Preventing Return Oriented Programming Attacks By Preventing Return Instruction Pointer Overwrites

Nick Allgood <allgood1@umbc.edu> *Enis Golaszewski <golaszewski@umbc.edu>*
Ben Smith <bsmith13@umbc.edu> *Alexander Spizler <spizler1@umbc.edu>*

I. Introduction

Over the last decade, buffer overflows have been the most common form of security vulnerabilities[1] This was true in 2015 as well as the previous three decades. Stack based buffer overflows were particularly dangerous because they frequently give instant arbitrary execution on the stack. Therefore, the majority of protections created by processor and operating system manufacturers have related to preventing *execution* on the stack and not preventing the buffer overflow. While this is a reasonable step to take, because of the focus on preventing only execution, it left the door open for return oriented programming attacks.

Return oriented programming is an attack that is based on overwriting the return instruction pointer on the stack in order to gain execution of specific code fragments within the legitimate application. When the "return" instruction is called, the next instruction to be executed is popped off the stack. If that instruction is the end of a function than you can arbitrarily execution with whatever the end of the function is and then return, which again pops the next instruction off the stack. Each of these function parts is called a *gadget* [2]. By chaining together these gadgets, interspersed with parameters for the gadgets, the attacker is able to execute legitimate code to achieve malicious purposes. "Attackers can use return oriented programming to circumvent the OS protection that prevents stack portions from being executed."[2]

A. Removing Access to the Return Instruction Pointer

In the past 40 years, no one has figured out a universal solution to prevent buffer overflow vulnerabilities; therefore we focus on preventing the attacker from overwriting the return instruction pointer. Without access to the return instruction pointer, the attacker cannot execute the gadgets which form the core of return oriented programming. The method of removing access to the return instruction pointer involves changing the CALL and RET instructions to place the return instruction pointer into cache memory. Cache memory is not directly accessible via user-mode or kernel mode code; it can only be modified indirectly by the processor. The cache used to store return instruction pointers will only be modified indirectly via the call and return instructions. Because there are no known write attacks against cache that don't involve the *manner* in which cache is being used i.e.prime+probe, the return instruction pointers will be protected from overwriting [3].

II. Characteristics of the Application

In order to calculate the expected impacts of our changes to the instruction set architecture, we selected four programs and analyzed their max stack depth and number of call and return instructions in comparison to overall instructions. The max stack depth is necessary to determine how often we will overflow the cache. The number of calls and returns is necessary for calculating any change in performance based on our modification of the call and return instructions. The four programs are: a python web-scraper scraping a website [10], resizing an image with ImageMagick [8], compiling hexedit [9][12][13], and running a simple server using netcat [11]. Below are the results of our analysis:

Program	Call / Return Count	Instruction Count	C/R %	Max Stack Frames
Python web-scraper	19569	355729	.055	587
hexedit	1366	37009	.037	197
ImageMagick	3126	69455	.045	128
netcat	567	22311	.025	107

Collecting this information involved using Valgrind's callgrind [7] for the stack frames and the Intel PIN tool [6] for counting instructions. The plugin used for PIN and can be found in **Appendix B**.

III. Description of your Design

A. Architecture Design

Our new architecture requires two hardware additions to the processor: a single register and an on-chip cache. The new register holds a single return instruction pointer at the top of the return instruction pointer stack. The on-chip cache holds several of the most recent return instruction pointers, with the rest of the return instruction pointer stack held in main memory. The additional hardware requirements required for branch prediction are discussed in **Section III C.**

As an example, given a 32KB return instruction pointer cache and 64-bit addresses, the return instruction pointer cache can hold 4,096 addresses. If the return instruction pointer stack has 8,192 addresses, then the top address is stored in the return instruction pointer register, the top 4,096 addresses are stored in the return instruction pointer cache, and the other 4,096 addresses are stored in main memory.

We implement the return instruction pointer cache as a circular buffer with cache blocks that can hold several addresses. In order to prevent any stalls due to L1 to main memory miss penalties, we implement the cache with blocks large enough to read or write faster than functions are called or returned by a program. We calculate this block size, $minBlockSize$, by solving the following inequality:

$$totalMissPenalty < \frac{minBlockSize}{maxCallRate},$$

where $totalMissPenalty = baseMissPenalty + b * minBlockSize$, given that miss penalty is a linear function of cache block size. This miss penalty must be less than the time it takes to load or store cache blocks from a non-trivial program. Solving for $minBlockSize$, we get:

$$minBlockSize > \frac{baseMissPenalty}{(1/maxCallRate)-b}.$$

Following the previous example with a 32KB cache, let $baseMissPenalty$ = 100 cycles. Also, given that no compiler will generate code where a call target is another call instruction, let $maxCallRate$ = 1/2 calls/cycle. Real values for $maxCallRate$ should be much less than this. Finally, for simplicity let b = 1. Therefore we have a $minBlockSize$ = 100 cycles / 1 cycles/call = 100 calls. Therefore, the cache block needs to hold 100 addresses which comes to at least 6,400 bits (800B) with a 64-bit address. In this example, we round up to 1KB blocks which can hold 128 64-bit addresses per block and 32 blocks total in the cache.

B. Circular Buffer Cache

Given a return instruction pointer cache that can hold C addresses, let each cache block hold B addresses. In the examples above, C = 4,096 addresses and B = 128 addresses. Also, let the current size of the return instruction pointer stack be N addresses. Remember that N can be much larger than C where N - C addresses are held in main memory. To implement the circular buffer, let the cache position for the top of the return instruction pointer stack be T where T = N mod C. T is not a cache index as it can point to any of the C addresses and any of the B addresses in a cache block. Let S be the cache position for the bottom most address in the stack that is held in the cache. For clarity, an address is held in S while T does not yet hold an address.

For each CALL, the new return instruction pointer is pushed to the return instruction pointer register and immediately also moved to position T in the return instruction pointer cache and T is incremented by 1. On a RET, after the return instruction pointer is read from the return instruction pointer register, the return instruction pointer held at position T-1 in the return instruction pointer cache is moved to the return instruction pointer register and T is decremented by 1. To prevent stalls from rapid function calls, if T + B > S (mod C) then we push all B addresses in the cache block at S to main memory. Similarly, to handle rapid function returns, if

T < S + B (mod C) then we load B addresses of the return instruction pointer stack from main memory to the cache block at S - B (mod C). Finally, we use a single bit, G, to indicate whether N > B. This is necessary to deal with initial conditions when the return instruction pointer is small.

Let us use an example where C = 16 and B = 4. Initially, when N = 0, we also have G = 0, S = 0, and T = 0. Because G = 0, we do not load addresses from memory.

```
|0123456789ABCDEF| C=16, B=4, N=0, G=0
|T---------------| T=0
|S---------------| S=0
```

After the first function call, the return instruction pointer register holds the first return instruction pointer. This value is immediately also pushed onto the return instruction pointer cache at position T. The results in the following cache:

```
|0123456789ABCDEF| C=16, B=4, N=1, G=0
|-T--------------| T=1
|S---------------| S=0
```

After four more function calls, G is now set to 1 and we have:

```
|0123456789ABCDEF| C=16, B=4, N=5, G=1
|-----T----------| T=5
|S---------------| S=0
```

When N = 12, the cache is close to getting full but still has just enough room before it triggers a push to main memory.

```
|0123456789ABCDEF| C=16, B=4, N=12, G=1
|------------T---| T=12
|S---------------| S=0
```

After one more function call, N = 13, T = 13 and T + B > S (mod C). That is, 13 + 4 (mod 16) = 1, and 1 > 0. This results in the cache pushing B addresses to main memory.

```
|0123456789ABCDEF| C=16, B=4, N=13, G=1
|-------------T--| T=13
|----S-----------| S=4
```

Now, let our program return from 5 functions such that N = 8 and T = 8. Because T - B >= S (mod C), we do not yet load addresses from main memory.

```
|0123456789ABCDEF| C=16, B=4, N=8, G=1
|--------T-------| T=8
|----S-----------| S=4
```

When one more function returns, we now have N = T = 7, and T < S + B. That is 7 < 4 + 4. Because G = 1, this causes a block of B addresses to be loaded to the cache block at S - B = 4 - 4 = 0.

```
|0123456789ABCDEF| C=16, B=4, N=7, G=1
|-------T--------| T=7
|S---------------| S=0
```

Finally, as functions return, N and T decrement until they both reach zero. When N = B, G is set to 0, preventing any unnecessary address loads from memory.

C. Branch Prediction

Branch prediction is a key architectural feature present in virtually all modern pipelined microprocessor architectures. By guessing the outcome of a conditional branch, speculative execution of subsequent instructions can begin before it is concretely known if the current branch is taken or not. Skadron et. al. [14] discuss this problem in depth and discuss a variety of solutions including a dedicated register for repairing stack state, which we include in our design. A processor that is able to accurately predict branches can continue issuing instructions into its pipeline instead of stalling until a branch outcome is known. In return oriented programming, mispredictions can result in the speculative execution of instructions that corrupt the call stack. In this section, we analyze the effect of mispredictions on our return instruction pointer stack.

One of the key challenges during speculative execution of instructions is rolling back to legitimate states on mispredictions of a branch. The return instruction pointer stack, if modified during speculative execution, must be restored to its previous state if a misprediction occurs. We consider two cases: speculative execution of instructions that do not result in main memory operations, and execution of instructions that do. For any pipeline depth P, we can accurately predict if main memory access for the return instruction pointer stack is likely. Consider the following stack state.

```
|0123456789ABCDEF| C=16, B=4, N=5, G=1
|------T---------| T=6
|S---------------| S=0
```

In the state above, speculatively executed call and return instructions can safely decrement and increment T without any risk of main memory access provided P < 7. If P >= 7, there is a risk of main memory access when issuing seven consecutive call instructions. If a misprediction occurs, rolling back the state of the return instruction pointer stack is a simple matter of restoring T back to its value when the first branching instruction was issued. The original value of T can be kept in a special purpose register, used for rolling back the state of the stack. We consider a more challenging state below:

```
|0123456789ABCDEF| C=16, B=4, N=8, G=1
|--------T-------| T=8
|----S-----------| S=4
```

The state above presents an interesting challenge - if a single return instruction is executed speculatively, it will result in main memory access to bring return addresses into the stack. This becomes a serious problem if we've mispredicted, as the branch misprediction penalty on many processors now becomes hundreds of cycles when we find ourselves having to roll back. For the ease of rolling back, main memory access should be delayed, if possible, until the branch outcome is known. If delaying main memory access is impossible (due to many CALL or RET instructions issuing into the pipeline), one should stall in the pipeline so that a misprediction does not result in hundreds of wasted cycles.

IV. Justification and Analysis

Assumptions

We perform the following calculations for an Intel Core i7 2.6Ghz processor [5]. We use a standard *DWORD* size of 4-bytes. For simplicity, we also make the assumption all memory accesses will be done on a single core and memory access will be local. We will also be assuming a maximum stack size of 8MB per-process and all call and return instructions will be 4-bytes in length. We are using Linux kernel version 2.6.31 for our data but all of the software should run and give similar results on any Unix or Unix-like operating system.

A. Reduction in Stack Frames

We calculate the reduction in stack frames with: $Max\ Stack\ Depth\ *\ DWORD\ Size$. We used the datasets mentioned above to measure these values on real programs:

Program	Max Stack Frames (Original)	Max Stack Frames (After ROP)	Max Stack Frames % Reduction
Python web-scraper	587	555	6%
hexedit	197	165	17%
ImageMagick	128	96	25%
netcat	107	75	30%

One thing to notice is that while the stack depth is reduced overall, there is not a decrease in memory consumption since the call and return addresses must still be stored for a process.

B. Block Size Calculations

The following data was calculated using the equations layed out in **Section IV** and the $baseMissPenalty$ was taken from the table in **Appendix A**. *Let the constant* $b = 1$, $baseMissPenalty = 156\ cycles$, and $maxCallRate = \frac{1}{2}$. We are assuming the worst possible case for the maximum call rate but in reality the results should be significantly less. In our findings, no more than one half of all instructions should be calls or returns, so this upper bounds should capture all programs a processor might reasonably execute. Listed below are the minimum block size required for the sample programs if they were utilizing our modifications:

Program	**Base Miss Penalty Cycles**	**Max Call Rate**	**Constant b**	**Minimum Block Size (After ROP)**
Python web-scraper	156	$\frac{1}{2}$	1	312
hexedit	156	$\frac{1}{2}$	1	312
ImageMagick	156	$\frac{1}{2}$	1	312
netcat	156	$\frac{1}{2}$	1	312

C. Branch Predictions

To determine the impact of branch prediction on our proposed changes to the instruction set architecture, we ran similar tests to obtain branching data for our four programs. In return oriented programming, branch misses can corrupt the call stack, requiring one or more repair mechanisms to intervene. The effects of branching, particularly branch misses, is expanded upon in our architecture. Below, we present our findings collected using *perf*, a performance analyzing tool available in Linux kernel version 2.6.31 and above. The commands run with *perf* are identical to those in the previous sections.

Program	**Instructions**	**Branches**	**Branch Misses**	**Branch Miss %**
Python web-scraper	455,781,153	99,945,729	2,599,529	2.56%
hexedit	2,995,084,405	662,228,379	22,714,113	3.34%
ImageMagick	277,730,508	15,177,315	313,565	2.07%
netcat	837,154	162,149	10,435	6.44%

From these results, we assume that we can expect branch misses in most common applications, justifying the adjustments for combating call-stack corruption and avoiding needless main memory access. In particular, for more synchronous applications, such as servers listening on sockets, we see that there can be a potentially high rate of branch misprediction (i.e. 6.44%).

V. Conclusions

Preventing return oriented programming attacks by preventing return instruction pointer overwrites offers a unique and robust way to offer an additional level of code security at the hardware level. By implementing a relatively simple circular buffer, we are able to provide adequate protection against malicious attempts to modify the return instruction pointer with minimal overhead. While researching this problem set we discovered that Intel has created something in the same vein called Control-flow Enforcement Technology [4]. Their design similarly removes access to the Return Instruction Pointer; but does so by creating a shadow stack within memory. They then verify return calls to ensure that the Return Instruction Pointer has not been modified. Intel's design is effective but much more convoluted than our design. Intel's solution is reverse compatible which was likely a key requirement of their design process. Nevertheless while our design is more efficient, hardware changes are required.

The design detailed within this paper turned out positively. It appears to be an elegant solution to a difficult problem. Our research finding Control-flow Enforcement Technology by Intel reinforced our belief that our general concept was accurate. Due to the fact our idea involved hardware changes, we were unable to conduct actual tests of our architectural changes. It is our sincere hope that someday our research comes to fruition in real life systems.

References:

[1] Chau, Tran Thi Minh. *An Analytical Study of Buffer Overflow Vulnerabilities*. Diss. 2015.

[2] Prandini, Marco, and Marco Ramilli. *Return-oriented programming.* IEEE Security & Privacy 10.6 (2012): 84-87.

[3] Oren, Yossef, et al. *The Spy in the Sandbox: Practical Cache Attacks in Javascript and Their Implications*. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.

[4] Intel Corporation. *Control-flow Enforcement Technology Preview.* Revision 2.0. 334525-002. Published July 2017. Accessed May 11, 2018. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[5] Levinthal, Dr. David. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors.* Version 1.0. Published 2009. Accessed May 11, 2018. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[6] Naftaly, S. Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool.* Published June 13, 2012. Accessed March, 13 2018. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[7] Valgrind. *Callgrind: A Call-Graph Generating Cache and Branch Prediction Profiler.* Copyright 2000 - 2017. Accessed March 13, 2018. http://valgrind.org/docs/manual/cl-manual.html

[8] ImageMagick. *Convert, Edit, or Compose Bitmap Images @ ImageMagick.* Copyright 1999 - 2018. Accessed March 15, 2018. https://www.imagemagick.org/script/index.php

[9] Rigaux, Pascal. *hexedit - view and edit files in hexadecimal or in ASCII.* Accessed March 17, 2018. http://rigaux.org/hexedit.html

[10] TheNiqabiCoderMum. *A simple Python script and GUI to scrape images from websites using Python.* Published Aug 15, 2017. Accessed March 17, 2018. https://github.com/nqcm/web-scraper-python

[11] Giacobbi, Giovanni. *The GNU Netcat Project.* Published November 1, 2006. Accessed March 17, 2017. http://netcat.sourceforge.net/

[12] The Free Software Foundation. *GCC, the GNU Compiler Collection.* Version 6.4. Published May 4, 2017. Accessed March 15, 2018. https://gcc.gnu.org/

[13] The Free Software Foundation. *GNU Make*. Version 4.2.1. Copyright 1988 - 2016. Accessed March 17, 2018. https://www.gnu.org/software/make/

[14] K. Skadron, P. S. Ahuja, M. Martonosi and D. W. Clark, "Improving prediction for procedure returns with return-address-stack repair mechanisms," *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, Dallas, TX, 1998, pp. 259-271.

Appendix A

1. Memory access latencies [5].

Memory Accessed	Latency
L3 Cache Hit Unshared	40 Cycles
L3 Cache Hit Shared Line	60 Cycles
Remote L3 Cache Hit	75 Cycles
Local RAM	60ns / 156 Cycles

$$\begin{aligned} Cycles_{RAM} &= Time * Clock\ Rate \\ Cycles_{RAM} &= 60(10^{-9}) * 2.6(10^{9}) \\ Cycles_{RAM} &= 156 \end{aligned}$$

Appendix B

1. PIN tool plugin [6].

```
/*BEGIN_LEGAL
Intel Open Source License

Copyright (c) 2002-2016 Intel Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.  Redistributions
in binary form must reproduce the above copyright notice, this list of
conditions and the following disclaimer in the documentation and/or
other materials provided with the distribution.  Neither the name of
```

```
the Intel Corporation nor the names of its contributors may be used to
endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR
ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
END_LEGAL */
#include <iostream>
#include <fstream>
#include "pin.H"

ofstream OutFile;

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;
static UINT64 ccount = 0;
static UINT64 rcount = 0;
static UINT64 max_depth = 0;
static UINT64 temp_depth = 0;
static UINT64 fail = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

VOID count(INS ins, VOID *v){
    if (INS_IsDirectCall(ins)){
        if (INS_IsSyscall(ins)){
            cout << "SYSCALL!!" << endl;
}
        ccount++;
        temp_depth++;
        cout << std::hex<<  INS_Address(ins) << std::dec << " " <<
(INS_Disassemble(ins)) << endl;
        cout << temp_depth << endl;
    }
    else if (INS_IsRet(ins)){
        rcount++;
        temp_depth--;
        cout << std::hex<<  INS_Address(ins) << std::dec << " " <<
(INS_Disassemble(ins)) << endl;
```

```
        cout << temp_depth << endl;
    }
    max_depth = (temp_depth > max_depth) ? temp_depth : max_depth;
    fail = (temp_depth < 0) ? 666 : fail;
    icount++;
}

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed

    //INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "ccount " << ccount << endl;
    OutFile << "rcount " << rcount << endl;
    OutFile << "Count " << icount << endl;
    OutFile << "maxdepth " << max_depth << endl;
    OutFile << "final temp, for debugging: " << temp_depth << endl;
    OutFile << "if not 0, you failed " << fail << endl;
    OutFile.close();
}

/* ===================================================================== */
/* Print Help Message                                                    */
/* ===================================================================== */

INT32 Usage()
{
    cerr << "This tool counts the number of dynamic instructions executed" << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

/* ===================================================================== */
/* Main                                                                  */
/* ===================================================================== */
/*   argc, argv are the entire command line: pin -t <toolname> -- ...    */
/* ===================================================================== */

int main(int argc, char * argv[])
```

```
{
    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());

    // Register Instruction to be called to instrument instructions
    //INS_AddInstrumentFunction(Instruction, 0);
    INS_AddInstrumentFunction(count,NULL);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```