Analysis of Return Oriented Programming and Countermeasures

by

Eric K. Soderstrom

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Signature redacted

Author_____
Department of Electrical Engineering and Computer Science
September 5, 2014

Certified by_ Signature redacted _____
Professor Martin Rinard Thesis Supervisor
September 5, 2014

Certified by_____ Signature redacted _____
Dr. Hamed Okhravi Thesis Supervisor
September 5, 2014

Accepted by_____ Signature redacted _____
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Acknowledgments

This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Also, many thanks to Richard Skowyra for his help, particularly with regards to helping put together a real-world ROP side-channel exploit against Apache.

Finally, my sincere thanks to Stelios Sidiroglou-Douskos for his insight into all things related to control flow integrity, and his continual encouragement and help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Attackers have relatively success in defeating modern defensive techniques by using an exploitation method known as "code reuse." This class of exploitation techniques makes use of the lack of memory safety in C which allows an attacker to redirect a program's control flow to pre-existing snippets of code. Code reuse attacks have historically been a powerful and ubiquitous exploitation technique [2]. Even as recently as 2014 there has been an outbreak of these code reuse attacks, targeting such applications as Adobe, Internet Explorer, and Firefox [5]. Many defensive countermeasures have been taken by the security community, ranging from data execution prevention to varying degrees of code randomization. This thesis can roughly be broken into two halves:

1. Show how code reuse attacks can leverage timing information in order to break many existing defenses.

2. Investigate how control flow integrity can be improved upon as a countermeasure to code reuse attacks.

In chapters 2 and 3, we show how the fundamental assumption behind popular code reuse countermeasures can be circumvented. Specifically, we examine defenses, known as code diversification, that rely on randomizing address space in order to obfuscate the locations of sensitive pieces of code. Code diversification defenses make the assumption that an attacker cannot gain access to the diversified code. Exploits

of the past have proven that if this assumption can be broken, it is possible to craft powerful code reuse exploits that circumvent many of the existing countermeasures to code reuse attacks using side-channel attacks. However, in order to break this assumption, these exploits rely on the pre-existence of a buffer over-read vulnerability [6], thereby limiting the number of potential target applications to only applications in which such a vulnerability exists. We demonstrate that if a code reuse vulnerability is already present, the direct memory disclosure vulnerability is superfluous.

The fundamental idea is that an attacker can analyze the output and timing information from a remote machine in response to crafted input. This general technique, known as a side-channel attack, has an infamous history of effectively leaking sensitive information in crypto implementations, perhaps most notably against the openssl implementation of RSA. In general, side-channel attacks utilize execution timing, cache contents, power consumption, fault analysis, and other observable quantities in order to ascertain information about the state of a system. The technique has been used to great effect in the world of cryptography, and in this thesis we demonstrate how many of the same ideas can be applied to the world of systems security. However, rather than using side-channel information to leak sensitive keys, it is possible to use the same information to leak sensitive memory contents of diversified code.

The results show that memory disclosure using side-channel attacks is not only possible, but in fact quite practical to implement. We consider three classes of fault analysis side-channel attack and five classes of timing side-channel attack, and provide examples of where such vulnerabilities might exist and what their limitations might be. In order to characterize the effectiveness of these attacks, we consider how feasible it would be for an attacker to uniquely determine the location of gadgets within the `libc` library. We primarily examine `libc` because it is automatically linked with most programs, and contains system call wrappers that allow for meaningful exploits.

Finally, using the resulting timing information, we show that an attacker can reconstruct with a fairly high degree of fidelity the original memory contents of the target application. Using pattern matching algorithms it is possible for an attacker to infer memory contents even in the presence of network latency and jitter. We describe

the details of this novel attack technique and construct a sample payload against an Apache web server.

In order to combat this class of powerful code reuse attacks, research has turned towards investigating a technique to verify valid control flow a priori, known as control flow integrity [1]. In chapters 4 and 5, we discuss the details of control flow integrity and why it has shown such promising potential as a preventative measure against code reuse attacks. We discuss current research in the field of control flow integrity and its real-world implementations. Then, we examine some of the shortcomings of current implementations, and discuss how they can be improved upon by generating more precise control flow information. We present a method for evaluating the precision of the resulting control flow graphs by borrowing principles from the fields of graph theory and network analysis, and tailoring the principles to apply more directly to control flow integrity in the context of code reuse attacks. We also present our findings. Finally, in chapter 7 we conclude and describes how future work might make use of these findings to better inform control flow integrity style defenses. Additionally, chapter 7 includes discussion of how dynamic analysis can be further refined.

Chapter 2

Side-Channel Attacks through Code Reuse

2.1 Background

2.1.1 Buffer Overflows

A buffer overflow is a type of memory safety violation, in which a program attempts to write data beyond the length of the buffer. This is a particularly notorious problem for languages that lack basic memory safety, such as C and C++. If the buffer is located on the stack, this presents an opportunity for an attacker to overwrite a return address or function pointer [7]. Subsequently, when the next return statement is executed, or the overwritten pointer is called, the attacker gains direct control over the control flow transfer.

Initial buffer overflow exploits would typically fill the buffer with malicious shell code, and then redirect control flow to that shell code. This particular vulnerability was effectively dealt with by simply marking all data pages as non-executable, thus preventing an attacker from ever being able to execute his own malicious code on the target platform. However, this did not deal with the underlying problem of giving an attacker control over function pointers and return addresses in the first place.

This opened the door for code reuse attacks [2]. In a code reuse attack, a buffer

overflow vulnerability is exploited to overwrite a return address with the location of a snippet of code already present in the address space of the target program, known as a gadget. In the most basic form of a code reuse attack, a return address is overwritten with the entry point to a function, often a system call wrapper in `libc`. For calling conventions that specify arguments to be passed on the stack, this makes it trivial to call the `libc system()` function with "/bin/bash" as the argument to gain complete access to the machine. Even without such a calling convention, an attacker can often make use of other available gadgets in order to set up the registers in such a way as to allow arbitrary argument passing. [2]

2.1.2 Code Diversification

Various forms of code diversification techniques have been proposed and implemented as countermeasures to code injection attacks. The fundamental idea behind most of these techniques is to in some way randomize the location of code in order to prevent an attacker from being able to redirect control flow to some known location.Perhaps the most widely used is address space layout randomization (ASLR), wherein the base addresses of the stack, heap, and libraries are all randomized. This makes it difficult for an attacker to correctly determine the location of injected shell code, in the case of code injection attacks, as well as the locations of useful gadgets within libraries, in the case of code reuse attacks.

Pappas et al. [11] present a method of in-place code randomization, in which certain instructions are replaced with equivalent alternate instructions in order to break useful gadget sequences. Hiser et al. [12] analyze a very fine-grain code diversification technique, in which every instruction's location is randomized, and a virtual machine is used to track the proper order of instruction execution. Franz and Homescu et al. [13] propose different schemes of randomly inserting NOP instructions into code in order to alter gadget locations. Wartell et al. [14] examine code randomization at the basic block level.

2.1.3 Return Oriented Programming

A more subtle form of code reuse attack, known as return oriented programming (ROP), utilizes short snippets of code, rather than complete function calls. An attacker will first accumulate a set of gadgets from the address space of the victim process. These gadgets are typically 2 - 7 instructions in length, and end with a `return` instruction. By using a buffer overflow to overwrite the return address on the stack with the location of a gadgets, an attacker can effectively transfer control to any of the available gadgets. Because each gadget ends with a `return`, it is possible to to place the addresses of several gadgets on to the stack, and have them execute sequentially. In effect, developing a ROP exploit can be thought of as programming in assembly, where the instruction set one uses is limited to the set of available gadgets in the victim process address space.

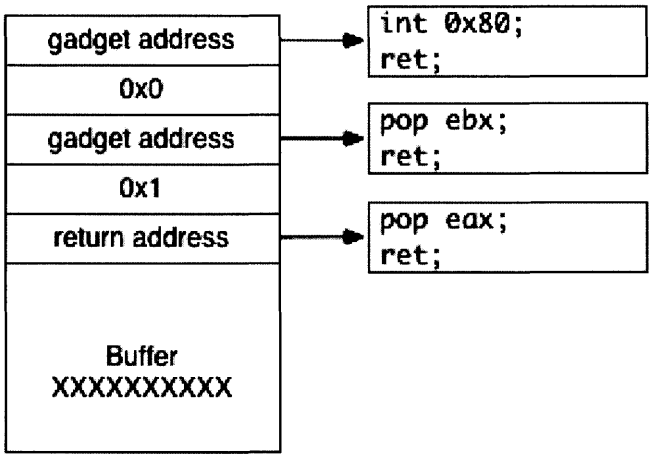

Figure 2-1: basic ROP that calls exit()

Figure 2-1 shows the stack layout during a very simple ROP attack. There is a buffer on the stack that has been overflown. The attacker has overwritten the return address with three gadgets, the effect of which is to pop the value 1 into register `eax`, 2 into register `ebx` and then call `int 0x80`. That is, the program will call the `exit()` system call with status 0, which will cause the thread to terminate with exit status 0. This particular ROP attack is relatively benign, as the thread could simply be restarted. We will show how this technique can be used in conjunction with a

side-channel attack to cause real damage even in the presence of code diversification defenses.

2.1.4 Attacks on Diversified Code

Even environments with both data execution prevention and code diversification in place can still be susceptible to exploitation. Sacham et al. [2] proved that on 32-bit architectures, which use only 16 bits for address randomization, entropy exhaustion attacks can totally defeat ASLR within a matter of minutes. Additionally, Boneh et a. demonstrated that in some cases it is possible to bootstrap enough gadgets, even against a closed-source binary, to construct a payload that will dump memory contents over the wire. 64-bit architectures do have sufficient randomization to prevent brute force attacks against ASLR, but are still susceptible to memory disclosure vulnerabilities. The now infamous Heartbleed bug showcased the prototypical example of direct memory disclosure, in the form of a buffer over-read. Note that all of these techniques for defeating ASLR rely on the presence of two distinct vulnerabilities.

1. A vulnerability, such as a buffer over-read, that can directly disclose memory contents to the attacker, and thus defeats ASLR.

2. A vulnerability, such as a buffer over-flow, that allows can attacker to overwrite a function pointer or return address, and thereby execute a code reuse attack.

Hund et al. [15] show that it is possible to utilize side-channel information based on cache hits in order to defeat some forms of code diversification. This is similar in principle to the technique we present in this thesis, though it relies on the attacker having access to the victim's cache. We make the much weaker assumption that a remote attacker only has access to information returned by the victim machine.

2.2 Fault Analysis

In order to execute a fault analysis side-channel attack, we consider a scenario in which a remote server has a buffer overflow vulnerability. Through this vulnerability,

```
1 recv(socket, buf, input);
2 if (arr[index])
3     rv = SUCCESS;
4 else
5     rv = ERROR;
6 send(socket, &rv, length);
```

the attacker may overwrite stack variables, and gain information about the process address space based on the server's response. We present three possible classes of fault analysis attack.

2.2.1 Data Overwrite

In the following example, we presume that the `index` variable can be overwritten with an attacker-controlled value. `index` is used to index into an array, `arr`, and the resulting value is used as the predicate for an `if` statement. In this case, the value returned by the server will depend on whether the value found at `arr[index]` is a null byte. In this example, an attacker could construct a sequence of payloads that scan the address space for null bytes. This null byte profile can be compared to the known null byte profiles of gadgets found within a library, thereby enabling the attacker to leak the locations of these gadgets.

2.2.2 Data Pointer Overwrite

Similarly, by overwriting a data pointer directly, an attacker can infer information about chosen locations in the program's address space. In the following example, we consider the case where an attacker can overwrite the `ptr` variable used in the body of the `while` loop. In this case, the value returned over the socket depends on the number of contiguous bytes required to sum to a value greater than 100. By varying the locations pointed to by `ptr` the attacker can generate a profile of returned values from the server. As in the preceding example, this profile can be compared to locally-generated profiles for gadgets. Comparison of the locally-generated gadget profiles and the remote profile allows the attacker to indirectly infer locations of gadgets,

thereby breaking code diversification.

```
1 recv(socket, buf, input);
2 sum = i =0
3 while (sum < 100)
4     sum += ptr[i++]
5 send(socket, &i, length);
```

2.2.3 Function Pointer Overwrite

Finally, we consider the case where an attacker is able to overwrite a function pointer. In this example, the attacker can overwrite `funcptr`. `funcptr` is then called, and the return value is sent back directly over the socket. By creating an offline profile of `libc` gadgets with deterministic return values, it is possible to scan `libc` until a sufficient set of such gadgets has been discovered to build up a more powerful code reuse attack.

```
1 recv(socket, buf, input);
2 rv = (*funcptr)();
3 send(socket, &rv, length);
```

2.3 Timing

The second class of side-channel attacks we explore in this thesis are timing side-channel attacks. The idea here is that an attacker makes use of the amount of time it takes for a server to respond to a malicious query. This requires that the server response time should be in some way influenced by the malicious input, and that the attacker is able to record sufficiently many response times with sufficient accuracy so as to mitigate noise from network jitter. Riedi et al. [16] show that the effects network jitter can fairly effectively be reduced by repeated sampling. We apply the same principles here in order to leverage minute differences in execution time in order to infer details about code content and code location.

```
1 if (input == 0)
2     i = i * 2;
3 else
4     i = i + 2;
```

Figure 2-2: crafted input

2.3.1 Crafted Input

Using the crafted input, an attacker need not exploit any memory safety vulnerability at all. Instead, simply by choosing suitable input values, he is able to execute different control flow paths, the timing of which can leak information about the diversified code. Namely, the defensive techniques proposed by Hiser et al. [12] and Franz et al. [13] both involve the random insertion of NOP instructions in order to randomize the locations of gadgets. In the following example, an attacker is able to choose input that will exercise either the consequent statement, or the alternative depending on the chosen value for input. By comparing the time difference in execution to the expected time difference in execution, an attacker can determine the relative number of NOP instructions inserted into either the consequent or the alternative basic blocks. This information greatly reduces the possible NOP-insertion cases that would need to be considered for a brute force attack.

In cases where an attacker is able to overwrite a variable used in the predicate for an `if` statement with no corresponding `else` clause, then the timing difference can directly reveal the number of NOP instructions inserted into the consequent basic block.

```
1 if (input == 0)
2     foo();
3     bar();
4 send(socket, SUCCESS, length);
```

This technique relies on having essentially perfect knowledge of the victim architecture. CISC architectures may lead to unpredictable timing patterns, as instructions can have widely varying execution times. RISC architectures are somewhat more predictable in timing, as the simplified instruction set has less variance in instruction

execution time.

2.3.2 Data Overwrite

A more powerful form of timing attack can be executed if the attacker is able to exploit a buffer overflow vulnerability and overwrite a stack variable that is used in determining control flow. Consider the scenario in listing 2.1, in which an attacker might be able to overwrite the `index` variable, which is used to index into a character array. By modifying the `index` variable appropriately, the attacker can cause the character pointer `arr + index` to point to arbitrary byte locations, even within the text segment of the executable. By observing execution time differences between these locations, an attacker is able to build up a null byte profile, and compare it with a known null byte profile for various gadgets. This technique is quite similar to the fault analysis side-channel attack that makes use of a data overwrite, except it does not rely explicitly on the server sending back explicit information about which basic block was executed. Rather, the time of execution is enough to determine which basic block was executed. This form of exploit will often result in a segmentation fault, in cases where `arr + index` lies outside of the process address space. This isn't necessarily a problem, as most web servers will restart a crashed thread without re-randomizing the process address space due to performance considerations. This means an attacker is able to scan through memory locations until he finds valid location within the process address space. At this point, he can continue scanning to generate a null byte profile.

```
1 char arr[10];
2 if (*(arr + index) == 0)
3     i = i * 2;
4 else
5     i = i + 2;
```

Listing 2.1: data overwrite

2.3.3 Data Pointer Overwrite

Overwriting a data pointer itself can also leak information about a randomized address space. In figure 2.2, an attacker can overwrite `ptr` with a chosen location in memory. The execution time of the `while` loop will then be proportional to the dereferenced pointer value and can be used to approximately reconstruct the dereferenced value.

```
1 int i = 0;
2 while (i < *ptr)
3     i++;
```

Listing 2.2: data pointer overwrite

Even if the overwritten data pointer is not directly used in any control-flow influencing predicates, it is still possible to use execution timing to gain valuable information about arbitrary memory locations. If an overwritten data pointer is dereferenced and used in a computation, the execution time of that computation can depend on the dereferenced value. In figure 2.3 an overwritten integer pointer, `ptr` is dereference, and used in a multiplication. If the dereferenced value is a power of 2, the compiler will likely turn the instruction into a bit shift, which will execute more quickly than a multiplication instruction. This subtle difference in timing can allow an attacker to generate a profile of memory locations that contain a power of two.

```
1 int i = 3;
2 i = i * (*ptr);
```

Listing 2.3: data pointer overwrite not used directly in control flow

2.3.4 Code Pointer Overwrite

Finally, we consider the scenario in which an attacker can overwrite a function pointer or return address directly. If no form of code diversification were enabled, the attacker could simply perform a traditional code reuse attack. However, even with some form of code diversification, the time of execution for calling chosen locations in the process address space is still revealing. Many functions in `libc` have distinct timing

```
1 recv(sock, buf, input);
2 void (*fptr)() = &bar;
3 char buf[100];
4 (*fptr)();
```

characteristics, and can be identified based on execution time alone. Note that an overwritten function pointer does not even need to point to the entry point of a valid function. It is possible to overwrite a function pointer or return address with a location that corresponds to any instruction. Of particular importance is the `return` instruction, because it has the shortest execution time, and will generally not cause a segmentation fault when called. By using a buffer overflow vulnerability to scan through address space, an attacker builds up a profile of `return` instructions, which he can compare the known `libc` profile of return instructions.

Once again, this technique relies on targeting a server that restarts threads without re-randomizing address space, as making calls to locations outside the process address space will result in a segmentation fault.

Chapter 3

Side-Channel Evaluation

In the previous section we proposed different high-level ideas for leveraging fault analysis and timing information to leak details about code randomization. In this chapter, we explore those ideas more fully and evaluate exactly how much information can be leaked, and what practical effect it will have with regards to a real-life exploit.

3.1 Metrics

In order to measure how useful the information leaked by various side-channel attacks actually is, we define the notion of an uncertainty set size (USS). We define the uncertainty set size of a given gadget as follows:

$$USS_p(f) = |\{h, USS_p(h) = USS_p(f), f \neq h\}|$$

Simply put, the USS of a given function, f, under a given profiling technique p, is the number of other functions in the proccess address space that have an indistinguishable profile from function f. In the trivial case of a direct memory disclosure, the USS profile for a given funciton would be the machine code instructions for that function, and every non-identical function in the address space would have a USS of 0. Because the most dangerous and powerful function are those that can be used to make system calls, we pay special attention to identifying the number of leaked

functions that contain gadgets that can be used to make system calls, as well as the number of distinct gadgets overall contained in the identified functions.

We use this metric to characterize the uniqueness of functions found within texttlibc. As motivation for using different approximate byte value metrics to profile functions, we observe that the byte value distribution in the C standard library is fairly structured. Figure 3-1 shows the byte value distribution over textttlibc.

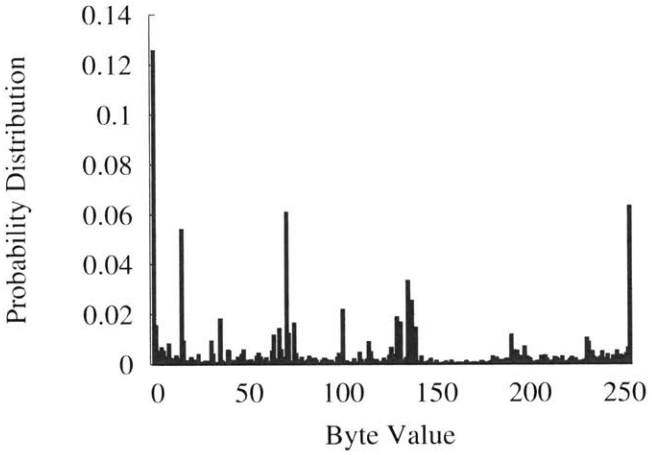

Figure 3-1: Byte value distribution in textttlibc.

What's more, we can compute the information entropy of textttlibc as follows:

$$H(X) = -\sum_{i=0}^{i=255} p(x_i) log_2 p(x_i)$$

Where $p(x_i)$ is the probability mass function for a byte with value i. The information entropy of a distribution can be used to calculate an upper bound on the data compression ratio for a file. Therefore, we can use compression algorithms to gain a fairly accurate approximation for the information entropy of the standard C library. The Lempel-Ziv coding (LZ77) gives a compression ratio of 2.3 for textttlibc, which gives us an upper bound of 0.44 on the information entropy for textttlibc. Given the highly structure byte value distribution, evidenced by high compressibility and low information entropy, it is reasonable to use byte value profiles identify various functions within textttlibc. The following profile metrics are intended for profiling textttlibc in the presence of medium-grained ASLR, in which the base address of

textttlibc as well as function entry points have been randomized, but no basic-block level randomization has been done. We wish to develop a profile for each function within textttlibc that will allow an attacker to uniquely distinguish between functions, and thereby identify locations of important gadgets. Each of this metrics can be used to locate function entry points, by looking for the boilerplate function entry point instruction sequences. Namely, saving the old EBP register, updating the stack pointer. We assume the entry point to a function is known, and an attacker wishes to profile the byte values contained in the function. Entry points can also be identified through the code pointer overwrite method, described in chapter 4. Namely, by overwriting a function pointer, and subsequently calling that pointer, the calling thread is likely to segfault, unless the called value happens to correspond to either a function entry point, or the address of a return instruction. Because return instructions can be easily identified by their unique timing, the remaining non-segfaulting address locations comprise the set of function entry points in textttlibc. We characterize the uniqueness of these functions under various profiling methods by computing the given uncertainty set size.

We evaluate the USS profiles of functions in textttlibc under various side-channel profiling methods discussed in chapter 2. Table 3.1 summarizes these results. By successfully identifying a textttlibc function, an attacker has access to all gadgets contained within that funciton. We find that most profiling techniques are able to uniquely identify the majority of distinct gadgets contained in textttlibc, where we consider a gadget to be "uniquely identified" if it is contained in a textttlibc function with USS 0 under a given profiling method. We also note that even functions that are contained in large sets ($USS > 1$) can have value to an attacker. The system call wrappers in textttlibc share very similar profiles under most of the following profiling methods, due to their very similar semantic structure.

Table 3.1: Gadgets Leaked

Information Leaked	Total gadgets	Distinct gadgets	Syscalls
All functions	24102 (100%)	2059 (100%)	60 (100%)
Zero bytes	13691 (56.8%)	1947 (94.6%)	4 (6.7%)
Return instructions	10106 (41.9%)	1720 (84.0%)	1 (1.7%)
Crashes	13989 (58.0%)	1999 (97.1%)	3 (5.0%)
Return Values	12236 (50.8%)	1995 (96.9%)	14 (23.3%)
Timing	14165 (58.8%)	1972 (95.8%)	16 (26.7%)

3.1.1 Null Byte Profiles

The *data overwrite* and *data pointer overwrite* vulnerabilities described methods by which an attacker can use timing or fault analysis to determine whether the value corresponds to `0x00` or not. Figure 3-2 illustrates the cumulative distribution function for the uncertain set size of functions within textttlibc. 38% of these functions are uniquely identifiable (USS = 0) with only 4 null byte locations leaked, and 62% are uniquely identifiable if all null byte locations within that function are revealed.

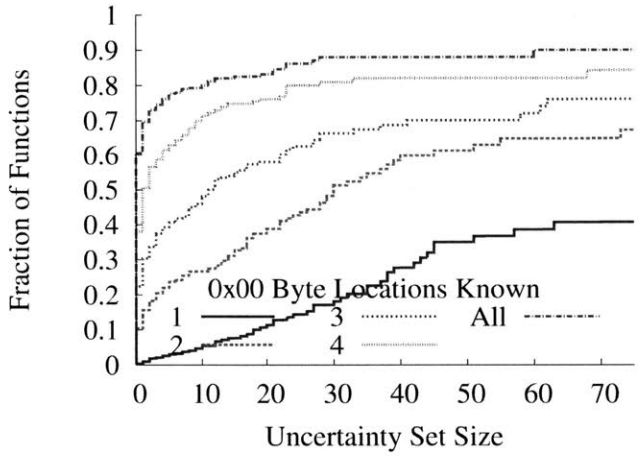

Figure 3-2: Uncertainty set size of textttlibc functions upon leaking null byte locations.

Null byte profiling reveals a particular group of textttlibc functions, of USS 108, that all show the same null byte profile. This group corresponds to the collection of textttlibc system call wrappers, which only differ in the value they load into register `eax` before execution `int 0x80`.

3.1.2 Return Instructions

By overwriting a code pointer or return address, we described how a timing side-channel attack can reveal the location of return instructions. Figure 3-3 shows the CDF for uncertainty set size depending on the number of leaked return instructions. As return instructions are less prevalent than null bytes, we expect them to reveal less about the target textttlibc function. This does seem to be the case, as only 42%

of functions have an USS of 0 with two return instructions leaked. This value does not increase appreciably for leaking more `return` instruction locations, as many functions simply do not contain more than two `return` instructions.

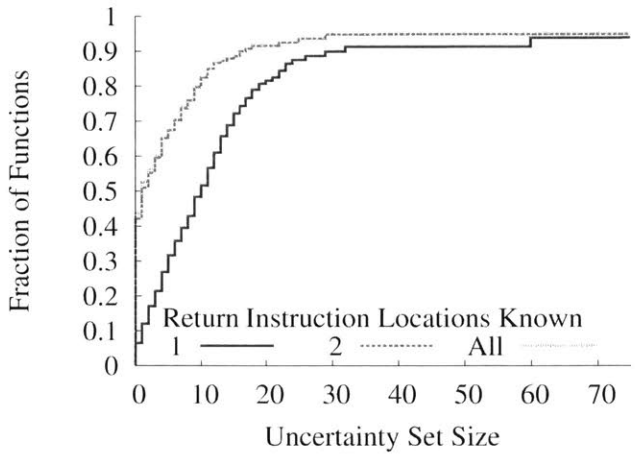

Figure 3-3: Uncertainty set size of textttlibc functions depending on number of leaked return instructions.

3.1.3 Crashes

In some cases, it might not be possible to obtain precise enough timing information to identify return instructions using the code pointer overwrite method. In those situations it is still possible to leak information about diversified code by identifying crash sites. Figure 3-4 shows the cumulative distribution function of USS depending on the number of known crash sites in a given textttlibc function. Because the vast majority of locations do result in a crash, many crash locations are needed before one can construct very many uniquely identifying crash site profiles. With 60 known crash sites, it is possible to uniquely identify only 23% of functions, and with all crash sites leaked, it is possible to uniquely identify 56% of functions.

3.1.4 Return Values

Additionally, if the attacker is able to find a vulnerability in which the value returned by the server is influenced by the value of an overwritten variable, then that infor-

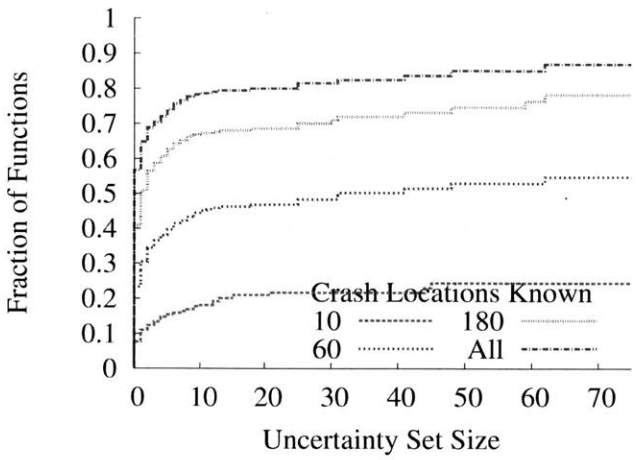

Figure 3-4: textttlibc function CDF as crash sites are learned.

mation can be used as well. Often, the return value will depend on the arguments to the function, which can be difficult to control for an attacker who does not even know what function he is calling. However, the x86 calling convention specifies that return values are passed from callee to caller through the `eax` register or the ST0 x87 register. If the attacker manages to overwrite a function pointer with the address of a non-crashing location within a function, some of these locations can result in deterministically setting the `eax` or x87 ST0 registers. Figure 3-5 shows the CDF for the USS of textttlibc functions depending on the number of known return values. With a single known return value, 12% of textttlibc functions have a unique return value profile, and with all possible return values leaked, 57% of functions are uniquely identifiable.

3.1.5 Timing

Finally we evaluate how execution timing profiles can be used to identify functions. We discussed how execution timing can be used to identify return values, but there are also other locations within functions that can successfully be called without resulting in a crash. As shown in figure 3-6, an attacker can learn 10% of functions with a single valid execution timing data point, 38% of functions with two valid timing data points, and 60% of functions with all valid timing data points.

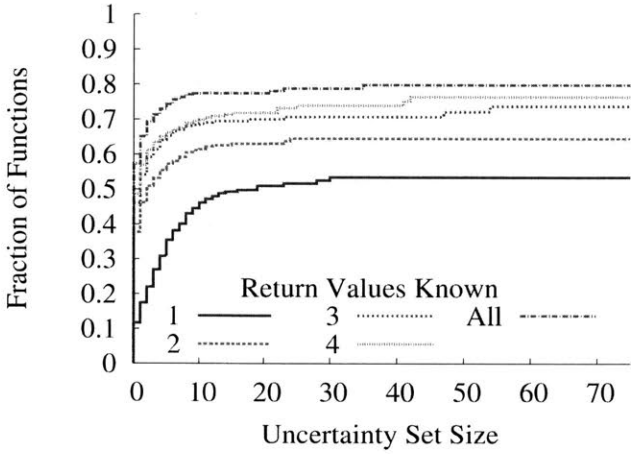

Figure 3-5: textttlibc function CDF as return values are learned.

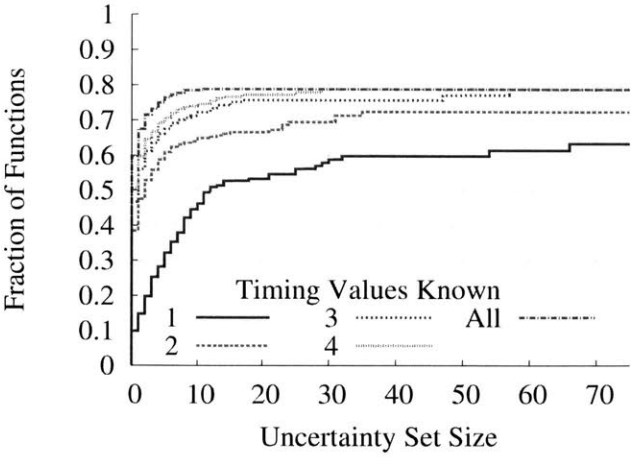

Figure 3-6: textttlibc function CDF as execution timing is learned.

3.2 Practical Side-Channel Exploits

3.3 Measurements

To evaluate the efficacy of a timing side-channel attack, we set up an Apache HTTP server version 2.4.7. We performed our attack over an 802.11g wireless network, as well as a wired LAN environment in order to evaluate the effect of increased network latency and jitter. Additionally, we consider four possible potential code diversifications defenses: coarse-grained ASLR, medium-grained ASLR (function reordering), fine-grained ASLR (basic block reordering), and NOP insertion.

We assume the existence of a stack buffer overflow vulnerability, similar to CVE-2004-0488. Although there are no known vulnerabilities of this type in Apache 2.4.7, we use assume the existence of a typical stack-based buffer overflow for the sake of evaluation. The Apache log.c source code contains a `for` loop, in which the condition dereferences a pointer. We assume the attacker can overwrite the `fmt` pointer with a chosen location, and thereby influence the execution timing. The following code snippet shows the vulnerable code.

```
1 for (i = 0; i < fmt->nelts; ++i)
2     ap_errorlog_format_item *item = &items[i];
3 for (i = 0; i < fmt->nelts; ++i)
4     ap_errorlog_format_item *item = &items[i];
```

3.3.1 Bootstrapping a Timing Attack

When computing the delay time, we use only the first percentile (queries faster than 99% of all queries made) in order to mitigate the effect of non-Gaussian noise introduced by chance network effects [16]. Figures 3-7 and 3-8 show the cumulative delay of the first percentile of server response times for 11 different byte values for `fmt->nelts`. The response time grows linearly with slope directly proportional to the byte value. No byte value exhibited standard deviation grater than $0.557ms$ over LAN and $0.715ms$ over 802.11g wireless.

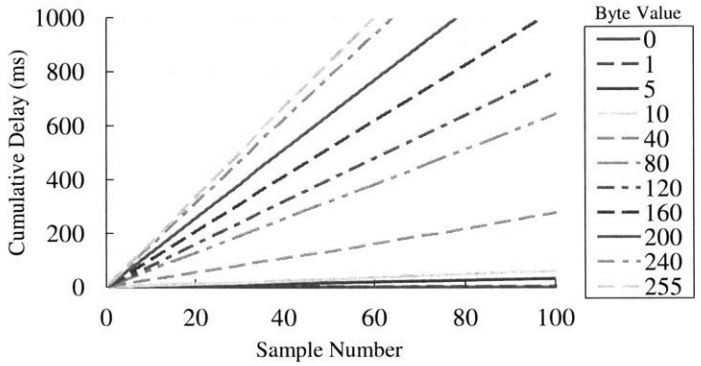

Figure 3-7: Cumulative delay for various byte values against Apache 2.4.7 over LAN.

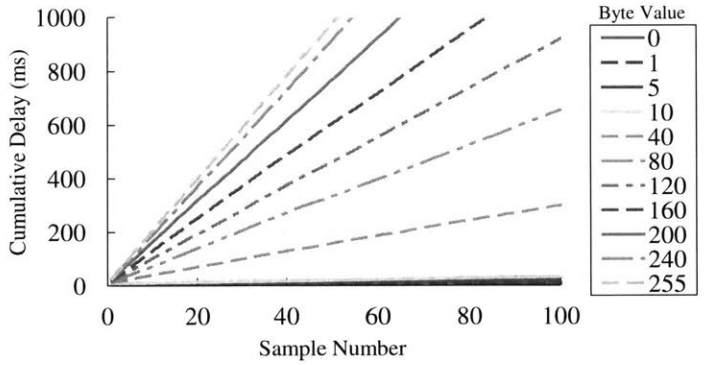

Figure 3-8: Cumulative delay for various byte values against Apache 2.4.7 over 802.11g.

To implement the timing side-channel attack, an adversary simply overwrites the `fmt` pointer with subsequent memory locations, until a location is found that does not result in a crash. The attacker then sends multiple identical queries and records the server response time. He collects the cumulative server response delay from these queries until the expected cumulative delay between subsequent byte values differs by a suitable margin. The attacker repeats this process for a range of valid memory addresses, and thereby collects a sequence of approximate byte values.

If coarse-grained ASLR is in place, then the base address of textttlibc is unknown, but there is no function-level code diversification. In this case, we can leak the base address of textttlibc by leveraging existing fuzzy substring-matching algorithms to match the approximate byte sequence inferred through the timing side-channel attack against the known byte sequence of textttlibc. If an attacker is able to estimate byte values with ± 15 and a 6% margin of error, it is possible to identify the base of textttlibc with only 13 contiguous measured bytes for 54% of possible measurement offsets, and with 40 contiguous measured bytes for 85% of possible measurement offsets. It takes approximately 5000 queries to calculate a byte value based on timing within the 6% margin of error.

At 5000 samples per byte, and an average LAN network delay of $8.46ms$, an attacker can identify a single byte in $43.2sec$, and therefore can identify the base address of textttlibc in less than $561sec$ for more than half the time. Wireless is a bit slower, at $41.94ms$ per query, it will take $209.4sec$ per byte, and a total expected time of $2722.2sec$ to identify the base location of textttlibc.

3.3.2 Coarse-Grained ASLR

The standard implementation of ASLR is "coarse-grained", meaning only the base addresses of the stack, heap, and libraries are randomized. This means that an attacker already knows the locations of all gadgets within the executable itself. The Apache 2.4.7 execution is not position independent, so all of its gadgets are known a priori to an attacker. We show by construction that it is possible to construct a pure ROP payload whose execution time is dependent on the value of an attacker-supplied

address. The ROP payload is essentially equivalent to the following pseudocode, where an attacker will vary the address pointed to by `ptr`.

```
1 i = 0;
2 while (i < 500000 * (int) *ptr)
3     i++;
```

We emphasize that this ROP attack does not rely on the existence of a conditional loop with a conveniently vulnerable data pointer. Rather, the loop is entirely constructed from gadgets that already exist in the Apache executable. This gives the attacker the ability to increase the effectiveness of the attack, by multiplying the value at a given location by a large fixed constant (500000 in this example) in order to amplify the timing differences between different byte values. In this case, an attacker can send only a single malicious payload per byte, and reduce the overhead of network delay for repeated sampling. We estimate that using this optimized ROP payload, an attacker can reasonably learn a given byte value in $1.39sec$ over either LAN or wireless.

Even though the location of gadgets within the Apache executable are known, there are no sufficiently powerful gadgets (system calls) to perform a dangerous attack. So the attack process in its entirety will look something like this:

1. Collect gadgets from the fixed-location target executable.

2. Use this collection of gadgets to construct a timing loop, whose execution time depends on the value of a chosen location in memory.

3. Send the ROP payload to the victim machine, scanning through sequential byte locations in textttlibc.

4. Use the fuzzy substring matching algorithm to identify the base address of textttlibc, thereby breaking ASLR.

5. Construct a final, malicious payload, that can make use of system call gadgets contained in textttlibc.

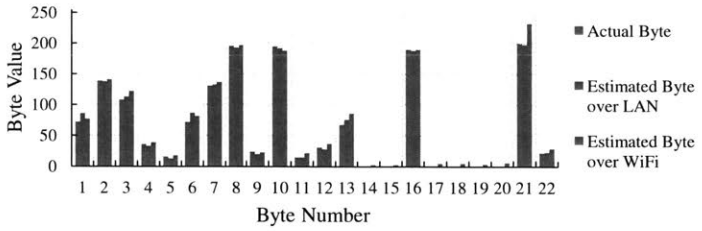

Figure 3-9: Estimated and actual byte values for a chosen offset in textttlibc.

We tested this ROP payload in our experimental setup against Apache 2.4.7 and gtextttlibc2.16. We found that 22 contiguous bytes were required to completely determine the randomized textttlibc base address. This value falls nicely into the theoretical expected range (13 - 40) required byte measurements required, and was possible to perform in only $30.58sec$ by using the ROP optimization described above.

3.3.3 Medium-Grained ASLR

In medium-grained code diversification, such as address space layout permutation (ASLP), function entry points within libraries as well as executables are randomized. In this case, an attacker will not have access to a pure ROP payload, as shown in section 3.3.2. In this case, an attacker must make use of a vulnerable variable or pointer that leaks information through either of the previously described fault analysis or timing side-channel attacks. The attack proceeds as follows.

1. Identify a pointer overwrite vulnerability, such as the Apache vulnerability in section 3.3.

2. Overwrite the vulnerable pointer with arbitrary locations, until non-crashing regions are found.

3. Identify potential function entry points, by identifying byte sequences that correspond to the boilerplate function preamble (saving EBP, updating EBP, etc.).

4. Profile the function using the timing side-channel attack, and compare to the known profiles of functions in textttlibc and the executable.

5. Repeat steps 3 and 4 until enough dangerous gadgets have been found to enable a return-to-textttlibc attack.

3.3.4 Fine-Grained ASLR

Finally, the most resilient form of ASLR is fine-grained ASLR, in which even basic blocks can be re-ordered. Even in this worst-case scenario, if an attacker has access to a buffer overflow that overwrites some timing-dependent variable, he could still slowly attempt to profile textttlibc until encountering a basic block that makes a system call. gtextttlibc2.16 is 1.3MB in size, with 60 system call wrappers. Assuming basic block randomization, we expect to have to profile at least 22,000 bytes until encountering an aligned basic block that makes a system call. At 43.2 seconds per byte over LAN, we would expect such an attack to take approximately 11 days, and at 561 seconds per byte over wireless, we would expect 143 days.

This timing side-channel attack is prohibitively slow against fine-grained ASLR, particularly over a wireless network. However, we point out that even if the earliest discovered gadgets are not system call wrappers, they can still be used to speed up the discovery process. If, during the discovery phase, the attacker learns the locations of the enough gadgets to construct the optimized ROP timing attack, described in section 3.3.2, then it would be possible to speed up the attack considerably. If the first gadgets discovered were sufficient to build this optimized ROP payload, then at a rate of $1.39sec$ per byte, the full attack against fine-grained ASLR would only take 8.5 hours. However, it is difficult to compute an accurate expected time for this type of multi-phase attack. The gadgets we used in constructing the optimized ROP payload against Apache relied on a few very uncommon gadgets, so finding the gadgets to construct the optimized ROP payload would be more time consuming than simply discovering system call gadgets in the first place. It is an open question as to whether or not an attacker could build this type of optimized ROP payload with very common gadgets, in which case timing side-channel attacks would certainly be more feasible against even fine-grained ASLR systems.

Chapter 4

Control Flow Integrity

In response to the growing recognition for the ability of memory disclosure vulnerabilities to completely mitigate all practically enforced defenses against code reuse, research has lately turned towards a technique known as control flow integrity (CFI). The basic idea behind CFI is to first generate a control flow graph (CFG) for a given program beforehand. A CFG will have one node per basic block, and a directed edge between two basic blocks if and only if a control flow transfer is made between those basic blocks during valid execution of the program. The CFG is then enforced at runtime, so even if an attacker is able to overwrite a return address or function pointer, he is not able to divert control flow in any unintended ways. CFI only needs to enforce indirect control flow transfers, namely indirect jumps and returns, because only indirect control flow transfers can potentially have their targets maliciously influenced by an attacker.. CFI is particularly robust in that it does not rely on any "secret state," unlike code diversification defenses which require the randomized state of the proccess address space to remain secret in order to be effective. As we have shown in chapters 2 and 3, defensive measures that rely on hidden state are even more susceptible to memory disclosure than previously thought, so the fact that CFI does not rely on hidden state makes it particularly appealing.

Figure 4-2 illustrates a simple control flow graph for a simple merge sort program. This diagram represents the "ideal" case, in which each point of control flow transfer has only a single valid target, so an attacker would have absolutely no freedom in

```
1  MergeSort(A[], B[], n) {
2      SplitMerge(A, 0, n, B);
3  }
4
5  CopyArray(B[], iBegin, iEnd, A[]) {
6      for(k = iBegin, k < iEnd; k++)
7          A[k] = B[k]
8  }
9
10 SplitMerge(A[], iBegin, iEnd, B[]) {
11     iMiddle = (iEnd + iBegin) / 2
12     SplitMerge(A, iBegin, iMiddle, B);
13     SplitMerge(A, iMiddle, iEnd, B)
14     Merge(A, iBegin, iMiddle, iEnd, B);
15     CopyArray(B, iBegin, iEnd, A);
16 }
17
18 Merge(A[], iBegin, iMiddle, iEnd, B[]) {
19     <merge sorted arrays A and B>
20 }
```

Figure 4-1: Merge sort pseudocode

overwriting return addresses or function pointers, as any deviation at all from normal control flow could immediately be detected. The typical program does not have such an ideal CFG, as indirect control flow transfers could potentially have any number of valid targets, and it is difficult to determine beforehand the set of valid targets. Runtime enforcement also presents a difficult challenge. In order to see widespread adoption, CFI should have fairly minimal runtime overhead, in the range of 116%. We shall defer the question of runtime enforcement to future research, and instead focus on the theoretical limits of CFI. Namely, we examine how to produce the most precise CFG possible. As motivation, chapter presents current work in the domain of practical CFI enforcement, and describes why these efforts have proven insufficient.

4.1 Practical CFI

The two most promising forms of practical CFI implementation are Compact Control Flow Integrity and Randomization (CCFIR) [9] developed by Song et al. and bin-CFI [8], proposed by Zhang et al. CCFIR is somewhat stricter, but requires debug and

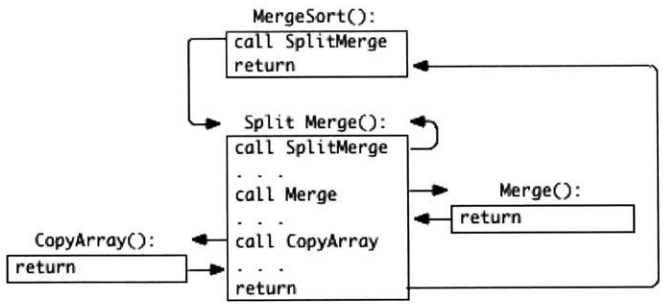

Figure 4-2: Control Flow Graph for Merge Sort

relocation information. bin-CFI is less precise, mainly because it uses only 2 IDs instead of 3, but can be implemented on commercial off-the-shelf software, without the need for debug or relocation information. The two approaches are fairly similar in principle. Both of them label the targets of indirect control flow transfers with an ID. CCFIR uses a three ID scheme, with a separate ID for each of the following:

1. Function calls and indirect jumps.

2. Returns from non-sensitive functions.

3. Returns from sensitive functions.

Where they define "sensitive functions" as system call wrappers in textttlibc. CCFIR then enforces these allowable control flow transfers at runtime by including a "springboard" code section, which contains stubs for direct jumps for each valid jump target, and each valid return target. Indirect control transfers are then only allowed to jump to locations within the springboard section, with type-checking to guarantee that the springboard stub's ID matches the intended control flow transfer ID. However, there is nothing to prevent an attacker from choosing from *any* of the stubs within the entire springboard section (with appropriate ID) when overwriting a function pointer or return address. This is the fundamental problem with practical CFI. A clever enough attacker may still find a suitable way of linking together the available gadgets to construct an exploit. The bin-CFI approach is fairly similar to CCFIR, but makes use of only two IDs instead of three, thus making it even more susceptible to exploitation.

1. All function returns and jumps.

2. Function calls.

ROPguard follows a similar idea. Although it is not exactly categorized as CFI, as it does not strictly follow a notion of a control flow graph, it does behave similarly. ROPguard identifies a set of "critical functions", namely CreateProcess, VirtualProtect, VirtualAlloc, LoadLibrary, and various file I/O functions, and attempts to enforce the guarantee that these critical functions can only be called in an intended manner. Upon calling a critical function, ROPguard employs a few heuristics to determine whether the call was done safely or not. These heuristics include: checking for a valid return address, inspecting the stack for the critical function's address, and unwinding stack frames (if the program has been compiled to use register ebp as a frame pointer). While these are all useful and practical heuristics, ROPguard suffers from many of the same problems as practical CFI implementations, in that it is not restrictive enough in truly constraining control flow transfer to only the intended targets. Critical functions can still be called directly through wrapper gadgets, and the possibility still exists for an attacker to setup a stack frame in such a way as to control arguments to these critical calls.

4.2 Limitation of Practical CFI

The motivation behind using only 2 - 3 IDs, rather than attempting to implement "ideal CFI", as described by Abadi, is twofold. Firstly, it greatly reduces the runtime overhead of enforcing a complex CFG, and secondly it circumvents the difficult problem of generating a very precise CFG. It was believed that this limited form of practical CFI was sufficient to prevent attacks. Both CCFIR and bin-CFI reported a vast reduction in the number of available gadgets to an attacker, but the fact remained that these practical CFI implementations still leave quite a lot of flexibility to the attacker when choosing control flow transfer targets. Figure 4-3 illustrates this point. The red edges indicate transfers that would be allowed under practical CFI

implementations, that should not be allowed in ideal CFI. Some of these spurious red edges have been omitted for clarity, but they indicate that in practical CFI, any return address can transfer control to essentially *any* return site, and any indirect call can essentially transfer control to *any other* entry point.

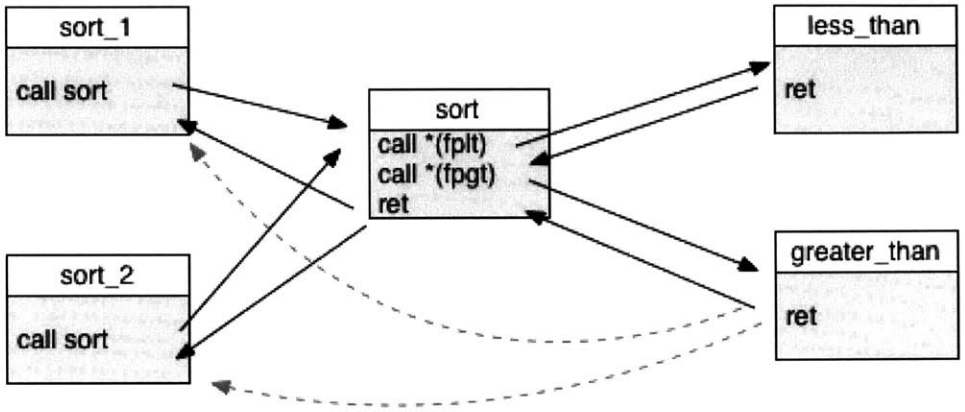

Figure 4-3: Practical CFG

Portokalidis et al. [10] recently showed that even with these practical CFI enforcement techniques in place, there is still sufficient freedom in control flow transfers to construct a meaningful exploit [10]. They showed a proof of concept exploit against Internet Explorer in which it is possible to chain gadgets together in a manner that results in calling VirtualProtect to make the program's code segment writable, and then inject a shellcode with memcpy. Furthermore, the authors go on to identify the number of sensitive gadgets in various common Windows dynamic-link libraries that are not sufficiently protected through practical CFI. They examine dynamic-link libraries from IE9, Adobe Reader, Firefox, and various Microsoft Office applications. The results showed that the most vulnerable library, xul.dll from Firefox, presented 644 insufficiently protected critical gadgets, and even the least vulnerable library, shell32.dll, still presented 5 unprotected critical gadgets (with most other libraries falling in the 100 - 200 range). This proof of concept exploit proves that 2 - 3 ID schemes for CFI are not sufficient, because the overly permissive control flow graphs leave too much leeway to an attacker.

Chapter 5

Improving Control Flow Integrity

5.1 Pointer Analysis

Most of the uncertainty in the generation of control flow graphs comes from the use of function pointers. A function pointer's value is not necessarily known at compile time, so it can be difficult to determine the set of possible valid values. The corresponding node in a control flow graph for a function pointer about which no information can be learned, will have edges connecting to every other valid function entry point. This leaves a lot of freedom to an attacker in choosing how to order the execution of gadgets in constructing his malicious payload. There are some pathological cases in which truly no information can be learned about the targets of a function pointer, as in the code snippet in figure 5.1, in which the function pointer `foo` has it's value determined by the commandline argument to the program.

```
1 void (*foo)();
2 foo = argv[1];
3 foo();
```

Listing 5.1: Dereferencing an unknown function pointer.

However, in most cases, existing static analysis techniques are able to determine sets of potential targets taken by a function pointer. Conventionally, these sets are divided into "must alias", "may alias", and "no alias" classifications. Every function

in a pointer's "must alias" set will be pointed to at least once during the programs execution, no function in the pointer's "no alias" set will be pointed to, and functions in the "may alias" set may or may not be pointed to.

In this chapter, we describe existing pointer analysis techniques, their shortcommings, and what we have done to build upon these techniques in order to more accurately build control flow graphs.

5.1.1 Flow Sensitivity

Flow-sensitive pointer analysis would take into account the order of execution of instructions. Consider the example in code snippet 5.2. Pointer `p` first points to `x`, then is used as an argument in calling function `foo`, then points to `y`, and is finally used as an argument in calling function `bar`. Although both `x` and `y` are in pointer `p`'s "must alias" set, we can see by inspection that when `foo` is called with `p` as an argument, `p`'s value can only ever be `&x`, and similarly when function `bar` is called, `p`'s value will only ever be `&y`, based on the order of execution of instructions in this simple program. Flow-sensitive analysis is provably NP-hard for all but the most trivial of programs, so it is not used in practice.

```
1 p = &x;
2 foo(p);
3 p = &y;
4 bar(p);
```

Listing 5.2: Flow-sensitive use of function pointers.

5.1.2 Context Sensitivity

When a function pointer takes its value from the return value of a function call, it's difficult to determine practically what that value might be. Most existing static pointer analysis techniques make a "worst call" assumption when considering the set of return values from a function. In the following example, we have a function `foo` which takes a function pointer as an argument, and simply returns that function

```
1 fptr foo(fptr x){return x;}
2 {
3     a = foo(a);
4     b = foo(b);
5 }
```

Figure 5-1: Context-sensitive use of function pointers

pointer. In this simple program, the function `foo` sometimes returns `a` and sometimes returns `b`. An algorithm that is not context-sensitive would consider that any call to `foo` could potentially return either `a` or `b`, and does not consider the arguments with which `foo` was called. This would result in the spurious points-to relationships $a \rightarrow b$ and $b \rightarrow a$.

5.2 Static Analysis Algorithms

In practice, there are two primary algorithms used for static points-to analysis: **Andersen's** algorithm and **Steensgaard's** algorithm. In Andersen's algorithm, the source code of a program is parsed line by line for assignment statements of the form $p = (\&a|a| * a)$ or $*p = (\&a|a| * a)$. Each assignment statement is used to update a directed graph of points-to relationships. For example, the assignment statement $p = \&a$ would add the edge $p \rightarrow a$, indicating that p may point to a. Similarly, the assignment $p = q$ will add edges $p \rightarrow r$ for all nodes r that are also pointed to by q. If a subsequent statement changes the set of locations that can be pointed to by q, we need to add them to the set of locations that can be pointed to by p as well. Steensgaard's algorithm is similar, but limits the outdegree of nodes in the points-to graph to 1. This is done by combining multiple nodes into a single equivalence class, if all nodes in that equivalence class have in-edges from the same parent node. This optimization reduces the running time from $O(n^3)$ to $O(n)$ at the cost of some precision.

In our static analysis for generating control flow graphs, we use an implementation of pointer analysis similar to Andersen's algorithm. We additionally make use of the fact that the pointers we are interested in analyzing are function pointers. Therefore,

after we generate a set of functions that "may" be pointed to by a given function pointer, we can filter out any of these functions that do not have the same parameter type signature as the function pointer in question.

5.3 Hybrid Control Flow Graphs

The main practical limitations of the precision of static pointer analysis techniques are the lack of flow-sensitivity and the lack of context-sensitivity. In this section we describe how pointer analysis, and therefore control flow graphs, can be improved by incorporating runtime information into our analysis. We use LLVM to transform source code into LLVM intermediate representation (LLVM IR). We are then able to instrument this intermediate representation of the code in such a way that the resulting executable records the execution of each basic block and each indirect control flow transfer. We then run the instrumented executable with a suitably broad range of sample input test cases, and use the recording control flow transfers to generate a "dynamic" control flow graph. Every control flow transfer in this dynamic CFG represents a valid edge. This dynamic CFG will necessarily be a subset of the static CFG produced using algorithmic analysis. Every edge capturing through this dynamic analysis will necessarily be an edge in the "true" control flow graph of valid control transfers. However, there may be some edges in the static control flow graph, that are not present in the dynamic control flow graph, that may still represent legitimate control flow transfers. This can be the case if our sample test input does not exercise every part of the program.

The advantage of dynamic analysis is that it comes with context-sensitivity and flow-sensitivity built-in. Because the dynamic CFG is generated by actually running the program, the order of instruction execution, and the values returned by function calls will always be taken into account when generating such a CFG. The figure below shows the CFGs produced through our static and dynamic analyses for the simple program below. Because static analysis cannot be flow-sensitive and still run in polynomial time, the static CFG has a spurious edge between main and foo. The

Figure 5-2: CFGs that demonstrate imprecisions in flow-sensitive code in static analysis (left) and dynamic analysis (right)

dynamic analysis on the other hand, will never have this spurious edge, because no actual execution of this code will result in the function foo being called. In this case, the dynamic CFG is more precise than the static CFG, as it is strictly a subset of the static CFG, and omits no edges that represent valid control flow transfers.

```
1 void (*fp)();
2 fp = foo;
3 fp = bar;
4 fp();
5 return;
```

However, we can also have cases in which the dynamically generated CFG does omit edges that represent valid transfers. In the following code snippet, the program calls function `large` only if it has be run with more than 1000 commandline arguments. Otherwise, it calls function `small`. If our set of input test cases does not happen to include a case where the program is called with more than 1000 inputs, the dynamic CFG will never include this valid execution path, whereas the static CFG does correctly capture it, as shown in listing 5.3.

```
1 int main(int argc, char **argv) {
2     if (argc > 1000)
3         large();
4     else
5         small();
6 }
```

Listing 5.3: Rarely exercise execution path.

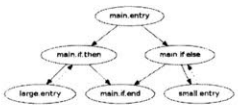

Figure 5-3: Static analysis CFG

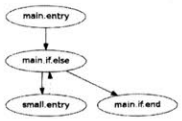

Figure 5-4: Dyanmic analysis CFG

5.4 Analysis

In this section we present the metrics and collected data that characterize precisely how a control flow graph is changed with the inclusion of dynamic information. We present 4 different graph metrics, describe their relevance in the domain of CFI, and present the data collected for our test suite of applications.

5.4.1 Per-Node Metrics

As a simple metric for characterizing the reduction in complexity of a control flow graph, we can use **cyclomatic complexity**. Cyclomatic complexity is a measure for the number of linearly independent execution paths through a directed graph with one entry point and one exit point. In our case, this direct graph corresponds to the control flow graph in which each node is a basic block, and each edge represents a valid control flow transfer between basic blocks. The cyclomatic complexity can be computed directly using the close formula:

$$M = E - N + 2P$$

Where M is the cyclomatic complexity, E is the number of edges in the CFG, N is the number of nodes, and P is the number of connected components.

Degree centrality is another relatively straightforward way to characterize a CFG's complexity. The indegree centrality for a node is simply the number of directed edges that point towards that node. In the context of a CFG, this is the number of basic blocks that are permitted to make a control flow transfer to that node. Similarly, the outdegree centrality of a node is the number of basic blocks to which it is permitted to transfer execution. The normalized degree centrality for a node is the total fraction

of nodes in the graph with which that node shares an edge. It gives a measure of the number of potential targets at any given control flow transfer.

Betweenness centrality is a per-node metric from network theory that represents the number of shortest paths between all pairs of nodes in the network that pass through a given node. That is, the betweenness centrality g for a node v is given by:

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Where $\sigma_{st}(v)$ is the number of shortest paths between s and t that pass through node v, and σ_{st} is the total number of shortest paths between node s and t.

We can think of betweenness centrality as a rough metric for how easy it is to reach a given basic block. If an attacker is stringing together specific gadgets in order to construct a malicious payload that still respects a program's CFG, he is likely to need to make use of the shortest paths between these specific sensitive gadgets. Because nodes with high betweenness centrality are more likely to appear in CFG-respecting ROP payloads, we can use this set of nodes with high betweenness scores as a minimal set of critical program points to monitor. We describe in greater detail in the chapter on future work how a defensive tool might make use of this information for monitoring and preventing ROP attacks.

We also measure the closeness centrality for nodes in static and dynamic control flow graphs. Closeness centrality is defined as the inverse of the "farness" of a node, and "farness" is is the sum of the values of a node's shortest paths to all other nodes in the network. The average closeness centrality of a graph then gives a measure of how tightly linked the graph is.

5.4.2 Results

For our test setup, we used examined four applications present in the LLVM test-suite module: bison, dparser, spiff, and SIBsim4. Table 5.2 shows the calculated cyclomatic complexity, average node degree centrality, average node betweenness centrality, and average node closeness centrality metrics for inter-procedural control flow graphs gen-

erated for these applications. Cyclomatic complexity saw the greatest change in value between the static and dynamic analyses, ranging from an 83% reduction in the case of spiff, to a 55% reduction in the case of bison. This decrease is essentially proprtional to the decrease in total number of edges in the control flow graph. Degree centrality exhibited similar behavior, with decreases ranging between 25% and 38%. By comparison, betweenness centrality remained relatively stable, seeing a reduction of less than 5% for all applications other than bison, which saw a reduction of 16%. This indicates that the edges that are missing from dynamic analysis have a disproportionately minor effect on the pair-wise shortest paths betwen nodes. Finally, the average closeness centrality also saw reductions between 65% and 39%, suggesting that the total distance between any pair of nodes in the control flow graph was increased by approximately a factor of 2, on account of the missing edges in the dynamic analysis that were present in the static analysis.

Table 5.1: CFG reduction in complexity

Application	Cyclomatic Complexity	Avg Degree Cent.	Avg Betweenness Cent.	Avg Closeness Cent.
bison (static)	351	0.04374	0.0119	0.368
bison (dynamic)	157	0.0332	0.0143	0.225
dparser (static)	588	0.029	0.0084	0.339
dparser (dyamic)	96	0.018	0.0086	0.1184
spiff (static)	131	0.060224	0.0219	0.371
spiff (dynamic)	22	0.0397	0.023	0.14116
SIBsim4 (static)	104	0.06697	0.0244	0.3765
SIBsim4 (dynamic)	22	0.04178	0.0255	0.177

Table 5.2: Change in graph complexity metrics in dynamic CFG generation in comparison to static analysis.

Chapter 6

Discussion of Future Work

6.1 Refinement of Dynamic Analysis

It is possible to make use of the frequency of execution of paths in a CFG to further refine CFI. Namely, we could use the frequency of execution of certain control flow paths as a means of detecting ROP attacks. If we observe certain infrequent execution paths being used more frequently than expected, it may be an indication of a ROP attack. In order to be effective, this technique would have to rely on principled input sampling that accurately reflects the expected real-world input distribution.

Furthermore, it could be advantageous to collect detailed path profiling information, rather than simply viewing a CFG as a collection of basic block nodes and control flow transfer edges. As a simple example, consider the `if-else` code snippet shown below.

```
1 if (input > THRESHOLD)
2     optimized_algorithm()
3 else
4     slow_algorithm()
```

Static CFG analysis will produce an edge for both function calls `optimized_algorithm` and `slow_algorithm`. Even our dynamic analysis will simply have two edges in the CFG for each of these function calls, assuming the input set is broad enough to cover both cases. Nothing in the CFG indicates that not both of these control flow trans-

fers can occur in the same execution of the surrounding function. More generally, the CFG edges that are taken during execution of a program influence which CFG edges can be considered valid during subsequent control flow transfers. The propagation of this information between edges in a CFG may not always be as apparent as in this straightforward `if-else` example. Borrowing from the existing machine learning techniques, it may be possible to produce a control flow graphs whose edge transfer probabilities can be computed based on previous edges traversed.

6.2 Enforcement

Conventional control flow integrity enforcement will either always allow or always disallow a control flow transfer, based on its presence or absence in the control flow graph. Using dynamic analysis as a heuristic to inform our CFG means that control transfers are not strictly classified as "always allowed" or "always disallowed". Namely, a control flow transfer that is permitted in our static CFG, but does not appear in the dynamic CFG will be of ambiguous validity. It is possible that this transition could be the result of imprecise static analysis, or it could be the result of a dynamic input sample set that does not fully exercise all valid control flow paths. In these uncertain cases, we should consider various defensive techniques that can be used that do not break valid programs.

Code annotation. It is possible to inspect CFGs for nodes with very large out-degree in the statically generated CFG, and relatively small out-degree in the dynamically generated CFG. This would likely indicate the usage of an ambiguous function pointer with many possible targets, but in practice the programming only ever intends the function pointer to take on a small set of possible values. In such cases, one could vastly simplify a control flow graph by prompting the programmer to annotate his or her usage of these ambiguous function pointers.

Stack unwinding can be used to detect the presence of a ROP attack in progress. The idea behind stack unwinding is to use the state of the stack to reconstruct the previous function calls. During a ROP attack, where return addresses and saved frame

pointers have been overwritten with gadget locations, we would expect to be unable to accurately unwind the stack. However, full stack unwinding is not always feasible, even in correctly functioning programs. In cases where the compiler optimizes away usage of base pointers, it can be impossible to perform stack unwinding. We would have to weigh the trade-offs of the performance overhead incurred by occasionally unwinding the stack, as well as opting out of compiler optimizations that do away with frame pointers.

Address space re-randomization. We have already shown the ineffectiveness of address space randomization in the presence of ROP attacks due to side-channel memory disclosure. However, when used in conjunction with control flow integrity, it may be possible to achieve protection that surpasses the capabilities of either one alone. When a control flow transfer occurs that is suspicious, due to its infrequency or nonexistence in the dynamic CFG, but is not prohibited, due to its presence in the static CFG, it could be advantageous to perform some re-randomization of memory. This type of defense would be especially effective in preventing bootstrapped ROP attack, that relies on first leaking information from the target, and subsequently exploiting the leaked information to deploy a second leaked payload. The exploit we generated against Apache falls into this category, as well as the exploit against Internet Explorer developed by Boneh et al. [4].

Chapter 7

Discussion of Future Work

7.1 Refinement of Dynamic Analysis

It is possible to make use of the frequency of execution of paths in a CFG to further refine CFI. Namely, we could use the frequency of execution of certain control flow paths as a means of detecting ROP attacks. If we observe certain infrequent execution paths being used more frequently than expected, it may be an indication of a ROP attack. In order to be effective, this technique would have to rely on principled input sampling that accurately reflects the expected real-world input distribution.

Furthermore, it could be advantageous to collect detailed path profiling information, rather than simply viewing a CFG as a collection of basic block nodes and control flow transfer edges. As a simple example, consider the `if-else` code snippet shown below.

```
1 if (input > THRESHOLD)
2     optimized_algorithm()
3 else
4     slow_algorithm()
```

Static CFG analysis will produce an edge for both function calls `optimized_algorithm` and `slow_algorithm`. Even our dynamic analysis will simply have two edges in the CFG for each of these function calls, assuming the input set is broad enough to cover both cases. Nothing in the CFG indicates that not both of these control flow trans-

fers can occur in the same execution of the surrounding function. More generally, the CFG edges that are taken during execution of a program influence which CFG edges can be considered valid during subsequent control flow transfers. The propagation of this information between edges in a CFG may not always be as apparent as in this straightforward `if-else` example. Borrowing from the existing machine learning techniques, it may be possible to produce a control flow graphs whose edge transfer probabilities can be computed based on previous edges traversed.

7.2 Enforcement

Conventional control flow integrity enforcement will either always allow or always disallow a control flow transfer, based on its presence or absence in the control flow graph. Using dynamic analysis as a heuristic to inform our CFG means that control transfers are not strictly classified as "always allowed" or "always disallowed". Namely, a control flow transfer that is permitted in our static CFG, but does not appear in the dynamic CFG will be of ambiguous validity. It is possible that this transition could be the result of imprecise static analysis, or it could be the result of a dynamic input sample set that does not fully exercise all valid control flow paths. In these uncertain cases, we should consider various defensive techniques that can be used that do not break valid programs.

Code annotation. It is possible to inspect CFGs for nodes with very large out-degree in the statically generated CFG, and relatively small out-degree in the dynamically generated CFG. This would likely indicate the usage of an ambiguous function pointer with many possible targets, but in practice the programming only ever intends the function pointer to take on a small set of possible values. In such cases, one could vastly simplify a control flow graph by prompting the programmer to annotate his or her usage of these ambiguous function pointers.

Stack unwinding can be used to detect the presence of a ROP attack in progress. The idea behind stack unwinding is to use the state of the stack to reconstruct the previous function calls. During a ROP attack, where return addresses and saved frame

pointers have been overwritten with gadget locations, we would expect to be unable to accurately unwind the stack. However, full stack unwinding is not always feasible, even in correctly functioning programs. In cases where the compiler optimizes away usage of base pointers, it can be impossible to perform stack unwinding. We would have to weigh the trade-offs of the performance overhead incurred by occasionally unwinding the stack, as well as opting out of compiler optimizations that do away with frame pointers.

Address space re-randomization. We have already shown the ineffectiveness of address space randomization in the presence of ROP attacks due to side-channel memory disclosure. However, when used in conjunction with control flow integrity, it may be possible to achieve protection that surpasses the capabilities of either one alone. When a control flow transfer occurs that is suspicious, due to its infrequency or nonexistence in the dynamic CFG, but is not prohibited, due to its presence in the static CFG, it could be advantageous to perform some re-randomization of memory. This type of defense would be especially effective in preventing bootstrapped ROP attack, that relies on first leaking information from the target, and subsequently exploiting the leaked information to deploy a second leaked payload. The exploit we generated against Apache falls into this category, as well as the exploit against Internet Explorer developed by Boneh et al. [4].

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *the 12th ACM conference on Computer and communications security (CCS)*, 2005.

[2] H. Sacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[3] Papas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programing using in-place code randomizations. In *Security and Privacy*, 2012.

[4] D. Boneh, D. Mazieres, A. Mashtizadeh, A. Belay, A. Bitau, Hacking Blind, 2014.

[5] X. Chen, ASLR Bypass Apocalypse in Recent Zero-Day Exploits, In *FireEye* blog, 2013

[6] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, Breaking the memory secrecy assumption, In *the Second European Workshop on System Security (EUROSEC)*, 2009.

[7] Aleph One, Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14-16, 1996

[8] M. Zhang, R. Sekar, Control Flow Integrity for COTS Binaries, In *22nd USENIX Security Symposium*, 2013

[9] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, Practical control flow integrity and randomization for binary executables, In *Proceedings of the 2013 Security and Privacy Symposium*, 2013

[10] E. Goktas, E. Athanasopoulos, H. Bos, G. Portokalidis, Out of control: overcoming control-flow integrity, IEEE, 2014.

[11] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy* (2012).

[12] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, Where'd my gadgets go? In *Security and Privacy* 2012.

[13] M. Franz, S. Brunthaler, P. Larsen, A. Homescu, and S. Neisius, Profile-guided automated software diversity, In *the 2013 IEEE/ACM International Symposium on Code Generation and Optimizatio (CGO)* 2013.

[14] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, Binary stirring: self-randomizing instruction addresses of legacy x86 binary code, In *Computer and Communications Security (CCS)*, 2012.

[15] R. Hund, C. Willems, and T. Holz, Practical timing side channel attacks against kernel space aslr. In *Security and Privacy*, 2013.

[16] S. A. Crosby, D. S. Wallach, and R. H. Riedi, Opportunities and limits of remote timing attacks, *ACM Transactions on Information and System Security*, 2009

[17] OSWALD, D., RICHTER, B., AND PAAR, C. Side-channel attacks on the Yubikey 2 one-time password generator. In *Research in Attacks, Intrusions, and Defenses (RAID)*. 2013.

[18] ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. G-free: Defeating return-oriented programming through gadget-less binaries. In *ACSAC'10* (2010).

[19] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS* (2004).

[20] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption. In *EUROSEC* (2009).

[21] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Security and Privacy* (2013).

[22] TUNSTALL, M., MUKHOPADHYAY, D., AND ALI, S. Differential fault analysis of the advanced encryption standard using a single fault. In *the International Conference on Information Security Theory and Practice (WISTP)* (2011).