University of Cincinnati

Date: 12/6/2013

I, Matthew L White , hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Science.

It is entitled:

Defeating ROP Through Dynamically Encrypted Return Addresses

Student's name: **Matthew L White**

This work and its defense approved by:

Committee chair: John Franco, Ph.D.

Committee member: Prabir Bhattacharya, Ph.D.

Committee member: Paul Talaga,

9047

Defeating ROP Through Dynamically Encrypted Return Addresses

A thesis submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

Master of Sciences

in the School of Computing Sciences and Informatics

of the College of Engineering and Applied Sciences

by

Matthew L. White

B.S. Purdue University

May 2008

Committee Chair: John Franco, Ph.D.

Abstract

Operating systems have been evolving to provide defenses, including Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR), for some common attack vectors. However, as defenses increase, so does the cleverness of attackers. A more recent attack vector seeks to bypass the most common defense mechanisms by needing only to execute instructions that already exist in the program being run. This method, Return Oriented Programming (ROP), has allowed for new exploitation vectors on even the most up-to-date operating systems. A ROP attack is performed by an attacker who first gains control of a program and then executes a series of short instructions (gadgets) that already exist within the running application. This paper introduces a new method to mitigate ROP attacks that removes the control an attacker has in selecting and executing arbitrary portions of code. This is accomplished through the use of dynamic analysis and instrumentation to both identify when a program calls a function and to encrypt the return address that will be used.

Acknowledgements

The author would like to thank his advisor, Dr. John Franco, for his constant guidance, as well as committee members Dr. Prabir Bhattacharya and Dr. Paul Talaga for their support. The author would also like to thank his dear friend Richard Wartell, whose expertise is a constant source of inspiration. The author extends his gratitude to his co-workers Ryan Smith, Creighton Long, Wyatt Neal, Eric Schwartz and Dan Searles, whose insights and feedback helped to make this research possible.

Most importantly, the author extends his heartfelt thanks to his wife and family, whose love and encouragement is never ending.

Contents

Abstract **ii**

Acknowledgements **iv**

List of Figures **viii**

1 Introduction **1**

1.1 Background 1

1.1.1 Return Oriented Programming 2

1.2 Goals 5

1.3 Organization 5

2 Related Work **6**

2.1 Hardware Protections 6

2.2 Compilation Protections 7

2.3 Instrumentation Protections 8

3 Using Binary Instrumentation for Return Address Encryption **10**

3.1 The Case for Dynamic Instrumentation 10

3.1.1 DynamoRIO Binary Instrumentation Framework 11
3.2 Assumptions 11
3.3 Description 12
3.3.1 Basic Block Callbacks 14
3.3.2 `CALL` Instruction Instrumentation 15
3.3.3 `RET` Instruction Instrumentation 16
3.3.4 Basic Block Instrumentation 17
3.4 ROP Protection Example 19
3.5 Encryption Keys 20

4 Challenges and Limitations **22**
4.1 x86 Challenges 22
4.2 Transparency 24
4.3 A Refined Approach 24
4.3.1 Implementation 25
4.3.1.1 `CALL` Instruction Instrumentation 25
4.3.1.2 Memory References 26
4.3.1.3 Address List Management 27
4.3.2 Analysis and Impact 29
4.3.3 Limitations 30

5 Performance and Analysis **31**
5.1 Benchmarks 31
5.2 Performance 32
5.3 Analysis 34

6 Conclusion **38**

6.1 Future Work 39

6.1.1 Optimized Instrumentation 39

6.1.2 Memory Writes to Encrypted Addresses 40

6.1.3 64-bit Architecture Support 40

6.1.4 Multiple Keys 40

Bibliography **42**

List of Figures

1.1 Anatomy of a ROP Attack 4

3.1 `CALL` Instruction Transformation 15

3.2 `RET` Instruction Transformation 17

3.3 Basic Block Transformation 18

3.4 Simulated Vulnerable Example 19

4.1 `CALL` Without `RET` 23

4.2 PIC Code Base Address 23

4.3 The Need For Address List Management 28

4.4 `RET` Referencing a Non-Encrypted Value 29

4.5 `TEST` With an Encrypted Value 29

5.1 Average Time Per Run (seconds) 33

5.2 Time Impact Relative To Base System 33

5.3 Average Instructions Per Basic Block 35

5.4 Number of Basic Blocks 36

5.5 Percentage of `CALL` and `RET` Instructions 37

Chapter 1

Introduction

1.1 Background

The world is more of an interconnected hub than ever before with nearly 40% of the world population (and over three quarters of the population of developed nations) having access to the internet [11]. People now have access to an endless amount of content and services located worldwide. As more services are being moved to the web, however, the threat from malicious users continues to increase. These threats come in many forms, from the occasional script kiddie to the more dangerous hacktivism, and even in the form of a large botnet able to pool resources on demand that can be controlled by a single person [9, 14]. Now there are even researchers shining light on alleged government sanctioned intrusions [13]. The easiest targets for these groups of malicious users are applications that are not designed with security in mind or that are written improperly such that they may be exploited.

CPU manufacturers and operating system developers strive to protect their prod-

ucts from applications that are not written securely. One of the most common protection mechanisms included in modern operating systems is Data Execution Prevention (DEP). With DEP, an operating system and CPU cooperate to enforce a policy that denies execution of data that is contained in marked segments of memory. This helps to prevent attackers from gaining control by overwriting values on an application's stack during a buffer overflow attack and then forcing the application to continue execution from the newly written values [2].

Address Space Layout Randomization (ASLR) is another common security method used in modern operating systems. ASLR randomizes the starting address of memory into which libraries are loaded and even the location of an application's stack [1]. This is useful for preventing return-to-libc attacks which are able to bypass DEP restrictions. In return-to-libc, an attacker is able to modify the return address of a function so that the program resumes execution in another function that is already loaded (typically within `libc`) [22]. With ASLR in place, the address of the desired function is not constant, limiting success rates of the attack.

Although these mechanisms help reduce the attack space, a relatively new offensive technique called Return Oriented Programming (ROP) is able to craft malicious code in a way that can defeat some of the newest attack prevention methods implemented in modern operating systems including DEP and even ASLR.

1.1.1 Return Oriented Programming

ROP attacks are similar to return-to-libc attacks in that they rely on code that already exists. However, ROP attacks do not rely upon complete functions (e.g. calling

the `libc` function `system()`). Additionally, it has been shown that since x86 has a variable length instruction set, an attacker does not even need to rely upon instructions that were originally intended by the compiler [20]. An attacker performing a ROP attack must first identify sequences of assembly instructions within an application that accomplish a specific goal. These are normally very short sequences, referred to as *gadgets*, of just a few instructions that end with a control transfer instruction (typically a `RET` instruction). Once an attacker has identified enough gadgets, they can be chained together to perform arbitrary tasks that were unintended by the original application. This is normally done by utilizing a buffer overflow to control the call stack and overwriting it with the addresses of the desired gadgets. Note that unlike normal buffer overflows where an attacker's shellcode is written to the stack, this overflow is used only to write a sequence of addresses that contain the location of gadgets. When these addresses are chained together appropriately, arbitrary instruction sequences can be executed [20]. This exploit vector is summarized in Figure 1.1.

The defenses discussed in Section 1.1 are not effective in defeating ROP. DEP is bypassed because ROP does not require any code to be executed from the stack. Instead it only needs to be able to write the chained gadgets to the stack and then have those addresses read later on (such as when a `RET` instruction is encountered at the end of a gadget). Also, ASLR has been been shown not to be sufficient in defeating ROP. In their paper, Schartz et, al. [19] show that major operating systems, including Windows, OS X, and Linux, each have their own deficiencies implementing ASLR that allow for exploitation.

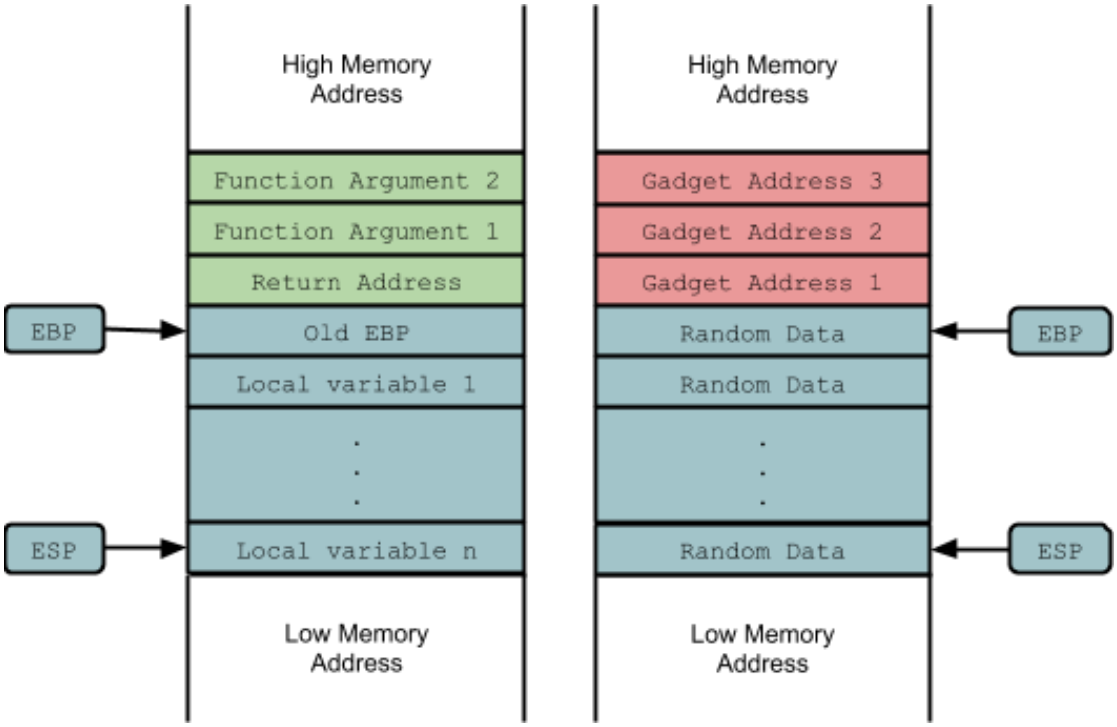

Figure 1.1: Anatomy of a ROP Attack: The left side denotes a normal stack frame for a function. Labeled are the bounds of the stack frame with the `EBP` and `ESP` registers as well as the layout of the passed-in function arguments and local variables. On the right, a ROP attack has been performed. Using normal buffer overflow techniques, the attacker first overwrites all data on the stack leading up to the return address (local variables and the old `EBP` value) with random data. Afterwards, the attacker then chains together the addresses of the gadgets. In this manner, when a function completes and the `RET` instruction is encountered, program control is transferred to *Gadget Address 1.* As each `RET` instruction is encountered at the end of each gadget, the next *Gadget Address* will be used.

1.2 Goals

The goal of this thesis is to explore a new ROP protection method that uses on-the-fly encryption and decryption of return addresses at runtime. Additionally, because of the high cost of releasing software and the large amount of legacy software that is no longer maintained, this thesis is focused on an approach that requires neither access to source code nor recompilation. Finally, the focus of this thesis will not be to stop buffer overflows and allow an application to recover, but rather prevent a malicious user from launching arbitrary commands and gaining control of a system. This is done by utilizing the DynamoRIO Dynamic Instrumentation Tool Platform. Details about DynamoRIO can be found in Section 3.1.1.

1.3 Organization

This thesis is organized as follows: Chapter 2 reviews related work in the area of ROP mitigation techniques. Chapter 3 introduces the proposed method to help eliminate ROP attacks. Chapter 4 discusses the limitations of the method described in the previous chapter and presents improvements to enhance applicability. Chapter 5 investigates the performance of the proposed method. Chapter 6 presents a summary and conclusion of the research as well as an examination of future work.

Chapter 2

Related Work

There have been various defense mechanisms studied and proposed to defeat ROP attacks. Suggested methods have a wide ranging focus, including hardware modifications, compilation methods, and instrumentation protections.

2.1 Hardware Protections

SmashGuard is a hardware-based implementation to protect return addresses that uses an additional hardware stack that is only accessible from the privileged kernel level. When a `CALL` assembly instruction is encountered, the return address is pushed onto this new stack. `RET` instructions compare the addresses on each of the stacks and ensure no changes were made [17].

In a paper by Xu, et al. [24], two additional approaches are presented. In the first approach, the authors propose splitting the stack into two: a control stack to be used for function return addresses only, and a data stack for everything else. The second method described, called Secure Return Address Stack (SRAS), pushes a copy

of the function return address onto the "return address stack" when a function is called. Upon function return, when the `RET` instruction is fetched, the next value in the address stack is popped. Since a duplicate copy of the return address is stored, any manipulations can be detected. SRAS is able to perform various counter-measures (such as raise an exception) when a mismatch occurs.

Hardware based approaches can offer great protection with little or no overhead incurred. However, the barriers to entry in the hardware market are not normally feasible for short term, widespread adoption.

2.2 Compilation Protections

StackGuard is a protection method that helps to prevent buffer overflows through the placement of a canary word next to the return address on the stack. It then monitors this canary value for changes and can flag overwrites as malicious activity. This helps to control generic buffer overflow attacks as well as ROP attacks [7]. Canaries can still be susceptible to certain types of buffer overflows. One such case is when an attacker knows the canary value before hand. Then, when performing the overflow, an attacker overwrites the canary with the same value that previously existed. Another deficiency of canaries is that an attacker can simply overwrite other application pointers (e.g. function pointers) and leave the return address alone [5].

Return Address Defender (RAD) is another compilation-based protection method that creates a copy of the return address for a function into a section of memory that is surrounded by regions marked as read-only. RAD also modifies a function's prologue and epilogue and checks the return address against the secured copy prior to

the jump back to the caller [10].

PointGuard offers another technique that uses a modified GCC compiler to encrypt all pointer values until they are dereferenced. This does not prevent attackers from corrupting data, but does prevent them from gaining control of an application [6].

Another tool, G-Free, does not directly modify a compiler, but does modify the compilation process. It introduces a stage that accepts the output of the compiler and performs some processing before sending the output along to the assembler. G-Free seeks to provide two protection mechanisms. First, it removes unintended or unaligned instructions by utilizing `NOP` instructions and then it fixes up address offsets that may have been affected by the insertion of additional instructions. G-Free also takes steps to encrypt return addresses and decrypt them immediately before the `RET` instruction executes [15].

The above methods offer solutions that provide good protection mechanisms with generally low overhead. However, because they rely on either modified compilers or compilation processes, they require recompilation from source. This is normally not feasible, especially given the large amount of software that is either not open source or no longer maintained.

2.3 Instrumentation Protections

One technique that has been explored is using dynamic instrumentation to create shadow stacks for storing copies of certain application stack values. Two proposals utilizing this method are TRUSS and ROPDefender. These proposed solutions use a shadow stack to store a copy of return addresses as they are pushed onto an

application's stack via a `CALL` instruction. When a `RET` instruction is encountered, the instrumentation client compares the return address in the application's stack to what exists in the shadow stack. If these differ, they assume something malicious happened. TRUSS uses the DynamoRIO instrumentation framework while ROPDefender uses Pin [21, 8].

In a paper by Wartell [23], a method called STIR is presented which uses static instrumentation to randomize the location of basic blocks in a binary. STIR works by first statically disassembling a binary into its basic blocks and creating a lookup table indicating where each block is located. A new binary is created from the original that modifies the original binary header to include a library import that will, at each load of the application, use the lookup table to randomize the location of all the basic blocks. Wartell has shown that this adds more entropy to a binary than the more standard ASLR approach discussed in Section 1.1.

Static disassembly and instrumentation can allow a mitigation technique to perform the bulk of the processing tasks prior to an application running. This means that when an application is run after the static instrumentation applied, the application incurs only a small amount of overhead. However, completely accurate disassembly of arbitrary x86 applications is not always possible due to variable length and unaligned instructions, interleaving of code and data, and jump addresses that are computed at runtime [23]. This can lead to potential problems when an application runs if all code paths are not modified correctly.

Chapter 3

Using Binary Instrumentation for Return Address Encryption

This chapter presents an approach to ROP attack defense using the DynamoRIO dynamic binary instrumentation framework to encrypt and decrypt return addresses at runtime.

3.1 The Case for Dynamic Instrumentation

Chapter 2 presented an overview of several methods previously researched - each with their advantages and disadvantages. One of the main goals of this thesis, as stated in Section 1.2, is to enable users to protect their systems without relying upon outside maintainers of software or necessitating the purchase of new hardware. This leaves out any approach that requires either specialized hardware or a recompilation of source code. Additionally, static disassembly was left out as an approach due to complications shown in Section 2.3. This leaves dynamic instrumentation as an ideal

candidate for the type of protection method suggested in this thesis.

3.1.1 DynamoRIO Binary Instrumentation Framework

DynamoRIO (DR) is an open source binary instrumentation framework which provides an application programming interface (API) that allows for complete code manipulation of an application at runtime. Using the API, developers write *clients* that register for various events such as the creation of a basic block or the creation of a new thread by the application [4]. This allows a client developer to manipulate, add, or remove instructions prior to being executed by the processor. When a user wants to run a program under control of a DR client, they first invoke the DR binary and pass both the desired client and the target application as parameters. This allows DR to control the application and allow the client to manipulate the runtime dynamically. DR was chosen as the instrumentation framework due to efficient, low-overhead implementation [18] and its comprehensive and easy-to-use API. Furthermore, DR and clients written for it can run on both Windows and Linux, allowing for a broad range of application targets.

3.2 Assumptions

The following assumptions were made for this approach:

1. **Exploitable:** An adversary is able to launch a ROP attack on the given binary.

2. **Black Box:** There is no access to source code for the given binary. This could be because the application is no longer supported/maintained, or is a closed

source program where code is proprietary.

3. **Isolated:** The adversary does not have physical access to the machine executing the applications. More on this assumption is detailed in Section 3.5.

3.3 Description

As described in Section 1.1.1, attackers must gain control of the call stack of a program to perform a successful ROP attack. Once this is done, they can place the addresses of the identified gadgets to call short arbitrary instruction sequences that end with a `RET` instruction. In this manner, attackers can rely on the functionality of the x86 architecture to continuously pop the next gadget off the top of the stack and continue execution.

This thesis uses the dynamic instrumentation capabilities provided by DR to encrypt return addresses on an application's stack at runtime. By doing this, when a `RET` instruction is encountered, the address on the stack will first be decrypted before program control is transferred.

In the typical ROP scenario, an attacker will overwrite the return address on the stack with the address of the first gadget (as seen in Figure 1.1). Now, however, the overwritten address will first be decrypted upon function return. Since attackers do not know the encryption key, they will not be able to properly encrypt the address of their intended gadget such that it will be decrypted to the appropriate address. When the malicious address is decrypted, program control is transferred to an undesired location (most likely leading to an eventual crash of the application). A discussion of using and protecting encryption keys can be found in Section 3.5.

To implement this thesis, a DR client was written using the framework's API. This is a well-documented process on DR's website [3]. By writing a DR client and utilizing the DR client API, this program will perform the following high-level steps:

1. Utilize the `drmgr` extension to register callbacks for basic block creation events.

    - This is one of the primary extensions exposed by DR to its clients that allows for arbitrary instruction manipulation and it works on the level of *basic blocks*. These are defined by DR as being a set of instruction sequences that end with a control transfer instruction. Each basic block has one entry point and one exit point [4]. This means that most basic blocks will end with either a `CALL`, `JMP`, or `RET` instruction.
    - The callbacks will allow the client to take control every time a basic block is about to be inserted into DR's cache. DR will pass the client several parameters, including a list of instructions in the basic block that the client can inspect and manipulate.
    - `drmgr` provides a mechanism for managing thread local storage (TLS) to allow the client to preserve information across callbacks for each thread within the running application. In a client, this takes the form of a `C` structure whose members are defined by the client developer.

2. Inspect the instructions of each basic block for a `CALL` or `RET` instruction.

    - `CALL:` An x86 `CALL` instruction is instrumented by inserting code just prior to the `CALL` to load the TLS field (described in Step 1), and set a flag that a `CALL` has been encountered. This flag will be utilized in Step 3. Refer to

Section 3.3.2 for a detailed description.

- `RET`: an x86 `RET` instruction is instrumented to insert some additional instructions prior to executing the return. This allows for cleanup of the stack manipulation that occurred when the `CALL` instruction was instrumented. Refer to Section 3.3.3 for a detailed description.

3. Insert code at the start of each basic block to check if the block has been reached by a `CALL` instruction, and if so, encrypt the return address. Refer to Section 3.3.4 for a detailed description.

3.3.1 Basic Block Callbacks

The basic block callback routines provided by DR through `drmgr` are only called by DR whenever a basic block is about to be entered into DR's cache. DR makes use of this cache to dramatically reduce the amount of overhead incurred by a typical dynamic instrumentation framework [4]. Because DR will not execute the client's callback routine when pulling a basic block from cache, a client typically can only instrument a basic block once.

The use of the cache means that it is important for the client to make a distinction between instrumentation time and runtime. Instrumentation time describes client code that is run during one of the callbacks. Runtime describes application code that is executed natively by the CPU after instrumentation has occurred. Since instrumentation time normally only occurs once per basic block, all instructions must be instrumented on the first pass. Any modifications made to the basic block will be stored in the cache along with the original basic block contents, and will be executed

each time the block is pulled from the cache [3].

3.3.2 `CALL` Instruction Instrumentation

When a `CALL` instruction is encountered, a flag is set in the TLS so that the next basic block can determine whether or not the return address must be encrypted. This is accomplished by using the DR API convenience function `drmgr_insert_read_tls_field` to load the TLS into a register. Next, the API function `INSTR_CREATE_mov_imm` is used to move the value `1` (true) into the appropriate member of the TLS structure to indicate a `CALL` has occurred. Refer to Section 3.3.4 for how this flag is checked.

The last step taken into consideration for instrumenting `CALL`s, is that loading the TLS field into a register effectively clobbers the original value of that register. Because of this, the client must first be sure to insert instructions to save a register to scratch space prior to loading the TLS. DR provides a mechanism for spilling registers using what it calls *spill slots*. Original register data can be saved to these spill slots and later restored with the API routines `dr_save_reg` and `dr_restore_reg`.

```
1 call    0x77379f4b
```

```
1 mov     dword ptr [fs:0x00000eec], eax
2 mov     eax, dword ptr [fs:0x00000ef4]
3 mov     eax, dword ptr [eax+0x00000248]
4 mov     eax, dword ptr [eax]
5 mov     eax, dword ptr [eax+0x04]
6 mov     dword ptr [eax+0x01], 0x00000001
7 mov     eax, dword ptr [fs:0x00000eec]
8 call    0x77379f4b
```

Figure 3.1: `CALL` Instruction Transformation: The figure on the left represents the application's original `CALL` instruction. The figure on the right shows the instrumentation applied by the client.

Figure 3.1 shows an example transformation of a `CALL` instruction. Lines 1 and 7 show the effect of `dr_save_reg` and `dr_restore_reg`. Lines 2 - 5 load the flag that will be set by first loading the TLS field into `EAX`, and then using offsets to load the address of the flag field within the TLS. Finally, line 6 moves the immediate value `1` (true) into `EAX` to set the flag.

3.3.3 `RET` Instruction Instrumentation

In x86, the `RET` instruction pops the return address off the top of the stack and sets the `EIP` register to this value. This tells the processor the address of the next instruction to execute [12]. Because return addresses are being encrypted, the value on the stack must first be decrypted to recover the original value in order for the `EIP` register to be set appropriately.

To do this, an instruction is inserted directly before the `RET` that will decrypt the value just prior to being used. The DR API function `INSTR_CREATE_xor` is used to create an `XOR` instruction to perform an exclusive-or between the encryption key and the encrypted return address value at the top of the stack. As with the `CALL` instruction, care must be taken to ensure application state is not affected. Since the `XOR` instruction can affect the `EFLAGS` register, the API functions `dr_save_arith_flags_to_xax` and `dr_restore_arith_flags_from_xax` are used. Note that because these functions clobber `EAX`, it must first be spilled and then restored afterwards using the method described in the previous section.

Figure 3.2 shows an example of a `RET` instruction transformation. Lines 1 and 7 show the effect of `dr_save_reg` and `dr_restore_reg`. Lines 2, 3, 5, and 6 save

```
1 ret     0x000c
```

```
1 mov   dword ptr [fs:0x00000eec], eax
2 lahf
3 seto  al
4 xor   dword ptr [esp], 0x12345678
5 add   al, 0x7f
6 sahf
7 mov   eax, dword ptr [fs:0x00000eec]
8 ret   0x000c
```

Figure 3.2: `RET` Instruction Transformation: The figure on the left represents the application's original `RET` instruction. The figure on the right shows the instrumentation applied by the client.

and restore x86 flags. Line 4 performs the XOR decryption by applying the key (`0x12345678`) to the return address that is located at the top of the stack.

3.3.4 Basic Block Instrumentation

At the start of each basic block, instructions must be inserted to check if the block was reached via a `CALL` instruction, meaning that the return address must be encrypted. This requires several steps:

1. Load the TLS field (once again using `drmgr_insert_read_tls_field`).
2. Load the flag field into `ECX` by inserting a `MOV` instruction.
3. Check if the flag used in Section 3.3.2 is set by inserting a `JECXZ` instruction.
    - If the flag is not set, jump to the label inserted in Step 6.
4. Clear the TLS flag.
5. Insert an `XOR` instruction to encrypt the return address.

- **Note:** Immediately after a `CALL`, the return address is at the top of the stack and is easily accessed using `ESP`.

6. Insert a label that the `JECXZ` instruction from Step 3 will use to jump to if the TLS flag was not set, effectively bypassing encryption.

```
1  mov     dword ptr [fs:0x00000ee8], ecx
2  mov     ecx, dword ptr [fs:0x00000ef4]
3  mov     ecx, dword ptr [ecx+0x00000248]
4  mov     ecx, dword ptr [ecx]
5  mov     ecx, dword ptr [ecx+0x04]
6  mov     ecx, dword ptr [ecx+0x01]
7  jecxz   <label>
8  mov     dword ptr [fs:0x00000eec], eax
9  lahf
10 seto    al
11 mov     ecx, dword ptr [fs:0x00000ef4]
12 mov     ecx, dword ptr [ecx+0x00000248]
13 mov     ecx, dword ptr [ecx]
14 mov     ecx, dword ptr [ecx+0x04]
15 mov     dword ptr [ecx+0x01], 0x00000000
16 xor     dword ptr [esp], 0x12345678
17 add     al, 0x7f
18 sahf
19 mov     eax, dword ptr [fs:0x00000eec]
20 <label>
21 mov     ecx, dword ptr [fs:0x00000ee8]
```

Figure 3.3: Basic Block Transformation: These instructions are inserted into every basic block to determine if it was entered via a `CALL` instruction by checking a flag in the TLS.

Figure 3.3 shows the instructions inserted at the beginning of every basic block. Lines 1 - 6 save the `ECX` register and load the TLS flag field. If the field was not set, its value will be `0`, causing the application to jump from line 7 to line 20 (note: to improve readability, memory addresses were removed and the mnemonic "<label>" was used). If the TLS flag field was set, execution continues to line 8. Lines 8 - 10 save another register as well as `EFLAGS`. Lines 11 - 15 clear the TLS flag field and line

16 performs XOR encryption on the return address (using the key `0x12345678`). The remainder of instructions on lines 17 - 21 restore saved flags and registers.

3.4 ROP Protection Example

Now that the protection mechanism has been discussed, an example program will be reviewed to show that the method fundamentally works. Figure 3.4 is an example program that simulates a ROP attack by having a function modify its own return address and has been adapted from the article "Smashing The Stack For Fun And Profit" [16].

```
1  void foo(int a) {
2      char buffer1[4];
3      char *ret;
4
5      ret = buffer1 + 8;
6      (*ret) += 8;
7  }
8
9  void main() {
10   int x;
11
12   x = 0;
13   foo(0x1234);
14   x = 1;
15   printf("x = %d\n",x);
16 }
```

Figure 3.4: Simulated Vulnerable Example

In Figure 3.4, the `main` function first sets the variable `x` to 0, and then calls `foo`. Inside of `foo`, the variable `ret` is set to the address of the return address on the stack by offsetting the local variable `buffer1` (as seen in Figure 1.1, local variables are placed on the stack 8 bytes below the return address). Finally, the `ret` variable

is dereferenced and has its value increased by 8. This effectively forces `foo` to no longer return to the expected location within `main` (line 14: `x = 1;`), and instead skips this line of code and returns directly to the `printf` call. Because of the skipped assignment, the program now outputs "`x = 0`".

When the code of Figure 3.4 is run under control of the on-demand encryption client, the program starts off behaving normally until `foo` attempts to return. When this happens, it first decrypts the return address value on the stack. Since the encrypted value was modified, the return address is no longer accurate when it is decrypted, leading to undefined behavior (most likely a segmentation fault).

3.5 Encryption Keys

The description in Section 3.3 shows how an adversary will be thwarted by attempts to manipulate the return address. Any return address value that a malicious user writes to the stack will first be decrypted to a new value, resulting in the inability to gain control of the program. Although effective, it should be noted that in 32-bit address space the available number of keys is 2^{32}, or about 4.3 billion. This is a relatively small number that is susceptible to a brute-force attack.

Protection of encryption keys is vital to the reliability of this system. An attacker should not be able to obtain or derive key information. DR and its clients execute in a separate process from that of the instrumented application. Additionally, it is an assumption of this thesis that an attacker does not have physical access to the machine running the application. Major operating systems use strict access control mechanisms to protect processes from reading and writing the data of other processes

without appropriate permissions. An attacker attempting to control a system through a ROP style attack would crash an application before getting to the point needed to read memory from other processes. Because of this, an attacker would not be able to obtain key information without first compromising the system in some other fashion, rendering the need for obtaining the encryption key moot.

It should also be noted that since the assumption is made that the adversary does not have physical access to the machine, a system administrator could employ external protection mechanisms to reduce an attacker's chance of a successful brute force attack. Some options would be to throttle connections to remote services or to enforce strict firewall filters.

There are some considerations that could be made to expand the available key space while also increasing the number of keys used within a process to reduce the ability of an attacker to employ brute force tactics, but these are left as a subject of future work. This topic is explored in Section 6.1.

Chapter 4

Challenges and Limitations

The method proposed in Chapter 3 presents a straightforward approach for implementing encrypted return addresses and dealing with on-demand requirements. Although some good results are obtained (as presented in Chapter 5), the x86 architecture presents many challenges that prevent the previous method from being a general solution. These challenges are discussed here. Also, a new approach to address these limitations is considered and analyzed.

4.1 x86 Challenges

x86 is a variable length, complex instruction set, and many instructions are able to reference memory indirectly through either a register, the stack, a constant address, or a scaled value [12, 23]. It is perfectly acceptable for an application, during its normal operation, to read from or even write to its own call stack. In fact, this can be a common scenario for many types of applications. In a paper by Bruening, some of these scenarios are listed [4]. Among them:

1. `CALL` **without** `RET`**:** Useful for finding the address of the current instruction:

```
1                call   next_instr
2  next_instr:   pop eax
```

Figure 4.1: `CALL` Without `RET`

2. **Obtaining a Base Address:** Useful for PIC code determining the base address of the application:

```
1            call   PIC_base
2            ...
3 PIC_base: mov    ebx, [esp]
4            ...
5            ret
```

Figure 4.2: PIC Code Base Address

These two scenarios help show the limitations of the method described in Chapter 3. By applying the previously described method to scenario 1, instructions are inserted prior to the `CALL` to indicate that the next basic block should encrypt the return address. The next basic block, which contains the code from line 2, will have instructions inserted at the beginning to perform encryption first. When the `POP` instruction executes, the `EAX` register will hold an encrypted value.

Scenario 2 will encounter a similar situation. The basic block starting at the label `PIC_base` will have instructions inserted that will encrypt the return address because the block is entered from a `CALL` instruction. The `EBX` register will contain an encrypted value once the `MOV` instruction on line 3 executes since the instructions to decrypt the return address will not be inserted until the end of the basic block (just

before the `RET` instruction).

In these scenarios, if the application attempts to use the encrypted value, the behavior is undefined. In many cases though, the application will be attempting to access memory it does not have permission to access, most likely causing a segmentation fault.

4.2 Transparency

There are steps taken by the encryption client to keep its modifications to the application transparent. This includes ensuring original register contents are saved prior to clobbering their values, as well as saving and restoring the x86 `EFLAGS` register when inserted instructions may affect flag values. However, complete transparency is not maintained since an application's stack is manipulated. As seen in the previous section, this can lead to unintended consequences and is a major drawback of the current implementation.

4.3 A Refined Approach

Because of the challenges listed in Sections 4.1 and 4.2, many real world programs cannot run under the previously described method. A new approach, considered below, uses the following basic steps to help address these limitations:

1. Utilize the `drmgr` extension to register callbacks for basic block creation events.
    - This step is identical to the previous method.

2. Inspect the instructions of each basic block for a `CALL` instruction or any instruction that reads from memory.

    - **CALL:** This step is also identical to the previous method, however additional bookkeeping steps are taken to track encrypted return addresses. See Section 4.3.1.1 for a detailed description.
    - **Memory Reference:** Any instruction that references memory must be inspected to determine if the address referenced must first have the referenced value decrypted (note that this includes `RET` instructions). This is done by comparing the referenced memory address to the list of encrypted addresses that is constructed when encountering `CALL` instructions. Details of this step are in Section 4.3.1.2.

3. Insert code at the start of each basic block to check if the block has been reached by a `CALL` instruction, and encrypt if necessary.

    - This step is identical to the previous method. See Section 3.3.4 for details.

4.3.1 Implementation

The following sections examine the specific steps needed to implement the refined approach.

4.3.1.1 `CALL` Instruction Instrumentation

In this updated approach, `CALL` instructions are instrumented in the same fashion as in Section 3.3.2. However, an extra step is taken to track the current stack position

where an encrypted address is held and store this value in a sorted list. This is so that, when a memory reference occurs, the DR client is able to compare the address being read against the list of addresses that contain encrypted values.

4.3.1.2 Memory References

Upon encountering an instruction that references memory, the DR client must first determine if this is a direct or indirect memory reference. For direct memory references, the target address can be calculated statically within the `drmgr` basic block callback. However, indirect memory references can change during runtime and must have code instrumented to calculate this value every time the basic block is executed [4, 12]. One exposed API function that can handle this requirement nicely is `drutil_insert_get_mem_addr` which will instrument instructions to obtain the memory address referenced by an operand. Once this memory address is obtained, the DR client instruments additional code into the application to callback into the client where it can determine if further instrumentation is needed to perform decryption. This is accomplished by comparing the referenced address that was just obtained to the list of addresses containing encrypted values. If there is a match, the DR client must instrument code to perform on-demand decryption.

Once the client knows to decrypt, it must then understand the type of instruction that is causing the memory reference so that it knows how to decrypt properly. This leads to three cases that must be handled separately.

1. **RET Instructions:** The steps to decrypt are identical to those listed in the previous method found in Section 3.3.3.

2. **Comparison Instructions**: For instructions that do not store a value, but merely make a comparison (such as `TEST` and `CMP` instructions), the DR client must insert code prior to the instruction to decrypt the value on the stack, and then code following the instruction to re-encrypt the value. This must be done so that the comparison can be performed on unencrypted values.

3. **Source/Destination Operand Reference:** When an operand of an instruction contains a memory reference to an encrypted value, it must first be decrypted before it can be used. Also, the value on the stack should be re-encrypted after the instruction executes so that the protection mechanisms still apply. An exception to this rule occurs when an instruction's destination operand contains a reference to an encrypted memory location. In this case, re-encryption is not performed. For example, in the instruction "`xor [ebp+4], 0x10`", the operand `[ebp+4]` is both a source and destination. Assuming this references an address containing an encrypted value, the value will first be decrypted before the `XOR` executes. However, since the memory reference operand is also a destination, the value is not re-encrypted afterwards. The ramifications of this scenario are discussed in more detail in Section 4.3.3.

4.3.1.3 Address List Management

As an application executes, `ESP` is constantly changing. Therefore, it is critical to carefully maintain the list of encrypted return addresses and ensure that items are properly removed when they no longer contain encrypted values. To emphasize this point, Figure 4.3 shows a scenario where it is necessary to remove stale items from

the encrypted address list. When FOO returns on line 7, execution continues at line 2 after ESP is increased [12]. Without proper address list management, the DR client will think that the reference to ESP on line 3 contains an encrypted value since the PUSH instruction on line 2 will decrease the stack pointer to a value that used to be encrypted. The solution to this is to constantly monitor ESP. When ESP changes, any value in the encrypted address list lower than the current value of ESP must be removed from the list.

```
1       call   FOO
2       push   eax
3       mov    ebx, [esp]
4       ...
5   FOO:
6       ...
7       ret
```

Figure 4.3: The Need For Address List Management

Another item to consider is outlined in step 3 of the previous section. When a destination operand refers to an address containing an encrypted value, the value at the memory address is not re-encrypted after the instruction executes. This means that the address should be removed from the list of encrypted addresses. This will allow the program to continue functioning properly since the DR client will determine that the RET instruction associated with this previously encrypted address is no longer encrypted, and consequently will bypass decryption.

4.3.2 Analysis and Impact

This chapter laid out the challenges and limitations of encrypting return addresses. Additionally, it sought to lay out a framework for a more generic approach. Although a complete implementation of this refined approach was outside of the scope of this thesis, efforts were made toward a proof-of-concept to determine its effectiveness and relative impact.

Initial indications showed success in handling some previously identified problem spots in code. The following two situations were targeted during testing since both of these would fail under the previous implementation. In Figure 4.4, the `RET` instruction is not referencing an encrypted value since the address to which it will return is pushed onto the stack immediately beforehand. Under the new prototype implementation, the DR client sees that the stack address referenced is not in the list of encrypted addresses and no decryption code will be inserted. Figure 4.5 was also tested with the new DR prototype. In this example, `[eax + 0x44]` references an address that contains an encrypted value. The DR client sees that the application is referencing an encrypted value and ensures that the value is first decrypted, and then re-encrypted after the `TEST` instruction executes.

```
1 push    eax
2 ret
```

Figure 4.4: `RET` Referencing a Non-Encrypted Value

```
1 test    dword ptr [eax+0x44], 0x01000000
2 jnz     0x772cd600
```

Figure 4.5: `TEST` With an Encrypted Value

Although only a prototype of the refined approach was implemented, it is worth noting its impact. The new approach now requires instrumentation of every operand that references memory. This means that every operand of every instruction within every basic block must be reviewed to determine if there is a memory reference. If there is, code must be inserted to determine the referenced memory address. This action incurs a significant runtime cost.

Another source of overhead comes in the form of *clean calls*. A clean call is a way for a DR client to insert (in a transparent fashion) a callback from application code back into a function in the client [3]. The prototype uses the clean call mechanism to compare a referenced memory location to the list of encrypted addresses that it is storing. Although convenient, clean calls can incur a substantial runtime cost, and cause a large impact when used frequently. A full implementation would need to take these issues into account to ensure manageable overhead.

4.3.3 Limitations

One major limitation that should be noted is that this method does not attempt to protect applications that modify their own return values. This is done so that security of this approach is not compromised in a way that would inadvertently allow an attacker to find a gadget that would allow for their memory writes to be encrypted. It is sometimes necessary for applications to manipulate their return addresses; however, investigating novel ways of allowing this action while not compromising security is left as a discussion for future work (see Section 6.1.2).

Chapter 5

Performance and Analysis

This chapter explores the performance impact of the dynamic encryption client built using the method described in Chapter 3. Due to challenges discussed in Sections 4.1 and 4.2, not every application is able to run under the protection of this ROP prevention method. However, several programs were identified that help to benchmark this approach and determine its impact.

5.1 Benchmarks

As mentioned above, the DR encryption client can only run on some applications. Any application requiring either a graphical user interface or the loading of shared libraries is unable to execute properly, limiting the benchmark tests to command line applications only. Additionally, this thesis only targeted 32-bit code. The following benchmark applications were used in gathering performance data:

- **Quick Pi:** Calculates 2 million digits of Pi.

- **md5:** Digests 100 million characters.
- **miniz:** Lossless data compression library that contains a benchmark application.
- **Echo Server:** Winsock program which receives data and transmits the same data back out through the same socket. The Winsock client sends 1,128 messages before closing the socket. Client and server code were adapted from the Microsoft MSDN developer website.
- **Python:** Script that Base64 encodes a 6.18MB file.
- **du:** Disk Usage utility from the Sysinternals Suite that gathers statistics on a directory tree including 4,897 directories and 37,692 files totaling 16.3GB.
- **AES:** Portable Advanced Encryption Standard implementation that encrypts a 95MB file with a 256-bit key using 1,024 iterations.

5.2 Performance

An average runtime of 50 runs for each of the benchmarks outlined in the previous section was measured under the following conditions:

- **Base System:** Intel Core i7 Q 820 @1.73GHz, 6GB RAM, Windows 7 Ultimate.
- **DR:** DynamoRIO without any clients.
- **DR+Encryption:** DynamoRIO with the encryption client.

Figure 5.1 shows a comparison of runtimes when the benchmarks were run using each of the above conditions. Figure 5.2 helps to show the amount of slowdown relative

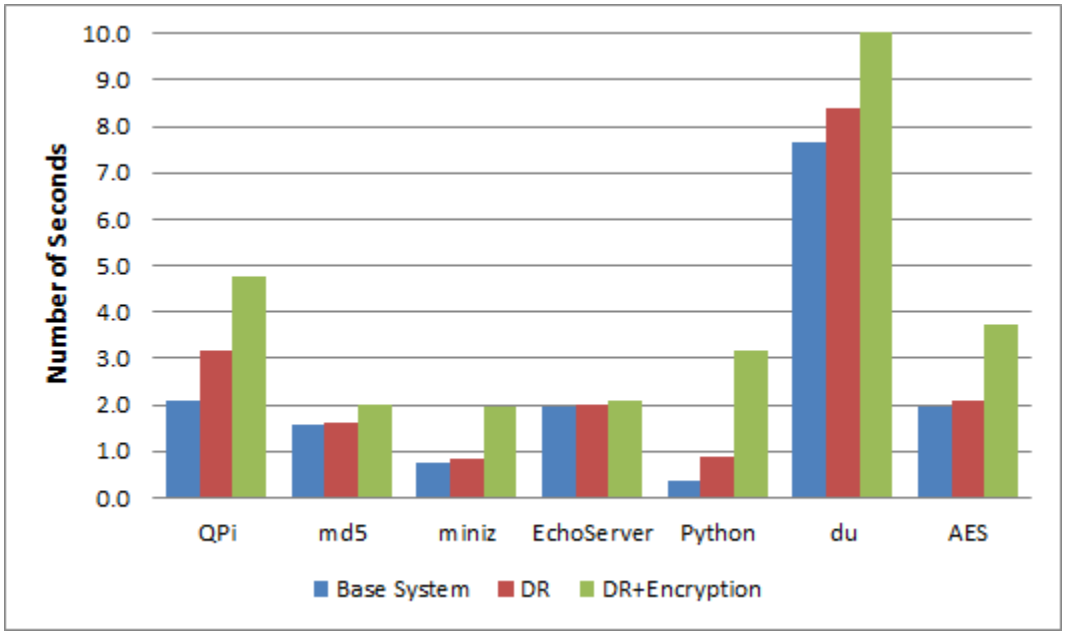

Figure 5.1: Average Time Per Run (seconds)

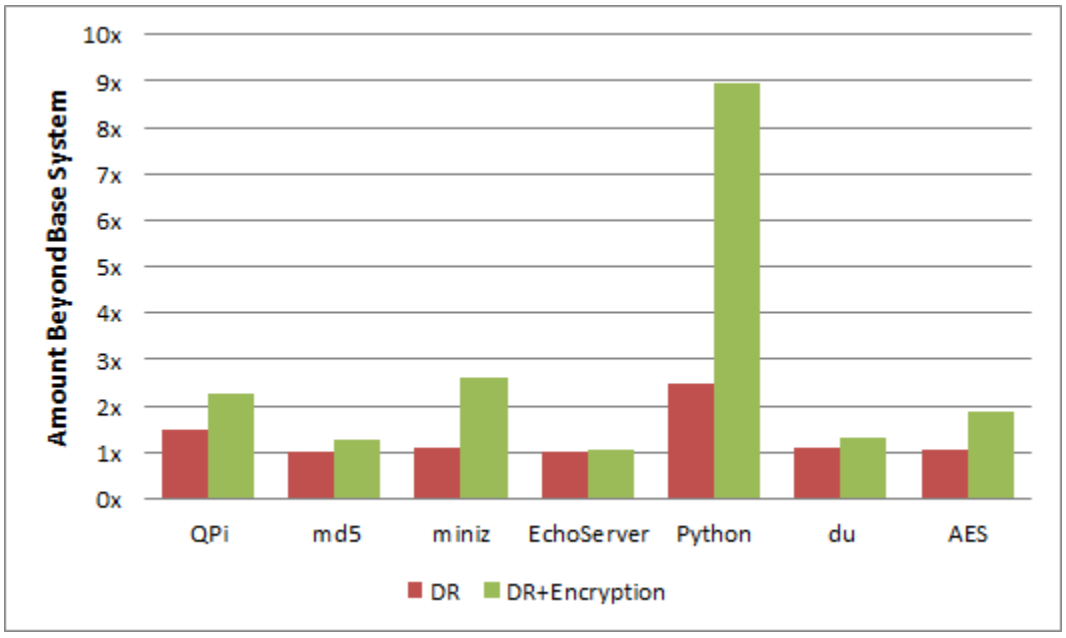

Figure 5.2: Time Impact Relative To Base System

to the base system. Experimental results show that DynamoRIO itself (without any clients) imposes an average of 1.37x slowdown, with ranges from 1.02x through 2.49x. When the encryption client is used with DR, an average slowdown of 3.02x is induced, with ranges from 1.29x through 8.96x. It should be noted that these averages do not include the runs for EchoServer. This was done to eliminate any skew induced by sockets since network latency was not measured.

5.3 Analysis

Performance Overhead

The overhead induced by the encryption client stems from a few main factors. First, DR must now call into the client for every basic block created, at which point the client inspects every basic block for `CALL` or `RET` instructions. Additionally, if either one of these instructions is found, a total of 7 instructions are added to the basic block that will be executed on each pass (see Sections 3.3.2 and 3.3.3).

Another large factor is that every basic block (regardless of containing a `CALL` or `RET` instruction) has several instructions added to determine if the return address should be encrypted, and if so, perform the encryption. This amounts to 20 additional instructions inserted into each basic block. However, the only time all 20 instructions are executed are in the case of the basic block being entered via a `CALL`. Otherwise, only 8 additional instructions are executed (see Section 3.3.4 for details).

The last major source of overhead deals with extra processing required by DR when it needs to interface with a client. Once the encryption client has manipulated a basic block, DR must perform its own analysis and modifications of instrumentation

performed by the client to ensure that it maintains control of the application.

These factors are the main contributions to the overhead seen by the encryption client. Most benchmarks saw a fairly similar overhead when compared to DR without a client. However, the Python benchmark in particular was an outlier, and an explanation into the cause of this will be given here.

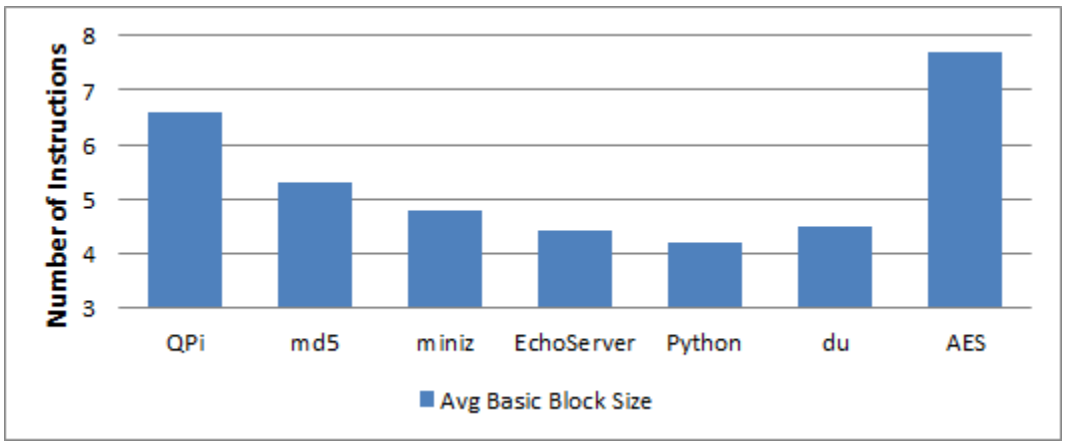

Figure 5.3: Average Instructions Per Basic Block

Figure 5.3 displays the average number of instructions contained within a basic block. The Python benchmark contains the lowest average, meaning that there is an increased relative overhead due to instructions inserted at the start of each block. Additionally, Figure 5.4 shows both the number of unique basic blocks encountered, as well as the total number of basic blocks executed. The Python benchmark contains the largest number of unique basic blocks while also a comparatively small number of overall basic blocks executed. Since DR must callback into the client on every basic block created as well as perform post processing of the blocks after the client performs manipulation, additional overhead is incurred.

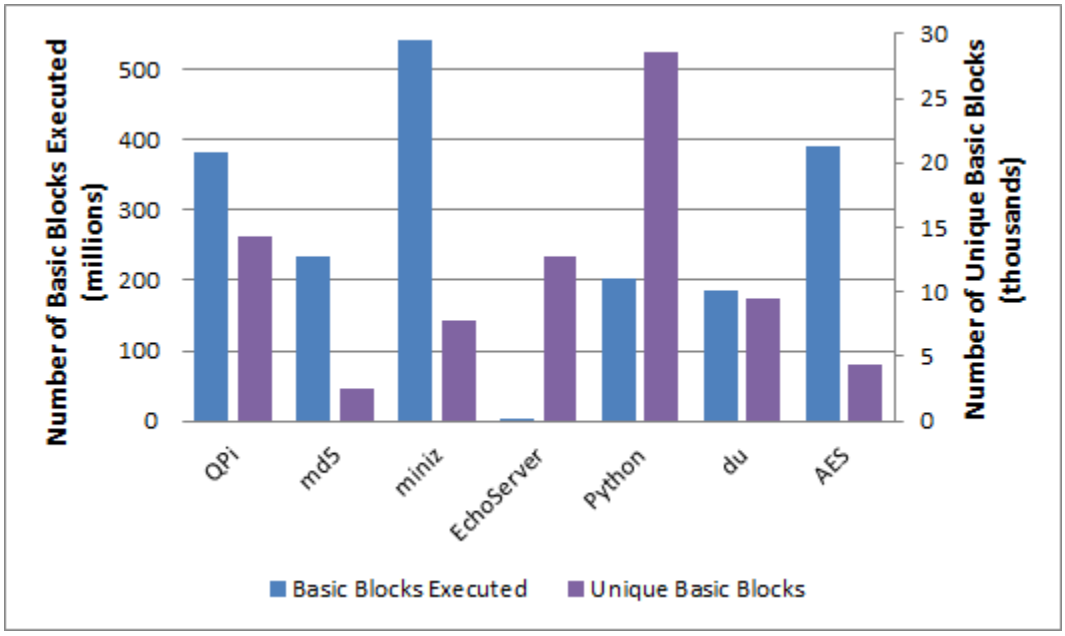

Figure 5.4: Number of Basic Blocks

Another major source of overhead stems from the abundance of `CALL` and `RET` instructions within the Python benchmark. Figure 5.5 shows that nearly 5% of the instructions encountered in the application are one of these control transfer mechanisms. Not only do each of these instructions have additional instructions instrumented, in the case of `CALL` instructions (which is about half of the instructions represented in the graph), the following basic block will require the execution of all 20 instructions that were instrumented.

Although most prominent in the Python benchmark, the above factors are the major source of overhead in all applications that run with the protection of the encryption client.

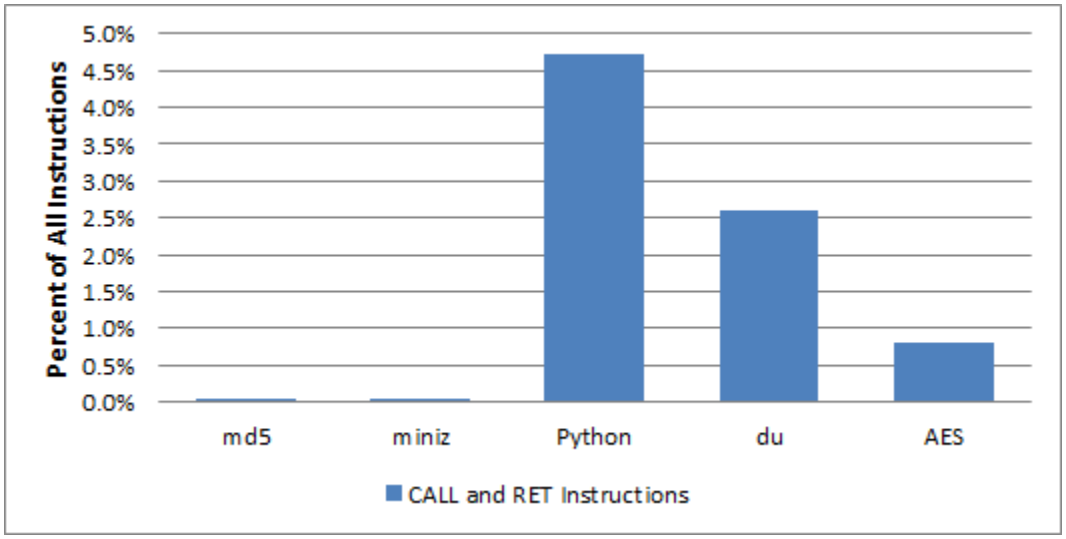

Figure 5.5: Percentage of `CALL` and `RET` Instructions. Note: Information was not obtained for QPi and EchoServer due to complications with DR's included instruction counting client.

Memory Overhead

The memory footprint of the encryption client is negligible, allocating only a tiny amount of heap space per thread for the TLS. This is used to store the flag indicating whether or not a `CALL` instruction was encountered in the previous basic block. Although the encryption client adds little memory overhead itself, DR does have its own memory overhead that is well documented. A complete analysis of the memory footprint required by DR can be found in the paper by Bruening [4].

Chapter 6

Conclusion

ROP protection is a critical need as attackers and their tools become more and more sophisticated. The method of encrypting return addresses is a novel extension of the related work outlined in Chapter 2. By eliminating an attacker's ability to create ready-made exploits that could target any number of computers, overall security is greatly increased. Chapter 3 explored a method for encrypting return addresses as well as the necessary steps needed for such a method to work. Furthermore, a simulated ROP attack was outlined in Section 3.4 to show the protection method working as designed. Chapter 4 outlined several challenges encountered including the complexities of x86 as well as considerations needed for application transparency. Finally, in Chapter 5, the return address encryption method was evaluated to determine its impact on an application.

The goal of application level protection from ROP attacks through encrypted return addresses was achieved; however, there were certainly some limitations reached. One of the biggest limitations was that the encryption client did not provide complete

application transparency. This allowed an application to access encrypted values, causing an application to not function properly. Another potential limiting factor was the amount of runtime overhead incurred. Since a 3x slowdown could be unreasonable in many situations, especially for some remote services, some users may not opt for this extra protection.

6.1 Future Work

Although there are some limitations encountered that could hinder adoption in the real world, there are several areas in which improvements could be made to aid adoption.

6.1.1 Optimized Instrumentation

In the current implementation, every basic block includes a series of instructions to test whether or not the block was entered through a `CALL` instruction. This is a costly approach and there could be a method employed by the client to perform analysis that could eliminate the need for the check in every basic block.

Another optimization could be included that would remove the need for the flag indicating a `CALL` instruction was encountered by deleting the `CALL` and substituting that instruction for a `PUSH`/`JMP` combination. This would allow the encrypted return address to be pushed onto the stack prior to the control transfer instruction. This method would have to ensure that indirect branches have their jump targets calculated correctly and in an efficient enough manner to ensure this method reduces overhead.

6.1.2 Memory Writes to Encrypted Addresses

As discussed in Section 4.3.3, the methods explored in this thesis do not protect an application that modifies its own memory addresses. Although this is a relatively uncommon scenario, several complex applications do employ methods where this could be more prevalent (such as in self-modifying code). The refined approach could be expanded to allow for the encryption of addresses that have been overwritten by the application. To ensure this is done safely, one possibility could be to utilize taint checking such that any memory associated with user input does not get encrypted.

6.1.3 64-bit Architecture Support

The work in this thesis targeted the IA-32 architecture, however, DR also provides support for 64 bit machines and applications. By adding 64-bit support to the encryption client, a larger number of applications could utilize its protection. Another benefit of using 64-bit addresses is that the key space is expanded to 2^{64}, or 1.8447×10^{19}.

6.1.4 Multiple Keys

One approach for reducing an attacker's ability to attempt a brute force attack would be to use multiple keys within the DR client. This could be done by changing key generation from a single key created at application initialization to creating keys on-demand when a `CALL` instruction is encountered. This would introduce the need for keeping track of the stack address into which the encrypted value is placed. When a `RET` is encountered, it must first lookup the stack address storing the encrypted return address to acquire the correct key to use. In this manner, an attacker must

now correctly determine multiple encryption keys instead of just one.

Bibliography

[1] Address Space Layout Randomization. `http://pax.grsecurity.net/docs/aslr.txt`. Accessed: 4/7/2013.

[2] Data Execution Prevention. `http://technet.microsoft.com/en-us/library/cc738483.aspx`. Accessed: 4/7/2013.

[3] Derek Bruening, Qin Zhao, and Reid Kleckner. DynamoRIO Dynamic Instrumentation Tool Platform. `http://dynamorio.org/`. Accessed: 5/19/2013.

[4] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[5] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, 1999.

[6] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association.

[7] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, Berkeley, CA, USA, 1998. USENIX Association.

[8] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[9] F-Secure. Threat Report: H1 2012. Technical report, F-Secure, 2012.

[10] Fu-Hau Hsu and Tzi-Cker Chiueh. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 409–, Washington, DC, USA, 2001. IEEE Computer Society.

[11] Information and Communication Technologies. The World in 2013: ICT Facts and Figures. Technical report, International Telecommunication Union, 2013.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[13] Mandiant. APT1: Exposing One of China's Cyber Espionage Units. Technical report, Mandiant, 2013.

[14] McAfee. McAfee Threats Report: First Quarter 2012. Technical report, McAfee, 2012.

[15] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating Return-Oriented Programming Through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[16] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996. Accessed: 9/21/2013.

[17] Hilmi Özdoğanoğlu, T.N. Vijaykumar, Carla E. Brodley, and Benjamin A. Kuperman. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transactions on Computers*, 55(10):1271–1285, October 2006.

[18] Arkaitz Ruiz-Alvarez and Kim Hazelwood. Evaluating the Impact of Dynamic Binary Translation Systems on Hardware Cache Performance. In *IEEE International Symposium on Workload Characterization*, IISWC, Seattle, WA, September 2008.

[19] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *In Proceedings of USENIX Security 2011*, 2011.

[20] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[21] Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. Transparent Runtime Shadow Stack: Protection against malicious return address modifications, 2008.

[22] Solar Designer. Getting around non-executable stack (and fix). Technical report, bugtraq, 1997. `http://imchris.org/projects/overflows/returntolibc1.html`.

[23] Wartell, Richard. *Rewriting x86 Binaries Without Code Producer Cooperation.* PhD dissertation, The University of Texas at Dallas, 2012.

[24] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture Support for Defending Against Buffer Overflow Attacks. In *The Second Workshop on Evaluating and Architecting System Dependability (EASY'02)*, 2002.