow the buffer and overwrite other critical data also on the program stack deliberately, such as the return address of the function. Thus, when the function returns, the execution flow can be directed to the malicious code injected by the attacker. Also, parameters or other local variables can also be easily overwritten to create security holes. For example, sensitive data might be exposed or super user privilege might be granted illegally.

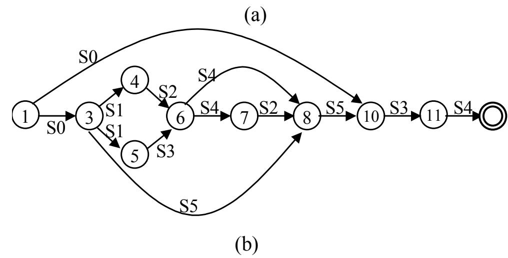
Anomaly Detection via System Call Monitoring

System calls are generated as the program interacts with the kernel during its execution, such as the *fopen*, *fgets*, *fclose* in Figure 1.a. [2] argues that system call trace might be a good starting point for detecting anomaly. System call trace can be considered as a distilled execution trace, leaving many program structures out. As can be seen from Figure 1, if the return address on stack is tampered, the program cannot return to the normal program point after *read_input_file* in function *main* and call *fclose*, instead, the malicious code might invoke other system calls, for example *fork*, in order to damage the system, which means the system call trace is likely to be discernible from the normal case. Although system call trace is a great simplification of the whole program activities, storing and checking against all normal system call traces is a tremendous design effort. The classical solution is based on FSA [6][7].

System Call Monitoring Based on FSA

```
1.  S0
2.  while(…)
3.      S1;
4.      if(…)S2;
5.      else S3;
6.      if(S4)…;
7.      else S2;
8.      S5;
9.  }
10. S3;
11. S4;
```

(a)



(b)

Figure 2. FSA with pc as states.

Next, we give an example to explain how FSA techniques work and the previous work on incorporating additional program information into the FSA model to improve detection capability.

In Figure 2, each program statement invoking a system call becomes a state on the state machine diagram. The transitions between states are triggered by system calls. Each transition edge in the FSA is labeled by the triggering system call and the target state is determined by the feasible control flow. The state machine can be easily constructed through static analysis of the program [3] or dynamic learning of the system call trace and program counter [4]. The state machine is typically non-deterministic, e.g. the state 3 on Figure 2.b may lead to state 4 or state 5 after S1 is called. A non-deterministic state machine can be converted to a deterministic one with an increase of states.

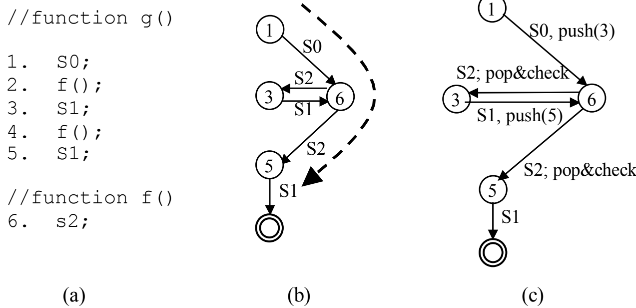


Figure 3. Impossible path problem and FSA with pc and return address stack (modified example from [4]).

One significant property of establishing the FSA based on static program analysis is that no false positives will be generated, due to the conservative nature of static analysis. Thus, all normal executions will not trigger alarms and if the state machine reaches the error state, there is a guarantee that something is anomalous.

The above basic FSA model has the weakness of impossible path problem[3], which is illustrated in Figure 3.

In Figure 3.a, function f() is called twice in function g(), therefore we will have the FSA constructed as in Figure 3.b. If buffer overflow happens in f(), the attacker can change the return address to 5, which is an illegal path. However, from the FSA constructed, the path from state 1 to state 6 to state 5 is still legal, leaving the buffer overflow undetected. In other words, syscall trace S0,S2,S1 becomes possible in this code segment. Figure 3.c shows how the abstract stack model [3] solves this problem. It maintains a return address stack along with the FSA. In the example, when we take the path from state 1 to state 6, 3 is pushed to the stack. Upon transition from state 6 to state 3, we pop the top of the stack and check if it accords with the new state. In this way, path from state 1 to 6 to 5 is found to be invalid because the pop-up state does not match the destination state after the transition from state 6 to 5.

However, [5] points out that even the call stack model can miss important anomaly. In Figure 4, although the condition in line 2 is false, i.e. this is not a super user, the attacker can cause a buffer overflow to return from f() to line 7 instead of line 4, which actually grants him super user privilege. This attack is not detectable under abstract stack model, since the system call trace as shown in Figure 4.b is the same as an execution path going from line 2 to 6 to 8 to 7, which is a legal path for super user. The above example shows that the granularity at system call level is not fine enough to detect many obvious anomalies in reality.

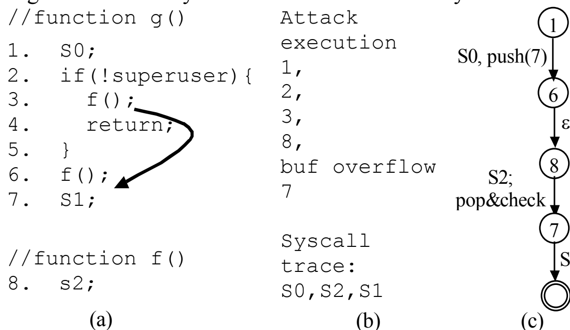


Figure 4. FSA with return address stack (simplified example from [5]).

To detect such anomaly, more program information, esp. control flow information, has to be incorporated. Recently, [5] proposes to constructs two hash tables to store not only the return addresses but so-called *virtual paths*. Virtual paths record procedure entry/exit points traversed between two system calls. By keeping tracking of more program information at runtime, detection capability is improved (the example in Figure 4 is shown to be detectable). However, the approach in [5] is still unable to detect attacks with finer granularity.

```
//function g()
1.  if(!superuser){
2.    f();
3.    return;
4.  }
5.  //become superuser
6.  execve("/bin/sh");

//function f()
8.  no syscall but overflows;
```

Figure 5. A simple example to evade anomaly detector in [5]

Figure 5 shows a simplified example extracted from [5]. When there is no system call in function f(), a buffer overflow attack can easily escalate the privilege without being detected

Control Flow Monitoring

As concluded in [5], the ability of anomaly detections systems heavily relies on the granularity it monitors. An anomaly detector solely relying on system call trace is crippled if an attack takes place between two system calls. Also, it is easy to think about other examples that can foil the anomaly detector even if limited program control flow information is considered, such as the example in Figure 5.

We argue that a granularity at control flow level is fine enough to detect most possible attacks. Most attacks would require a change in program control flow. We believe tampering a program without changing its control flow is very difficult. For integer programs, normally there is a change in control flow (a jump like instruction) in every 6 to 10 instructions, which means control flow level is very fine grained. To enforce *control flow level monitoring and detection*, first we need to make sure the program is sequentially executed except when *jump instructions* are encountered. *Jump instructions* include branch instructions, function calls/returns and any other instructions that could change program PC in a non-sequential way. For jump instructions, the anomaly detector should further verify whether the jump target follows the program normal behavior. This includes checking whether the jump target is possible and normal. For direct branches and direct function calls, the jump target is known during compilation. For indirect jumps, the compiler can determine a set of possible jump targets through pointer analysis. In real programs, an indirect jump will not have many possible targets. In fact, many of them have only one possible target. Jumping to other impossible targets surely implies anomaly. Moreover, through profiling we find the most commonly taken target(s) for a given jump and use that as a normal behavior. For each function return instruction, we should keep track of the call site it corresponds to and make sure the execution is back to the correct call site. By doing so, tampered return address on the stack through buffer overflow can be easily detected.

Obviously, control flow level monitoring achieves the finest granularity among all anomaly detection techniques proposed so far. It *subsumes* the system call trace based anomaly detection since all paths between two syscall calls are verified, down to basic block level. Performing detection at such a low level brings great potential to detect future attacks exploiting subtle control flow

deviations. As shown in Figure 5, only one anomalous return grants the attacker super user privilege.

Drawbacks of the Software Solutions

Though it is obvious that finer monitoring granularity brings better detection capability, up until now, no solution based on control flow level monitoring has been proposed. The major reason is the performance overhead. Software based anomaly detection systems suffer from huge performance degradation due to inserted monitoring code, even operating at the system call granularity. According to our experience, a FSA based software anomaly detector can degrade the performance of monitored program by tens of times. For example, with monitoring enabled, sending an email using *sendmail* program takes around several tens of minutes [3]. Moreover, anomaly detection software, as any software, can be attacked itself. Thus, the security for the whole system can be hardly guaranteed.

The above arguments prompt us to implement control flow level monitoring at hardware level. In this paper, we propose a hardware based anomaly detection system monitoring at control flow level. Our approach offers multiple advantages over software based solutions including near zero performance degradation, much stronger detection capability and zero latency reaction upon anomaly thus much better security.

3. Introduction to XOM Secure Processor Model

In this section, we introduce the well-known XOM secure processor model [8], which has been widely accepted in the secure architecture domain. Our hardware anomaly detection system is based on XOM.

Under XOM model, only the central processor chip is assumed to be secure. The processor securely possesses the private part of a public/private key pair. The private key is the root of the security in XOM model. A session key for symmetric encryption/decryption is chosen by the software vendor every time the software is released to the customer. All the code and data of the software are initially encrypted using the session key. The session key itself is encrypted using the customer processor's public key then decrypted by the processor. Symmetric encryption is used to reduce the overhead of decrypting the possibly huge program code and data. XOM model provides fundamental support for copy protection since the released program binary (encrypted using the session key) can only be executed on the right XOM processor (the wrong processor will get the wrong session key). Memory integrity checking is introduced later in the XOM model to further enhance its security [10][11]. Integrity checking guarantees that the contents in external memory cannot be modified by an external attacker without being detected.

To support multiple processes, a session key table is stored on chip to distinguish different processes. Once the execution of the current process is interrupted, the processor automatically encrypts and hashes all on-chip shared data like registers and stores the hash value along with the session key entry for that process. When the process is resumed, the processor restores the on-chip shared data and verify its validity.

Under XOM model, OS is not trusted. But there is a secure kernel residing in the processor and is trusted. Secure kernel has higher privilege than the untrusted OS and has access to all XOM hardware components like session keys and secure on-chip memory. Secure kernel provides basic supports for encryption/decryption, data tagging etc. and implements several special instructions for XOM mode, for example, *enter_xom* and *exit_xom*.

The current XOM model is able to guard against attacks on hardware components, e.g., attacks on insecure bus or external memory. It also protects programs from being exposed and tampered by other malicious processes (including the OS) running on the same processor. However, the XOM model is unable to detect intrusions, because intrusions take advantage of software bugs like buffer overflows, or misconfigurations, misuses etc., of the victim program. During intrusions, the attacker does not have to tamper the program first. On the other hand, the attacker exploits the problems in the program itself or the improper usage of the program and utilize the program to against itself. Under XOM model, the processor is completely unaware of the bugs or the misuses of the program. For example, upon a buffer overflow, the XOM processor will overwrite the call stack for the attacker, thinking it just performing common loads/stores for the program. XOM does not undertake bounds checking of the input and thus it is possible to send long strings to overwrite return addresses launching buffer overflow kind of attacks. Although it may be difficult to slip in malignant code, one could still crash the application leading to denial of service attack. Moreover, XOM model obviously cannot prevent misuses or misconfigurations of the program.

In our approach, isolation between different processes is still required. Also, the normal behavior profile must be initially stored in memory and must be immune from tampering. Hence, the XOM model forms a proper infrastructure for us to monitor control flow dynamically.

4. Hardware Based Dynamic Control Flow Monitoring

In this section, we elaborate our hardware based anomaly detection system, operating on program control flow level. Our approach achieves much better performance over software based solutions, since the monitor is entirely implemented in hardware and works in parallel with the instruction execution. While in system call tracing based software approaches, intercepting syscall alone can incur 100% to 250% overhead [4]. According to our experiences, the real performance degradation may reach tens of times. Our approach also achieves much stronger detection capability since we monitor the program at control flow level, which cannot be done in software without huge performance degradation. Finally, our approach offers much better security. The hardware is able to act immediately once an anomaly is detected, thus guarantees to prevent further damages to the system. In other words, our system offers *intrusion prevention* ability. *Intrusion prevention* cannot be achieved by software solutions, since it is not realistic for software solutions to monitor on instruction level. The granularity that a software solution can achieve is much larger, thus there is a delay between anomaly occurrence and anomaly detection. For example, for software anomaly detection systems monitoring system call traces, the anomaly will not be detected until the next system call. The delay could be utilized by the attacker and the system may have been tampered before the attack is detected and any action is taken. In addition, since our detector resides in the XOM processor, the attacker is not able to tamper the detector itself. As against this, in software based approaches, the anomaly detector is simply another program that is subject to all kinds of malicious attacks. Finally, unlike software based approaches, our anomaly detection system does not assume OS is secure and un-tampered. Our assumption is consistent with the

assumption of XOM model. Thus, under our approach, the OS is nothing but another software and it can be monitored too.

4.1 Architecture Overview

Figure 6 shows an overview of the components in our hardware anomaly detection system. Some components are inherited from the XOM machine model. By integrating dynamic control flow checking with XOM, we add anomaly detection ability to XOM. On the other hand, our scheme heavily relies on data confidentiality and integrity guarantee of XOM.

The processor chip (upper part) is physically secure and the memory (lower part) is subject to attacks. Inside the processor chip (shown in the upper left box), the processor maintains two master keys and a table records 3 entries for each process. On the upper right part, all process private data is stored. This area is saved and restored upon context switches. In addition to the processor pipeline, caches and other components, a detector accepts dispatched instructions and checks them against normal behavior profile and the RA stack. The checking is done in parallel with the instruction execution to reduce performance overhead. However, unless the checking is done, the instruction being checked cannot retire from the processor. We assume in-order retirement. Thus, we guarantee that any instructions following an anomalous instruction cannot change machine state. In other words, our system can *prevent intrusion* at the very beginning of it, offering much better security. As an example, in buffer overflow attack, the anomalous return instruction will be captured during its execution and the control flow will not be directed to the malicious code. The attacker has no chance to launch a successful attack at all.

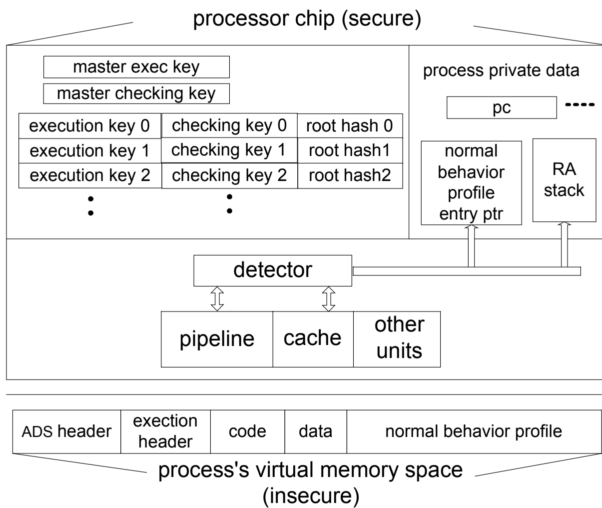


Figure 6. Architectural overview for dynamic control flow monitoring.

There are two master keys (the private part of public/private key pair) stored on-chip. One is called *master execution key* and the other is called *master checking key*. The program is encrypted using a session key contained in the execution header. The execution header is in turn encrypted by the corresponding public key of the master execution key. On the other hand, the normal behavior profile is encrypted with another session key contained in the ADS header. The ADS header is encrypted using the corresponding public key of the master checking key. To guarantee the integrity of the normal behavior profile, the authentic profile is signed by an authority. The authorities can be organized hierarchically and each software vendor can become a leaf of the trust hierarchy. Thus, each software vendor can generate normal behavior profile data for its software and sign it. The signature is then put into the ADS header. For clarity, we call the session key that is used to encrypt the program *execution key* and the one to encrypt the normal behavior profile *checking key*. Both of them are symmetric encryption keys that can be efficiently implemented in hardware. To support multi-processing, session keys of all active processes (either executing for stalled) are kept in a table, which is only accessible by the processor.

Upon starting a new program, the OS should layout the program code, static data and the normal behavior profile in the process's memory space, then a special instruction[1] with pointers to the execution header (if encryption is enabled) and the ADS header (if control flow checking is enabled) is executed. A hash tree for integrity checking is constructed in the external memory and the root hash is verified with the one contained in the execution header.

When a process is context switched, the hardware automatically encrypts the process's private data, which includes the data that is normally saved during context switches and that is special for XOM and dynamic control flow monitoring, like the normal behavior profile entry point and the RA (return address) stack. The process private data may reside in the external memory. The integrity of the private data is guaranteed by integrity checking scheme. The root hash of an active process never goes to external memory and is recorded in the same tables as the session keys are.

The detector utilizes the normal behavior profile and the RA stack to check instructions executed dynamically. Normal behavior profile contains several tables that record the normal execution trace the program follows. Upon reaching a *jump instruction* (defined in page 3), the detector searches the tables to determine if the control flow should jump and if it should, it further checks if the jump target is a valid and normal one. RA stack maintains all function return addresses above the current function on the call stack. Upon function calls, the return address is pushed to the RA stack for future checking when the function returns. RA stack also handles spilling/reloading return addresses when the on-chip area is not enough to hold all return addresses.

Finally, although the normal behavior profile and the spilled RA stack are in the same virtual address space as the program code and data, the processor ensures the running program cannot touch the data, instead only the detector can access them and monitor the program behavior based on them. The address space for normal behavior profile and RA stack is known to the OS as reserved for anomaly detection, therefore no space should be allocated from this area to the program. Any instructions attempting to read/write this reserved space will be detected and stopped by the processor. In this way, we achieve complete protection of the normal behavior profile and spilled RA stack from being tampered by the malicious program.

4.2 Checking Anomaly with Hardware

We now elaborate the hardware design to manage normal behavior profile and the RA stack. As we know, only a small number of instructions (about 10%) can cause non-sequential execution of the program, i.e. they are jump instructions. However, such instructions are not uniformly distributed in the code, i.e. the number of non-jump instructions between two jump instructions

[1] enter_xom (execution_header_pointer, ADS_header_pointer)

varies a lot. Another difficulty is that there are a variety of such instructions, like branch instructions, calls/returns, etc. Moreover, indirect branch instructions and indirect function call instructions can have many possible jump targets. On the other hand, the hardware design prefers simple, uniform data structure that can be handled efficiently.

We notice that jump instructions have their special properties. The majority of jump instructions (80%-90%) are direct branches or direct call instructions. The jump target of such instructions is known during compilation time. Therefore, for such instructions we only need to record the offset of the jump target to the current PC address and two boolean values indicating whether a taken case is normal and whether a non-taken case is normal. Some jumps are not biased, thus both taken and not-taken are normal.

Hash table is the natural way to store target address and normal behavior for direct jumps. To check a direct jump instruction, we first calculate the hash value of its address, then use the hash value as index to the hash table to retrieve the corresponding information. However, there are a couple of problems with a simple hash table design. First, a simple hash table cannot exploit the code locality. The hash table entries for two adjacent direct jumps could be far away. Moreover, the processor always fetches a cache block of data from external memory. Using a simple hash table, it is very possible that only one record in the fetched block is touched before its eviction, which is greatly inefficient. Second, hashing can cause collisions. To avoid fetching the wrong information for a jump instruction, each hash entry has to be tagged which could also waste space. For example, alpha processor has 64 bit address space, which means the tag field has 62 bits, whereas the offset field is only 21 bits. Due to the above reasons, our anomaly detection system abandons a simple hash table design and instead deploys a specially optimized one.

Checking Pipeline in the Detection System

Figure 7 shows an overall picture of our anomaly detector. During execution, each dispatched instruction is sent to the detector with its PC value (*cur_addr*) and the type of the instruction. The type can be “non-jump”, “direct branch”, “indirect branch”, “direct call”, “indirect call” and “return”. During decode stage, the type of the instruction can be easily obtained. For direct jumps, the offset field obtained during decode stage is added to *cur_addr* to get the target address (*next_addr*) and the *next_addr* is also sent to the detector. For indirect jumps, the target address (*next_addr*) generated may not be available when the instruction is dispatched, i.e., the target address has to be loaded into a register first. The target address will be sent to the detector as soon as it is available. By feeding the instruction into the detector as soon as it is dispatched, control flow checking and instruction execution can be performed in parallel and performance degradation is reduced as much as possible. Once the PC address (*cur_addr*) is available, the detector accesses the normal behavior profile to get the corresponding information. The accessing is done in parallel with instruction execution. Some of the dispatched instructions are wrongly speculated, the detector will check them anyway since the performance impact is minor and we achieve better security by checking more instructions.

The first stage in the detector is called *is_jump check*. It verifies whether the instruction at *cur_addr* should be a jump instruction. If the detector finds that the current instruction is a jump but it should not be, there must be an anomaly. All non-jump instructions are not further checked after the first stage. After the first stage, we isolate return instructions from other jump instructions, since return instructions only need to be checked against the top of the RA stack. In the second stage called *normal jump check*, we check jump instructions that are not returns. In this stage, we perform group-wise hashing on the current PC address (*cur_addr*) so that all direct jumps without collisions after hashing can be verified. Next, *special jump check* checks all the jumps left out by the previous stages. At the end of the pipeline, we push the return address of a call instruction to the RA stack. Here, the call instruction has been verified to be valid.

Checking stages are pipelined with frequently used stages stationed earlier. In Figure 7, *is_jump check* stages processes all instructions, but it can be done quickly. The *is_jump* stage filters all non-jump instructions to alleviate the burden of later stages. Similarly, *normal jump check* is slower and requires more data for checking purposes. However, it only looks at jump instructions, which constitute roughly 10% of all instructions. Slower input rate to this stage allows more time to be spent on each instruction without stalling the checking pipeline. Finally, *special jump check* is the slowest due to the irregularity of the data structure and poor locality in cache. But fortunately, very few instructions (less than 2%) reach this stage, leading to a very long interval between two requests to this stage. To amortize the disparity of inter-request intervals, we assume small size *request buffers* exist between any two stages. In practice, request buffers with 10-20 entries are generally enough for such purposes even in the worst case.

Function calls and returns are relatively infrequent events. Moreover, push and pop operations to RA stack can be done very efficiently. Thus return instructions checking has little impact on performance.

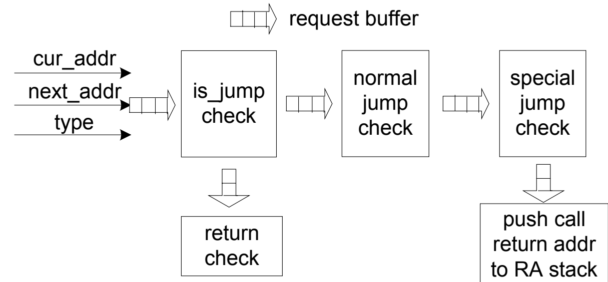


Figure 7. Control flow checking pipeline.

is_jump Check

is_jump stage checks all instructions against a table called *is_jump_table*, which is a part of normal behavior profile. As shown in Figure 8.a, the table is an array with M bits, where M is the number of instructions in the program. Each bit indicates whether the corresponding instruction is a jump instruction (value 1) or not (value 0). The address of the instruction being checked (*cur_addr* in Figure 7) is used as an index to retrieve the bit for that instruction. Obviously, since the array is directly indexed without hashing, it should have good spatial locality in cache. Secondly, the size of the array is much smaller than the original code, because only 1 bit is needed to represent an instruction (1/32 of the original code size if the instruction is 32 bit long). The checking speed is largely decided by the time to fetch the corresponding bit. In the worst case, the bit is in external memory and it would take tens of cycles to fetch it. However, upon a miss, a whole cache block is brought into the cache. Assume the cache line is 32B, that could satisfy the first stage checking of 256 instructions. Thus performance degradation should be minor.

If an anomaly is detected in this stage, i.e. a non-jump

instruction is found to be a jump instruction or vice versa, the detector knows it is absolutely an attack and it raises a *threat* exception. The exceptions are securely handled by the secure kernel, which is explained later.

Normal Jump Check

Although only a small percentage of instructions reach the *normal jump check* stage, they are sporadically distributed. Since a simple hash table design has the drawbacks mentioned before, we propose groupwise hashing. In Figure 8.b, the instruction address is divided into two parts: group ID and group offset. Assume there are N groups and each group contains K entries, then the ith group starts at address i*K. Inside each group, the group offset is first hashed then indexed into one of the entries in the group. The advantage of groupwise hashing is: 1) hashing saves space for non-uniformly distributed addresses of jump instructions. 2) hashing is only performed inside each group and each group has a number of sequentially stored entries. In this way, we can exploit more spatial locality, because adjacent branches are most likely in the same group, their information will be stored adjacently. Some branch addresses may be hashed to the same location, causing collisions. As we observe, with a reasonable hashing function and a proper setup of N and K, collisions rarely happen. In case of a collision, we indicate that the branch should be handled in the *special jump check* stage.

The data structure is detailed in Figure 8.b, each entry in the table is defined as *normal_jump_entry*. Since only direct branches are considered in this stage, each entry needs to specify whether taken or not-taken is considered as normal behavior or they both are normal. These two bits are learnt through profiling and training. The third field *is_special* is to indicate if there is a collision after hashing. Since the normal behavior profile is generated statically, we know whether an instruction after hashing will cause any collision. If so, the corresponding jump instruction should be processed in next stage. In this way, *normal_jump_entry* does not need to have a tag field, which save space greatly. The last field is a 21 bit offset, which is the maximal offset length in Alpha's instruction set.

For each checking request, *cur_addr* is split into group ID and group offset to locate the corresponding entry in the *normal_jump_table*. After getting the entry, if *is_special* bit is set, the request is simply forwarded to the *special jump check* stage. If *is_special* bit is not set, and *next_addr-cur_addr=size of one instruction*, the jump instruction does not jump, we verify if the *nottaken* bit is set in the entry. If *nottaken* bit is set, then the execution flow is in its normal behavior, otherwise there is an anomaly and the attacker raises a warning exception. If *next_addr-cur_addr<>size of one instruction* and *next_addr-cur_addr<>the offset field in the entry*, this jump instruction has jumped to some uncommon place, therefore the detector knows an attack is ongoing and raises a threat exception. Finally, if *next_addr-cur_addr<>size of one instruction* and *next_addr-cur_addr=the offset field in the entry*, the detector further checks if the *taken* bit is set. If it is set, the execution is normal, otherwise the detector raises a warning exception. In conclusion, in cases 2, there must be an attack. In case 1 and case 3, it is possible that the anomaly is caused by some corner cases.

In our scheme, we setup N and K properly so that each cache blocks contains normal branch information for each group. Thus, the branch information for a group is always fetched into the processor together, improving locality. Assume each cache block is 32B thus can contain 10 branch records. We assume only 8 of them are used to reduce collision. So each cache block records branch information for 8 branches. Assume on average, there is one branch instruction per 8 instructions, one cache block has the branch information for 64 instructions, i.e., 256B. Thus, in our design, the length of group offset is 8 bits. $N = \lceil M/2^8 \rceil$, K=10.

Special Jump Check

This stage handles all remaining instructions. They can be direct jumps that cause collisions in *normal jump check* stage, or indirect jumps (indirect branches, indirect calls etc.). To cover all cases, we construct a link list when collision happen--Figure 8.c. *special_jump_entry* stores the PC address (tag) of the corresponding entry so that colliding entries can be distinguished. Note, in our *normal_jump_entry*, there is no tag field. To record more than one jump targets, the size of a *special_jump_entry* is variable. The irregularity of data structure complicates *special jump checking* stage and leads to longer latency for each checking request in this stage. However, normally the number of possible targets is very limited (less than 3), and only less than 2% of instructions reach this stage, thus the performance impact is minor.

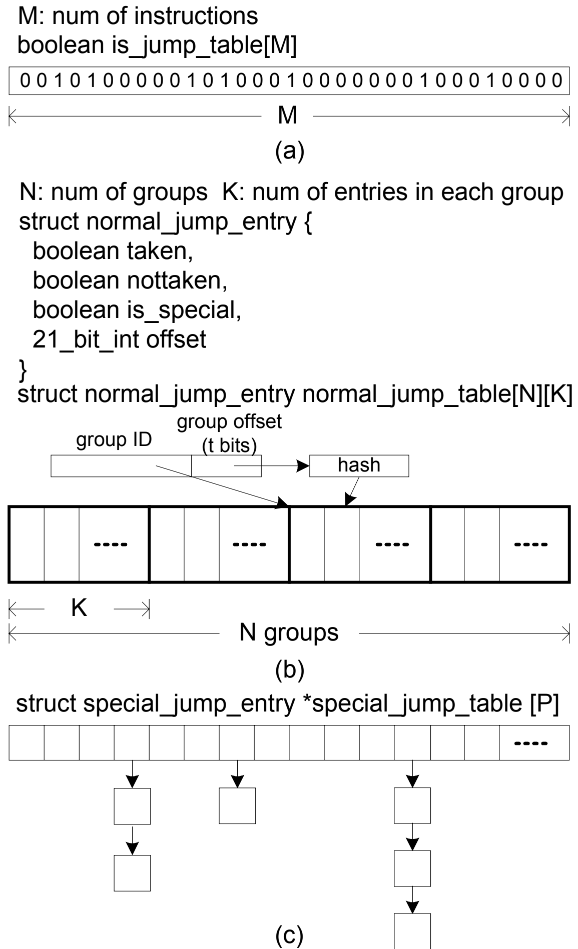


Figure 8. Data structure of the checking tables.

RA Stack, Function Calls and Returns

Function calls and returns involve operations on the RA stack. RA stack is simply a stack for pushing/popping return addresses. As shown in Figure 7, function returns are isolated after *is_jump*

check stage. Each return address of the return instruction is compared against the return address popped up from the top of the RA stack. If they do not match, the detector raises a *threat* exception. Function calls are nothing but direct or indirect branch instructions that are setting return address. They are checked in *normal jump check* and *special jump check* stages. After a function call instruction is checked, the detector pushes the return address of the function to the RA stack for later verification when the function returns.

RA stack only has a limited number of entries on chip. Due to recursive function calls, the call stack depth of a program can be very deep. However, only several entries on the top of the stack are active during any period. Thus, we adopt the idea of *register stack engine* that has been implemented on the Itanium processor [18]. When the on-chip RA stack is close to be full, the detector automatically spills half of the return addresses that are at the bottom of the on-chip RA stack to a reserved and protected memory space. On the other hand, when the return addresses stored on-chip are almost consumed (popped up), the detector fetches (restores) some return addresses back to the on-chip area. In other words, the detector manages to keep the top portion of the RA stack always on-chip. Since all accesses are to the top of the RA stack, this mechanism can successfully hide the latency to spill/restore return addresses between on-chip and off-chip space.

4.3 Handling Anomalies by Secure Kernel

As discussed above, the anomaly detector can raise two kinds of exceptions when an anomaly is detected. By *threat* exception, the detector indicates that there must be an attack ongoing. By *warning* exception, the detector indicates that there may be an attack or some corner cases not seen in training/profiling. The exceptions are handled by the secure kernel residing in XOM processor. Upon a *threat* exception, the secure kernel can dump current execution and detection context, such as the process private data, key table, etc., for further intrusion analysis, then halt the whole system to prevent further tampering to the system. Upon a *warning* exception, the secure kernel can respond in a less offensive way. It can estimate the threat faced by the system based on other information available, then makes a decision that whether to shutdown the system, to kill the anomalous process or to resume its execution.

4.4 Collecting Normal Behavior Profile

Most of the information in normal behavior profile can be collected by the compiler. For direct jumps, the compiler can easily get the correct offset during compilation. For indirect jumps that many have multiple targets, the compiler can get a set of possible targets through pointer analysis. Normally, the number of possible targets for indirect jumps are very limited. To tell whether a taken or not-taken direct branch is normal, profiling of normal program execution in a secure environment has to be done.

4.5 Other Hardware Techniques to Reduce Performance Penalty

In this section, we suggest several hardware optimizations to improve the performance of our anomaly detection system. In particular, we propose prefetching of normal behavior profile, separate cache space for normal behavior profile and customized BTB design.

Prefetching and Separate Normal Profile Cache

There is no upper bound for the size of the normal behavior profile, so it may be too big to be stored entirely on-chip. In some cases, we have to load the needed normal profile data from external memory and it might cost significant latency. In our scheme, the fetching of normal profile data is done in parallel with the execution of the instruction, thus the latency due to profile data fetching is largely hidden.

Prefetching may further reduce the data fetch latency. If a fetch of normal profile data causes a cache miss, the instruction execution may not be able to cover the latency fully. During our experiments, we observe that a significant part of misses due to normal behavior profile data fetching are caused by the fetching of *special jump entries*, due to their lack of locality. Thus, we propose compiler-assisted *special jump entry* prefetching. In particular, the compiler inserts prefetching instructions for all the indirect jump instructions in each function at the function entry point. The prefetching instruction just gives some hint to the processor, but does not grant any read/write access of normal behavior profile data to the program being monitored. Since there are only a few of indirect jumps in each function, the code increase is minor. Our results show that prefetching helps reduce performance degradation further.

Under our default scheme, the profile data share caches with original program code and data, which leads to competition of shared cache space and degradation of cache performance. Thus, we propose a separate cache for profile data to avoid the cache pollution. The cache can be very small compared with the size of original on-chip caches for the execution of the program.

Customized BTB

Branch target buffer (BTB) is widely used in modern processors to remember the branch target of a branch instruction and to help provide next instruction fetch address at the end of instruction fetch stage.

Assume a tagged BTB design, each BTB entry contains a jump PC address as tag and the target address of the jump instruction. It is easy to see that the structure is similar to the *normal_jump_entry* defined in Figure 8.b, leading to the chance of utilizing BTB to reduce performance overhead further. In our customized BTB design, each BTB entry not only records the target address of the jump but also includes two additional bits in the *normal_jump_entry* – taken and nottaken. In this way, each BTB entry can contain the normal behavior profile information for a direct jump. For indirect jumps with multiple targets, one BTB entry is not enough. Whenever a new direct jump instruction is updated into BTB, we not only update the target address of the BTB entry, but also the two additional bits. The BTB can function as before. The only extension is the two additional bits. With customized BTB design, for each direct jump instruction, if the jump hits in the BTB (which we will know in the instruction fetch stage) the detector does not need to fetch corresponding normal behavior profile data from the cache since it is already in the BTB. In other words, the customized BTB can behave like a cache for normal profile data of direct jump instructions. Large BTB is common in modern processors and it can achieve a good hit rate. Thus, the traffic due to *normal_jump_entry* fetching under our customized BTB design is reduced greatly.

4.6 Example Revisited

The example in Figure 5 illustrates an attack that all previous anomaly detection approaches cannot detect. Under our hardware anomaly detection system with dynamic control flow checking, when f() is called in line 2, the return address is pushed to the top of the RA stack. Therefore, upon returning from f(), a mismatch between the tampered return address and the return address popped

from the RA stack will definitely be caught by the detector.

5. Other Considerations

DLLs are shared by many processes and are loaded on demand. They can be protected in the same way as normal programs, as long as normal behavior profile data are created and loaded together with the DLL. The instruction addresses in the normal profile data for a DLL should be relative to the beginning of the DLL, so that they are independent to the actual location the DLL is loaded to the program's virtual address space. Another point is that normal behavior may be different when a DLL is invoked by different applications. So it is hard to define "normal" for a DLL in some cases. However, the CFG of the DLL is static and fixed. Our hardware anomaly detection system can verify the control flow of the DLL anyway. When invoking a function in a DLL, there should be some way to verify if the right DLL is called to avoid a Trojan version of the DLL. This can be fulfilled with certain authentication mechanism.

System calls like *fork* and *exec* family can create copies of the running process or overwrite it completely. Since the XOM architecture and OS [12] can handle this securely, our anomaly detection system simply follows their framework. A copy of the normal profile data can be created if the process is forked. Also, new normal profile data can be loaded to the memory space if an *exec* family system call is executed.

To support multi-threading, multiple RA stacks have to be deployed and each thread has its own RA stack.

The *setjmp*()/*longjmp*() functions are used in exception and error handling. *setjmp*() saves the stack context and other machine state for later recovery by invoking *longjmp*(). Thus, after *longjmp*(), the program resumes as if the *setjmp*() just returns. *longjmp*() will confuse RA stack checking since the function calling *longjmp*() never returns, as well as its active parent functions. *longjmp*() function is actually implemented by an indirect jump instruction. The possible jump targets can be collected through compiler analysis too. The peculiarity of *longjmp*() function is that the jump target can cross functions. To avoid the false positives caused by *longjmp*(), first the detector has to be aware of the particular indirect jump instruction, which can be done easily by monitoring PC addresses. Second, after the execution of this particular indirect jump instruction, the RA stack has to be recovered to the particular state when *setjmp*() is called. To support this, whenever *setjmp*() is called, a snapshot of current RA stack has to be taken and recorded by the detector for later recovering. The detector can detect the execution of *setjmp*() by monitoring PC addresses.

6. Evaluation

Our hardware anomaly detection system is proposed as an enhancement to the current XOM-based secure processor designs. In our experiments, we implemented a XOM-based secure processor with both confidentiality and integrity protection as our baseline processor. In particular, we implemented the OTP (one time pad) encryption scheme proposed in [9], which achieves very low encryption/decryption overhead. We also implemented the integrity checking scheme proposed in [11]. All the hardware modeling is done in Simplescalar toolset [16]. We choose all SPEC2000 integer programs as benchmarks. We perform our experiments based on SimPoint [17] to capture the characteristics of benchmarks quickly and accurately. Each benchmark is first fast-forwarded according to SimPoint then simulated by 100M instructions. The default parameters of our processor model is shown in Table 1.

Table 1. Default Parameters of the Processor Simulated

Clock frequency	1 GHz	L1 I/D	64K, 2 way, 2 cycle 32B block
Fetch queue	32 entries	Unified L2	512K, 4way, 32B block Latency 10 cycles
Decode width	8	Memory bus	200M, 8 Byte wide
Issue width	8	Memory latency	first chunk: 80 cycles, inter chunk: 5 cycles
Commit width	8	SNC cache	64K, 4way, 32B
RUU size	128	Encryption/ Decryption latency	50ns
LSQ size	64	TLB miss	30 cycles
Branch predictor	2 Level		

Figure 9 show the distribution of jump instructions. On average, 12% of total program instructions are jumps and only 1.5% of program instructions are indirect jumps (including indirect branches and indirect calls). The above result shows that our staged anomaly detection system is very reasonable. First stage is the fastest and processes all instructions. Only around 12% instructions reach the second stage, which is slower. Finally, only around 2% instructions reach the slowest third stage.

Figure 10 shows the performance degradation of our hardware anomaly detection system over the baseline processor. Normalized IPC over our default configuration is shown. By default, the normal behavior profile data shares L1 data cache and L2 cache with the original program code and data. There are no other architectural optimizations enabled in default configuration. Our default configuration is chosen to avoid the hardware cost and modification to current processor designs as much as possible. On average, the performance degradation due to anomaly detection under default configuration is only 0.96%. Benchmark *gcc* has worst degradation – 2.7%. Even without other optimizations, our anomaly detection system performs exceptionally well. The major reasons are: 1) The fetching of normal behavior profile data is performed in parallel with the execution of the instruction. Most of the profile data fetching latency is hidden. 2) We discard the simple hash table design and choose groupwise hash table to improve the locality of profile data accesses. 3) The size of the profile data is small. Normally the profile data is less than 20% of the program code size. Accesses to this small piece of data do not cause much cache pollutions and cache misses.

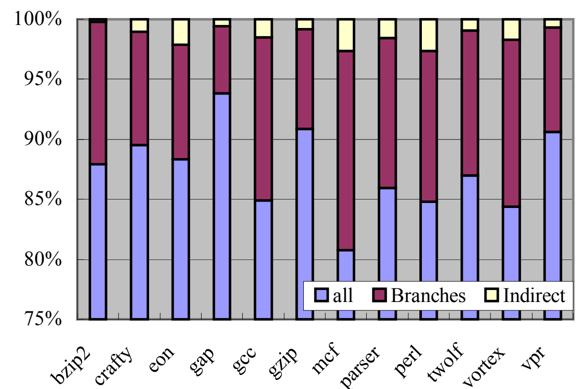


Figure 9. Branch Distribution

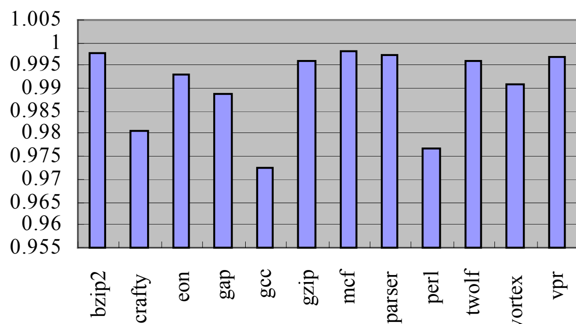


Figure 10. Normalized IPC under Default Configuration.

Figure 11 shows the effect of a separate cache for normal behavior profile data. Four configurations are shown. Default (without separate cache), 4K separate cache, 8K separate cache and 16K separate cache. The separate cache is 4-way and 32B block size. With a separate cache, the performance of original L1 and L2 caches will not be degraded, leading to performance improvement. Since many benchmarks have near zero performance degradation even under default configuration, a separate cache does little for them. For several benchmarks suffering relatively big performance degradation, a separate cache can reduce the degradation efficiently.

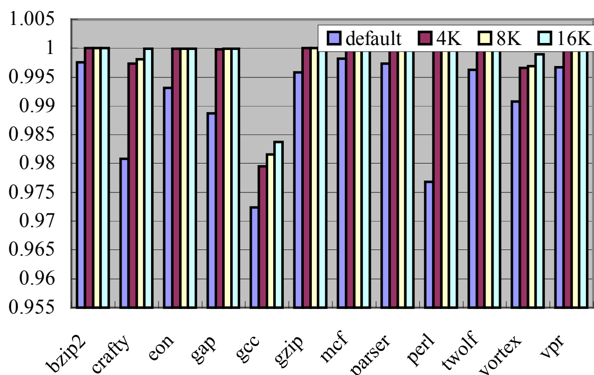


Figure 11. Separate Cache for Normal Behavior Profile

Figure 12 shows the results of prefetching and customized BTB optimizations. The IPC is normalized to the default configuration. From the results, prefetching is ineffective to improve the performance. Although prefetching can reduce latency of some accesses, it may also kick out some active blocks in cache thus degrade cache performance. To fully exploit the power of prefetching, it takes advanced prefetching algorithm to reduce the negative effect of prefetching as much as possible, which is not the main interest of our paper, since the performance degradation is already minor under basic configuration. On the other hand, our customized BTB design leads to apparent performance improvement for some benchmarks. BTB optimization is more efficient because it eliminates a significant portion of accesses to caches as explained earlier, without any side effect.

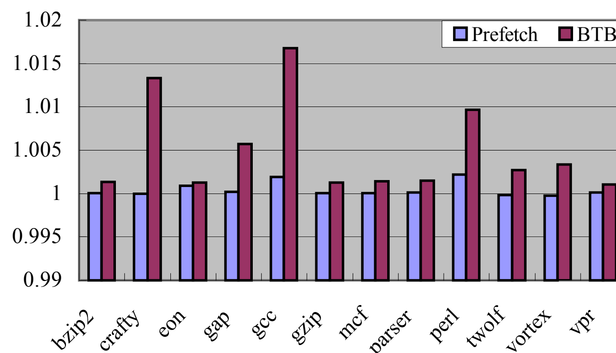


Figure 12. Prefetching and Customized BTB Optimization.

7. Conclusion

In this paper, we propose a hardware-based approach to verify the control flow of target applications dynamically and to detect anomalous executions. With hardware support, our approach offers multiple advantages over software based solutions including near zero performance degradation, much stronger detection capability and zero-latency reaction upon anomaly thus much better security.

The performance degradation is extremely low (worst case - 2.7% for gcc) mainly due to the efficient design of our staged checking pipeline (only 12 % instructions reach second stage and 2% reach the third stage). A second reason for low degradation is checking for intrusion is done in parallel with the instruction execution causing almost no extra overheads. The guarantees offered and low degradation make this scheme a practical approach to detect intrusion in hardware. This also allows intrusion prevention to be done which is not possible otherwise due to a considerable lag between the start and detection of the intrusion.

REFERENCES

[1] Allen Householder, Kevin Houle, and Chad Dougherty “Computer Attack Trends Challenge Internet Security”, *IEEE security and Privacy*, Apr. 2002.

[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, “A Sense of Self for Unix Processes,” *In Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[3] D. Wagner, D. Dean, “Intrusion Detection via Static Analysis,” *In Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[4] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” *In Proceedings of the 2001 IEEE Symposium on Security and Privacy,* 2001.

[5] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, Weibo Gong, “Anomaly Detection Using Call Stack Information,” *IEEE Symposium on Security and Privacy*, May, 2003.

[6] A.Kosoresow, S.Hofmeyr, “Intrusion Detection via System Call Traces,” *IEEE Software*, vol. 14, pp. 24-42, 1997.

[7] C. Michael, A. Ghosh, “Using Finite Automate to Mine Execution Data for Intrusion Detection: A preliminary Report,” *Lecture Notes in Computer Science (1907)*, RAID 2000.

[8] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” *ASPLOSIX,* Nov. 2000.

[9] J.Yang, Y.Zhang, L.Gao, “Fast Secure Processor for Inhibiting Software Piracy and Tampering,” *In Proc. 36th International Symposium on Microarchitecture,* Dec. 2003.

[10] E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", *Proceedings of the 36th International Symposium on Microarchitecture (MICRO)*, December 2003.

[11] B.Gassend, G.E.Suh, D.Clarke, M.v.Dijk, S.Devadas, “Caches and Hash Trees for Efficient Memory Integrity Verification”, *The 9th International Symposium on High Performance Computer Architecture (HPCA9),*Feb. 2003.

[12] David Lie, Chandramohan Thekkath and Mark Horowitz. “Implementing an Untrusted Operating System on Trusted Hardware”, *In Proceedings of the 2003 Symposium on Operating Systems Principles (SOSP03).* October, 2003.

[13] D. Wagner, P. Soto, “Mimicry Attacks on Host-Based Intrusion Detection Systems,” *ACM Conference on Communications Security*, 2002.

[14] H. S. Javitz, A. Valdes, “The SRI IDES Statistical Anomaly Detector,” *In Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1991.

[15] Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. C. Ko, M. Ruschitzka, and K. Levitt. In Proceedings of the 1997 IEEE Symposium on Security and Privacy. 1997.

[16] Doug Burger and Todd M. Austin. “The SimpleScalar Tool Set Version 2.0,” Technical Report 1342, University of Wisconsin--Madison, May 1997.

[17] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, “Automatically Characterizing Large Scale Program Behavior,” *ASPLOS 2002*, October 2002.

[18] Intel Corporation. Intel IA-64 Architecture Software Developer’s *Manual*. Santa Clara, CA, 2002.