

A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing

Dong Ye, David R. Kaeli
Department of Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts 02115
{dye, kaeli}@ece.neu.edu

Abstract

Buffer overflow vulnerability is one of the most common security bugs existing in today's software systems. In this paper, we propose a microarchitectural design of a return address stack aiming to detect and stop stack smashing. This approach has been used in other proposals to guard against buffer overflow vulnerabilities. Our contribution is a design that handle multi-path execution, speculative execution, abnormal control flow, and extended call depth. Our solution makes no assumption about the presence of architecturally visible calls and returns.

1 Introduction

The significance of buffer overflow vulnerabilities has been widely acknowledged [2, 9]. A major form of attack that exploits this vulnerability is to overflow the buffers allocated on the stack and overwrite the return address residing at a higher address. On the function return, the PC will address the injected attack code. Upon execution of the return, execution will be hijacked to the attack code, which obtains the privilege level that the hijacked process possesses at the point of being smashed [6].

Various software-based approaches have been proposed to tackle this problem. Inspecting the

number of security alerts reported by CERT, buffer overflow related problems accounted for more than half of all the alerts reported since 1997 [2]. Among the 28 CERT advisories reported in 2003, 24 of them were due to buffer overflow vulnerabilities or exploitation of buffer overflow vulnerabilities [2].

In recent years a number of hardware-based approaches have been proposed to address stack smashing attacks [5, 7, 10]. We believe that due to the sheer frequency of the stack smashing attacks and the serious consequences resulting from these attacks, stack smashing exploits require hardware-based countermeasures.

A Reliable Return Address Stack (RRAS) is a microarchitecture-level return address stack (μ RAS) that can guard against stack smashing attacks. The μ RAS has been widely employed in microprocessors for decades to improve the control-flow prediction accuracy for function returns [8]. Because return addresses stored on the microarchitectural stack are only updated upon function calls and returns, it is impossible for memory-based attacks such as stack smashing to overwrite the return addresses on the μ RAS. However the return address that resides on top of this stack is not always correct, so we cannot count on it for detecting attacks. A new implementation of the μ RAS is required.

In this paper, we first present security expo-

sures related to using a μ RAS to provide the correct return address, followed by both a structural and algorithmic description of our new RRAS design. We also compare the RRAS with other published designs in the literature. We then conclude the paper and provide directions for future work.

2 Design Issues

To guarantee the correctness of the return address, the μ RAS has to address the following problems, all of which lead to a mismatch between the return address on the μ RAS and the address found on the program stack (even before it is smashed):

1. The return addresses on the μ RAS can be lost between context switches.
2. The μ RAS is a finite-sized table. The runtime nesting depth of function calls may exhaust this depth.
3. In multi-threaded processors, calls and returns issued by different threads can be interleaved.
4. In processors that support speculative execution, calls and returns along a speculative path may update the μ RAS, but may be squashed later.
5. Calls and returns may not be properly nested; examples include *setjmp()/longjmp()* in C and exception handling in C++ and Java.
6. Some *hacker code* may use architectural call and return instructions for purposes other than function calls and returns, which can in turn lead to two situations: i) a *call without a corresponding a return* and ii) a *return without a corresponding call*. The scenarios will cause the conventional μ RAS to fail to produce the proper return address.

To accommodate arbitrary levels of nesting depth, it is necessary to provide a reserved memory structure for a μ RAS spill. This backup memory needs to be allocated in a separate segment so that a potential buffer overflow vulnerability existing in another segment does not affect the backup memory. Given the necessity of this backup memory, the RRAS includes backup memory as part of the thread state. This mechanism also helps out with context switch management by speculating and recovering the return addresses of all the outstanding function calls. A per-thread return address stack is also necessary in order to properly handle multi-path execution [3].

A traditional μ RAS assumes that the return address associated with the last call is on the top of the stack. Most RAS mechanisms can only index the top entry on the stack, while in the reality, the correct address may be just a bit deeper on the stack. This situation can easily occur due to control speculation or call/return sequences that are not properly nested.

To address these issues, the RRAS searches the entire μ RAS to locate the correct return address, searching from the top down. The rationale for this approach is:

1. Most of time, the correct return address is on the top of the stack. If we look one entry deeper (just one entry deeper), we can greatly increase the chance of locating the correct return address [8]. The cases where the correct return address resides deep on the stack are rare and handling these cases is already very costly.
2. No special repair mechanism is needed to synchronize the microarchitectural stack with the program stack.

The RRAS makes two changes to the traditional μ RAS to enable us to locate correct return addresses reliably and efficiently:

1. Return addresses on the RRAS can only

be popped upon commitment of function returns (though return addresses can be pushed before a function call commits), and

2. Each return address on the RRAS is accompanied by the entry address of the called function.

Upon commitment of a function return, the RRAS pops all the entries above the located entry. This insures that the RRAS remains closely (though not necessarily fully) synchronized with the state of the program stack, and also facilitate future searches.

Recursion tends to generate frequent function calls and can consume many entries on the μ RAS. Although correctness of return address is still ensured by providing backup memory, it comes at the cost of expensive memory operations. Using modest record keeping, the RRAS can identify direct recursion on the fly.¹ The RRAS captures multiple executions of the same direct recursion with a single entry on the microarchitectural stack. It does not need to push multiple identical return addresses onto the stack for direct recursions. Instead, it is able to reuse the single return address entry for all of the returns associated with the direct recursion.

Both the traditional μ RAS [8] and previous enhancements to the μ RAS [5, 7, 10] depend upon two properties being present to function properly:

1. the instruction set architecture must provide explicit instructions for implementing calls and returns, and
2. any code running on the platform must al-

¹Direct recursion is the most frequently used form of recursion. Our design of the RRAS addresses direct recursions with special hardware, but leaves the handling of indirect recursions to memory backup. However, the same hardware approach can be easily extended to handle indirect recursion. The cost of this hardware increases linearly with the maximum distance of indirect recursions that can be handled without resorting to memory backup.

ways use these instructions for implementing function calls and returns.

We will refer to the execution state where these two prerequisites are present as the *restrictive mode*.

The RRAS is able to function properly, even if either one, or both, of these conditions is not met (we call this state as the *non-restrictive mode*). The RRAS is designed to identify the essential runtime behavior of a function (i.e., the return address of a function will be the instruction address after the call site address, and then dynamically redirect fetching accordingly. Since the RRAS only handles true function calls and returns (versus architectural calls and returns), the *hacker code* usage of call and return instructions is treated simply by the RRAS as normal jump instructions.

3 Implementation

Next we present the implementation logic of the RRAS.

3.1 Structure

Each return address is paired with the entry address of the called function. The entry address is actually the destination address of this call. Maintaining a pair of addresses enables the RRAS to identify function calls solely based on an address basis, and is independent of needing explicit call and return opcodes. The logic for the identification process was first described by Kaeli and Emma [4]. A special table is added to augment the RRAS. This table is called the *Address Pair Table* (APT). The APT stores the entry and exit address pairs of all *active* functions. The APT supplies the associated entry address, given the exit address, upon a function return. This entry address is all that is needed to locate the return address on the microarchitectural stack in the case of a non-recursive call. When a function is called recursively, multiple entries on the

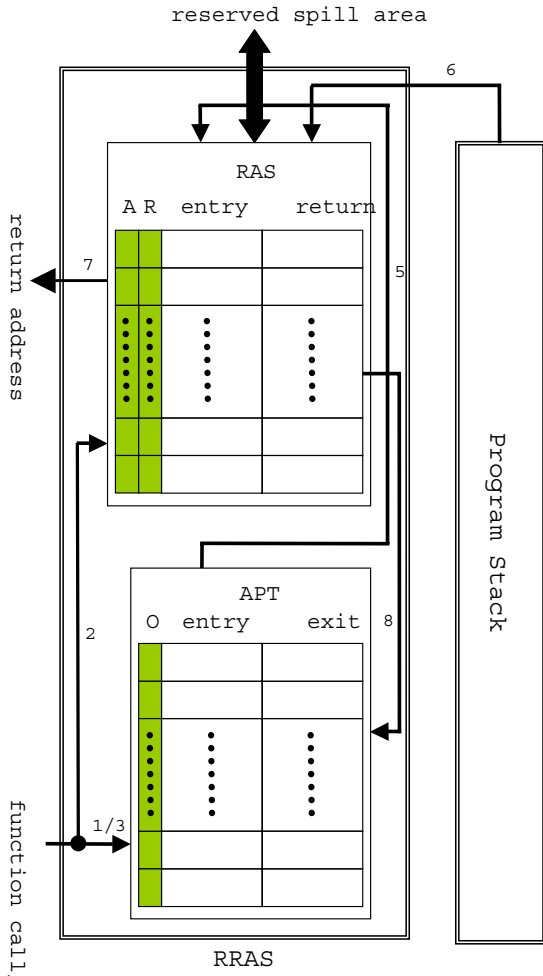


Figure 1: Organization of a Reliable Return Address Stack.

RAS will have identical entry addresses, though it may have different return addresses. Also, we need to consider the case where function calls and returns that are not nicely nested. Given these cases, we cannot guarantee the correctness of the return address using an entry on the RAS by only matching the function entry address associated with this stack entry. The APT also works with the traditional BTB to differentiate between normal jumps and jumps for implementing function calls and returns when running in an unrestricted mode.

Figure 1 shows the structure of the RRAS. We introduce 3 bit-wide tags which are associated with each on the stack. The bits A , R , and O are defined as follows:

- An entry on the RRAS with $A == 1$ indicates that there are additional entries below it that have the same entry address.
- An entry on the RRAS with $R == 1$ indicates that this entry is for a direct recursion.
- An entry in the APT with $O == 1$ indicates that there is at least one entry currently on the RAS having the same entry address. This means an entry that is going to be pushed onto the RAS with the same entry address is a recursive function call and thus the A bit of this entry should be set.

3.2 The Address Pair Table

We use the sample program shown in Figure 2 to illustrate how the APT functions. A function call/return in the RRAS is identified using the algorithm described in [4]. After being identified, the address pair for the call and return will be recorded in the APT. Some further enhancements to the RRAS needed to identify calls and returns include:

- storing information about the calling functions (i.e., entry/exit address, as well as an indication of whether an invocation of this function is still outstanding on the call path) in the APT,
- differentiating between jumps used for calls and returns versus normal jumps by interrogating the APT or the BTB, and avoiding stack updates in the latter case, and
- treating jumps as potential calls and returns if not found in either the APT or the

MAIN:		PRINT:	
<i>instr. addr.</i>	<i>instructions</i>	<i>instr. addr.</i>	<i>instructions</i>
100	CALL @PRINT	700	.
110	.	710	.
.	.	.	.
140	CALL @PRINT	.	.
150	.	.	.
.	.	.	.
.	.	800	RETURN

Figure 2: A sample program.

BTB (this is essentially the identification process).

We will use the example in Figure 2 to step the identification of a function entry/exit pair in the APT, and the process of locating the correct return address stored in the RAS.

1. When the **CALL** at 100 is executed, the target of 700 (i.e., the entry address of the subroutine **PRINT**) is checked against the APT along path 1. Suppose this is the first time that **PRINT** is called, so no associated entry has been created on the APT. Since we are not sure whether an instruction is a normal jump (we treat all architectural call/return instructions as normal jumps initially) or a jump to implement function call, a new entry is created in the RAS along path 2, with an entry address of 700, and a return address of 110. Since this is the first time the **PRINT** function has been encountered during execution, the APT will not produce a match on the entry address of 700, and the “A” and “R” bits of this entry are both clear.
2. When the **RETURN** at 800 is executed, this exit address is checked against the APT along path 1. Since no match is found on this first invocation, the return address on the program stack is fetched along path 6. The return address is used to find a match on the RAS, searching from the top down.

On a match, the associated function entry address of the matched entry (700), and the function exit address (800), will create a new record in the APT (sent along path 8). At the same time, this return address is sent along path 7 to the processor to redirect execution. Then the *R* bit of the matched entry is checked. If the *R* bit is set, all of the entries above it (but not including the matched entry) are popped from the RAS. If the *R* bit is clear, this matched entry is also popped. The check of the *R* bit is needed when we want to reuse this entry if it is associated with a direct recursion. For each of the popped entries, if its *A* bit is clear, the entry address is sent to the APT along path 8 to clear the *O* bits of all the APT records with the same entry address. The check of the *A* and *O* bits is necessary to maintain the semantics of the *O* bit of the APT.

3. When the **CALL** at 140 is executed, similar operations are performed as in step 1. In this case, we will have a match in the APT. On this match, the *O* bit of the matched record is checked:

If the *O* bit is set, a new entry with an entry address of 700, and a return address of 150 and is generated with the *A* bit set. Before we can create this entry on the RAS, we need to compare it to the top entry of the RAS.

If they match (i.e., if both the function entry and return addresses from this *to-be-pushed entry* match the corresponding fields in the top entry of the RAS), no new entry will be created on the RAS. But the *R* bit of the top entry will be set. If we do not match on both addresses, then a new entry is created on the RAS.

If the *O* bit is clear, a new entry is created, with an entry address of 700, a return address of 150 and the *A* bit cleared, on the RAS. Meanwhile, the *O* bit in the APT is set to indicate that now there is a function with an entry address of 700 that is outstanding on the call path.

4. When the **RETURN** at 800 is executed, this exit address is checked against the APT. We will find a match and its associated entry address is 700. This function entry address is used to search entries on the RAS, starting from the top of the stack, and working down. On a match, the *A* bit is further investigated:

If the *A* bit is clear, the return address associated with the matched entry on the RAS is used to redirect execution, and the *R* bit of this entry is checked further:

If the *R* bit is clear, the matched entry, and all those above it, are popped.

If the *R* bit is set, the entries above this entry, but not including the matched entry, are popped.

If *A* is set, then some entry (or entries) deeper in the stack has the same function entry address. The return address is fetched from the program stack along path 6. The entry address 700 and the return address 150 are used to

find a match in the RAS. If we find a match on both the function entry address and the function return address, then we know the return address has not been overwritten since the same copy also exists on the μ RAS and it is used to redirect execution. The matched entry's *R* bit is checked further:

If the *R* bit is clear, the matched entry, and all those above it, are popped.

If the *R* bit is set, the entries above it, but not the matched entry, are popped.

For each entry popped from the RAS, if its *A* bit is clear, its entry address is sent to the APT and the *O* bits of all the APT records that have the same function entry address are cleared.

4 Evaluation

Since the RRAS requires us to pop entries from the hardware stack only upon commitment of the function return, we should keep function calls from pushing an entry onto the μ RAS if there are any function returns that are on a speculative path. In case there are function calls issued that are on a speculative function return path, and this path is squashed, the entry pushed by the subsequent function calls would be prematurely popped. Our preliminary evaluation focuses on the performance impact of this situation.

Figure 3 shows the impact of managing a RRAS while running nine integer SPEC2000 benchmarks. We use the SimpleScalar [1] sim-outorder framework using the default configuration. In our sim-rras machine, we stall issuing a function call whenever there is a function return outstanding. Given this conservative model, the data we show in Figure 3 is a pessimistic estimate of the performance imposed by this requirement.

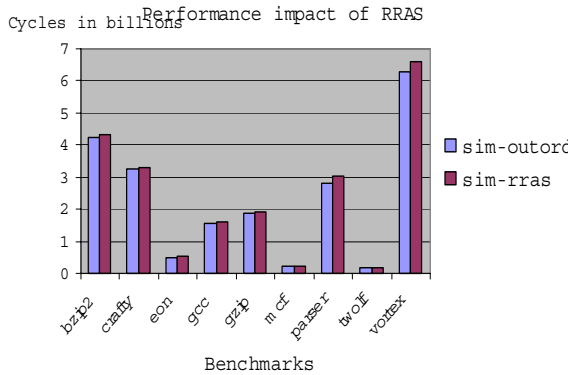


Figure 3: Performance impact of stalling function calls whenever there is an outstanding return on the RRAS.

As we can see, the penalty associated with this restriction is small for SPEC2000int.

5 Conclusion

This paper describes the design of a reliable return address stack that can both detect and recover from stack smashing attacks. In contrast to other proposed mechanisms that extend the return address stack to defend against stack smashing attacks [5, 7, 10], the contributions of our RRAS mechanism include:

- The RRAS does not require function calls/returns to use explicit call and return operations, and further, does not depend on the availability of architecturally visible call and return instructions.
- The RRAS handles irregularly nested calls and returns properly and thus does not require recompilation or binary modification.
- The RRAS does not have to maintain perfect synchronization between the μ RAS and the program stack. Thus no special repair mechanism is needed in the RRAS.
- The RRAS also provides a solution to the issue of microarchitectural stack spills/refills

due to recursive calls. This will decrease the frequency stack spills and thus mitigate one of the largest reasons for performance loss when using a return address stack for defending against stack smashes [5, 7, 10].

Future work will study the impact of stack spills and will also run a set of virus benchmarks that attempt to compromise the system. This future work should help to quantify both reliability and performance associated with the RRAS.

References

- [1] D. Burger and T. Austin. The simplescalar tool set version 2.0. Technical Report 1432, University of Wisconsin–Madison, 5 1997.
- [2] CERT CC. Cert advisory. <http://www.cert.org/adversories/>.
- [3] S. Hily and A. Sez nec. Branch prediction and simultaneous multithreading. In *Parallel Architectures and Compilation Techniques*, 1996.
- [4] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th International Symposium on Computer Architecture*, pages 34–41, 1991.
- [5] J. McGregor, D. Karig, Z. Shi, and R. B. Lee. A processor architecture defense against buffer overflow attacks. In *IEEE International Conference on Information Technology: Research and Education*, 2003.
- [6] Adelph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
- [7] Y.-J. Park and G. Lee. Repairing return address stack for buffer overflow protection. In *1st Conference on Computing Frontiers*, 2004.

- [8] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *31st International Symposium on Microarchitecture*, pages 259–271, 1998.
- [9] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.
- [10] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer. Architecture support for defending against buffer overflow attacks. In *2nd Workshop on Evaluating and Architecting Systems for Dependability*, 2002.