

Neutralizing Manipulation of Critical Data by Enforcing Data-Instruction Dependency

Chandra Sharma
Kansas State University
ch1ndra@ksu.edu

Nathan Miller
Kansas State University
nathan232@ksu.edu

George Amariuca
Kansas State University
amariuca@ksu.edu

ABSTRACT

In this paper, we propose a new approach to neutralize attacks that tamper with critical program data. Our technique uses a sequence of instructions as a trap against the illicit modification of the critical data. In a nutshell, we set up a dependency such that the continued execution of the program is contingent upon the successful execution of the instruction sequence and the successful execution of the instruction sequence is contingent upon the integrity of the critical data. In particular, we discuss a specific implementation of our technique focusing on a critical data that is often subject to malicious manipulation: the return address of a function. We show that our technique can be an effective countermeasure to defend against attacks that overwrite the return address to divert control to a malicious code. We further show that our technique offers significant protection without resorting to complementary defenses such as ASLR, DEP or StackGuard.

CCS CONCEPTS

• Security and privacy → Mobile platform security; Trusted computing; Software security engineering.

KEYWORDS

control-hijack; critical instructions; return-address-instruction dependency, code-stack

ACM Reference Format:

Chandra Sharma, Nathan Miller, and George Amariuca. 2020. Neutralizing Manipulation of Critical Data by Enforcing Data-Instruction Dependency. In *Hot Topics in the Science of Security Symposium (HotSoS '20)*, April 7–8, 2020, Lawrence, KS, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3384217.3385620>

1 INTRODUCTION

Computer software encapsulates several types of critical data that are of interest to potential attackers. The end effect of manipulating these data ranges from something simple like allowing access to a locked software feature to something more intricate like providing full access to the system. More importantly, these data are

influential in determining the control flow of the program and therefore, tampering with them opens up unlimited possibilities. It is, therefore, imperative that such data be shielded from manipulation.

In this paper, we present a novel approach to effectively defend against attacks that tamper with critical data. Our approach is centred around the concept of *critical instructions*, which are instructions that are used as a trap against the unintended modification of the critical data. An important characteristic of critical instructions is that the continued program execution is contingent upon the successful execution of the critical instructions, which in our technique is contingent upon the integrity of the critical data.

One type of critical data that receives the attention of most attackers is the return address of a function. The return address is a part of the activation record placed in the stack by a function, when it calls on another function. By manipulating the return address in the activation record, the attacker is able to redirect the program execution to an arbitrary address and execute arbitrary code. Such manipulation of the return address, hence forth referred to as a *return-manipulating attack*, is often achieved by means of exploiting a buffer-overflow vulnerability. The lack of proper bounds-checking while referencing data in the buffers often allows an attacker to write data beyond the capacity of the buffer and to the adjacent parts of the memory allocated for other program data. Such overflows are quite common in the stack and often the subject of a general class of attacks known as *stack-smashing attacks*.

To exemplify a use case of our technique, this paper focuses on attacks that overwrite the return address with the purpose of redirecting control flow to arbitrary code. To defend against such attacks, we enforce a dependency between the return address and a set of critical instructions. While there are numerous approaches to enforce such dependency, our example implementation uses a relatively straightforward approach: upon calling a function, the instruction sequence responsible for returning back to the caller is masked with the return address saved on the stack, and placed on a secondary stack, outside the reach of the attacker. To return from the callee, the program execution is diverted to the secondary stack, where the critical instruction sequence is unmasked with the current value of the return address, and executed. The masking operation needs to be carried out before an anticipated illicit modification of the return address, while the unmasking-and-execution operation after the illicit modification, but before any significant damage can be done. It is thus natural to incorporate masking into the callee function's prologue, and unmasking in its epilogue.

While the impending discussion will mostly revolve around a specific implementation of our approach, the underlying principle can have many different implementations in many different domains. For example, the concept of critical instructions is not just specific to x86 programs; the same concept can be applied to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotSoS '20, April 7–8, 2020, Lawrence, KS, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7561-0/20/04...\$15.00

<https://doi.org/10.1145/3384217.3385620>

secure programs for other processor architectures as well. Moreover, although our example implementation uses a simple XOR mask, our technique is perfectly compatible with more sophisticated encoding-decoding schemes.

The remainder of the paper is organized as follows: Section 2 provides a brief overview on some well-known countermeasures against the attacks in question, and the limitations of those countermeasures. Section 3 is divided into several subsections and provides a detailed exposition of our technique. Section 4 discusses a prototype implementation of our technique. Section 5 makes functional comparison of our technique with other similar techniques. Finally, Section 6 makes concluding remarks and discusses future work.

2 BACKGROUND

Ever since return-manipulating attacks became commonplace, numerous defenses have been proposed to neutralize them. While an ideal defense would offer a complete solution that is provably secure, that incurs negligible overhead and that can be deployed without significant changes to the hardware and software, it is hard to find such a solution under real world constraints. As a consequence, existing defense strategies are based on some tradeoffs. Here, we discuss some of the countermeasures and highlight the tradeoffs they make.

Address Space Layout Randomization (ASLR) encompasses those countermeasures that randomize the base address of critical program sections when the program is loaded in the memory. By randomizing the base address of the code section, data section and the stack region independently, ASLR aims to break the execution flow required for a successful exploit by making it harder for the attacker to pin-point the exact location of the exploit code. Shacham et al. [15] studied the effectiveness of the ASLR system and concluded that, in many cases, it is fairly easy to brute-force past this defense. The security flaw of ASLR stems from the fact that the randomization of addresses is limited to the address space available. Furthermore, not all bits can be used for randomization due to various constraints – e.g., some randomized program segments must align to a page-boundary. The fact that the address bits available for randomization usually form a proper and rather small subset of the entire address poses serious problems, especially for 32-bit architectures. In recent years, numerous enhancements for the ASLR system have been proposed, particularly focusing on fine-grained randomization to increase the entropy. **Address Space Layout Permutation (ASLP)** [13] is one such defense strategy that builds upon ASLR. Essentially, ASLP achieves fine-grained randomization by randomly reordering functions within the code segment and data objects within the data segment. Another defense, dubbed **Instruction Location Randomization** [11] randomizes the location of every instruction in the memory. Although this class of defenses look like a promising solution to the problem at hand, the attack models for these defenses fail to account for multiple memory disclosure vulnerabilities. Consequently, these defense strategies have been the subject of successful memory disclosure attacks [17].

StackGuard and similar countermeasures [6, 20] make use of a canary value to detect tampering with the saved return address. Typically, a canary value is placed between the local variables, and

the saved frame and instruction pointers in the stack. These measures make the assumption that in order to overwrite the saved instruction pointer (return address), the attacker must overwrite the canary, which can be detected. Hence, the canary value is chosen such that it is either hard to replicate (Terminator canary) or, hard to guess (Random canary and Random XOR canary). The vulnerability of the StackGuard and similar countermeasures comes from the fact that an attacker need not overwrite the canary value before he can overwrite the saved return address. For example, by exploiting a format string vulnerability, an attacker can overwrite the return address without modifying the canary [2]; if the implementation uses a terminator or a random canary, the tampering is never detected. Furthermore, implementations that make use of a random canary or a random XOR canary rely on the secrecy of some random value that is either used as a canary itself or used to generate a canary. The implicit assumption is that it is infeasible for the attacker to know this value. However, Strackx et al. showed that this assumption does not always hold as the attacker may be able to retrieve the secret value by exploiting a buffer-overread vulnerability [19].

PointGuard encrypts all code pointers before storing them in memory and decrypts them when they are de-referenced [5]. The encryption-decryption scheme is based on a bitwise XOR operation: the encryption involves XORing a pointer with a random value generated at runtime, and the decryption involves XORing the encrypted value with the same random value. PointGuard makes the assumption that without the knowledge of the random value used for encryption, the attacker cannot make a useful prediction about the decrypted value; if the attacker overwrites an encrypted code pointer, it is very likely that decryption of the pointer results in an unexpected value that leads to program crash. The security offered by PointGuard mostly relies on the secrecy of the random value, and consequently, PointGuard is also vulnerable to buffer-overread attacks [19]. Furthermore, since pointers are addresses, encrypted pointers only offer an entropy of 2^{32} for a 32-bit system. In the best-case scenario (from the defender's point of view), the odds for a successful attack is 1 in 2^{32} . However, the odds significantly improve to 1 in 2^8 , if the attacker only needs to overwrite a byte of the encrypted pointer [2, 22].

Instruction Set Randomization (ISR) randomizes the instruction set for every process running on the system [12]. This approach offers solid protection against code-injection attacks; without the knowledge of the randomization key, any injected code is most likely to be treated as invalid. This is analogous to executing ARM instructions on an x86 processor. Of course, there could be some instructions that are common to both processors, but any such instruction will have different semantics and will therefore be unlikely to do anything meaningful for the attacker. The major flaw with Instruction Set Randomization is that it incurs high overhead [22]. Unless natively supported by hardware, such high overhead makes ISR prohibitively expensive to implement. Software emulated versions of ISR such as Randomized Instruction Set Emulator (RISE) has been found to be vulnerable to chosen key attacks [21]. Also, ISR inherently does not protect against code-reuse attacks as any code in the process' memory will be valid for that process [18]. In conclusion, the performance limitations of ISR may often outweigh its security benefits.

Control Flow Integrity enforces a rule that a program execution must stick with its Control Flow Graph [1]. By constraining the control flow to those locations that are determined before the execution actually begins, CFI aims to mitigate all those attacks that rely on manipulating the control flow of the program. The Control Flow Graph is created by taking into account all possible sites of control transfer; each node in the graph represents a site and an edge from one node to another represents the control flow from a source to a destination. The enforced rule requires that every control transfer during the program execution must correspond to an edge from a source node to a destination node in the graph. While CFI promises a great deal of security, it is known to incur considerable overhead and has some compatibility problems [24]. In recent years, numerous enhancements to CFI have been proposed [24, 25], but the security implications of these enhancements are not very encouraging [8, 10].

Code Pointer Masking (CPM) [14] aims to place a constraint on the control flow of the application and is therefore a close neighbor of CFI. By imposing a restriction on the addresses that a function can return to, CPM offers reasonable protection against attacks that rely on manipulating the return address to divert the control flow to arbitrary addresses. CPM takes it even further: it imposes a restriction on all code pointers in addition to the return addresses. It does this by applying a mask to all code pointers before they are used for control transfer. For masking the return addresses, the mask is created by OR-ing all the return sites of the program determined before the program is executed. The mask for the function pointers, on the other hand, is created from the addresses of the functions in the program code. In essence, the mask acts as a filter to limit the range of addresses to which program control can be transferred. The security of CPM depends on the effectiveness of the mask, which in turn depends upon the number of different addresses that are used to create the mask. Furthermore, the number of overlapping bits in the addresses also accounts for the efficacy of the mask. For effective protection against return-manipulating attacks, it is desirable that the program has fewer return sites to limit the range of the addresses that the function can return to. Unfortunately, such constraints are often impractical.

Return Address Defender (RAD) [3] and similar countermeasures [4, 7, 9, 23] separate critical program data from the main stack and place them in an alternate memory location. The underlying design incentive stems from the observation that overflows are common in the stack and control data are the primary targets. By separating regular data from control data, these countermeasures ensure that an attacker is unable to leverage a buffer overflow and use it to manipulate control structures. More recent defenses of this type make use of multiple stacks [23], each assigned to hold different data sets based on criticality. Furthermore, each stack is protected from the other by using a *guard page* to ensure that overflow in one stack does not affect the other. However, all these countermeasures rely on an assumption that the attacker cannot access the memory region where the control data are saved. In practice, such assumptions do not always hold and sometimes a single indirect pointer overwrite can be exploited to bypass these defenses [22].

3 RETURN-ADDRESS-INSTRUCTION DEPENDENCY

3.1 Overview

Most of the existing defenses involve *transforming some address/-pointer parameter*; for example, ASLR randomizes the base address of the program-segments, ASLP randomizes the address of the functions, PointGuard encrypts all the pointers with some random value and Code Pointer Masking masks the code pointers with a pre-determined bit pattern. In general, the address parameter is the primary subject transformation. By contrast, our proposed technique takes a different approach, based on the observation that, in the course of an attack, the normal execution of a program is diverted right after the victim program executes some pre-determined instructions. For example, the execution of the *ret* instruction by the victim program is where the normal execution takes the diversion intended by the attacker. This gives us a new perspective: just as the integrity of the return address is critical from the defender's point of view, the integrity of the *ret* instruction is critical from the attacker's point of view. The *ret* instruction, in essence, is a bridge for execution-diversion. Should this bridge collapse, the attack is essentially thwarted.

The core of our technique is using the *ret* instruction as a trap, the bait for which is the return address saved in the stack. This requires enforcing a dependency between the *ret* instruction and the return address such that tampering with the return address will alter the *ret* instruction. This technique can be extended further to incorporate other critical instructions resulting in a long sequence of critical instructions that must be executed in the unmodified form for the program to continue. As altering a single bit of the return address essentially alters the instruction sequence due to the dependency, if an attacker tampers with the written address, the program terminates abruptly during the unsuccessful execution of the instruction sequence thereby, eliminating the attacker's prospect of taking control over the victim program. To setup the dependency, we use the following approach:

- In the function prologue, we encode/mask a sequence of critical instructions with the return address saved on the stack.
- In the function epilogue, we decode/unmask the encoded sequence with the return address available on the stack, and then proceed to execute it.
- All encoded/decoded instructions are written to a code-stack: a stack that is set apart for a program to facilitate execution of temporary code. Our code stack setup will be similar to that used by Younan et al. [23] in their multi-stack defense approach and Dang et al. [7] in their *parallel shadow stack* setup, one major difference being that our stack will hold code instead of data. The code stack is, by default, assumed to be writable+executable.
- Each function will be allocated a stack frame in the code stack to store the encoded/decoded instructions. The frame allocated for a function in the main stack will be the same size as that allocated in the code-stack. The implication is that a function's frames will be at a constant offset from one stack to the other.

Our approach mandates that if one of the encoded instructions in the sequence is decoded incorrectly, the resulting instruction must contribute to breaking the execution flow. To achieve this, we logically divide the x86 instruction set into two groups. Group A includes those instructions that the attacker might use to his advantage. These include all forms of control transfer instructions such as different variations of *jmp*, *call* and *ret* instructions. Group B includes all those instructions that do not have any value to the attacker. We ensure that all unsuccessful decodings (as a result of tampering with the return address) result in instructions in the second group. Note that the execution of an unsuccessfully decoded instruction may not necessarily result in immediate program termination as the execution may slip to subsequent instructions. A careful coordination with the subsequent instructions is required to ensure program termination.

3.2 Attack Model and Assumptions

Before delving into the specifics of our technique, it is important to define an attack model and make explicit all the assumptions. To define the security scope of our proposed return-address-instruction dependency (RAID) approach, we assume that the adversary (a) cannot manipulate other code pointers, and (b) cannot overwrite the encoded instructions in the code stack. The first assumption is simply stating that the RAID technique is designed for, and effective against, those attacks that overwrite the return address to manipulate the control flow of the program. These include attacks that use a buffer overflow vulnerability, indirect pointer overwriting, or format string vulnerabilities to overwrite the return address saved on the stack. We do not include those attacks that manipulate other code pointers to hijack control, although we believe it is possible to extend our approach to protect against these classes of attacks as well. Furthermore, although we haven't accounted for the attacks that manipulate the frame pointer, it is straightforward to extend our approach to protect against those attacks as well. The second assumption is rather realistic, and very commonly employed (either explicitly or implicitly) in the literature [3, 4, 9, 23].

To establish that the RAID technique offers significant protection without resorting to secondary defenses, we also need to clarify the following allowances. (1) There needs be no canary in place, and thus the adversary can freely overwrite the return address. (2) ASLR may be turned off or on, at the preference of the adversary. (3) The adversary knows what set of instructions are encoded by RAID. (4) The adversary knows the return address used to encode those instructions. (5) The adversary knows the algorithm used to encode/decode the instructions.

3.3 Enforcing Dependency with the *ret* Instruction

We start by considering *ret* as the only instruction in our instruction sequence. We now look into techniques to enforce a dependency between the *ret* instruction and the return address. As the return address is four bytes long while the *ret* instruction is only one byte in size, it not possible to efficiently enforce bit-to-bit dependency. At this time, we therefore, consider only the least significant byte of the return address. We now present an example:

```
main:
  push ebp
  mov ebp, esp
```

```
...
call somefunc
add esp, 16
leave
ret

somefunc:
; Get the Least Significant Byte of the return address
mov bl, BYTE [esp]
; Encode the ret instruction (0xC3) with the return address
xor bl, 0xC3
; Write the encoded instruction to code-stack
mov BYTE [ebp+code_stack_offset], bl
; The usual function prologue
push ebp
mov ebp, esp
...
leave
; Get the Least Significant Byte of the return address
mov bl, BYTE [esp]
; Decode the encoded instruction
xor bl, BYTE [ebp+code_stack_offset]
; Write the decoded instruction
mov BYTE [ebp+code_stack_offset], bl
; Start executing the decoded instruction
lea ebx, [ebp+code_stack_offset]
jmp ebx
```

Figure 1a and Figure 1b show the code stack right before and after the execution of *somefunc*'s modified prologue. Notice that the code stack is padded with *hlt* (0xF4) instructions to ensure program termination if the execution slips. Also, for illustration, we assume the return address of *main* is 0x08048024. Notice in Figure 1b that the XOR operation between the least significant byte of the return address (0x24) and the *ret* instruction (0xC3) yields 0xE7.

F4	F4	F4	F4
F4	F4	F4	...

(a) Code-stack right before the execution of the modified prologue

E7	F4	F4	F4
F4	F4	F4	...

(b) Code-stack right after the execution of the modified prologue

Figure 1: Code-stack before and after the execution of *somefunc*'s modified prologue

Now, let us assume that, in between the execution of *somefunc*'s prologue and epilogue, the attacker overwrites the return address with 0x10204354. Notice that the least significant byte of the return address has changed and therefore, the encoded instruction won't decode to the anticipated 0xC3 (the *ret* instruction). Instead, it decodes to 0xB3, the result of XOR between 0xE7 and 0x54.

After the function epilogue decodes the encoded instruction, the execution jumps to the address of the decoded instruction. Note that the machine code 0xB3F4 translates to the assembly instruction *mov bl, 0xF4* which, in itself, does not terminate the program. Instead, it lets the execution fall through to the next instruction, the HLT instruction (0xF4). It is this instruction that terminates the program, cutting the attacker from taking control.

Although for the values assumed in the previous example the decoded instruction contributed to the termination of the program, this may not always be the case. For example, let us assume that the attacker manages to overwrite the return address (0x08048024) with 0x10204325. After the encoded byte (0xE7) is decoded with the illicitly modified least significant byte of the return address (0x25), it results in 0xC2. Note that the machine code 0xC2F4F4 translates to the assembly instruction *ret 0xF4F4*. Clearly, this instruction does

not contribute to the termination of the program, but rather it transfers control to the modified return address (and pops 0xF4F4 bytes off the stack in the process). This is what we call a *false trap*, to reflect a case where our purported trap fails its purpose.

One way to avoid false traps altogether is to carefully choose the bits of the instruction sequence that are to be encoded/decoded. In some cases, it takes exhaustive trial and error to determine all those bits that can be safely encoded/decoded. In others, superficial analysis involving common sense is enough. For the *ret* instruction (0xC3), we exhaustively tested all possibilities and found that encoding/decoding the high nibble (0xC) is optimal. Note that each nibble corresponds to 4 bits, so this allows for 2^4 different decoded values: 0x03, 0x13, 0x23, 0x33, 0x43, 0x53, 0x63, 0x73, 0x83, 0x93, 0xA3, 0xB3, 0xC3, 0xD3, 0xE3 and 0xF3. Figure 4 comprehensively lists of all possible byte sequence in the code stack when only the high nibble of the *ret* instruction is encoded-decoded.

03 F4	add esi, esp	83 F4 F4	xor esp, 0FFFFFFF4
F4	hlt	F4	hlt
...
13 F4	adc esi, esp	93 F4	xchg ebx, eax
F4	hlt	F4	hlt
...
23 F4	and esi, esp	A3 F4 F4 F4 F4	mov ds:0xF4F4F4F4, eax
F4	hlt	F4	hlt
...
33 F4	xor esi, esp	B3 F4	mov bl, 0xF4
F4	hlt	F4	hlt
...
43 F4	inc ebx	C3	ret
F4	hlt
...
53 F4	push ebx	D3 F4	invalid opcode
F4	hlt	F4	hlt
...
63 F4	arpl sp, si	E3 F4	jecxz -10
F4	hlt	F4	hlt
...
73 F4	jae -10	F3 F4	repz hlt
F4	hlt
...

Figure 2: A listing of all possibilities when the high nibble of the *ret* instruction is encoded-decoded

Some interesting facts are revealed by Figure 2. First, some of the padding *hlt* (0xF4) bytes are often a part of the operand to some other instruction. Second, the number of padding bytes (0xF4) that are treated as an operand to an instruction differs between different possible instructions; for example, the *mov ds:0xF4F4F4F4* instruction takes four 0xF4 bytes as its operand while the *inc ebx* instruction takes none. This information is very helpful in determining the least number of *hlt* instructions (0xF4s) that must be appended to ensure that the execution does slip. As it is evident from the figure, the *mov ds:0xF4F4F4F4*, takes the maximum number, 4; therefore, we need a *hlt* sled composed of at least 5 *hlt* instructions so that at least one of them is spared to halt the execution. We note that the minimum size of the *hlt* sled is instrumental in determining the size of the frame to be allocated for each function.

Yet another interesting observation is that not all instruction sequences are guaranteed to end up in an *hlt* instruction. While the instruction sequence C3 F4 ... (assembly: *ret hlt* ...) will not reach an *hlt* instruction by design, there are two other possible instruction

sequences that may not end up in an *hlt*: *jae -10* and *jecz -10*, both of which jump 10 bytes backward (when the corresponding condition is true). To prevent execution slip, a fix is to insert a *hlt* instruction at the jump offset. This approach, however, imposes a loose restriction on the length of the critical instruction sequence: in between the *hlt* sleds, the critical instruction sequence must be at most 10 bytes in length. As this number is fairly large to suit our needs, we did not examine the possibilities of circumventing the restriction.

3.4 Enforcing Dependency with the *leave* Instruction

Modern compilers for C and C++ languages often generate a *leave* (0xC9) instruction right before a *ret* instruction in the compiled function. The purpose of the *leave* instruction is to release the stack frame just before the callee function returns. An after-effect of the execution of *leave* is that it makes the *esp* point right above (at one-DWORD-higher address than) the saved *ebp* in the current function's stack frame. The subsequent *ret* will then pop the value at that position into the *eip* register. Unless illicitly modified, this value is the return address of the calling function. It is thus clear that *leave* plays a crucial part in selecting the value that is to be interpreted as the return address.

The *leave* instruction is also an interesting candidate for a trap sequence. Before we dig into the specifics, let's examine the effect of *leave* in the execution of a program by means of a trivial C program and the corresponding assembly code.

```

int add(int x, int y) {
    int result = x+y;
    return result;
}

int main() {
    int sum = add(10,20);
    return 0;
}

add:
    push    ebp
    mov     ebp, esp
    sub     esp, 4
    mov     edx, DWORD PTR 8[ebp]
    mov     eax, DWORD PTR 12[ebp]
    add     eax, edx
    mov     DWORD PTR -4[ebp], eax
    mov     eax, DWORD PTR -4[ebp]
    leave
    ret

main:
    push    ebp
    mov     ebp, esp
    sub     esp, 4
    push    20
    push    10
    call    add
    add     esp, 8
    mov     DWORD PTR -4[ebp], eax
    mov     eax, 0
    leave
    ret

```

Figure 3 shows the layout of the main stack right before the execution of the *leave* in the *add* function.

Now, let's assume that the *leave* instruction in the *add* function is omitted altogether from the compiled program. When this program is executed and the execution gets to the *ret* instruction in the *add* function, it will load the value in the *result* variable into the

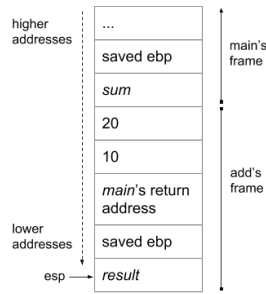


Figure 3: Partial stack layout before the execution of *leave* in the *add* function

eip register. Whether or not the execution will terminate depends on the value of *result*—one instance where the program is surely terminated by the operating system is when *result* holds a NULL value.

Note that, for our purpose, replacing *leave* with some other instruction is equivalent to omitting it altogether – provided, of course, that the replacing instruction does not modify the stack pointer and lets the execution fall through to the subsequent *ret* instruction.

Before we further examine the possibility of incorporating *leave* into our critical instruction sequence, let's look into the slightly tweaked assembly code of the *add* function in the above program.

```
add:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov edx, DWORD PTR 8[ebp]
    mov eax, DWORD PTR 12[ebp]
    add eax, edx
    mov DWORD PTR -4[ebp], eax
    mov eax, DWORD PTR -4[ebp]
    mov esp, 0
    leave
    ret
```

From the normal execution point of view, the two assembly codes have the same functionality despite the added line of code (marked in bold). However, for what we're about to discuss, this change makes an important difference. Let's again assume that the *leave* instruction in the *add* function is omitted from the compiled program. This time, when the program is executed and the execution gets to the *ret* instruction in the *add* function, the program is guaranteed to raise a protection fault and terminate.

As we are interested in instructions that do not alter the program semantics during normal execution but contribute to program termination in the course of an attack, *leave* is, therefore, an interesting candidate for a trap sequence. As we did for the *ret* instruction, we ran tests to determine which bits to encode in order to avoid false traps. In the case of *leave* (0xC9), we found that encoding-decoding the low nibble (0x9) is optimal. When the low nibble of the encoded instruction is decoded, it allows for 16 different possibilities: 0xC0, 0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7, 0xC8, 0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE and 0xCF. Figure 4 lists some of the possible byte sequences in the code stack corresponding to these possibilities. Note that the first byte corresponds to the *leave* instruction and the second byte corresponds to the *ret* instruction followed by a fixed number of *hlt* instructions (although in the figure we've opted not to show the *hlt* instructions where they are irrelevant).

C0 C3 F4 F4	rol bl, 0xF4 hlt	C8 C3 F4 F4	enter 0xF4C3, 0xF4 hlt
C1 C3 F4 F4	rol ebx, 0xF4 hlt	C9 C3	leave ret
C2 C3 F4	ret 0xF4C3	CA C3 F4	retf 0xF4C3
C3	ret	CB	retf
C4	invalid opcode	CC C3	int 3 ret
C5	invalid opcode	CD C3 F4	int 0x03 hlt
C6 C3 F4 F4	mov bl, 0xF4 hlt	CE C3	into ret
C7 C3 F4 F4 F4 F4 F4	mov ebx, 0xF4F4F4F4 hlt	CF	iret

Figure 4: A listing of some possibilities when the low nibble of the *leave* instruction is encoded-decoded

We should note that Figure 4 is not a comprehensive listing of all possibilities. For the sake of brevity, we assumed that the bits of the return address used to encode/decode the *ret* (0xC3) instruction are unmodified, and so we fixed the second byte to 0xC3. A comprehensive listing all the possible byte sequences is available on the first authors' website.

Referring to Figure 4, we make few crucial observations. The shaded rows show the cases where the return address has been illicitly modified but the execution does not end up in the *hlt* instruction. Clearly, before the execution can get to one of the *hlt* instructions, the control is diverted by the execution of a *ret*, *retf* or *iret* instruction. Essentially, in each case, one of these instructions has replaced the *leave* instruction and therefore, its effect is completely lost when the decoded sequence is executed. Note that the *ret*, *retf* and *iret* instructions all load the *eip* register with the value determined by the *esp* register, but since there is no *leave* to adjust the *esp*, we cannot make any prediction where the value will come from. Hence, depending upon the compiler that generated the compiled code, the popped value may or may not be within the attacker controlled buffer. To address the potentially security pitfalls, we tweak the *esp* register to point to an inaccessible address (a NULL address) right before the function executes its epilogue. With this tweak, when the *ret* instruction tries to pop the return address off the stack, the CPU will raise a General Protection Fault. It should be noted that this does not impact the normal execution of the program (where a *leave* instruction precedes a *ret* instruction in the code stack) as *leave* will adjust the *esp* to its correct position.

3.5 Fabricating Critical Instructions

Thus far, we have incorporated into our critical sequence all those instructions that constitute the function epilogue. Unfortunately, the two instructions, *leave* and *ret*, exhaust the list but only provide for 8-bit dependency with the return address. Next, we discuss fabricating other critical instructions, and placing them into the sequence, so that we can enforce one-to-one dependency with all the bits of the return address. But before we delve into the details, it

is important to define certain requisite properties of the fabricated instructions:

- The fabricated instructions must not break the normal execution of the program or alter the intended program semantics.
- The fabricated instructions must either break the execution, or contribute to breaking the execution, when any of the bits used to encode/decode them is altered. Hence forth, the instructions that break the execution will be called *terminators*, and the instructions that contribute to breaking the execution will be called *contributors*.

While there are numerous instructions that can be fabricated as critical instructions, we only examine a few in this paper. We start off by considering another *leave* instruction, as we have already analyzed the security implications of using it as a trap instruction and found it to be a perfect fit. As the functionality of *leave* is to release a function's frame, we cannot, however, incorporate another *leave* into our sequence without first fabricating a fake function frame. For this part, we'll jump directly to an example of fabricating a fake function frame and skip trivial details.

add:

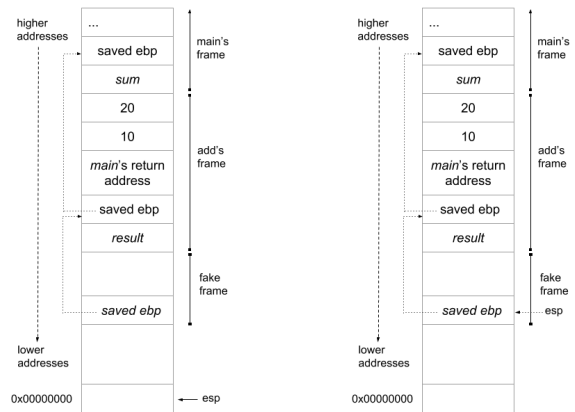
```
push ebp
mov ebp, esp
sub esp, 4
mov edx, DWORD PTR 8[ebp]
mov eax, DWORD PTR 12[ebp]
add eax, edx
mov DWORD PTR -4[ebp], eax
mov eax, DWORD PTR -4[ebp]
sub esp, someOffset
push ebp
mov ebp, esp
mov esp, 0
leave
leave
ret
```

The above code is a slightly modified version of the *add* function we used as an example in the previous section. The lines marked in bold reflect the code that creates and releases the fake function frame. The assembly instructions *mov esp, 0* and *sub esp, someOffset* ensure that the stack pointer is outside the current function's frame when the execution gets to the first and the second *leave* instructions, respectively. Clearly, the value for *someOffset* must be chosen by the compiler during program compilation.

It is fairly intuitive that the fabricated *leave* does not alter the program semantics. Unfortunately, it cannot be relied upon for program termination following the tampering with the return address. To elucidate, let us assume that the first *leave* instruction in *add*'s epilogue is replaced with some other instruction that lets the execution fall through to the second *leave* instruction. After the replacing instruction is executed, the execution will simply move to the second *leave* instruction. Figure 5a shows the stack layout right before the execution of the second *leave*. Now, when the second *leave* is executed, it will position the stack pointer to the address of the last saved frame pointer as shown in Figure 5b. When the subsequent *ret* is executed, it will interpret the saved frame pointer as the return address and start executing data in the stack as instructions. Since the data in the stack is most likely controlled by the attacker, he is able to execute arbitrary code. It is therefore apparent that adding another *leave* instruction in our sequence is a bad idea.

It is interesting that although one instance of the *leave* instruction can be used as a critical instruction, multiple instances of the same instruction are not necessarily appropriate for the same use.

This leads us to another significant finding: the instruction(s) preceding the candidate instruction and the instruction(s) following the candidate instruction also determine the efficacy of the candidate instruction for use as a critical instruction. In general, if the instruction is a *terminator*, its effectiveness is determined by the instruction(s) preceding it. This is because the effect of executing an instruction, critical or otherwise, is dependent on the execution environment set by the preceding instructions. For instance, the value popped in the *eax* register during the execution of the *pop eax* instruction depends on the position of the stack pointer, which in turn depends on the execution of one or more preceding instructions that manipulate the *esp* register. On the other hand, if the instruction is a *contributor*, its effectiveness is determined both by the instruction(s) preceding it (if there are any) and the instructions immediately following it. For instance, the *ret* instruction which we discussed earlier is a *contributor*, so it relies on the following *hlt* instruction(s) to terminate the execution when incorrectly decoded.



(a) Stack layout before the execution of the second *leave*

(b) Stack layout after the execution of the second *leave*

Figure 5: Partial stack layout corresponding to two execution points in the *add* function's epilogue

Although the fabricated *leave* instruction, on its own, cannot be used as a critical instruction, it may be possible to pair it with some other instruction and produce a critical pair of instructions. Simple intuition suggests that the other instruction must relate to the *leave* instruction, or else there is no point of pairing them. Essentially, we need to instantiate a dependency between the *leave* instruction and the other instruction. Without discussing too many specifics, let us look at one such instruction by means of an example:

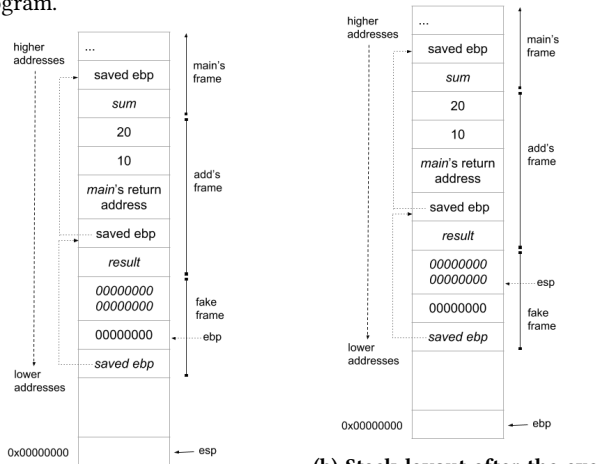
add:

```
push ebp
mov ebp, esp
...
sub esp, someOffset
push ebp
mov ebp, esp
mov esp, 0
inc ebp
dec ebp
leave
leave
ret
```

In the above example, we fabricated a *dec ebp*(0x4D) instruction in the *add* function we defined earlier. The *dec ebp* and *leave* instructions are related in the sense that *dec ebp* decrements the frame

pointer which is the implicit operand to the *leave* instruction. In the example, we also added a complementary instruction, *inc ebp*, to ensure that the program semantics are preserved.

Before discussing the aptness of the *dec ebp* instruction for inclusion in the sequence of critical instructions, we make a note that although the fabricated *leave* instruction is a part of the critical instruction pair, it will not be encoded/decoded as we found it to be insecure for our purpose. The functionality of the fabricated *leave* is, therefore, similar to that of a *hlt* instruction in our sequence: the instruction itself won't be used to enforce a dependency with the return address but will be a part of the instruction sequence to break the execution flow when some other instruction is incorrectly decoded. However, unlike the *hlt* instruction, the fabricated *leave* instruction will be executed during a normal execution of the program.



(a) Stack layout before the execution of the *dec ebp*

(b) Stack layout after the execution of the fabricated *leave* instruction

Figure 6: Partial stack layout corresponding to two execution points in the *add* function's epilogue

Figure 6a shows the partial stack layout right before the execution of the *dec ebp* instruction in the *add* function. We should note that for this part, we're assuming that whenever a fake function frame is created, its contents are cleared to zeroes. This is a fairly practical assumption as the compiler can easily be tweaked to attain this effect. Next, to examine the effect of incorrectly decoding the *dec ebp* instruction, we will assume that it is replaced with some other instruction that lets the execution fall through to the fabricated *leave* instruction. Now, let us examine the stack layout after both the replacing instruction and the fabricated *leave* instruction are executed. Figure 6b shows the partial stack layout after the execution of the fabricated *leave* instruction but before the execution of the next *leave* instruction. At this point, we request the readers to keep track of the stack and the frame pointers while we consider two possibilities.

First, we'll assume that the following *leave* and *ret* instructions are unmodified—as in the case where these instructions are correctly decoded because the bits of the return address used to encode them are not tampered with. In this case, when the second of the *leave* instructions is executed, the program will raise a protection fault and terminate. It is immediately obvious that the fabricated *leave*

instruction itself is not the one terminating the execution, rather it serves to alter the execution state which causes the next *leave* instruction to raise a protection fault.

The next possibility is when one or both of the trailing *leave* and the *ret* instructions are modified—that is, when these instructions are incorrectly decoded because the bits of the return address used to encode them are tampered with. Assuming four bits of each of the *dec ebp*, *leave* and *ret* instructions are encoded-decoded, analyzing if tampering with the return address breaks the execution will generally require analyzing 2^{12} possibilities. This is due to the fact that the execution of an instruction may alter the execution environment for the subsequent instructions. Further, when one of the instructions is incorrectly decoded, the resulting instruction may take the following instructions as its operands, and since the x86 architecture provides for variable length instructions, this makes it difficult to ascertain the number of following instructions that will be interpreted as operands. Fortunately, the need to test for all 2^{12} possibilities can be greatly reduced by grouping the many resulting instructions into equivalence classes, such that all the instructions in one equivalence class have a similar effect on the program execution. For instance, if a critical instruction *X* decodes to one of the instructions in the set $A = \{inc\ eax, inc\ edx, inc\ esi\}$, and if a critical instruction *Y* decodes to one of the instructions in the set, $B = \{leave, ret, retf, irt\}$, then the effect of the execution of any of the instructions in Set A is orthogonal to the effect of the execution of any instruction in Set B. For our security evaluation, only one of the instructions in Set A needs to be considered. Of course, if the sequence contains a third critical instruction, say *Z*, then we need to consider *X*, *Y* and *Z* together to determine the equivalence classes.

After running comprehensive tests to determine which bits of the *dec ebp* can safely be encoded (and subsequently decoded), we found that encoding the lower nibble of the *dec ebp* instruction is optimal for our purpose. Figure 7 shows all possibilities when the lower nibble of the *dec ebp* instruction is decoded.

40	inc eax	48	dec eax	44	inc esp	4C	dec esp
C9	leave	C9	leave	C9	leave	C9	leave
C9	leave	C9	leave	C9	leave	C9	leave
...
41	inc ecx	49	dec ecx	45	inc ebp	4D	dec ebp
C9	leave	C9	leave	C9	leave	C9	leave
C9	leave	C9	leave	C9	leave	C9	leave
...
42	inc edx	4A	dec edx	46	inc esi	4E	dec esi
C9	leave	C9	leave	C9	leave	C9	leave
C9	leave	C9	leave	C9	leave	C9	leave
...
43	inc ebx	4B	dec ebx	47	inc edi	4F	dec edi
C9	leave	C9	leave	C9	leave	C9	leave
C9	leave	C9	leave	C9	leave	C9	leave
...

Figure 7: A listing of all possibilities when the low nibble of the *leave* instruction is encoded-decoded

Referring to Figure 7, we see that all the instructions reflecting the incorrectly decoded *dec ebp* instruction are single-byte instructions and therefore do not take any subsequent bytes as operands. Further, most of these instructions manipulate the general purpose

registers and have orthogonal effects to the execution of the incorrectly decoded *leave* and *ret* instructions. It is also apparent that none of the resulting instructions cause immediate program termination when executed. Rather, they let the execution slip through to the following *leave* instruction. However, none of these instructions are control transfer instructions, and by following the execution flow and the stack pointer, we can see that in all cases, the execution is terminated upon the execution of one of the following *leave*, *ret* or, *hlt* instructions. Of course, when all the encoded instructions are correctly decoded, the execution continues normally without any change in the intended program semantics, which is desirable. We conclude that the *dec ebp* can safely be used as a critical instruction.

An interesting aspect of our technique is that, even when an instruction can not be used to enforce dependency with the return address, it may still be possible to use it as a critical instruction by setting up a dependency with some other instruction. In essence, we can create a complex dependency chain where the dependency from one instruction can be passed down to some other instructions. Furthermore, the same technique can be used to fabricate other critical instructions to extend the protection to all 32 bits of the return address. In fact, the *leave* instruction can be linked to multiple instances of the *dec ebp* instruction, and following our previous observation that the *dec ebp* instruction can be encoded in such a way that it decodes to a single byte instruction, it is possible to incorporate multiple instances of the *dec ebp* instruction into our sequence and use them as critical instructions.

4 IMPLEMENTATION

We developed a prototype implementation of RAID for the x86 architecture. The end product of our implementation is a C-compiler, *ucc_raid*, designed to run on Linux machines. Like its parent compiler, *ucc*[16], it only supports a subset of C-89 standard. We note that our implementation is only meant to serve as a *proof-of-concept*, and hence, inherently suffers from several limitations. One such limitation is that the implementation only protects 12 bits of the return address. More importantly, our implementation is not optimized in any way and therefore, exhibits significant performance overheads. Nevertheless, the implementation serves to show that RAID offers a solid protection against tampering with the return address without interfering with the intended program semantics.

4.1 Implementation Details

ucc_raid uses two stack: a regular stack and a code stack. Because we wanted to keep things simple with *ucc_raid*, we opted to split the main program stack into two and use a chunk as a code stack and the remaining chunk as a regular stack. The code stack is at 64 kilobytes from the regular stack, and this sets a 64-kilobytes size limitation on the regular stack. It is important to note that this is a soft limitation and can easily be adjusted by adjusting the total size of the main program stack, and more importantly, the offset to the code stack from the regular stack.

4.2 Modification to Prologue and Epilogue

ucc_raid modifies the prologue to encode three critical instructions: *ret*, *leave* and *dec ebp*. As discussed before, only those bits of the critical instructions that are known to be secure are encoded. The

instructions that are encoded in the prologue of a function are decoded in the same function's epilogue before being executed. Further, in an epilogue, *ucc_raid* also sets up the *hlt* sled in the code stack and adds other dependent instructions. Moreover, a fake function frame is created in the regular stack. All modifications made to a function's prologue and epilogue as implemented in *ucc_raid* can be found in the appendix of the paper.

4.3 Performance

To understand how implementing RAID impacts the compile-time performance of a compiler and run-time performance of an output executable, we ran benchmarks with our prototype implementation, *ucc_raid*, and compared results with its parent compiler, *ucc*. Our tests show that RAID has negligible compile-time overhead. However, the run-time overhead of the unoptimized implementation is significant. In our tests, we hit the runtime overhead of up to 900% which decreases proportionally with the decrease in number of function calls in the source program. Upon tuning the implementation and analyzing the execution time, we learned that the encoding and decoding operations themselves do not incur significant overhead, rather, jumping back and forth between the code located in two different segments, namely the code segment and the stack segment, is the source of the overhead. We strongly believe that allocating the space for the code stack in the code segment, rather than in the stack segment, significantly improves performance. This, however, requires an operating system modification for a secure and efficient implementation.

5 FUNCTIONAL COMPARISON WITH PRIOR WORKS

In Section 2 we discussed some of the existing defenses against return-manipulating attacks (and buffer overflow attacks, in general). Here, we provide a more in-depth comparison between our RAID approach and a related class of these existing defenses, highlighting the similarities/differences and the limitations/advantages.


Instruction Set Randomization (ISR): ISR randomizes the instruction set on a per-process basis. The similarity between ISR and our technique lies in the parameter that is transformed: a set of instructions. Another similarity lies in the intended effect: both techniques rely on the execution of invalid instructions to cause abnormal program termination in the course of an attack. However, there are also important differences. First, our technique does not randomize the instructions, but rather uses them as a trap. The direct consequence is that our approach, unlike ISR, is not compromised by the revelation of the key used for the transformation. Further, we do not randomize the whole instruction set, but rather we just encode/decode a small sequence of instructions. This makes our technique more practical.

RAD, Stack-Shield, Dual-Stack and Multi-Stack: RAD and similar countermeasures are based on the concept of isolating critical information away from the main stack. While RAID may not hold obvious advantage over RAD and similar countermeasures, it is important to understand that RAID is meant to illustrate a specific implementation of a more general technique. The underlying

data-instruction dependency technique, which RAID is based on, is extremely effective in the following setup: given some critical data and a non-recursive and non-reentrant code that is always executed right before the use of the critical data, the tampering with the critical data can be neutralized by enforcing dependency with the code. The same technique can be applied in the reverse application—to protect against tampering with the code. Notice that the requirement of a code stack is specific to RAID and not the underlying data-instruction dependency technique. In some cases, under certain assumptions, the requirement of a code stack can be slacked for alternate implementations of RAID. Under such implementations, the encoding/decoding operations are performed directly over the actual program code without moving the instructions to a separate executable stack. An example implementation, which assumes writable code segment, is provided in the appendix.

PointGuard: PointGuard encrypts/decrypts all code pointers by applying an XOR mask of a randomly generated key. Our technique bears two important similarities with PointGuard. First, the XOR masking/unmasking process used in our approach is similar to that used by PointGuard. Second, our approach relies on abnormal program termination to stop the attacker from taking control over the program, as does PointGuard. On the other side of the spectrum, the main difference between our technique and PointGuard lies in the parameter that is masked/unmasked: PointGuard masks the code pointers while our technique masks a set of critical instructions. Another important difference is the parameter used as a key for the masking process: PointGuard uses a randomly generated value, but our approach uses the return address of the caller function. In terms of security, unlike PointGuard, our approach does not rely on the secrecy of the key, or the unpredictability of the decoded value and is therefore immune to buffer-overread attacks and lucky guessing.

6 CONCLUSIONS

In this paper, we discussed a new technique to neutralize attacks that tamper with critical data. We introduced RAID as a specific implementation of our technique focused on attacks that manipulate the return address. RAID does not rely on complementary defenses such as ASLR, DEP and StackGuard and can be used to protect other critical data such as the frame pointer. Broadly speaking, RAID uses a sequence of critical instructions as a trap against tampering with the return address of a function. The trick is to ensure that the trap is triggered whenever any bit of the return address is modified. We demonstrated that RAID can be extremely effective against such modifications. Future work will be focused on the construction of trap gadgets, that are orthogonal to each other in the sense that each trap is triggered independently of whether the previous traps were triggered or not. The trap gadget architecture has the potential to ameliorate the effort involved in the trap design process, and should allow RAID to easily scale up to 64-bit architectures. Additional future work could focus on smart trap designs, that minimize the computational overhead incurred at run time. 

7 ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1527579 and 1619201.

REFERENCES

- [1] Martin Abadi, Mihai Budiu, Ulfr Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 340–353.
- [2] Steven Alexander. 2005. Defeating compiler-level buffer overflow protection. *The USENIX Magazine; login* (2005).
- [3] Tzi-cker Chiueh and Fu-Hau Hsu. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Distributed Computing Systems, 2001. 21st International Conference on*. IEEE, 409–417.
- [4] Marc L Corliss, E Christopher Lewis, and Amir Roth. 2005. Using DISE to protect return addresses from attack. *ACM SIGARCH Computer Architecture News* 33, 1 (2005), 65–72.
- [5] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, Vol. 12. 91–104.
- [6] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.
- [7] Thurston HY Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 555–566.
- [8] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection.. In *USENIX Security Symposium*, Vol. 2014.
- [9] Michael Frantzen and Michael Shuey. 2001. StackGhost: Hardware Facilitated Stack Protection.. In *USENIX Security Symposium*, Vol. 112.
- [10] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 575–589.
- [11] Jason Hise, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. 2012. ILR: Where'd my gadgets go?. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 571–585.
- [12] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 272–280.
- [13] Chongkyun Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 339–348.
- [14] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. 2011. Code pointer masking: Hardening applications against code injection attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 194–213.
- [15] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.
- [16] sheisc. [n.d.]. UCC: A Lightweight Open-Source C Compiler for Research and Education. <https://github.com/sheisc/ucc162.3>, Last accessed on 2019-07-11.
- [17] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Lieben, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 574–588.
- [18] Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The Effectiveness of Instruction Set Randomization.. In *USENIX Security Symposium*, Vol. 10.
- [19] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*. ACM, 1–8.
- [20] Perry Wagle, Crispin Cowan, et al. 2003. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*. Citeseer, 243–255.
- [21] Yoav Weiss and Elena Gabriela Barrantes. 2006. Known/chosen key attacks against software instruction set randomization. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 349–360.
- [22] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 17.
- [23] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. 2006. Extended protection against stack smashing attacks without performance loss. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE, 429–438.
- [24] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 559–573.
- [25] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries.. In *USENIX Security Symposium*. 337–352.

A MODIFICATION TO PROLOGUE AND EPILOGUE WITH *ucc_raid*

The following code shows all modifications made to a function's prologue as implemented in *ucc_raid*.

```
# -(65536)(%esp) to -(65536+3)(%esp) reserved for a hlt(0xF4) sled
# Encode the higher nibble of the ret instruction (0xC3)
movb (%esp), %bl
andb $0x0f, %bl
xorb $0x0c, %bl
shlb $4, %bl
movb %bl, -(65536+4)(%esp)
# Encode the lower nibble of the leave instruction (0xC9)
movb (%esp), %bl
shrb $4, %bl
xorb $0x09, %bl
movb %bl, -(65536+5)(%esp)
# -(65536+6)(%esp) is used in the epilogue for another leave
instruction
# Encode the lower nibble of the dec ebp instruction (0x4D)
movw (%esp), %bx
shrw $8, %bx
xorb $0x0D, %bl
movb %bl, -(65536+7)(%esp)
# Original prologue
pushl %ebp
movl %esp, %ebp
```

The following code shows all modifications made to a function's epilogue as implemented in *ucc_raid*.

```
# Set up a hlt(0xF4) sled
movl $0xF4F4F4F4, -(65536-1)(%ebp)
# Decode the higher nibble of the ret instruction
movb 4(%ebp), %bl
andb $0x0f, %bl
shlb $4, %bl
xorb %bl, -(65536)(%ebp)
# Overwrite the lower nibble of the ret instruction
andb $0xf0, -(65536)(%ebp)
orb $0x03, -(65536)(%ebp)
# Decode the lower nibble of the leave instruction
mov 4(%ebp), %bl
shrb $4, %bl
xorb %bl, -(65536+1)(%ebp)
# Overwrite the higher nibble of the leave instruction
andb $0x0f, -(65536+1)(%ebp)
orb $0xC0, -(65536+1)(%ebp)
# Add another leave instruction
movb $0xC9, -(65536+2)(%ebp)
# Decode the lower nibble of the dec ebp instruction
movw 4(%ebp), %bx
shrw $8, %bx
andb $0x0f, %bl
xorb %bl, -(65536+3)(%ebp)
# Overwrite the higher nibble of the dec ebp instruction
andb $0x0f, -(65536+3)(%ebp)
orb $0x40, -(65536+3)(%ebp)
# Shift up the decoded instructions and use the freed space to
expand the size of the hlt sled
movl -(65536+3)(%ebp), %ebx
movl %ebx, -(65536+7)(%ebp)
movl $0xF4F4F4F4, -(65536+3)(%ebp)
# Checkpoint (Edit to add a Breakpoint(0xCC) if required)
movb $0x90, -(65536+8)(%ebp)
# Save the pointer to the decoded instructions
leal -(65536+8)(%ebp), %ebx
# Create a fake function frame
pushl $0x00000000
pushl %ebp
movl %esp, %ebp
```

```
pushl $0x00000000
incl %ebp
# Jump to the decoded instructions
pushl %ebx
ret
```

B AN ALTERNATIVE IMPLEMENTATION OF RAID

In the following example, we show an example implementation of RAID which does not require a second (executable) stack. The critical data to protect is the return address of the function, *somefunc*. This implementation assumes that the code segment is writable and that *somefunc* is a non-recursive and non-reentrant function.

```
somefunc:
# Get the return address in bx register
mov bx, WORD PTR [esp]
# Encode the leave and ret instructions with bits 0-15 of the return
address
xor WORD PTR [address_of_leave_ret], bx
# The usual function prologue
push ebp
mov ebp, esp
...
...
# The modified epilogue
# Get the return address in bx register
mov bx, WORD PTR 4[ebp]
# Decode the leave and ret instructions with bits 0-15 of the return
address
xor WORD PTR [address_of_leave_ret], bx
leave
ret
```

Next, we consider the *add* function from before and provide a concrete example.

```
add:
# Get the return address in bx register
mov bx, WORD PTR [esp]
# Encode the leave and ret instructions with bits 0-15 of the return
address
xor WORD PTR [add_epi], bx
# The usual function prologue
push ebp
mov ebp, esp
sub esp, 4
mov edx, DWORD PTR 8[ebp]
mov eax, DWORD PTR 12[ebp]
add eax, edx
mov DWORD PTR -4[ebp], eax
mov eax, DWORD PTR -4[ebp]
# The modified epilogue
# Get the return address in bx register
mov bx, WORD PTR 4[ebp]
# Decode the leave and ret instructions with bits 0-15 of the return
address
xor WORD PTR [add_epi], bx
add_epi:
leave
ret
```