Repairing Return Address Stack for Buffer Overflow Protection

Yong-Joon Park
Department of Electrical and Computer Engineering
University of Illinois at Chicago
Tel: 1-312-413-3140
ypark3 @uic.edu

Gyungho Lee
Department of Electrical and Computer Engineering
University of Illinois at Chicago
Tel: 1-312-413-9657
ghlee @uic.edu

ABSTRACT

Although many defense mechanisms against buffer overflow attacks have been proposed, buffer overflow vulnerability in software is still one of the most prevalent vulnerabilities exploited. This paper proposes a micro-architecture based defense mechanism against buffer overflow attacks. As buffer overflow attack leads to a compromised return address, our approach is to provide a software transparent micro-architectural support for return address integrity checking. By keeping an uncompromised copy of the return address separate from the activation record in run-time stack, the return address compromised by a buffer overflow attack can be detected at run time. Since extra copies of return addresses are already found in the return address stack (RAS) for return address prediction in most high-performance microprocessors, this paper considers augmenting the RAS in speculative superscalar processors for return address integrity checking. The new mechanism provides 100% accurate return address prediction as well as integrity checking for return addresses. Hence, it enhances system performance in addition to preventing a buffer overflow attack.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and information System**]: Security and protection – invasive software, unauthorized access

General Terms

Security

Keywords

computer security, intrusion tolerance, buffer overflow, computer architecture.

1. INTRODUCTION

Buffer overflow vulnerabilities constitute a significant portion of overall attacks at present. By overflowing a buffer near a return address at run-time stack, an attacker can alter the control flow of a program, which may activate the victim system into privileged mode and execute an arbitrary code on the victim's system. This vulnerability has been exploited by several notorious worms such as Morris worm [13], Code Red worm [8], W32/Blaster worm [12] and others. Since the Morris worm incident, buffer overflow attack problems have been one of the most critical security issues and have been studied extensively. However, these vulnerabilities are still the most prevalent type of security problem. According to ICAT statistics [18] for March 2003, seven of the ten most popular vulnerabilities are buffer overflow vulnerabilities. A similar survey [statistic] is available from CERT. Figure 1 shows the total number of advisories and the number of advisories related to buffer overflow: buffer overflows were 54.1% in the year 2002 and 74.1% in the third quarter of 2003 of the total advisories.

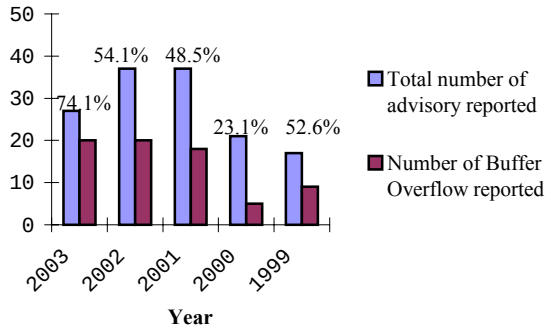


Figure 1. CERT Advisories Report (Oct. 16, 2003)

Buffer overflow vulnerability exploitations are still one of the major security issues, although numerous defense mechanisms were introduced and security patches have been released. There are multiple reasons for this. One reason is "afterward-patch". Programs are still written and distributed with the vulnerabilities, and patch can be made only after the vulnerabilities are exploited. Recently, the W32/Blaster worm [12] exploited buffer overflow vulnerability in Microsoft Windows RPC implementation. Another reason for exploitation may be program source code accessibility. Most software solutions [5, 6, 24] are compiler-driven patches, meaning they require source code changes and recompilation. It is difficult and sometimes even impossible to obtain source code for commercial software. Another reason can be the number of programs that need to be patched. There are

CF'04 April 14-16, 2004, Ischia, Italy.

numerous existing copies of legacy software that require patching, making it impossible to patch all existing vulnerable programs.

Our approach is to provide software transparent micro-architecture support for return address integrity checking. By keeping an uncompromised copy of return address, a compromised return address at run-time stack can be detected. Since extra copies of return addresses are already found in *return address stack* (RAS) in high-performance microprocessors with return address prediction, this paper considers augmenting the RAS in modern speculative superscalar processors with return address prediction. A simple RAS management scheme used in current processors cannot assure the correctness of return address integrity checking because the contents of the RAS can be corrupted even without any attack. For the return address integrity checking against buffer overflow attack, the RAS should always be able to provide uncompromised correct return address. However, the contents of the RAS can be corrupted due to its limited size in practice and its speculative execution. This paper proposes a new RAS management mechanism to guarantee uncorrupted RAS contents so that it can be used for return address integrity checking.

The RAS employed for return address prediction is fairly limited in size because most programs have relatively a small call-depth. For example, the Alpha 21264 processor has 32 RAS entries [10]. The P6 processor has 16 RAS entries [19]. Yet, deeply nested calls or recursive functions in some programs are able to cause an over-run corrupting the RAS since the size of the RAS is limited. Therefore the size of the RAS must be large enough to accommodate an arbitrary call-depth. By spilling return addresses in the RAS to a reserved memory area, the illusion of an infinite-size RAS can be created. However, another vulnerability can be generated from using memory: an adversary may be able to tamper the spill area in memory. To check the integrity of RAS spill area in memory, we uses a memory authentication scheme using collision-resistant hash trees [22], similar to the one proposed by Blaise et al [15].

In speculative superscalar processors with return address prediction, calls and return instructions speculatively update the RAS based on the prediction at instruction fetch stage. The RAS is corrupted by the speculative update if the prediction turns out to be wrong. Instead of updating the RAS at the fetch stage, our new RAS management mechanism updates the RAS at the instruction commit stage. In order to update the RAS at the instruction commit stage and provide return address prediction value during instruction fetch for speculative execution, the new management scheme uses *shadow state registers* (SSR) to provide a return address prediction value. One can also say that the SSR is kind of a "reorder buffer" (ROB) for the RAS. After a return or a call instruction is committed, the RAS is updated from the SSR as the register file in out-of-order processors is updated from the ROB at instruction commit stage. In this way, RAS corruption from a mis-speculated return or from call instructions can be prevented with a small overhead. This overhead can be negligible and offset by performance enhancement from reduced return address mis-prediction.

Non-local control transfer is another concerned issue of the RAS. For instance, the language C has system functions called setjmp(), and longjmp(). Since these instructions can bypass multiple stack frames without maintaining the RAS, they cause a misaligned stack frame. If we assume that only unmatched call/return sequence can corrupt RAS, non-local control transfer problem can be solved by pushing the target address into RAS and popping RAS until matched return address is found. If the matched return address is not found in the entire RAS including the spilled area, one can assume that undesired modification of the return address is found. Context switch also can cause misaligned RAS. Extending the spill/fill mechanism can solve the problem caused by a context switch: when a context switch occurs, entries of the switched context are spilled and entries of the switching context are filled. Here, you should explain the problem of the context switch before you talk about the solution (the spill/fill mechanism).

The remainder of this paper is organized as follows. Section 2 studies buffer overflow attack. Section 3 describes the new RAS management scheme. The simulation result and evaluation are in Section 4. Section 5 summarizes related works. The discussion is in section 6 and section 7 concludes the paper

2. BUFFER OVERFLOW ATTACKS

Buffer Overflow Attack is the most common attack to gain control of a victim system both locally and remotely. To control the victim system, an attacker has to gain sufficient privilege. However, an attacker does not have the privilege to control a victim system in most cases. Therefore, an attacker subverts the function of a privileged program and injects its attack code to be executed with the privilege. To achieve the attack, an attacker typically follows three steps:

1. Inject attack code or find suitable existing code for attack.
2. Change the control flow of the privileged program so that the attack code can be executed with sufficient privilege.
3. Execute desired code.

In order to achieve these steps, the following conditions should be met: the attacker must be able to change the control flow of the privileged program so that the attacker can compromise the victim system and the attack code should be placed in an executable area or already exist in the code area. One popular attack is known as stack smashing. A stack contains parameters of called functions and return address. A stack smashing attack fills up the stack area and modifies the return address to an attacker's desired location. In this attack method, an adversary can achieve first two steps easily. An adversary fills the stack area with the desired attack code and replace return the address with the location of the attack code.

2.1. Buffer overflow error

A buffer overflow is the result of streaming a large amount of unexpected data into a buffer. The problem arises because while a stack grows down, a buffer grows up in the run time stack area. So if a buffer overflow is generated it overwrites the old function pointer (FP) and return address. Figure 2 shows a typical buffer overflow coding error.

This code generates a segmentation violation error. The function *func()* copies a supplied string without bounds checking by using *strcpy()*. Simple *strcpy()* copies the contents of **str (string)* into *buffer[]* until a null character is found in the string. Notice the buffer size is much smaller than the string size; hence a buffer overflow is generated. After filling the buffer with 'A', the code will overwrite the old FP and return address and even **str* with 'A.' Since the ASCII value for 'A' is 0x41, the overwritten value for the return address is 0x41414141. This is

outside of the process address space, causing the segmentation violation. This vulnerability results in critical security problems. Since stack area is executable area, arbitrary code can be executed in the stack. By injecting an attacker's desired attack code instead of 'A' and modifying the return address to point to the attack code, an attacker can gain control of a victim system.

```
void func(char *str){
     char buffer[16];
    strcpy (buffer,str);}
void main(){
    char string[256];
    int i;
     for ( i=0; i< 255; i ++)
         string [i]= 'A';
    func(string);}
```

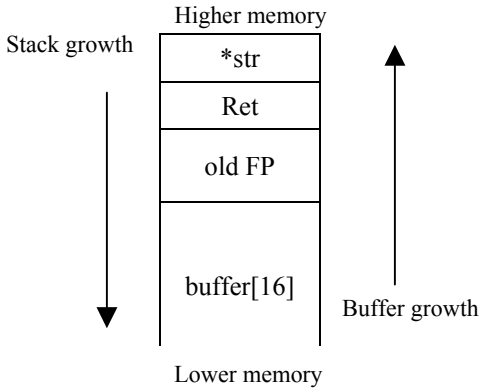


Figure 2. Typical buffer overflow error

2.2. Attack code (payload)

Attack code can be a hacker's own program or system library routine [1, 3, 19]. Figure 3 shows the example code to spawn a shell in a C program in Linux system. This code is small and very simple, but it provides a powerful shell for hackers. The hackers can via a buffer overflow put the shell code in the buffer area as a payload. From that they can gain complete control of the victim system. Another attack code can be a system library routine.

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

```
"\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";
```

Figure 3. An example C code for spawning a shell and its binary code

It is also possible to use existing library routines as malicious code [3]. There is a way to discover the address of useful library routines that can be used to download the executable program of the attacker's choice. The routine saves executable code as a file and automatically executes the downloaded program from Internet. In this case, buffer overflow attack facilitates downloading a virus or malicious code without user's consent.

W32/Blaster worm [12] exploits the buffer overflow vulnerability in Microsoft RPC implementation to generate Denial of Service attack on Microsoft windows update server. The worm scans a vulnerable system and generates buffer overflow attack on the vulnerable system to download *msblast.exe* file from the compromising host. After downloading the file, it executes the file to propagate to other system.

3. MICRO-ARCHITECTURAL DEFENSE MECHASIM (RAS MANAGEMENT)

3.1.RAS size and Overflow/Underflow control

Since the RAS has fixed number of entries, more calls than just the RAS entries can corrupt the RAS, which is a circular buffer. Thereafter, corresponding returns cause underflow by popping empty stack. In order to avoid return address corruption resulting from capacity limitation, the RAS is spilled and filled in case of stack overflow and underflow. To monitor the number of entries in the stack, a bottom of stack (BOS) pointer is introduced. Hence the number of stack entries can be calculated from the difference between the top of stack (TOS) and the BOS. When the number of entries in the RAS exceeds a certain amount, a portion of the RAS is spilled into backup storage. A chunk of return address from the RAS is spilled from the BOS into memory and then the BOS pointer is adjusted to point to the bottom of the RAS. Note that the RAS is a circular buffer. When the number of contents is below a given threshold, the spilled return addresses are loaded back from backup storage. A chunk is filled from backup storage to RAS and then the BOS pointer is adjusted. Since only a limited number of portions is spilled and filled, the operation can be paralleled with other instructions. Therefore, the overhead from accessing memory to perform a spill and fill operation can be minimized.

The number of spill and fill operation is affected by the size of RAS and the size of a chunk. Since a deeper RAS can reduce the frequency of spill and fill operation, a deeper RAS is desirable. Most programs in SPEC2000 benchmark show the maximum call-depth less than 64.

The size of a chunk can also affect the frequency of a spill and fill operation. In other words, the sequence of call and return instructions can affect the frequency of spill and fill operation. For instance, after spill operation, when consecutive calls are issued more than the size of a chunk, another spill operation will occur. Hence, smaller chunk size would cause frequent spill and fill operations. However, if the size of the chunk were improperly large, it would cause an immediate spill/fill operation after the fill/spill operation.

3.2.Backup storage protection

Although RAS spill and fill procedures are not visible in a program, an adversary could access backup storage and change data since backup storage is in a memory area. Therefore, the reserved memory area should be protected from any unauthorized (try using words other than malicious) access. In our RAS management scheme, we consider two different approaches. One

approach is to utilize virtual memory protection at an OS level. Most microprocessors support virtual memory protection at the page level. The size of a page is fixed at 4 KB for the IA-32 processor and from 8KB up to 64KB for the Alpha 21264. In a 4 KB-page, 1024 32-bit return addresses can be stored. If the number of spilled return address is greater than 1024, another 4 KB-page can be reserved. In this way, unprivileged backup storage access can be prevented. However, if the number of spilled return addresses is much smaller than 1024, 4 KB of memory is wasted, which is considered to be very small overhead in our current system. If we assume that there is no physical attack, the virtual memory protection can preserve backup storage from malicious access in a trusted OS.

Another approach is to validate backup storage using memory authentication. Blaise et al implemented an integrated Merkle tree / caching scheme to efficiently authenticate all or part of the memory area [15]. A Merkle tree [22] is a hash construct that verifies the integrity of a data object. Each leaf is a data object and each node in a tree is hash value for concatenation of children nodes. Assuming that the root value is retrieved from a trusted source, the integrity of each data object can be verified with a small amount of hash data. Figure 4 shows the layout of a Merkle tree where *H* is a collision intractable hash function and symbol '+' denotes concatenation.

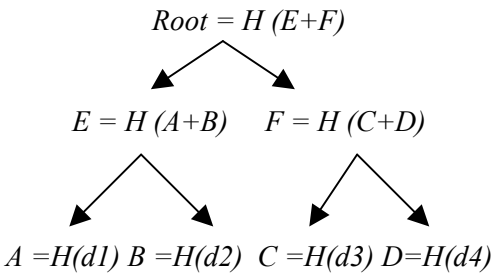


Figure 4. Merkle Tree

By checking the hash value with the value stored in its parents and by repeating verification until a value in a trusted source is found, we can verify the integrity of each node. For instance, when only "*Root"* hash value is in secure place, in order to verify the integrity of *d1*, calculations of hash values *A, B*, *C, D, E* and *F* are required. However if *E* is also in trusted source, the integrity of *d1* can be verified with only *B* and *E*. In practice, any one-way hash function such as MD2, MD5 [26], or SHA can be used as a collision-resistant function. In order to reduce the number of hash check, k-ary trees can be used. For a balanced tree, the number of hash checks to perform is $\log_k(N)$, where N is the amount of memory protected and k is the number of children nodes. The number of hash checks is same as the average depth of a tree.

Hash-tree based memory verification computes and checks a hash for every read from memory and it should compute and store a hash on write-back memory for a large amount of memory area in a secure processor such as XOM [14, 21]. If the memory authentication were implemented on normal processor, any memory write-back, which includes unauthorized memory write, would update hash value and memory. Hence it cannot detect unauthorized data modification. However, although our approach is based on a normal processor, this type of memory authentication can be applied to ensure the integrity of backup storage, because the memory authentication only verifies the backup storage, where is only accessed by the RAS, and the hash value is only updated by the RAS spill operation. In other words, RAS spill operation is trusted because it is transparent for software and independent from other running programs.

Figure 5 shows the high level schematic for backup storage authentication. A new hardware RAS Engine (RASE) manages the RAS. The RASE monitors the RAS, and if the number of RAS entries reaches a threshold, the RASE spills and fills the RAS into and from backup storage. During the spill and fill operation, a hash unit generates hash value for a spilled or filled chunk to check the integrity of the data. During spill operation, the hash value is stored into private memory area. When a chunk of return addresses is filled from reserved area in main memory, the hash unit calculates hash value and checks the integrity of the data. This authentication process can be performed simultaneously with other work. When a fill occurs, the spilled return address will be passed into the RAS from backup storage while the hashing unit checks the integrity of data by checking hash value in private memory and calculated hash value. Later, when the hashing unit detects unauthorized modification of data, it raises an exception to halt the program.

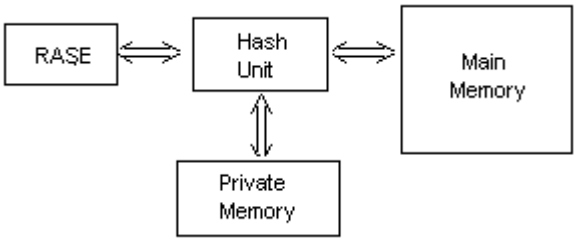


Figure 5. A high level schematic for backup storage authentication

To evaluate the cost of our new RAS management scheme we consider MD5 [26] algorithm. MD5 takes 512-bit block and generates 128-bit digest. Hence, 4K of hash values can be stored in 64KB private memory. When the hash value exceeds the available storage, two of the oldest hash values are concatenated and new hash value is generated. For instance, let's assume that private memory can have 3 hash values. When, in Figure 3, hash value *D* is stored after has value *A, B*, and *C*, parent hash value *E* is generated from the concatenation of hash value *A* and *B* to replace *A* and *B* with *D* and *E*. After verifying the corresponding chunk of hash value *D*, oldest two chunks are read and hash values A and B are generated to check the integrity of two chunks with hash value *E*. Upon successful verification, hash values of *D* and *E* are replaced with *A* and *B*.

3.3.RAS update and speculation.

There are two basic approaches to avoid RAS corruption resulting from a mis-speculated RAS update. One way is to preserve the history of the RAS, similar to Smith and Pleszkun's history file [14], and recover the RAS from a mis-speculated update. Skadron et al [23] has shown the repair mechanism to be very accurate, thereby improving performance relative to a stack

with no repair mechanism. The repair mechanism functions by storing the top of the stack pointer (TOSP) and the top of the stack pointer contents (TOSC) to shadow state registers and the RAS when a call or return instruction is fetched. After detecting a mis-speculation during the commit stage, the TOSP and the TOSC are restored to the RAS. Another approach to protect the RAS from a mis-speculated update is to keep uncommitted return or call state information in shadow state registers and update the RAS from it after the instruction is committed, similar to the future file. In this scheme, only the call instruction stores the TOSP and the TOSC into shadow state registers, while the return instruction stores only the TOSP into shadow state registers. When mis-speculation is detected, the shadow status registers are simply flushed. In contrast to the first approach of the RAS repair mechanism, the second approach saves newer state information to the shadow state register. Since the RAS repair mechanism updates the RAS speculatively in the instruction fetch stage, it updates the RAS more frequently than the second approach. How often the RAS is updated affects the frequency of the spilling/filling of the RAS contents to and from the reserved memory area. Hence, we chose the second approach.

RASE also manages the shadow status registers (SSR). If a return or call instruction is determined to be mis-speculated, it flushes the SSR. Figure 6 shows a high level schematic of our return address stack mechanism. There are two pointers – HEAD and NEXT for the SSR –, which behave similarly to the reorder buffer. The NEXT pointer points to the next available slot in the SSR. The HEAD pointer points to the SSR slot that contains the information of the RAS for the oldest return or call instruction not committed yet. The shadow register has two entries for each slot: the top of stack pointer (TOSP) and the top of stack contents (TOSC). When a call instruction is fetched, it updates the SSR with increased the TOSP and a return address as the TOSC. When return instruction is fetched, the SSR is updated with decreased the TOSP and invalidated the TOSC. It searches a return address from the NEXT pointer to provide a return address prediction value. When a call or return instruction is committed, the RAS is updated from the TOSP and the TOSC pointed to by the HEAD pointer. If the TOSC is invalid, only the TOSP updates the TOS in the RAS since the entry corresponds to a return instruction. The size of the SSR depends on the number of in-flight call and return instruction. For example, a MIPS R10000 processor supports up to four in-flight call and return instructions. Similar requirement apply based on pipeline depth between the instruction fetch and commit stages. For deeper pipelined processor, the depth of speculation will dictate the SSR size. When the SSR is full, the next instruction may overwrite the oldest instruction entries since the last instruction, which was pointed by the HEAD pointer, is already committed. The summarized steps are:

1. For the case where the NEXT and the HEAD pointers point to the same location, the TOSP points to the top of the RAS as the TOS does
a. When a return instruction is fetched, the value pointed by TOS in the RAS is the prediction value. TOSP becomes TOSP-1 and is stored into next available SSR slot (pointed by the NEXT pointer). The TOSC field is marked as invalid and the NEXT pointer is incremented by one.
b. When a call instruction is fetched, the return address (PC+ instruction size) is copied into the next available TOSC field. Increased TOSP is stored into the TOSP field in the SSR slot (pointed by the NEXT pointer) and then the NEXT pointer is incremented by one.

2. For the case where a call or return instruction is already fetched but not committed yet
a. When a return instruction is fetched, the previous slot of the slot pointed to by the NEXT pointer is referred to (this is confusing). If the TOSC is valid, the TOSC is a prediction value. If the TOSC is invalid, the content of the RAS pointed by the TOSP becomes a prediction value. A decremented TOSP is stored into the slot pointed to by the NEXT pointer and the NEXT pointer is incremented.
b. When a call instruction is fetched, the return address is copied into the TOSC and the incremented TOSP is stored in the slot pointed to by the NEXT, and the NEXT is incremented by one.

3. When a call or return instruction commits
a. When a call or return instruction commits, the RAS is updated from the shadow state registers pointed to by the HEAD pointer (i.e. when a call instruction is committed, the TOSC is pushed into the RAS and the top of the stack (TOS) is increased. When a return instruction is committed, the TOS is popped.), then the HEAD pointer is increased by one. If there is a fetched call or return instruction, step 2 is performed.
b. When an instruction is determined to be mis-speculated, the shadow state register is flushed and HEAD and NEXT pointers remain pointed to the same SSR slot. TOS becomes TOSP and the return address pointed to by the TOS in the RAS becomes the TOSC.

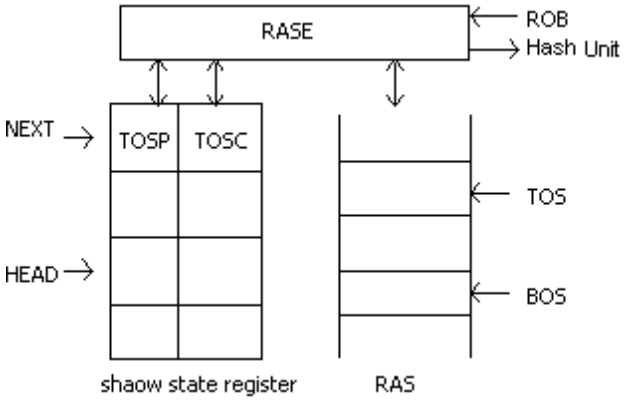


Figure 6. A high level schematic of RAS management

3.4 Non-local control transfer and context switch

Non-local control transfer is another issue of concern for the RAS. For instance, the language C has system functions called *setjmp()* and *longjmp()*. Since these instructions can bypass multiple stack frames without maintaining the RAS, they cause a misaligned stack frame. If we assume that only an unmatched call/return sequence can corrupt the RAS, the non-local control transfer problem can be solved by pushing the target address into the RAS and popping the RAS until a matched return address is found. If a matched return address is not found in the entire RAS

including the spilled area, we can assume that undesired modification of the return address is found. Context switch also can cause misaligned RAS. Extending the spill/fill mechanism can solve the problem caused by context switch: when context switch occurs, entries of the switched context are spilled and entries of the switching context are filled.

4. SIMULATION

We used the SimpleScalar tool set for the results reported in this section. Table 1 summarizes our baseline model, which is loosely modeled after the Alpha 21264. Since return address prediction can be done in fetch stage as BTB lookup, performance overhead is only added by the hash latency. It is established that the latency can be reduced to average 80 cycles with suitable skewing of the adders [15].

For an experiment in this section, three SPEC2000 CPU benchmarks are used: gap, mcf, and parser. Since other benchmarks have the maximum call-depth less than 32, their performance would be enhanced by more than 32 RAS entries. Since the sequence of call and return instruction significantly varies the simulation result, each benchmark was simulated from the beginning to the end. The simulation result only shows the normal situation for performance analysis; once the buffer overflow attack has occurred, the process cannot be recovered or performed any further.

Table 1 Baseline configurations for simulations

Architectural parameters	Value
Instruction-window size	64
Fetch/ Decode width	4/4 per cycle
Issue/ Commit width	4/4 per cycle
RAS	32
BTB	2048-entry, 2-way
L1 I-caches	64KB, 2-way, 32B line
L1 D-caches	64KB, 2-way, 32B line
L2 cache	Unified, 4MB, 4-way (LRU), 32B line
L1 latency	1 cycle
L2 latency	12 cycles
Memory latency	80 cycles
Hash latency	80 cycles
Chunk size	8

Figure 7 shows the result of the new RAS management scheme. It compares the overall performance, expressed in instruction retired per cycle (IPC), for two different schemes. The BASIC scheme uses the RAS repair mechanism that saves return address stack index and restores the saved index to the TOS at commit stage when mis-speculation is detected. The mechanism is provided by SimpleScalar tool set as a default configuration. However, the scheme does not provide perfect return address prediction. "RAS with spill/fill overhead" uses the message authentication scheme to protect the backup storage. In this scheme, the spill/fill overhead occurs from memory access latency and hash latency. For spill operation, the overhead only accounts memory access latency since the hash operation can be performed with the memory writing operation simultaneously. However, the fill operation overhead accounts both memory latency and hash latency since the hash operation can begin only after the memory read operation is finished. Table 2 summarizes the results with reference inputs in SPEC2000 benchmarks. The RAS has 32 entries and the size of the chunk is 8. It shows the number of executed instructions for each benchmark. Since the new RAS management scheme achieves 100 % prediction success rate, it only shows the RAS prediction success rate for BASIC scheme. The third row shows the total number of RAS spill operations for each benchmark. "Max spilled chunk," indicates the maximum number of chunks stored in backup storage at a given moment.

Table 2 Summary of result

	Mcf	parser	gap
Number of instruction	9168131541	13433257594	9591840728
RAS hit (BASIC)	98.9 %	94.13%	71.96 %
# of spill operation	25	456608	133034
Max spilled chunks	2	35	261

The BASIC scheme achieves a RAS hit rate of 98.9% for mcf, 94.13% for parser, and 71.96% for gap. Therefore even 100% RAS hit rate the performance improvement is not significant. However the latency of the spill/fill operation can be offset by the performance improvement from an accurate return address prediction. Figure 6 shows the IPC comparison between the BASIC and the new RAS management scheme.

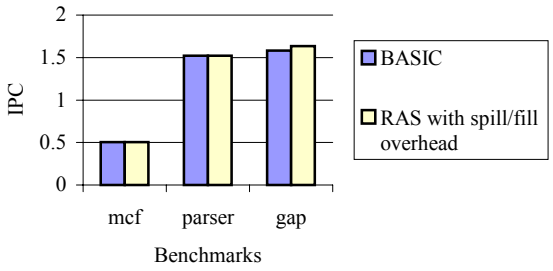
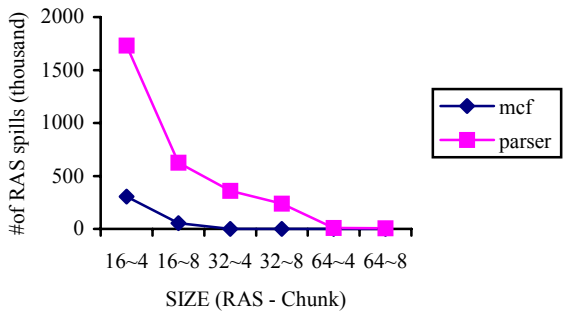


Figure7. Performance impact of new RAS management scheme

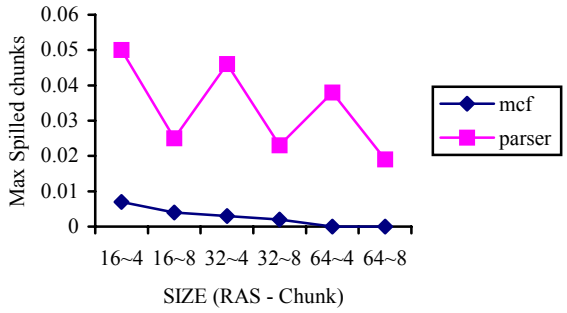
Deeper RAS and larger chunk size reduces the frequency of a spill/fill operation; therefore, the performance will be improved. In contrast, a more shallow RAS depth and a smaller chunk size will result in performance degradation. Figure 8 shows the result of simulation for train input with various sizes of RAS and chunk size. For instance, a 32-entry of RAS with 8-entry of chunk improves the overall performance by 2% compared to the performance of a 16-entry of RAS with a 4-entry of chunk for parser.

	mcf	parser
# of instruction	7136667402	7908917776
Max call-depth	37	208
#of RAS used	1201289	130794089

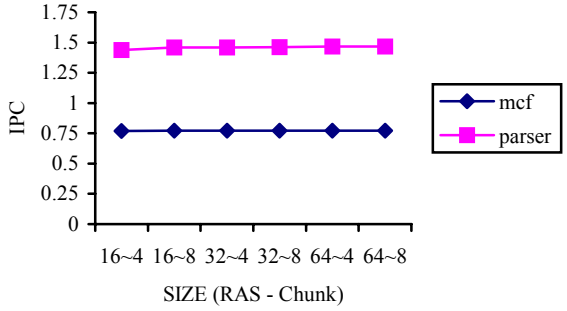
(a)



(b)



(c)



(d)

Figure 8. Simulation result for various sizes of RAS and chunk (a) Common result for various sizes of RAS and chunk (b) The number of RAS spills operation for various sizes of RAS and chunk. (c) Max spilled chunks for various sizes of RAS and chunk (d) IPC for various sizes of RAS and chunk

5. RELATED WORKS

There are numerous software defense mechanisms that can prevent achievement of one of the attack steps. One mechanism is to prevent buffer overflow so that it can, in turn, prevent a buffer overflow attack. Jones and Kelly's work [16] can prevent the injection of a payload and the modification of a return address by checking for array and pointer bounds in C program. Libsafe also can prevent the injection of a payload and the modification of a return address [2]. Libsafe intercepts all calls to library functions that are known to be vulnerable and substitute them with a safe version of corresponding functions. Another defense mechanism is to allow buffer overflow but preventing that an attacker changes the control flow of a victim system. PC encoding mechanism [24, 25] encrypts and decrypts return addresses with semantic encryption to prevent changing control flow from any unexpected modification of the return address. When a function is called, the return address is encrypted before being stored into an activation record. Similarly, at a function return, the target address is decrypted and its integrity is checked. Without knowing the encryption key, an attacker cannot obtain control of a victim system even if the attacker can overwrite contents of the activation record. StackGuard is a dynamic buffer overflow attack detection mechanism [6, 7]. StackGuard implements a canary word as a detection sensor on top of the return address in the activation record. The mechanism assumes that the canary word is overwritten when a buffer overflow occurs and overwrites the return address. However, it is possible to change return address without altering the canary value [3]. An attacker can bypass StackGuard and StackShield by using a vulnerable function pointer [20] and overwriting the base pointer value in the activation record. RAD [5] is another dynamic detection mechanism to prevent buffer overflow attack. RAD stores return address into safe memory area where when function is called. By comparing the copied return address and the user stack return address, it can detect the buffer overflow attack. Another mechanism prevents executing codes in stack area. The solar designer's non-executable user stack [9] allows the overwriting of stack contents, but malicious code in the user stack still cannot be executed.

6. DISCUSSION

Our new RAS management scheme can only protect stack-based buffer overflow attack. However heap area is also vulnerable to buffer overflow attack. The heap based buffer overflow attack mechanism is similar to the stack based buffer overflow; it overflows a vulnerable buffer which is near the function pointer value in the heap area and changes the function pointer value to redirect the control flow to compromise the victim system. Since the heap based buffer overflow attack exploits the vulnerable function pointer in the heap area, the new RAS management scheme cannot prevent the attack. Hence, our directions for future research are to build precise test bed for system call and extend the mechanism to check the integrities of indirect branches.

7. CONCLUSION

In this paper, we have described the implementation of the new RAS management scheme that prevents buffer overflow attack. The new RAS management scheme provides perfect return address prediction; as a result, the integrity of the return addresses can be checked to detect buffer overflow attacks. Although the new mechanism introduces overheads generated by spill/fill operations, it can be offset by the performance improvement from accurate return address prediction with security enhancement.

8. ACKNOWLEDGMENTS

This work is supported by the NSF grant CCR-0242222.

9. REFERENCES

[1] Aleph One. Smashing the stack for fun and profit, *Phrack Magazine, 7(49): File 14*,1996

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. *Proceedings of the USNIX Annual Technical Conference*, June 2000.

[3] Bulba and Kil3r. Bypassing StackGuard & Stackshield. *Pharck magazine vol. 11 Issue 56*

[4] P.Y. Chang, E. Hao, and Y.N. Patt. Alternative implementations of hybrid branch predictors. *Proceeding of Micro-28*, page 252-257, Dec. 1995

[5] Tzi-Cker Chiveh and Fu-Hau Hsu. RAD: A compile-time solution to Buffer Overflow Attacks. *Proceeding of 21st International conference on Distributed Computing system*, 2001

[6] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bake, Steve Beattie, Aron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Detection and prevention of Buffer-Overflow Attacks. *Proceeding of the* 7th *USENIX security symposium*, 1998

[7] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and defense for the vulnerability of the Decade. *DARPA Information survivability Conference and Expo DISCEX*, 1999

[8] Roman Danyliw and Allen Householder. CERT Advisory CA-2001-19: Code Red Worm Exploiting Buffer Overflow IN IIS Indexing Service DLL. *http://www.cert.org/advisories/CA-2001-19.html*, Jul. 2001

[9] Solar Designer. Non-Executable user stack. *http://www.openwall.com/*

[10] Compaq Computer Corporation. Alpha 21264/EV6 Microprocessor Hard-ware Reference Manual. Sept. 2000.

[11]DilDog. The Tao of Windows Buffer Overflow. *http://www.cultdeadcow.com/cDc_files/cDc-351/*

[12] Chad Dougherty, Jeffrey Havrilla, Shawn Hernan, and Marty Lindner. CERT Advisory CA-2003-20 W32/Blaster worm. *http://www.cert.org/advisories/CA-2003-20.html*

[13] Mark W. Eichin and Jon A.Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. *Proceeding of the IEEE Symposium on Research in Security and Privacy*, 1989

[14] J. E. Smith, and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Trans on Computer* 37:5,1988

[15] Blaise Gassend, G. Edward suh, Dwain Clarke Marten Van Dijk, Srivas Devadas. Cache and Merkle trees for efficient Memory Authentication. *Proceedings of the 9th High Performance Computer Architecture Symposium,* February 2003.

[16] R.W.M. Jones and P.H.J. Kelly. Backward-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the 3rd International Workshop on Automated Debugging*, 1997

[17] J. L Hennesy, D. A. Patterson. Computer Architecture A quantitative approach. Morgan Kaufman publisher Inc. 1996

[18] ICAT Metabase A CVE Based Vulnerability Database, http://www.icat.nist.gov/icat.cfm

[19] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual. 2003

[20] Klog. Frame pointer overwrite. *Pharack magazine vol.9. Isuue 55*

[21] David Lie, Chandramohan Thekkath, Mark Mitchell, and Patrick Lincoln. Architectural Supports for Copy and Tamper Resistant Software. *APOLS-IX 2000 Cambridge, Massachusetts*. 2000

[22] Ralph Merkle. Protocols for public key cryptography. *IEEE Symposium on Security and privacy*. Page 122-134, 1980

[23] K. Skadron, P. S. Ahuja, M. Martonosi and D.W. Clark. Improving prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms. *Proceedings of the 31st Annual ACM/IEEE international symposium on Microarchitecture*, page 259-271, Dec. 1998

[24] A. Tyagi, and G. Lee. Encoded program counter: Self Protection from Buffer Overflow Attacks. *Proceedings of International conference on Internet Computing (IC'2000)*, June 2000

[25] C. Pyo and Gyungho Lee. Encoding Function Pointers and Memory Arrangement Checking against Buffer Overflow Attack. *Proceeding of the Fourth International Conference on Information and Communications Security* (as Lecture Notes in Computer Science Vol. 2513, Springer-verlag), Singapore, Dec. 2002.

[26] R. Rivest. RFC1321: The MD-5 message-Digest Algorithm, 1992