Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code*

Jeff Seibert
MIT Lincoln Laboratory
244 Wood St.
Lexington, MA
jeffrey.seibert@ll.mit.edu

Hamed Okhravi
MIT Lincoln Laboratory
244 Wood St.
Lexington, MA
hamed.okhravi@ll.mit.edu

Eric Söderström
MIT CSAIL
32 Vassar St.
Cambridge, MA
e_k_s@mit.edu

ABSTRACT

Code diversification has been proposed as a technique to mitigate code reuse attacks, which have recently become the predominant way for attackers to exploit memory corruption vulnerabilities. As code reuse attacks require detailed knowledge of where code is in memory, diversification techniques attempt to mitigate these attacks by randomizing what instructions are executed and where code is located in memory. As an attacker cannot read the diversified code, it is assumed he cannot reliably exploit the code.

In this paper, we show that the fundamental assumption behind code diversity can be broken, as executing the code reveals information about the code. Thus, we can *leak information without needing to read the code*. We demonstrate how an attacker can utilize a memory corruption vulnerability to create side channels that leak information in novel ways, removing the need for a memory disclosure vulnerability. We introduce seven new classes of attacks that involve fault analysis and timing side channels, where each allows a remote attacker to learn how code has been diversified.

Categories and Subject Descriptors

[**Security and Privacy**]: Systems Security—*Information Flow Control*; [**Security and Privacy**]: Software and Application Security

Keywords

Information Leakage, Code Diversity, Memory Disclosure, Side-Channel Attacks, Address Space Layout Randomization

1. INTRODUCTION

Decades of research have gone into solving the problem of memory corruption bugs [39]. These bugs are particularly notorious, as they can often be exploited by a remote attacker to execute arbitrary code on the victim host, effectively compromising that machine. Recently, defenses have progressed enough to prevent attackers from being able to use such bugs to conduct code injection attacks, where an attacker writes code into memory and then causes it to be executed [39]. Attackers desiring to do malicious computations are instead forced to reuse code that already exists in the vulnerable binary [33]. Code reuse attacks have become a commonly used technique for exploiting memory corruption bugs. However, to conduct such attacks, detailed knowledge of the layout of code is needed, including *what* code is in memory, and *where* is it located.

Code diversification techniques have then recently been proposed as a way to defend against code reuse attacks [18, 23, 45]. These techniques preserve the high level semantics of the code, while changing the low level details of the code. Diversification techniques include changing *what* specific machine-level instructions are executed and changing *where* code is located in memory. These techniques rely on the assumption that since the attacker cannot read the code in memory, then he cannot know what code is there nor where it is located, resulting in the attacker not being able to reliably exploit the code.

Attackers have responded by utilizing entropy exhausting attacks [34] and memory disclosure vulnerabilities [36, 38]. Entropy exhausting attacks brute-force the memory space, leveraging the fact that some diversification techniques do not introduce enough randomness into the system and thus can be fairly predictable. This allows an attacker to eventually guess how code has been diversified. However, as entropy is increased through more fine-grained diversification, these attacks will be less effective.

Memory disclosure vulnerabilities leak information about the code by allowing an attacker to directly read contents of memory dynamically during runtime. This allows an attacker to know exactly how code has been diversified without guessing. However, these attacks often require finding two specialized vulnerabilities in the same code. One vulnerability to read unintended memory, facilitating memory disclosure, and another vulnerability to write to unintended memory, facilitating the code reuse attack.

In this paper, we demonstrate that the fundamental assumption underlying code diversity can be broken, allowing an attacker to know how code has been diversified without having to read memory. We show how executing the code leaks information about the code. Specifically, we illustrate and analyze how timing and fault analysis side channels can be used to leak information about the code. The attacks that we introduce expand the number of ways that an attacker can exploit memory corruption bugs even if the code has been diversified. These attacks also have significant advantages over previously discovered attacks. First, they allow leaking information about the code without needing an entropy exhausting, brute-force attack. Second, they only require a single vulnerability that allows an attacker to write to unintended memory, removing

*This work is sponsored by the Defense Advanced Research Projects Agency 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660309.

the requirement of a second vulnerability often needed for memory disclosures [36, 38]. As we will demonstrate, a single memory corruption vulnerability can be all that is needed to both learn the contents of memory and conduct a code reuse attack.

The attacks that we propose demonstrate that side channel attacks can be used by a remote attacker on diversified code, allowing information to be leaked about the code by both analyzing the output of the execution and timing the execution. Side-channel attacks have a long history of being used to leak secret information, however, mostly in the context of learning secret data. Most prominently, side channels have been used to subvert cryptographic applications that operate on some secret key [10, 28, 43]. Many types of side channels have been found and shown to be effective in recovering secret keys. These side channels include timing of execution [9, 10, 12, 27] and caches [2, 8, 41, 46], and analysis of power [24, 28, 32], acoustics [17, 42] and faults [5, 14, 20, 43].

The same side channels that can be applied to computationally secure cryptography can be applied to code diversification because they both have a similar security model. Both code diversification and computationally secure cryptography have been researched due to provably secure techniques being too impractical to implement and deploy. Instead, they fundamentally rely on randomization schemes that require secrets to be kept. However, these secrets are exposed when actual computation happens. The difference being that with code diversification the secret is the code instead of data.

Our findings reveal that information leak vulnerabilities may be even more plentiful and easier to find than what is commonly believed today [39]. Furthermore, code diversification only randomizes the low level details of the code, but by necessity needs to maintain the high level semantics of the code. We demonstrate that an attacker can leverage his knowledge of high level semantics of the code to learn how the low level details have been changed.

The contributions of this paper include:

• We introduce new classes of side channel attacks that result in information being leaked about diversified code. We show how leveraging memory corruption vulnerabilities through overwriting data variables, data pointers, and code pointers can all reveal information about the code through fault analysis and timing side channels. We introduce three fault analysis attacks and four timing attacks. For each attack, we describe its capabilities and limitations.

• We evaluate how effective different side channels are in determining how code has been diversified. As different vulnerabilities will allow different information leaking capabilities, we conduct experiments to determine how useful these capabilities are in identifying real code. As `libc` is linked to nearly every program, we measure aspects of it such as timing, return values, and instruction locations. We find that knowing such information can often reveal how code has been diversified, and depending on the information leak, can allow the attacker to find up to 97% of gadgets contained in `libc`.

• We demonstrate a side channel attack on a vulnerable Apache web server and show that with a single vulnerability that is a buffer overwrite, we can leak enough information to execute practical, malicious payloads. It is widely believed that to leak enough information to defeat the ubiquitously deployed ASLR an extra memory disclosure vulnerability is needed. However, we show how a memory corruption vulnerability can be used to gain the same information and then be used to conduct a code reuse attack.

The rest of the paper is organized as follows. We discuss related work in §2. We introduce our side channel attacks in §3 and demonstrate how effective different side channels are in §4. We present a framework for incrementally leaking information using side channels in §5 and demonstrate a practical side channel attack in §6. We discuss potential defenses in §7 and conclude in §8.

2. RELATED WORK

2.1 Code Diversification

Leveraging diversity as a means of mitigating memory corruption bugs has been utilized for many years [15]. Recently though, with the rising prevalence of code reuse attacks, many techniques have been proposed to diversify code. Address Space Layout Randomization (ASLR) [40] is deployed in most modern operating systems today, where the base addresses of the stack, heap, and libraries are randomized. However, most implementations do not randomize the location of the executable [34]. Several techniques have then been proposed to complement ASLR.

Some works have focused on diversifying binaries at the instruction level, where no source code or debugging information is available. Pappas *et al.* [31] find instructions in the code that can be replaced by other, equivalent instructions. This modifies many useful instruction sequences that an attacker might use to conduct his attack, without increasing the size or significantly increasing execution time of the binary. Hiser *et al.* [19] build a virtual machine that allows instruction level randomization, where each instruction can be placed randomly throughout memory. The virtual machine keeps track of the order in which instructions should be executed, and fetches and decodes them as they are needed. Homescu *et al.* [21] focus on designing a library that hooks into just-in-time (JIT) compilers and randomizes their emitted code by inserting NOP instructions randomly.

Other works have focused on the randomization of function or basic block locations, which is also called *fine-grained ASLR*. Kil *et al.* [26] propose to randomize the order of functions in the code and find that it has very low performance overhead. Wartell *et al.* [45] diversify at the basic block level, and instrument code to randomly place basic blocks every time the code is executed.

Compiler based techniques have also been proposed for diversification of code. Onarlioglu *et al.* [29] propose an approach which inserts NOP instructions in the code to correct the alignment and prevent unaligned free-branch instructions used in code reuse attacks. Franz *et al.* [16, 23] have built NOP insertion into LLVM, where NOPs are randomly placed throughout the code. Guiffrida *et al.* [18] also modify the LLVM compiler framework to cause functions to be randomly permuted and a random amount of padding to be inserted between functions.

2.2 Attacks on Diversified Code

Several attacks have also been proposed to thwart code diversification. Entropy exhausting attacks [34] have been shown to defeat ASLR when applications are restarted after crashing and the amount of entropy in the system is small, as an attack can brute-force the memory space. The BROP attack [6] demonstrates that information gained from detecting if a process crashes or not can be used to determine the locations of *pop* instructions and the location of the `send` function. BROP allows an attacker to directly read a closed source binary, as a copy of the in-memory binary is sent to the attacker. Such attacks can typically only be conducted on systems where repeated crashing goes undetected, the application is restarted after crashing, and memory is not re-randomized after restarting. We present and demonstrate several attacks that do not require crashing the application, thus can bypass such mitigations.

Most often used in practice today are memory disclosure vulnerabilities [38], which allow an attacker to read unintended parts of the code or pointers on the stack or heap. Such vulnerabilities allow attackers to read memory locations dynamically at runtime, thus learn exactly how code has been randomized. However, to effectively use a memory disclosure vulnerability to conduct an ex-

ploit, an attacker needs two vulnerabilities, one to read memory and another to overwrite memory.

Snow *et al.* [36] show that a single memory disclosure vulnerability can be abused repeatedly to make fine-grained ASLR ineffective. They designed and built a JIT compiler where a single payload can be run within a victim's web browser scripting environment to JIT compile a code reuse exploit. We note that while their JIT compiler framework currently abuses a memory disclosure vulnerability repeatedly, it could also be modified to use the information leak attacks that we present in this paper, removing the need for a second vulnerability.

Hund *et al.* [22] demonstrate that a local attacker can defeat kernel space ASLR by leveraging cache and TLB-based side channels. Their work focuses on leaking *where* kernel code is located in memory through measuring the execution time of memory accesses without crashing the kernel. Our work differs as we demonstrate remote side channel attacks that are not cache-based. Furthermore, we demonstrate how both *where* code is located and *what* code has changed can be leaked through actively influencing the program's execution. Techniques such as cache interface randomization [44] can be deployed to mitigate cache-based side channels.

Another address leakage attack without a memory disclosure vulnerability is proposed by Blazakis [7] which uses garbage collection to leak addresses. In this attack, a number of objects are created on the heap, some address guesses are placed on the stack, then all references to the objects are removed. If the objects are deleted after garbage collection the guesses were wrong and the procedure continues from the beginning, otherwise one of the guesses is correct. This attack only applies to garbage collected environments.

2.3 Side Channel Attacks on Cryptography Implementations

We review several kinds of cryptographic side channels that are applicable to code diversification, with an emphasis on the ones that can be conducted by remote attackers.

Timing. Cryptography implementations are often optimized to forgo certain operations when unnecessary, to reduce performance costs. These optimizations are usually dependent on the secret key, thus different execution paths will be taken for different keys. As different keys will cause different execution times, an attacker can time execution and use that information to infer the key. Many works have shown that real implementations are susceptible to such attacks and that they are not trivial to eliminate [9, 10, 12, 27]. Similarly, we will demonstrate how an attacker can use the time of execution as a way to infer how code has been diversified.

Fault Analysis. Cryptographic systems have been analyzed under the premise that an attacker that has physical access can control the environment of the device. Specifically, by controlling aspects such as temperature and power supply voltage an attacker can induce a fault in the device [5, 14, 20, 43]. Typically, these attacks can force bits to flip during the cryptographic computation process. The output can then be analyzed, and from this information the attacker can deduce the secret key. In this work, we take advantage of the memory corruption bug to similarly force memory to change. However, the major difference being that the attacker no longer needs to have physical access to cause memory to change.

Caches. Cache hits and misses leak information about the data and code recently accessed by a process. Since cache hits can shorten execution time respectively, an attacker can measure execution time and infer something about the memory being accessed. These capabilities have been shown to allow secret information such as cryptographic keys to be discerned by a local attacker [2, 8, 41, 46]. As mentioned earlier, cache attacks have been shown to allow a local attacker to defeat kernel space ASLR [22].

Physical Access. Numerous side channel attacks cannot be performed remotely. This is due to the fact that they require measurements that can only be done if the attacker has physical access to the machine. These include power [24, 28, 32], electromagnetic (EM) emanation [30], and acoustical [17, 42] analyses, where the attacker needs to measure the power consumed, the EM field produced, or the sound produced by the device, respectively. In all cases, the instructions executed often have a distinct power, EM, or sound measurement. An attacker conducting cryptanalysis then needs to infer what secret key caused that sequence of instructions to execute. However, to discern how code was diversified simply learning what instructions were executed could be sufficient.

3. SIDE CHANNEL ATTACKS ON DIVERSIFIED CODE

Code reuse attacks leverage that there is a known mapping between memory locations and gadgets. Code diversification techniques break this assumption by rearranging the mapping. Specifically, these techniques can change *where* in memory the code resides (e.g., ASLR [40], function reordering [18]) and they can change *what* machine-level instruction a memory address contains (e.g, NOP insertion [23], instruction substitution [31]). To rebuild the mapping, the attacker then must either choose a memory address and determine what gadget resides there, or choose a gadget and determine where in memory it resides. Different side channel attacks can accomplish these two tasks.

To conduct a side channel attack, an attacker needs to be able to receive feedback from a victim. This requires that the remote attacker be able to interact with the program through a networked or scripting environment. In a networked environment, a remote attacker will interact directly with the program and can receive some information through the network. In a scripting environment, a remote attacker can send a script which conducts side channel attacks and possibly JIT compile an exploit [36] after discerning enough information. These are similar requirements to previous entropy exhausting and memory disclosure attacks.

The type of side channel attack that can be conducted will largely depend on the vulnerability itself and the gadgets that are available to an attacker. Some vulnerabilities will be better at leaking information than others. This will often depend on what the high level semantics of the code are, whether the vulnerability allows modifying variables, data pointers or code pointers, and how an attacker can repurpose the code to leak information about the code. Furthermore, if an attacker has access to particular gadgets, even a small number of them, he can use them to build an information leak attack. While the example code snippets we present later are in a normal control flow paradigm, they could also be created using return-oriented programming (ROP) or other code reuse paradigms [11].

Utilizing a memory corruption vulnerability brings with it the possibility of causing crashes. An attacker may or may not be able to tolerate crashes. Some attacks can typically only be conducted on code that tolerates crashes by restarting [34]. Other code, such as the kernel, cannot tolerate crashes at all [22]. Different memory corruption attacks have varying amounts of risk in causing crashes due to invalid memory accesses. Simply modifying variables has very little risk, to cause a crash the overwritten value would need to be used as an index to a pointer and then cause the pointer to point to an invalid page of memory. Similarly, if an attacker does not know where valid memory pages are, then overwriting a data pointer can cause crashes. However, if valid memory pages are predictable, then overwriting a data pointer has no risk in causing

crashes. Modifying code pointers has the highest risk of causing crashes, since the pointer needs to not only point to valid pages, but it also needs to point to a valid instruction sequence and those instructions need to use the stack in a safe manner. Side channel attacks that use these different vulnerabilities also carry with them the same levels of risk.

In the rest of this section, we discuss fault analysis and timing side channel attacks. For both types of side channel attacks, we present multiple classes of attacks, that take advantage of memory corruption bugs in different ways. For each attack, we give a high level overview and a simple example of some vulnerable code that could be exploited to conduct the attack. We then describe what kind of information can be learned from the described attack and the limitations of the attack.

3.1 Fault Analysis Attacks

To specifically conduct a fault analysis attack, the attacker needs to be able to send some payload, receive the result of the execution, and then interpret the result to learn what was executed.

Overwrite Data. An attacker can overwrite some data that is used as an index to a data pointer, causing the pointer to go out-of-bounds. This can cause some computation to be done on a location of his choosing, rather than the intended data. If the data pointer is then pointing to the stack and incorporates a return address, the result of the computation could reveal *where* the code is located. If the pointer points to the code, and that is computed on, the result could reveal *what* changes have been made to the code.

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

The exact computation that is being done on the data and what can be learned from it will depend on the vulnerable piece of code. The above example represents a simple piece of code that can be abused to learn low level details of how code has been diversified. In the example, the attacker can use a buffer overflow to overwrite an index to a pointer, the dereferenced pointer's value is checked and if it is zero an error is sent, otherwise a value indicating success is sent. The attacker can cause the data pointer to point back to somewhere in the code segment, and then learn where the byte `0x00` is located. If the pattern of `0x00` located in the code is distinct, the attacker can learn how code has been diversified.

Limitations to this attack are largely dependent on how reversible the computation being done is and thus how much does the output reveal about the input. This attack is most effective when the computation being done has a one-to-one mapping between input and output. However, if the computation is reversible, then both where code is located and what code is there can potentially be leaked.

Overwrite Data Pointer. An attacker could overwrite a data pointer directly, which is similar to the previous attack, and can cause some computation to be done on a location of his choosing, rather than the intended data. However, unlike the previous attack, an absolute location in memory is chosen, this will directly reveal *where* code is and *what* changes have been made to the code.

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

In the above example, the attacker can overwrite a data pointer that is used in determining how many values in an array are needed to sum up to 100. This vulnerability could allow an attacker to overwrite the pointer to the code segment and learn information about it. Thus if the result of this computation on some byte sequence or pattern of byte sequences is distinct, then an attacker can determine how code has been diversified.

Limitations are similar to *Overwrite Data*, concerning the reversibility of computation.

Overwrite Code Pointer. An attacker can overwrite a code pointer to cause a computation of his choice to be done. If the result of that computation is distinct to some piece of code and is sent back, the attacker will learn *where* that code is located.

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

In the above example, the attacker can use a buffer overflow to overwrite a function pointer. The attacker can then cause some arbitrary piece of code to be executed. In this case, the attacker can learn information in two ways. First, if the attacker gets no result, this will most likely be due to the code crashing before it was able to send the result. The attacker potentially learns then that the address chosen was an invalid instruction or that the correct return address was prematurely popped off the stack. Second, if the attacker receives some value, then he can analyze the undiversified code to learn what parts of it were capable of giving that value.

Limitations to this attack are dependent on how distinct is the computation being done and if multiple pieces of code could compute the same result. We will explore this in § 4.

3.2 Timing Attacks

To conduct a timing attack, an attacker needs to be able to start a timer, send some payload to the victim to initiate and possibly manipulate execution, and then finally receive some signal when execution has finished, indicating that the timer can be stopped. Timing can then reveal information about the code itself. We note that an attacker could possibly conduct both a fault analysis and timing attack with the same payload.

Crafted Input. This attack is the most similar to cryptographic timing side channels, as the attacker does not exploit a memory corruption vulnerability, but instead chooses well-crafted input to exercise different paths in the diversified code.

If the instructions in the diversified code have then been modified, for example if NOPs have been inserted [16, 23], this can change the timing from the original code and can be measured. Thus, this attack can help determine *what* code has been inserted into the diversified code.

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

In the above example, the attacker wants to learn if NOPs have been inserted into block 1 or 2, causing statement 4 to have moved. The attacker can learn this by crafting his input so that both paths are eventually exercised. As adding NOPs will increase the time taken to execute, he can time the results, compare the timings to what the original code produced, and if it is longer, can calculate how many NOPs have been inserted.

Limitations to this attack include not being able to detect if the offset of the entire code segment has been shifted in memory (e.g., ASLR), as it can only detect scenarios where instructions with different timings have been inserted. Furthermore, as x86 and other CISC architectures have variable length instructions that can potentially take variable cycles to execute, multiple instructions of differing length may take the same time to execute. This can make

it more difficult for the attacker to discern exactly where gadgets have moved in memory. However, this generally will not be a problem for RISC architectures, which have the same length and timing characteristics for each instruction.

Timing attacks that are conducted over the network and not in a scripting environment will also have to deal with network jitter. Jitter becomes a problem when it represents a significant proportion of what is being measured. However, attackers can leverage previous work which has shown this can largely be overcome by taking multiple measurements and filtering them intelligently [12]. Furthermore, an attacker can also reduce the significance of jitter by controlling how many times a piece of code is executed during a single measurement, perhaps through influencing a loop counter variable, thus reducing the number of total measurements needed.

Overwrite Data. The previous attack described scenarios where the attacker only modified state that was intended to be modified by him. However, parts of the code may only be executed when certain variables are set in a particular way, and the attacker may not normally have control over them. The *Overwrite Data* attack can be used to to allow an attacker to modify certain variables so as to execute different pieces of code. Similar to the previous attack, this will allow one to learn *what* code has been inserted into the diversified code. This attack can also be used to overwrite an index to a pointer, causing data on the stack or code segment to be computed on, and allowing an attacker to learn both *what* code has been inserted and also *where* code is located.

```
1 if (config_variable == 0)
2   i = i * 2; // block 1
3 else
4   i = i + 2; // block 2
```

The above example is similar to the previous example, except there is a memory corruption vulnerability and the variable used in the conditional is intended to only be set by a local configuration file. If this variable has been set to 0, then the attacker will not be able to exercise the code in block 2. However, the attacker can modify the variable through the memory corruption vulnerability, and then make block 2 be executed and time the results.

```
1 if (ptr[index] % input == 0)
2   i = i * 2; // block 1
3 else
4   i = i + 2; // block 2
```

In the above example, an attacker can overwrite an index to a pointer, which the dereferenced value is then divided by some user input, and the remainder is compared to 0. If the value is 0, the multiplication in block 1 is executed, which takes longer to execute than the addition in block 2. An attacker can use this vulnerability to discover where code is. Since the index variable can be modified, the pointer can point to a value on the stack, such as a return address. Similar to previous cryptographic timing side channels [10], the attacker can guess values intelligently and determine if the value is a divisor of the return address or not. Eventually the attacker can determine what is the exact value of the return address.

With respect to network jitter, this attack has similar limitations to the previous *Crafted Input* attack.

Overwrite Data Pointer. An attacker can also overwrite a data pointer to reveal the contents of memory through timing execution. Specifically, an overwritten pointer that is later used in control flow decisions can reveal the bytes to which the pointer actually points. The data pointer can be overwritten to point to a return address on the stack, revealing *where* code is located, or a location in the code segment, revealing *what* code is located there.

```
1 i = 0;
2 while (i < ptr->value)
3   i++;
```

In the above example there is a vulnerability that allows an attacker to overwrite the data pointer. If the attacker overwrites it with an address in the code segment, the loop will execute that many number of times. For example, if the attacker is looking for a return instruction in x86 code, the loop will execute 0xc3 times when found. If the attacker knows how long it takes to execute the loop once, he can compare that to when it is executed multiple times and figure how how many iterations actually took place. This attack essentially allows one to find the value of arbitrary memory locations through timing.

The limitation of this attack is that it requires code that makes control flow decisions using an overwritten pointer. However, as we will show later, an attacker can use other attacks to bootstrap and find enough gadgets to create a control flow mechanism.

Overwrite Code Pointer. In this attack, control flow is directly hijacked by overwriting code pointers, such as return addresses and function pointers. While the attacker may not know what code he is actually executing, the attacker can time how long it takes to execute and discern some information about *where* code is located.

In the example below, there are three functions whose locations have been randomized, where $func1$ has a memory corruption vulnerability allowing a function pointer to be overwritten. Each function $func1$, $func2$, $func3$, takes a different, distinct time to execute, t_1, t_2, t_3, respectively. The attacker can hijack control flow to location 1, and when he finds that it takes t_2 time to execute, he knows that that location is the beginning of $func2$. Similarly, if he executes at location 4, he will find it takes t_3 time to execute and will know he is executing $func3$. Note that this attack works even if fine-grained ASLR is deployed on the system.

```
- func2 (int z)
1  x = 3;
2  z = x + z;
3  return z;
- func3 (int x, int z)
4  z = z * x;
5  return z;
```

```
- func1 ()
6  recv(sock, buf, input);
7  (*funcptr)();
8  send(sock, data, len);
9  return;
```

Similarly, the attacker can also try to find return instructions directly by executing at different locations, thus also determining the size of pieces of code. For example, if the attacker hijacks control flow four different times and points execution to locations 1, 2, 3, and 4, he will find it takes the smallest amount of time to execute location 3, where the return statement is. As the return instruction is the fastest piece of code that can possibly be executed, he will know he has found the return instruction by the timing being the shortest. The attacker then knows that location 1 is $func2$, as that is the only function with size 3.

Limitations to this attack are similar to *Crafted Input*, but at a coarser level. If functions, or basic blocks of code, have distinct timing or size characteristics, then functions can be discerned with this attack. If many pieces of code, on the other hand, have the same size and timing, then it can be difficult to differentiate what is being executed. In RISC architectures size and timing characteristics are correlated, thus knowing one reveals information about the other.

4. SIDE CHANNEL EFFECTIVENESS

We demonstrated in the previous section that, depending on the vulnerability, different information can be leaked about the code. In this section, we ask the question: *If we can only leak certain*

information about the code, how useful is that information? Effective side channels require that the phenomena being measured (e.g., power, acoustic signals, time, output) be distinct, thus revealing information about what is being executed. For example, some of the attacks previously described require that the size of functions, particular sequence of bytes in code, or execution time to be revealing of what code is actually being executed. Thus, how distinct these measurements are determines how precisely an attacker can identify what piece of code he is actually measuring. As `libc` is almost always linked as a library in real code, we analyze how distinct certain aspects of it are.

4.1 Metrics

To measure how distinct a function is, we define the *uncertainty set size* (USS) metric. The *USS* of a function is the cardinality of the set of other functions that have the same measurement associated with it. Thus a totally distinct function's USS would be zero. If two functions had the same measurement, each function's USS would be one, indicating that there is one other function that could be confused with it. In other words, the USS metric indicates how many functions have the same side-channel characteristic. If the set is small enough, then an attacker can do brute-forcing or use the intersection of USS for different attacks to reduce the overall uncertainty. We also measure how much information needs to be leaked by the attacker to learn the actual USS.

Note that in this section, all of our evaluations assume a function-level randomization, known also as "medium-grained" ASLR, where the location of functions are randomized, but the internals of the functions remain the same. This is done to evaluate how useful different types of side channels are in general. In Sec 6, we evaluate actual attacks against different code diversification techniques (i.e., NOP insertion and coarse-,medium-, and fine-grained ASLR).

Identifying how distinct a piece of code is does not necessarily answer what an attacker can accomplish once information is leaked. We also evaluate how many total gadgets an attacker can discover once he leaks information about the diversified code. However, if the same gadget is located in many pieces of code, an attacker only needs to know one of those locations to use that gadget. Therefore, we also evaluate how many *distinct* gadgets are available.

As discussed by Skowyra *et al.* [35], when building a practical code reuse payload, the most important property a gadget set must have is access to system calls, not Turing completeness. Malicious payloads such as uploaders, downloaders, backdoors, and root inserters all require access to system calls. Although system calls are widely available in `libc`, they are much less common in executables or other libraries. In fact, we analyzed all libraries linked to the Ubuntu Top 500 packages. This amounted to 3989 unique libraries. Of those, only three libraries outside of `libc` have access to system call gadgets: `libgomp`, `libxul`, and `libjvm`. As a result, even though the executable may contain many gadgets, an attacker still needs to find gadgets with access to system calls in linked libraries, primarily `libc`. Thus, we also measure how many system call gadgets can be found, to determine their prevalence in the code, but we note that an attacker only needs one system call gadget to write an exploit.

4.2 Byte Sequences

The *Overwrite Data* and *Overwrite Data Pointer* attacks can cause information to be leaked by overwriting an index to a data pointer, or a data pointer itself, to point to the code segment. In the previous section, we gave an example of a fault analysis attack where some calculation was done on the code, and also an example of a timing attack where a loop was executed a number of times based on the value of the code. However, an attacker might have some other way to leak information. Since different vulnerabilities will leak information in different ways, we explore how distinct code is and how a side channel attack could take advantage of this.

We begin by evaluating the probability distribution function of bytes in the code. Considering code is highly compressible, we expect that bytes in code do not follow a uniform distribution. Figure 1 confirms this, which shows a large skew towards certain bytes, while many bytes are used very infrequently. We find that the most common values are `0x00` and `0xff`, as these values are often used in constants. This suggests that since code is distinct, computing something on the code itself can reveal what the code is.

If an attacker can find the locations of specific bytes, we evaluate if the pattern of such locations are distinct to functions. For example, perhaps through the use of a vulnerability similar to those presented earlier, an attacker can learn where `0x00` bytes are located. We also evaluate how many locations need to be leaked to uniquely identify functions. Figure 2 shows a CDF of the fraction of functions that have a particular USS or less. We find that when only one location is leaked, very few functions have an USS of zero. However, as the number of locations leaked increases, the USS decreases. For example, with four locations leaked, 38% of functions are uniquely identifiable. Finally, if all the locations are leaked, we find that 62% of functions have distinct patterns, while 71% of functions have an USS of one or less.

Table 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%)

Table 1 describes how many gadgets are available when only certain types of information are leaked (the Galileo algorithm as described by Shacham [33] was used for these statistics). When all function locations are known, the attacker has access to over 24,000 total gadgets which are comprised of 2059 distinct gadgets. When 0x00 byte locations are leaked, an attacker can find the locations of only 56.8% of all gadgets. However, this still includes 94.6% of distinct gadgets. Syscall gadgets are also available to the attacker, while four in total are available, an attacker will only need one to write an exploit. Thus, malicious payloads can still be constructed with only this amount of information leaked.

Functions that have an USS greater than zero can still be useful, as these functions can be very similar internally. For example, upon examining the group of functions that have the largest USS of 108, we find that they all are wrappers for system calls. These include functions such as *bind*, *listen*, and *mprotect*, which are often important for malicious payloads. However, these functions only differ in their first instruction, which moves a constant signifying what system call is intended into a register. The rest of the function is identical across all 108, where the *syscall* instruction is executed and then return values are examined. A payload could be crafted where some other gadget is used to load the register, perhaps using a *pop* instruction, and then jumps directly to the *syscall* instruction.

4.3 Return Instructions

Several of the attacks described can help determine where both intentional and unintentional return instructions are in the code. By using the *Overwriting Code Pointer* attack and taking advantage of either fault analysis or timing side channels, we can reveal such information. If an attacker can discern this information but nothing else, we evaluate if he can determine what function he is actually

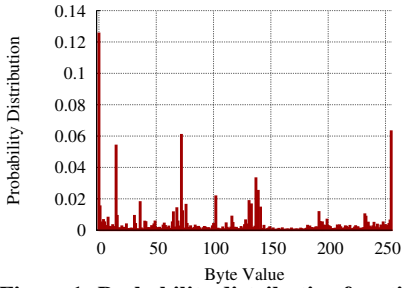

Figure 1: Probability distribution function of number of bytes found in code

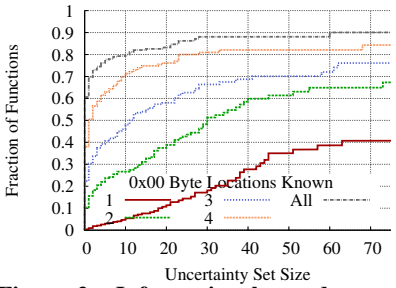

Figure 2: Information learned as more 0x00 byte locations are known

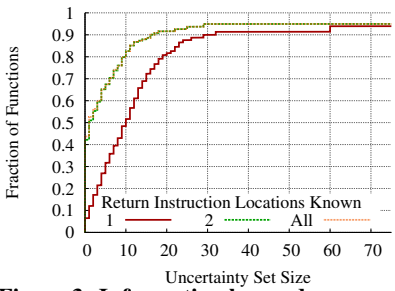

Figure 3: Information learned as more return instructions are known

measuring. Specifically, we evaluate if functions have distinct patterns of where return instructions are internally located.

Figure 3 shows a CDF of the fraction of functions that have a particular USS or less. We find that if the location of one return instruction is leaked, then 6.4% of functions are uniquely identifiable. However, if two locations are leaked, then 42.2% of functions are identifiable. Knowing two locations leaks almost as much information as knowing all locations, as we find if all locations are known then 42.6% of functions are uniquely identifiable. Furthermore, 90% of functions have an USS of 18 or less, meaning that this measurement greatly reduces the USS for most functions by a large amount. Similar to before, we find that a large number of system call wrappers are grouped together, thus have an USS greater than zero, but can be used as previously mentioned.

Leaking return instruction locations yields fewer gadgets than other information leaks (Table 1). However, 41.9% of all gadgets, 84.0% of distinct gadgets, and 1 syscall gadget are still available.

4.4 Output

If the attacker is able to conduct an *Overwriting Code Pointer* attack and can retrieve some information about the result of the execution, he can conduct some fault analysis and discern information about what was executed. In the best case, if the attacker could learn all the inputs used for a function and all the resulting outputs, he could potentially learn much about what was executed. However, we evaluate what an attacker can do with a much more limited amount of information, namely crashes and the return values of functions.

We evaluate first the USS of functions if the attacker is able to discern if the program crashes or not. To evaluate this, we overwrite a function pointer and test to see if the program crashes when the function pointer is called. We demonstrate our results in Figure 4. Many crash locations are needed before a significant number of functions are uniquely identifiable, over 60 locations are needed to know just 23% of functions. If all crash locations are known, over 56% of functions are distinct in their crash patterns and 78% of functions have an USS of 10 or less.

Learning crash locations allows more distinct gadgets (Table 1) to be leaked than the other types of information leaks that we investigate. Specifically, 97.1% of distinct gadgets are leaked, while 58.0% of all gadgets and 3 syscall gadgets are leaked.

While an attacker might be able to retrieve different memory values or registers after something is executed, a likely output is simply the return value of a function. In x86, this is the value found in the `%eax` register. Thus, we evaluate if the pattern of return values, when jumping into and executing different parts of a function, is distinct for a function. Figure 5 shows that when one return value is known, 12% of functions are distinct, and when two values are known, 38% of functions are distinct. When the attacker learns all values, 57% of functions are distinct, while 75% of functions have an USS of six or less.

The group of functions that have the largest USS of over 600 mostly consist of internal `libc` functions that are used by other functions. Thus, these functions are likely to not be directly needed for any common malicious payloads. We again find that wrappers to system calls are largely grouped together due to their internal similarity. The return value of executions also gives the ability to learn the most distinct gadgets (Table 1). 96.9% of distinct gadgets, 50.8% of all gadgets and 14 syscall gadgets are leaked.

4.5 Timing

An attacker will most likely be able to, at the very least, time execution. Thus, if he can use an *Overwrite Code Pointer* attack, and then time the resulting execution, he could possibly still gain a great deal of information about what is being executed. Similar to the previous experiment, we overwrite a code pointer to jump into and execute all locations within a function. We then evaluate how distinctive timing is to different functions.

Figure 6 shows that when only one timing value is known, only 10% of functions are uniquely identifiable. However, if two locations are known, the number jumps to 38%. If an attacker knows all the timing values, he can learn up to 60% of functions with an USS of zero and 76% of functions with USS of five or less. We find that timing is very effective in leaking gadgets. 58.8% of all gadgets, 95.8% of distinct gadgets, and 16 syscall gadgets are leaked.

5. PRACTICAL SIDE CHANNELS

In this section we describe our framework that can be used to practically leak information about the code incrementally. We focus on *Overwrite Data Pointer* timing side channel attacks, because as previous sections have demonstrated, they have the most potential for practical use. This is due to them having few limitations, they do not require the application to be able to crash and restart, they are flexible in the types of information they can leak, and they are effective in the amount of information they can leak.

Overview. Information leaks are most likely a stepping stone to other attacks. Thus once an attacker learns enough information about the location of certain gadgets, he will conduct the intended exploit (e.g., starting a shell). Furthermore, if an attacker learns certain other helpful gadgets, he can incorporate them into the attack as well. For example, if the attacker has found enough gadgets to create a loop, allowing him to reduce the needed timing precision, he would use them. Gadgets found in the executables are helpful in facilitating the information leakage, but they are often not enough for practical payloads as discussed earlier. An attacker can use these gadgets to find other system call gadgets in `libc`.

Thus, at a high level, our information leak framework iterates over three different steps. First, it evaluates a pool of currently known gadgets, and discerns if these gadgets can be turned into a new or more efficient information leaking capability. Second, it

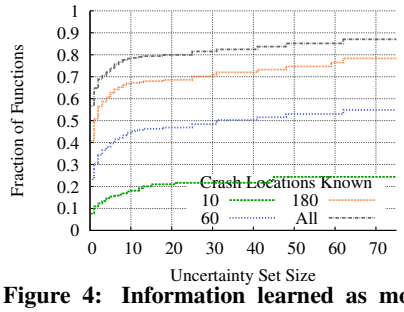

Figure 4: Information learned as more crash locations are known

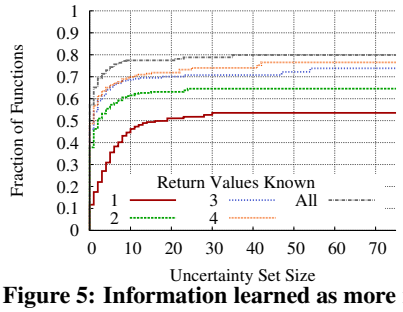

Figure 5: Information learned as more return values are known

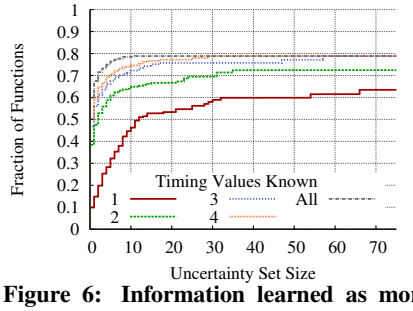

Figure 6: Information learned as more timing values are known

uses the most efficient capability to leak more information about the unknown parts of the code. Third, it updates the pool with any newly discovered gadgets. We stop when we have either leaked enough information to learn the exploit gadgets or have leaked the entire code.

Slow Side Channel Attack. If certain gadgets are available to use, then those are put into the pool of gadgets. Otherwise, we assume that initially the attacker can only conduct a slow timing attack, where a piece of code can only be executed once.

This attack requires that a piece of code is found where pointers can be manipulated such that the code performs some computation on itself and that the execution time is dependent on the input. Thus, if code such as a spinlock is found, we can use it to discern what code is at some memory address.

To know how long the piece of code should take to execute, we can use a database of timing values, similar to what was developed in §4. To get a proper timing baseline for the vulnerable remote host, we first send many requests without exploiting it. We time how long the baseline execution takes, which will be subtracted from any future timing measurements. We then start probing memory locations by overwriting the data pointer, and measuring the execution time. If a certain amount of timing precision is needed, the experiment is repeated.

Fast Side Channel Attack. If enough gadgets are found to create a ROP conditional jump, we use it to increase efficiency. The conditional jump allows a payload to be constructed where a piece of code executes x times. We can then reduce the number of measurements needed. For example, if enough gadgets are available we can implement a spinlock itself.

The timing payload will then contain either the memory address of a code pointer (e.g., a return address on the stack) or the code itself. A gadget then loads the memory address into a register and other gadgets somehow use that register as an iterator in the conditional jump. The timing result will then indicate how many times that piece of code was executed. This requires knowing how much time executing the piece of code once takes, which will be known through the database previously constructed. Eventually, as more memory addresses are measured, enough information will be leaked so as to learn a system call gadget. Once accomplished, we are able to compile the exploit based on the exact code in memory.

6. MEASUREMENTS AND RESULTS

We develop a tool to conduct a timing attack against Apache 2.4.7 and evaluate the accuracy of timing information for two types of networks: a 802.11g wireless network and a wired LAN with two routers. We evaluate our attack against four types of code diversification defenses: coarse-grained ASLR [40], function permutation (medium-grained ASLR) [26], basic block randomization (fine-grained ASLR) [45], and NOP insertion [16].

In our attack we use Apache HTTP Server 2.4.7 (the most recent version at the time of writing this paper) and `glibc` 2.16. A stack overflow vulnerability (CVE-2004-0488) from an earlier version of Apache was reapplied to create the initial vulnerability. The vulnerability allows an attacker to place arbitrary values on the stack.

Below is one example of a side channel vulnerable code from Apache, which is in Apache's /server/log.c file and is responsible for formatting error logs. We redirect the `fmt` pointer using the overflow vulnerability to a chosen byte in memory. The additional delay in processing the request is proportional to the byte value.

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

6.1 Slow Timing Attack

During the slow side channel attack, we redirect the loop iterator (`fmt->nelts`) using the Apache vulnerability to measure a chosen byte. This adds a small amount of delay into the query processing part of Apache. The success of our attack depends on the ability to observe the small delay caused by the loop over the network. Although this delay is small, by sending many queries to the server, the extra delay adds up over the baseline delay.

We measure the cumulative delay in two different setups: one in which the attacker is on the same 802.11g wireless access point as the victim and another one in which the attacker is three hops away from the victim on a wired LAN with two routers in between.

In each case, we start by measuring the time between an HTTP request and response 10,000 times to collect the timing samples (we will later reduce the sample size to its minimum). Since Crosby *et al.* [12] show that the first percentile yields the most precise timing measurement, we only keep the fastest 1% samples and discard the rest to account for abnormally large delays caused by various network conditions. Figures 7 and 8 illustrate the results. For the sake of space and readability, we just show the delay for eleven different byte values pointed to by `fmt`. The maximum standard deviation among all the LAN measurements for different byte values is 0.557 *ms* and for the wireless measurements it is 0.715 *ms*.

We use these measurements to estimate the bytes on the victim Apache machine. For the actual attack, we point the `fmt` to a chosen location. Given the slope of the cumulative delay, we estimate the byte stored at that location. We repeat the measurements for the subsequent bytes at that location.

To determine where that location is, while taking jitter into account, we develop and implement a fuzzy n-gram matching algorithm. This algorithm takes as input n measured bytes, where some measured bytes may be inaccurate due to noise, and will determine the most likely location in the `libc` library that is being measured. To accomplish this in an efficient manner, we build a trie, where for every offset in `libc` the next n bytes are put in the trie. We can then do a scoped depth first search on the trie, finding the strings that most closely match the measured bytes. We analyze `libc` to determine what is the expected number of measured bytes needed

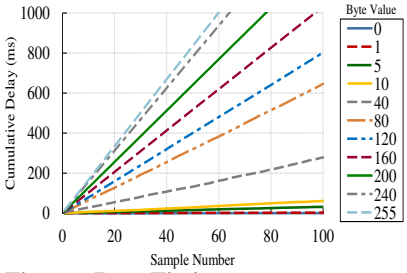

Figure 7: Timing measurement for Apache 2.4.7 over wired LAN

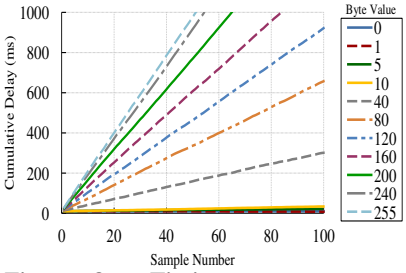

Figure 8: Timing measurement for Apache 2.4.7 over WiFi 802.11g

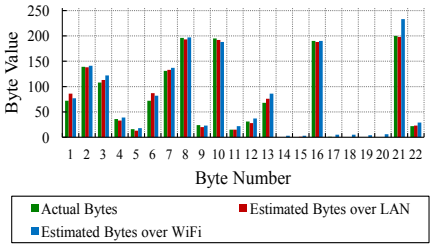

Figure 9: Estimated bytes using timing attack against Apache

to uniquely determine the offset, given a certain timing precision. We find that even if our timing precision is poor, the expected number of total measured bytes needed is low. For example, when any measured byte could be off by ± 15, tolerating a 6% margin of error, at most only 13 measured bytes are needed for the majority (54%) of offsets, and at most 40 measured bytes are needed for 85% of offsets. 6% is chosen based on the fact that for the code bases we have analyzed (`libc` specifically), with error rates higher than 6%, the fuzzy matching starts to return erroneous results.

To establish the minimum number of samples necessary for the slow timing attack, we repeat it with increasing sample sizes and calculate the average byte error. We start with a small sample size of 1000 and increase it until the error is less than the targeted 6%. We only keep the first percentile. Table 2 shows the errors for different sample sizes. For the slow timing attack, the minimum sample size acceptable is 5000 samples.

The amount of time it takes to collect one sample for the slow timing is approximately 8.64 *ms* over the wired network and 41.94 *ms* over WiFi. Since 5000 samples must be collected, the amount of time it takes to estimate one byte using the slow timing attack is 43.2 *sec* and 3.49 *min* for LAN and WiFi respectively.

6.2 Fast Timing Attack

After enough gadgets are found to build a simple ROP loop, the vulnerable code can be repeated many times (500,000 time in our case) locally on the victim's machine to reduce the network noise and speed up the attack. The ROP loop in pseudo-code is shown below. In the next section, we will describe how this ROP loop is constructed for Apache.

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

Since the error is small due to the ROP loop, our samples sizes can be very small (see Table 2). As such, we only keep the fastest sample. As can be observed from the table, the minimum number of samples necessary for the fast timing attack is 4 samples.

Table 2: The amount of error for different sample sizes

Sample Size	1,000	2,500	5,000
Slow Timing Error	22.79%	16.96%	5.89%
Sample Size	2	3	4
Fast Timing Error	20.56%	9.48%	5.7%

The time to collect one sample is mainly dominated by the ROP loop and is about 348 *ms* for both WiFi and LAN. Since 4 samples must be collected, the time to estimate one byte using the fast timing attack is 1.39 *sec* for both WiFi and LAN.

6.3 Coarse-Grained ASLR

In the coarse-grained implementation of ASLR, the location of the executable is not randomized. As a result, it is easy to construct the ROP loop using the gadgets in the Apache executable only (no linked libraries used). We know the locations of these gadgets from our local Apache copy. Thus, for the coarse-grained ASLR, only the fast timing attack is necessary.

We found 4175 gadgets in the Apache executable, although none of them has access to a system call or a sensitive function. We used 20 of these gadgets to implement the ROP loop. The actual ROP loop is illustrated in Figure 10 in the appendix. It implements a loop with the number of iterations equal to 500,000 times the pointed value. By redirecting the loop iterator into `libc` and estimating the bytes, we determine how it is shifted.

The ROP gadgets in the Apache executable, however, are not enough to build a practical payload (e.g. an uploader) because they do not have access to a system call. Since the location of `libc` linked to Apache is unknown, the attacker does not have access to system calls within `libc` either. Normally, the attacker needs another vulnerability for memory disclosure to locate `libc` or system calls within `libc` to build the complete attack. We demonstrate how just this one vulnerability can be used for both purposes, i.e., leaking instructions within `libc` and hijacking control using the actual payload. This is in spite of the fact that the vulnerability is a buffer-over-flow vulnerability, not a buffer-over-read one. Once we find the offset of `libc`, we will then use the same vulnerability to inject a ROP payload that accesses the system calls in `libc` to achieve the malicious behavior.

In our measurement, we first measured 20 bytes at the chosen location. In fact, the bytes were not enough to uniquely identify the location in `libc` as two locations matched those 20 bytes. By inspecting the 21st and 22nd bytes at those two locations in our local copy of `libc`, we realized that since the 22nd byte has very different values at those two locations, measuring two more bytes would uniquely identify the location. By measuring the two subsequent bytes, we determined that the location is indeed the 49th byte in the `__argp_fmtstream_putc` function. Figures 9 shows the actual bytes in that location and the estimated bytes over wireless and wired networks. Since we know from our local copy of `libc` that the 24921st byte after that is a system call (located in the `__lll_lock_wait_private` function), we used it to create the uploader which was the original attack goal.

Our measurement illustrates the strength of timing attacks as only 22 noisy bytes were enough to uniquely find the location in `libc`, a library of more than 1.3 MB. Note that in fact we were a little unlucky in our attack since for the majority of the cases only 13 bytes are enough. Since the gadgets in the Apache executable are already known, this attack takes about 30.58 *sec* (22×1.39 *sec*).

6.4 Medium-Grained ASLR

Function permutation such as the one implemented by ASLP [26] is a medium-grained form of ASLR in which function locations are also randomized within a library.

In this case, we still need to find a system call, but since the Apache executable itself is also randomized, we do not have access to the ROP loop gadgets. Our technique for attacking function per-

mutation is as follows. We redirect the vulnerable loop iterator to a chosen location to measure a number of bytes using the slow timing attack. We measure enough bytes to fingerprint the function. We know from our undiversified copy which functions contain a system call and which ones do not. If the function contains the system call, we are done. If it does not, we skip enough bytes to pass that function and start to measure the next function. We repeat this process until we find a system call.

The amount of measurement necessary for attacking medium-grained ASLR is the expected number of measured bytes needed to determine what function some piece of code is (13 bytes), multiplied by the expected number of functions we need to discover before finding one that has a system call gadget. We determine this latter value to be the number of functions in `libc` (3309), divided by the number of system call gadgets (60). Therefore the expected number of measured bytes needed is 717.

We performed this attack with different function permutations four times. The number of bytes measured before we found the system call for the four experiments were 303, 2661, 806, and 441 which is consistent with our expected number of measures. Since this attack on average requires 717 bytes with slow timing attack, it takes about 8.6 hours to complete on the LAN ($717 \times 43.2\ sec$).

6.5 Fine-Grained ASLR

Fine-grained form of ASLR can randomize code at the granularity of basic blocks. One such implementation is Binary Stirring [45]. Since basic blocks can be very small in nature, we conservatively estimate that every instruction in the basic block would need to be measured. As there are 218930 instructions in `libc`, we estimate that 3649 libc instructions would need to be measured before finding a system call. Furthermore, since the expected size of an instruction is 3.8 bytes, we expect 13866 measured bytes are needed to find a system call.

An alternative approach would be to try to find enough gadgets in Apache to build a ROP loop again. Note that since fine-grained ASLR also randomizes the basic blocks in the executable, the ROP loop we constructed for coarse-grained ASLR is not available to us because we no longer know the location of its gadgets. The most difficult gadget to find is the conditional jump gadget necessary for building the loop. A convenient way of building the conditional jump is by using instructions that use the carry flag as an input which are: ADC (add with carry), SBB (subtract with borrow), RCR, and LCR (right and left rotations with carry) in x86. Unfortunately there are only 27 instances of these instructions in Apache. Since Apache's binary is 831168 bytes, the expected number of measured bytes before we are able to construct the ROP loop is 30784 in this alternative approach. This is more than what is needed to find a system call using slow timing attack only (13866 bytes), so in our experiments we use the slow timing approach.

We performed this attack with different basic block randomization twice. The number of bytes measured before we found the system call for the two experiments were 7049 and 22942 which is consistent with our expected number of measurements.

Since this attack on average requires 13866 bytes with slow timing attack, it takes about a week to complete on the LAN ($13866 \times 43.2\ sec$). This time may look long, but the strength of this attack over brute-force or other memory disclosure attacks may justify its usage for an attacker. First, this attack does not rely on the weaknesses of specific fine-grained ASLR implementations. Second, it does not required a JIT environment such as the one used in JIT-ROP [36]. Third, it does not require an additional memory disclosure vulnerability. Fourth, it does not require a forking server and does not introduce numerous crashes such as the BROP attack [6].

6.6 NOP Insertion

If NOP insertion is used to diversify the code, we can first measure a sample of instructions to discover at what rate NOPs are inserted. For this experiment, we have setup the multicompiler developed by Jackson *et al.* [23] to perform our measurements. Our measurements show that after estimating about 30 instructions, the NOP insertion rate can be determined accurately. Since the expected size of an instruction is 3.8 bytes, the NOP insertion rate can be determined after estimating 114 bytes on average.

After the NOP insertion-rate is determined, the approximate location of every other instruction is known by the following formula:

$Diversified\ Loc. \approx Undiversified\ Loc. + (Undiversified\ Offset \times NOP\ Insertion\ Rate)$

As a result, we can directly jump to the system call in function `__lll_lock_wait_private`. To account for the error in the location because of the random NOP insertion, we scan 10 instructions before and after the diversified location to align correctly (76 bytes on average). Since this attack on average requires 190 bytes with slow timing attack, it takes about 2.2 hours to complete on the LAN ($190 \times 43.2\ sec$).

Note that the fine-grained ASLR achieves the best protection against this attack, because by leaking a number of bytes at a memory location, one cannot learn about any other part of memory. If the code has more dependencies (e.g. pointers from some locations to the others), the attack can succeed with fewer bytes.

7. POSSIBLE DEFENSES

Complete memory safety can mitigate the impact of the timing and fault analysis attacks. Note that attacks such as the *Crafted Input* timing attack can still leak information about the code, but such information will have little value in building a payload. However, complete memory safety with temporal and spatial safety properties incurs a very high performance overhead (often multiple times slowdown) which makes it impractical for many applications [39].

Previous work has suggested re-randomizing code pages during execution as a way to mitigate information leaks, as an exploit could be rendered ineffective by re-randomizing before it gets the opportunity to actually execute [36].

Weaker forms of memory defenses can mitigate certain types of timing and fault analysis attacks with lower performance overhead. However, the problems of weaker memory defenses include still relatively high performance overhead, false positive/negatives, source/binary compatibility, and modularity support [39]. Examples include data integrity techniques such as WIT [3], which stop the *Overwrite Data/Code Pointer* attacks, and control flow integrity [1], which stop *Overwriting Code Pointer* attacks.

Some weaker memory defenses, similar to code diversification, rely on randomization and secrets to be kept. Data Space Randomization [4] can help mitigate the *Overwrite Data/Code Pointer* and *Overwrite Data* attacks. Instruction set randomization [25] can help mitigate *Overwriting Data Pointer* as the attacker will only learn the encrypted code. Note that these defenses can themselves be attacked using the same side channel attacks described in this paper. For example, an attacker could conduct an *Overwrite Data Pointer* attack and an *Overwrite Code Pointer* attack on instruction set randomization, this would reveal both the encrypted and decrypted code, and then if the encryption scheme is a simple XOR [25, 37], the attacker can learn the secret key. However, since these defenses are not widely deployed in today's computing systems, we leave a systematic analysis to future work.

Both *Crafted Input* and *Overwrite Data* timing attacks rely on inserted or changed instructions increasing execution time. A pos-

sible defense then is to insert code that does not affect execution time. For example, inserting dead code that cannot be executed without hijacking control flow [18] can cause an attacker to misjudge the location of gadgets.

Side channels are often mitigated by causing every measurement to be the same, thus destroying the measurement's distinctiveness. For example, cryptographic timing side channels can be mitigated by causing every execution to take the exact same amount of time. Code diversification techniques currently do not explicitly attempt to mitigate side channels, although some diversification techniques can affect them. For example, NOP insertion [23] will affect the size and timing of functions. Although, NOP insertion or similar techniques do not try to make all functions have the same size and timing measurements and currently do not mitigate side channels, it is possible to conjecture about potential defenses based on this idea. However, deciding NOP locations based on timing characteristics to mitigate timing side channels will make NOP insertion more predictable which, in fact, defeats the original purpose.

Note that just adding random delays to the execution of the code cannot effectively mitigate side channel attacks [13].

8. CONCLUSION

Code diversity relies on the assumption that since an attacker can not read the code, he cannot reliably exploit the code. In this paper, we show that this assumption can be broken by applying side channel attacks that allow an attacker to leak information about the code by simply executing the code. We have demonstrated how a memory corruption vulnerability can facilitate fault analysis and timing attacks on diversified code. We have also shown through analysis on real code that discerning how code has been diversified can be easily achieved. Our results reveal that while code diversity raises the bar for attackers, it is not a panacea for memory corruption vulnerabilities. We believe that as more randomization techniques are deployed in practice, attackers will rely more on side channel attacks to leak enough information to actually exploit the code.

We leave the evaluation of side channel attacks against other code bases and the analysis of exploiting multiple side channel attacks with the same payload to future work.

9. REFERENCES

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Computer and Communications Security (CCS)* (2005).

[2] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *CHES* (2010).

[3] AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. Preventing Memory Error Exploits with WIT. In *Security and Privacy* (2008).

[4] BHATKAR, S., AND SEKAR, R. Data space randomization. In *DIMVA* (2008).

[5] BIHAM, E., AND SHAMIR, A. Differential fault analysis of secret key cryptosystems. In *CRYPTO* (1997).

[6] BITTAU, A., BELAY, A., MASHTIZADEH, A., MAZIERES, D., AND BONEH, D. Hacking blind. In *Security and Privacy* (2014).

[7] BLAZAKIS, D. Leaking addresses with vulnerabilities that can't read good. SummerCon '13. http://www.trapbit.com/talks/Summerc0n2013-GCWoah.pdf.

[8] BONNEAU, J., AND MIRONOV, I. Cache-collision timing attacks against AES. In *CHES* (2006).

[9] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *ESORICS* (2011).

[10] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *USENIX Security Symposium* (2003).

[11] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *CCS* (2010).

[12] CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security 12*, 3 (Jan. 2009), 17:1–17:29.

[13] DURVAUX, F., RENAULD, M., STANDAERT, F.-X., VAN OLDENEEL TOT OLDENZEEL, L., AND VEYRAT-CHARVILLON, N. Efficient removal of random delays from embedded software implementations using hidden markov models. vol. 7771 of *Lecture Notes in Computer Science*. 2013, pp. 123–140.

[14] DUSART, P., LETOURNEUX, G., AND VIVOLO, O. Differential fault analysis on AES. In *ACNS* (2003).

[15] FORREST, S., SOMAYAJI, A., AND ACKLEY, D. Building diverse computer systems. In *HotOS* (1997).

[16] FRANZ, M., BRUNTHALER, S., LARSEN, P., HOMESCU, A., AND NEISIUS, S. Profile-guided automated software diversity. In *the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2013).

[17] GENKIN, D., SHAMIR, A., AND TROMER, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. Tech. rep., 2013.

[18] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *the 21st USENIX Security Symposium* (2012).

[19] HISER, J., NGUYEN-TUONG, A., CO, M., HALL, M., AND DAVIDSON, J. W. ILR: Where'd my gadgets go? In *Security and Privacy* (2012).

[20] HOCH, J. J., AND SHAMIR, A. Fault analysis of stream ciphers. In *CHES* (2004), Springer, pp. 240–253.

[21] HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. librando: Transparent code randomization for just-in-time compilers. In *CCS* (2013).

[22] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy* (2013).

[23] JACKSON, T., HOMESCU, A., CRANE, S., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. Diversifying the software stack using randomized nop insertion. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling 100* (2013), 151–174.

[24] JOYE, M., PAILLIER, P., AND SCHOENMAKERS, B. On second-order differential power analysis. In *CHES* (2005).

[25] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *CCS* (2003).

[26] KIL, C., JUN, J., BOOKHOLT, C., XU, J., AND NING, P. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *ACSAC* (2006).

[27] KOCHER, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO* (1996).

[28] KOCHER, P. C., JAFFE, J., AND JUN, B. Differential power analysis. In *CRYPTO* (1999).

[29] 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).

[30] 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.

[31] 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).

[32] PROUFF, E., RIVAIN, M., AND BÉVAN, R. Statistical analysis of second order differential power analysis. *IEEE Transactions on Computers 58*, 6 (2009), 799–811.

[33] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS* (2007).

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

[35] SKOWYRA, R., CASTEEL, K., OKHRAVI, H., ZELDOVICH, N., AND STREILEIN, W. Systematic analysis of defenses against return-oriented programming. In *RAID*. 2013.

[36] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy* (2013).

[37] SOVAREL, A. N., EVANS, D., AND PAUL, N. Where's the feeb? the effectiveness of instruction set randomization. In *the 14th conference on USENIX Security Symposium* (2005).

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

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

[40] THE PAX TEAM. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[41] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology 23*, 2 (Jan. 2010), 37–71.

[42] TROMER, E., AND SHAMIR, A. Acoustic cryptanalysis : on nosy people and noisy machines. In *Eurocrypt Rump Session* (2004).

[43] 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).

[44] WANG, Z., AND LEE, R. B. New cache designs for thwarting software cache-based side channel attacks. In *the 34th Annual International Symposium on Computer Architecture (ISCA)* (2007).

[45] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *CCS* (2012).

[46] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *CCS* (2012).

APPENDIX

A. ROP TIMING ATTACK PAYLOAD

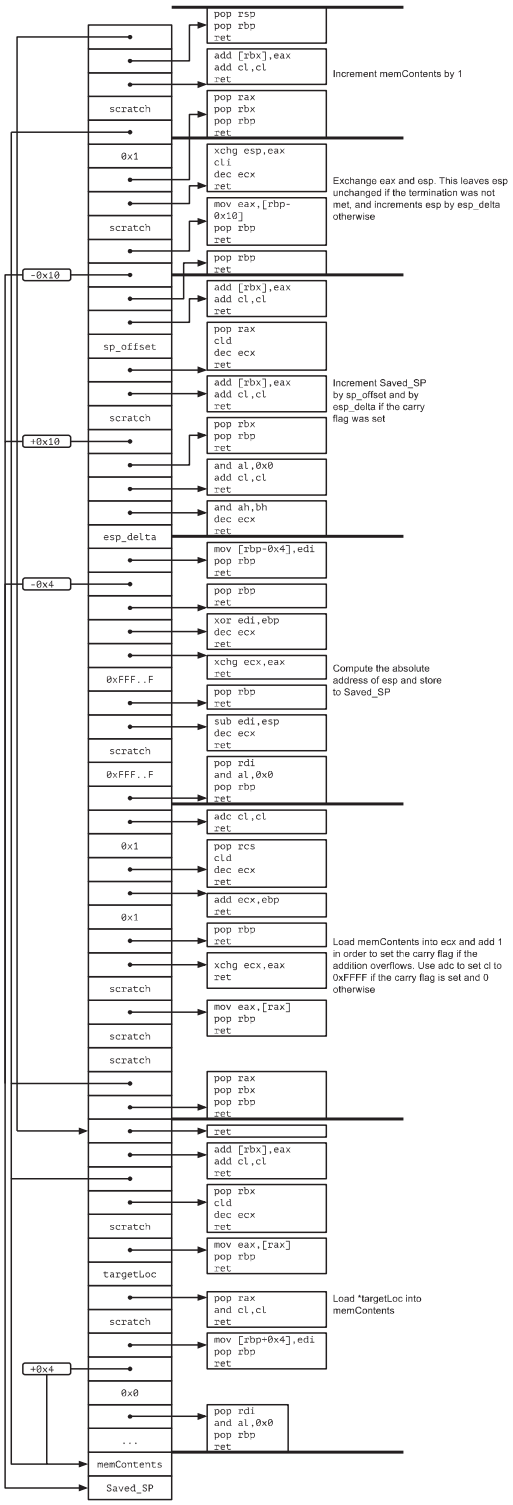

Figure 10: ROP payload used for the timing attack