An Evaluation in C: Surveying Control Flow Integrity

Michael Appel
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Email: moappel@andrew.cmu.edu

Jeff Brandon
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Email: jdbrando@andrew.cmu.edu

Tian Tan
Information Networking Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Email: tiant@andrew.cmu.edu

***Abstract*—Software vulnerabilities allow hackers to gain control of a program by redirecting the flow of execution to undesired code. It would be ideal for a system to have some built in protections to ensure that even vulnerable programs are constrained to their intended execution path. This is control flow integrity (CFI). The National Security Agency (NSA) has released a proposal for CFI, which utilizes hardware to overcome shortcomings found in CFI research. The NSA has also released instrumented ELF64 programs, a GNU library (GLIBC) and a loader, which performs standard functionality as their Linux counterparts. The goal of this paper is to evaluate the NSA's proposal for strengths and weaknesses by analyzing the given programs and libraries. We develop a new gadget discovery algorithm for this CFI system, develop tools to find and extract gadgets for analysis, and evaluate the provided binaries against CFI attacks from recent research.**

1. Introduction

Control flow is the sequence of instructions executed by a running program. Based on a programs state, this flow can take different paths by changing the location in the code the CPU fetches instructions. This location is pointed to by the address stored in the `eip` register of x86 systems. Beginning with the main function each possible branching point in the execution of a program (if statements, function calls, loops, etc.) delineates an atomic piece of a program, with respect to execution, called a basic block. The set of these basic blocks can be modeled as nodes in a graph, where a directed edge is drawn between two basic blocks if their instructions are connected in a valid execution of the program. This data structure is a control flow graph (CFG), which represents the model of possible execution paths a program can take. A programmer expects a running program to proceed only through the basic blocks they have defined. If the `eip` only ever fetches instructions from those nodes, and execution only traverses between nodes on those edges then the program has control flow integrity (CFI). The principle of CFI assures that execution only proceeds according to the original programmer's intent.

However, software attacks break this assumption of integrity primarily by either executing attacker controlled code somewhere in memory (a violation by adding nodes and edges to the CFG), or reordering the execution of pieces of code that are already loaded in memory (a violation by adding edges to the CFG). For example, a skilled attacker may overflow a buffer to overwrite a return address to a value of their choosing [1]. This historic stack smashing attack works by writing past the bounds of a local variable to cover the return address of the function's stack frame with the location of malicious code [1]. When the function attempts to return to the caller, the CPU pops the malicious address from the stack into `eip`, and the CPU is now fetching and executing the attacker's instructions.

In response to attacks like stack smashing, systems were designed with measures like non-executable stacks, called data execution prevention (DEP) in Windows. It is implemented in hardware using the NX bit of Intel chips, which allows the operating system (OS) to define regions of memory that cannot be executed. Attackers overcame DEP by developing heap-based attacks and techniques like ”return to libc” (*ret2libc*), return-oriented-programming (ROP), and jump-oriented-programming (JOP).

Heap-based exploits work by injecting code in non-stack memory locations, and therefore executing malicious code from the heap instead of the DEP protected stack [2]. In *ret2libc*, attackers store addresses of functions included in the program, which are necessarily not on the stack and therefore bypass DEP by picking particular functions to carry out arbitrary operations [3].

Shacham introduced ROP along with the concept of *gadgets* (a sequence of instructions ending with ret or similar control flow changing instruction), which makes return-type attacks more effective by eliminating the restriction to calling library functions [4]. In ROP, attackers craft the series of gadgets to perform an exploit in the face of DEP, but not the gadgets themselves [5]. Since the attacker reuses the program or library code, a non-executable stack will not prevent this type of exploit. With CFI, the number of available gadgets is greatly reduced because not every address is a valid control-transfer point. However, coarse-grained CFI only partially protects the program because attackers could point the return address to library functions which are powerful and valid control-transfer points [6].

Such functions have the ability to invoke a shell or change the permissions of memory pages to inject code [5].

These kinds of attacks prompted another defensive measure, address space layout randomization (ASLR), which randomizes included function addresses at load time, thus making it harder for attackers to guess the locations to fill their return chains. However, ASLR also has weaknesses, especially in 32-bit Linux and BSD machines [7], and several bypasses have been demonstrated theoretically and found in the wild [8], [9].

All of these attacks necessarily violate the CFI of the program. Therefore, assuring CFI in a system's execution prevents exploitation even if the code being attacked is vulnerable. Execution cannot jump to locations that are not defined by the program as a valid next instruction. Attackers also cannot reorder existing code in a meaningful way [10]. Therefore, CFI offers strong guarantees for protection against currently modeled attacks, but its complicated implementations need to be evaluated in order for it to be trusted.

Researchers and programmers have created solutions to software attacks as they arose [11], [12]. However, these software based solutions have been shown to be quickly defeated [13]. The result is an arms-race between attackers and defenders, where the defenders seem to always follow in the footsteps of new attacks. Therefore, a more effective solution is required to guarantee CFI. To that end, the NSA has developed an approach to CFI protections that combines hardware and software, and would require significant industry acceptance to be widely deployed. The NSA project functions as a starting point for organizations to begin researching the adoptions of the proposed strategies [14].

The proposal calls for the addition of only two features: landing point (LP) instructions and a protected shadow stack [14]. LP instructions are generated at compile time or inserted in existing binaries with reference source (i.e. recompiling). They provide hardware based, coarse grained guarantees in the points of possible changes in control flow by creating a bitmap of all possible branches.

The NSA implements three new instructions to instrument programs: call landing point (`clp`), jump landing point (`jlp`), and return landing point (rlp). `clp` is the first instruction of a function to disallow the program to call any address. `jlp` is supposed to be placed at the target address of jump instructions, but it becomes a challenge which will be covered in the Difficulties section. rlp is placed immediately after a call, to ensure that the program could not return to any attacker chosen addresses.

Shadow stacks complement LP instructions because they handle fine grained control at run-time [10]. The program saves copies of return addresses to the shadow stack when calls are made, and checks or overwrites the return address used by the retn instruction. This either restores execution preventing an attack, or halts the program logging awareness of the attack. The NSA notes that the shadow stack must be protected to avoid corruption by the very vulnerabilities it was designed to prevent.

The NSA has produced a detailed proposal of the possible system with recommendations for future work and adoption [14]. Along with the proposal, they have open sourced an instrumented C library and sample programs that are protected under their new scheme. While the NSAs CFI project is detailed and thorough, there has been little explicit study into the practical strengths and weaknesses of their approach. Thus our work is to reason about the defenses of the implementation, and provide proof of theory or empirical evidence of its threat reduction capabilities. This added confidence helps strengthen the argument for industry partners to move forward with adopting these kinds of protections, and implementing the changes necessary in hardware and system software design.

The outline of this paper follows. First, we will survey the related work in section 2, and analyze the most promising attacks in recent research against the NSA's proposal in section 3. In section 4, we will describe the testing environment setup to run instrumented programs on platforms that do not include the future hardware instructions or instrumented libraries. Next, we will define our gadget discovery algorithm, and provide manual analysis in section 5. Then we detail our exploit on a provided, minimal program in section 6. We also address future work to be completed in section 7 and provide some final remarks in section 8.

2. Related Work

The history of CFI research is rich and provides a strong example of the cat-and-mouse game often observed between attackers and defenders in the computer security field. Rather than discuss the history of all CFI solutions and attacks here, we will present the latest strategies for offense and defense that are the most relevant to our research.

One component of CFI research that is frequently discussed is the shadow stack. This method is the most common for enforcing fine grained CFI. Shadow stack implementation, security gains, and vulnerabilities have been discussed at length [6], [10], [15], [16], [17], [18]. The main contribution a shadow stack makes over a statically enforced CFG is seen in the case where a function is called from multiple locations, which is often the case in the real world.

Given only a CFG, a run time monitor only has knowledge of valid addresses that a function could return to during a valid execution, not which address specifically should be returned to after the current call completes. The traditional user-mode stack is vulnerable to corruption from vulnerabilities like buffer overflows. Thus, we cannot trust the return address that is stored on the traditional stack, that is defined by x86 architecture conventions.

The shadow stack is a data structure which mirrors the intended return addresses at the time a function is called. This allows for a dynamic run-time flow of information that can ensure a function returns to exactly the location it was called from. It is important to note that the shadow stack, if stored in unprotected memory, could also be corrupted, thus invalidating the security gains it offers. Therefore, it is important that the shadow stack be stored in a way such that it is protected.

Abadi et al. discuss inlined CFI Enforcement [10] which operates in a similar way to the proposed solution by the NSA [14]. Abadi et al. demonstrate low performance overhead that is practical for modern processors. Their solution involves modifying existing binary files, essentially recompiling them to support machine code level CFI policy enforcement.

Other significant contributions have been made by Carlini et al. in their paper *Control Flow Bending: On the Effectiveness of Control-Flow Integrity* [6]. In this paper the authors show that powerful attacks are still possible given a precise static CFI implementation. Control-Flow Bending refers to non-control-data attacks and represents a class of attacks that the NSA solution is potentially vulnerable to, but also beyond the scope of what the solution defends against. They provide an excellent argument on how CFI defenses should be measured in addition to attacks that reinforce the need for a shadow stack. They conclude that shadow stacks may have challenges, but CFI cannot be well-achieved without them. The authors oppose evaluating software with metrics like AIR [19] because they do not take into account the use of each remaining gadget and they do not account for a program's size in the percentage reduction.

A major contribution to CFI research made by Evans et al. provides an outline and proof of concept for a class of attack they call *Argument Corruptible Indirect Call Site* or ACICS [16]. This attack circumvents fine grained CFI and shadow stack protections by modifying a function pointer and the arguments provided to that function when it is called. This type of vulnerability does not break CFI, and is undetectable by a shadow stack because the exploit occurs on the forward edge of the call. An attacker would be able to call library functions of their choosing which is a serious breach in security. A common technique would be to make a call to `exec("/bin/sh")` to provide a shell.

An element of analysis we find relevant to our work is provided by Goktas et al. in their work titled *Out of Control* where they provide a framework for ROP Gadget analysis [5]. We aim to perform concrete analysis on the gadgets available to an attacker given the constraints imposed by the CFI implementation the team at the NSA has designed.

3. Attack Analysis

In this section we cover the latest research in attacking coarse and fine-grained CFI and shadow stack implementations. We perform manual analysis of the proposed attacks in each paper and prove why it would or would not work in the NSA's proposal. Our preliminary results are that most of the attacks are thwarted, however, several still exist.

3.1. Control-flow Bending

Carlini et al. evaluate attacks with and without a shadow stack implementation in *Control-Flow Bending* [6]. They also propose basic metrics to evaluate the defenses of CFI implementations [6] without problematic metrics like AIR [19]. The *Basic Exploitation Test* (BET) is a procedure to check for broken, insecure CFI which may be more effective than AIR.

A pass on the BET does not necessarily mean the CFI is secure, but it ensures it is not broken. The BET is the basic approach we have taken in our attack. Carlini et al. define this test as selecting a minimal program with a realistic vulnerability, and selecting attacker goals evaluate this possible exploit against a given CFI scheme. They propose the use of minimal programs since large programs make attacks easier because of a larger gadget pool.

The goals of the attacker in the BET may be arbitrary code execution, confined code execution, or information leakage [6]. Their example succeeds in the face of a very similar CFI scheme as the NSA proposal. They are effectively using LPI. Their program fails the BET because of an over-written return address that leads to several return chained gadgets. The NSA proposal would pass this test if the shadow stack was implemented because the chaining and initial overwrite would be caught.

The *control flow bending* attack, like ACICS gadgets, is one that CFI is not well prepared to deal with. In the case studies of attacks, Carlini et al. find that corrupting arguments to functions, including `printf`, provide enough input for attackers to achieve certain objectives. In the presence of a shadow stack the success is greatly reduced. In fact, they find that arbitrary code execution is not possible in any case. However, `printf` computation on memory and information leakage is still possible [6].

Like ACICS gadgets these attacks are not necessarily violating CFI. They are corrupting arguments to functions that are being mishandled. These defenses are outside the scope of CFI, but the attacks do reorder the sequence of basic blocks that are protected by this proposal.

3.2. Control Jujutsu

Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity provides a new type of gadget, ACICS, and an ACICS Discovery Tool (ADT), which can help automate this process. The tool finds Indirect Call Sites (ICS) and target function pairs, which would allow an attacker to influence the value used at that ICS.

Evans et al. use Apache and nginx to create proof of exploits. Their ACICS gadgets are successful in calling arbitrary functions that already exist. The attacks are especially powerful because they control the arguments to the arbitrary functions as well. If *Control Jujutsu* was applied in Carlini et al.'s BET it may not have been quite as successful since the attack surface and ACICS gadgets were so reduced.

Reordering and retargeting vulnerabilities, though, are not necessarily well-addressed by CFI and shadow stacks. The *Control Jujutsu* attack could impact an application protected by the NSA's proposal. Arguments are not protected in this scheme, and we show that an overwritten function pointer will, of course, be executed as a regular function call without violating CFI.

3.3. Losing Control

Losing Control by Conti et al. [17] presents several excellent attacks on software protected systems such as StackGuard [11] and StackArmor [20]. The attacks are on spilled registers containing CFI values, user mode return addresses from system calls, shadow stack exploits, and heap corruptions. Most of their attacks were presented with proofs-of-concept.

The first attack exploits the values spilled from registers by compilers under high register pressure or function call procedures. During compilation if registers are needed to execute some instructions, but their values must be preserved then they are saved onto the stack. Their values are restored before they are needed again or the callee returns, for example. This attack relies on the compiler spilling values like saved target addresses, which can be over-written by attackers. This assumes attackers can write to the stack before the value is restored. The NSA proposal defeats this attack because no registers are involved in the shadow stack. The instructions are placed on the shadow stack immediately by `clp` and checked during the `rlp` instruction. They are not stored in registers, and therefore they cannot be spilled and overwritten. A user-mode stack return address that is overwritten will be caught by the shadow stack.

Next, Conti et al. bypass CFI protections by altering return addresses from system calls that use `sysenter`. This exploit affects 32-bit x86 applications, even those running on 64-bit environments. When `sysenter` is used, or approximated for 32-bit programs in x86_64, the return address is saved to the user mode stack, which is restored more quickly when returning from privileged mode [17]. The attack utilizes a parallel thread in the some process to overwrite that user-mode return address. This attack fails in the NSA proposal again because the shadow stack addresses are saved in hardware secured memory. Despite the `sysenter` implementation, the shadow stack would retain the correct value.

Shadow stacks are examined directly, next. Conti et al. find that context switches between threads may not clear important values like the shadow stack location from registers. In the NSA proposal the shadow stack is stored in a separate page table. The `CR3` control register is used to handle addressing at mirrored offsets to the normal shadow stack. Therefore there is nothing to leak. Only `clp` and `rlp` instructions can interact with the shadow stacks page table. Therefore, this attack is also defeated.

Finally, they use JavaScript in Chromium to exploit a known vulnerability and eventually overwrite `vTable` pointers in `C++` objects. The `vTable` pointer references a read-only location in memory housing a table of function pointers for a class. These pointers can be overwritten in an attack that causes all function calls by that object to resolve to a malicious target sometimes shellcode [21]. This attack is possible, but constrained, under the NSA proposal. If the attacker can overwrite an object's `vPtr` then it must target executable code that begins with a `clp`. If the attacker can craft that code on an executable heap (e.g. JavaScript heap spraying) then adding a LPI is trivial. When the object's function is called, the malicious code will be instead. The initial arguments may not be under the attacker's control, but they gain full execution.

4. Testing Environment

Our testing environment is Ubuntu 14.04 AMD64, kernel version is 3.19.0-25. We started with Ubuntu 15.04, due to the fact that NSA programs are compiled with their own compiler, `gdb` and `objdump` refuse to run with them. We tried to find out why the programs failed to see if we could fix this problem, but it turned out to be extremely time consuming. Then we went for an alternate method, which was to set up virtual machines of different versions of Linux distributions and test `objdump` and `gdb` with NSA programs. Below is a list of distributions we have tested.

- Ubuntu 15.04 AMD64
- Kali Linux 2.0 AMD64
- Fedora 23 AMD64
- Ubuntu 14.04 AMD64

In conclusion, Ubuntu 14.04 is the only one we found in which `objdump` and `gdb` both work out of the box. However, `objdump` doesn't disassemble instructions correctly. The symbols that `objdump` uses don't align with the correct library function address that NSA programs call into, which lead to bad resolution of symbols and occasional misaligned instructions. This is not a big issue most of the time. With source code, we are able to fix symbols output by `objdump`.

5. Gadgets

In order to determine the strength of an attacker, we have analyzed the gadgets present in the instrumented binaries provided by the NSA [14]. First, we define a gadget discovery algorithm since these are not traditional ROP gadgets. We find the number of all possible gadgets, and filter those to reduce manual work load. Finally, we perform manual analysis on those filtered gadgets to determine their usability.

5.1. Gadget Discovery

Gadgets in this CFI implementation are simple to find, but difficult to define. Traditional ROP gadgets end with a `ret` statement, and start wherever the attacker has decided it is useful. The LPI instructions in the NSA proposal are the only possible starting points for gadgets [14]. These are defined as landing point gadgets (LPG). In an x86 architecture this reduces the starting point from any executable byte (assuming things like DEP are in place) to only those defined by LPI. However, the end of the gadget is harder to decide.

The end of a ROP gadget is where the stack pointer is adjusted and jumped to, but a LPG is ended where the attacker loses control. Usually this is at a branch statement, especially indirect calls and jumps. However, these short gadgets may be extended. If the attacker can influence the registers providing the indirect addresses, or if the attacker doesn't care what happens in a `call`, for example, because it returns to their desired code then the gadget is a larger virtual gadget comprised of the smaller LPG [14].

```
1 clp                    ; Must start with LPI
2 sub    rsp, 0x8
3 mov    r9d, r8d
4 mov    r8, rcx
5 mov    ecx, 0x1        ; Gadget could end here
6 call   0x30520
7 rlp                    ; Returns must hit an rlp
8 add    rsp, 0x8
9 ret                    ; Must go back to the caller
```

Listing 1. Example LPG call-preceded

The correct definition of the gadget, therefore depends on the attacker's point of view. If the attacker uses methods like those described in section 3 then they are using something like call-preceded programming [22]. This allows and may require more lengthy, complicated gadgets, like entire function calls. On the other hand, if the attacker is using small LPG while influencing indirect branches, then the end of the gadget is the first free branch. Listing 1 shows the logical gadget made of a `clp` gadget, starting at line 1, and a `rlp` gadget starting at line 7. The most successful attacks in the related work usually craft malicious argument values to functions, so the `clp` gadget may be more powerful when executed as a whole. Without a shadow stack the `rlp` gadget provides a useful stack pointer manipulation.

The counts of the number of possible LPI gadgets found in the first step of our discovery is shown in table 1. These counts do not reflect gadgets filtered for quality, which is covered in section 5.2. The AIR metric is given, but it should be taken with a grain of salt [6]. We calculate the AIR value according to the executable section size found with `size`, and with the unfiltered gadget counts which should show the least reduction. However, the problem identified by Carlini et al. is evident since all programs achieve similar scores, but the remaining gadget counts vary greatly.

The NSA proposal suggests that CFI and the shadow stack could be implemented separately. However, these numbers show that removing the `rlp` gadgets first, by implementing a shadow stack, is more useful in reducing the attack surface of the program. Our demonstrated exploit, based on the attacks cited in section 3 [6], [16], [17], also shows that the forward edge protections of CFI may not provide the guarantees desired. In essence, if the choice is implementing one or the either it seems a shadow stack is more appropriate to do first.

5.2. Gadget Analysis

In our analysis we defined gadgets as any that begin at a LPI, and continue until an indirect `jmp` (conditional or not) or `call`. We further filter these gadgets to a useful length of 10 instructions [22]. All gadgets are collected, regardless of a shadow stack implementation. If a shadow is in place, then `rlp` gadgets are discarded since it is impossible to use a `ret` or jump oriented programming (JOP) using a `pop` and `jmp` to that address [23].

File Name	Gadget	Possible	Practical
libc	`rlp`	10097	5022
	`clp`	2762	1004
	`jlp`	829	461
stack-functionptr	`rlp`	19	15
	`clp`	19	13
	`jlp`	7	7

TABLE 2: Counts of Practical Gadgets From `gadget-finder`

Table 2 reflects the amount of impractical gadgets filtered based on our definition. The overwhelming majority of gadgets belong to the `rlp` class. Our manual analysis found that `rlp` gadgets tended to be more useful than the others. There was a common pattern of a `call` shortly before the `ret` of a function. This places an `rlp` near the end which creates short, simple gadgets. Short gadgets are usually the most useful [4], [24] because they can achieve objectives without side-effects. This further suggests that a shadow stack protecting the backward edge of control flow may be important enough to implement before protecting the forward edge. This would reduce the attack surface considerably.

```
400707:    0f 1f 40 cc                rlp      <=
40070b:    c6 05 66 05 20 00 01       mov byte [rip+0
    x200566], 0x1
400712:    48 83 c4 08                add rsp, 0x8
400716:    5b                         pop rbx
400717:    5d                         pop rbp
400718:    c3                         ret
```

Listing 2. Example Difficult rlp gadget

Our analysis of the minimal program `stack-functionptr` [14] found no useful `clp` or `rlp` gadgets. However, we did find `rlp` gadgets with useful functions. These include stack pointer manipulation by arithmetic on `rsp` or `leave`, popping values into `rbp`, and one increment of a memory address but with side effects as in listing 2. A secured shadow stack would protect against all of these, however.

We were unable to find a useful gadget chain in libc to proceed with an attack despite the numerous gadgets. We examined the methods of JOP, ROP, and `clp` and `jlp` gadgets. Our search for JOP attack vectors was unsuccessful. The presence of LPI near a dispatcher gadget was not found [23]. Therefore, the virtual JOP program counter method would not be possible.

Our analysis of `clp` and `jlp` gadgets also did not prove fruitful. The gadgets had many side-effects, and we could not compile them into a working attack. The `clp` and `jlp` gadgets generally occur at the beginning or middle of

File Name	Size	`clp`	`jlp`	`rlp`	CFI only	AIR	CFI with shadow stack	AIR	ROP Gadgets
libc.so.6	146211	2762	829	10097	13688	0.9906	3591	0.9975	15977
ls	113494	364	325	1128	1817	0.9863	689	0.9948	1522
stack-functionptr	2133	19	7	19	45	0.9789	26	0.9878	74
stack-return	1924	17	6	17	40	0.9792	23	0.9880	77

TABLE 1: Possible LPI gadget counts with and without a shadow stack compared to ROP gadgets and AIR metric according to executable size

functions, respectively. This makes them exceptionally long, which increases the interactions they have with memory and registers. Each of these interactions can break a gadget, or the values a previous gadget set. For example, overwriting a needed value, or trying to write to a register's address that does not point to valid memory. This is why gadgets are best in shorter size [4], [24]. In the end, it was really only possible to treat these types of gadgets as full function calls, and attempt *ret2libc* style attacks. When used without proper context these calls will almost certainly cause the program to crash due to invalid memory references.

Another difficulty in the face of LPI when creating attacks, is that LPI practically forces gadgets to be aligned. The likelihood of the 4-byte LPI appearing unintentionally is intuitively much less than, for example, the single byte `ret` instruction: `0xC3`. Following, in an unintended gadget any ending branch must jump to an aligned LPI or an even less likely second unintended LPI. The likelihood of an unintended branch doing that is practically none. This is especially beneficial for defense because unintended gadget instructions account for a majority of found gadgets in libc [23].

This brings us to our next difficulty of ”compiling” gadgets. After hitting the first LPI, we found it very difficult to chain to the next, especially in the presence of a shadow stack. Without a shadow stack, the `rlp` gadgets seemed more useful and interesting, but again we were not able to create a working exploit from them.

6. Exploiting Instrumented Binary

This section details our attempted exploitation of `stack-functionptr` in order to conduct our own BET as defined by Carlini et al. The NSA provided minimal programs, and versions of `ls` and `libc` which are compiled with the LPI [14]. We provide a proof-of-exploit using the minimal program, `stack-functionptr` as seen in listing 4. Line 13 has a call to `memcpy` that is vulnerable to an overflow. We use this overflow to rewrite the address of the function pointer, `fptr`, with the value of the function `exit` in the procedural linkage table (PLT).

```
$ ./stack−functionptr 27 ‘perl −e ’print ”A”x24
    ,”\xb0\x05\x40”’‘
```

Listing 3. Program exploit

For example, running the program with the arguments in listing 3 will exploit the program's vulnerability. This results in a bending of the execution flow so that `foo` is not reached when called from line 14. This does not violate the proposed CFI and shadow stack, but instead leverages the same style of attacks in related work [6], [16]. These classes of attacks break the ordering of control flow, which is not necessarily accounted for in `clp` and `jlp` instructions. Only the shadow stack provides protection against the reordering of possible landing sites.

```
 1 void foo();
 2
 3 int main(int argc, char *argv[]) {
 4 void (*fptr)();
 5 char localbuf[16];
 6
 7     if (argc != 3) {
 8         printf(”Usage: %s length data\n”, argv
       [0]);
 9         exit (−1);
10     }
11 /* corrupt main() stack frame */
12     fptr = foo;
13     memcpy(localbuf, argv[2], atoi(argv[1]));
14     (*fptr)();
15 }
16
17 void foo() {
18     printf(”you are in foo\n”);
19 }
```

Listing 4. Vulnerable C Program

This exploit is trivial, and results in a simple denial of service by calling the exit function. In reality, any function could be called, but were constrained by an inability to control the arguments. The program is compiled for x86_64, which passes arguments in registers. This led us to a circular logic, where to corrupt the arguments to the `clp` gadget, we needed to first use other gadgets to populate those values, which leads back to the issue of chaining LPI gadgets.

However, even if this was a 32-bit program and passed arguments on the stack, we would not be able to overwrite the parameters for the function call. The variable we overflow, `localbuf` is above the addresses we would need to start overwriting, see table 3.

In essence, the restrictions in place by the NSA's proposal hampered efforts in making a serious attack by reducing the arsenal of gadgets, and using x86_64 programs which are better protected with register passed parameters.

However, as noted in section 3.1 and 3.2, if it possible to corrupt the arguments to functions, as in an ACICS gadget [16], then a successful attack may be mounted. Therefore, while this minimal binary passes the BET set by Carlini et al. because we did not completely achieve one of the attacker goals, if the shadow stack was not in place then the program would have failed the BET.

Address	Variable	Stack Pointers
7FFFFFD860		rbp
7FFFFFD858	fptr	
7FFFFFD850		
7FFFFFD848	localbuf	
7FFFFFD840	localbuf	
7FFFFFD838	argc	
7FFFFFD830	*argv	rsp

TABLE 3: Stack Frame Layout From `main` in `stack-functionptr`

7. Future Work

The next step in this work is to implement the protections on larger programs like Apache or nginx, and attempt to replicate the attacks discovered by Carlini et al. [6] and Evans et al. [16] The limitations of the current set are appropriate for BET, but may not provide enough attack surface to reveal as many gadgets as may be needed.

Most attacks presented here are possible because the shadow stack cannot prevent attacks on a forward call edge, like in a call to libc attack. Researching the benefits of marking function pointers with some additional information about their valid call targets could yield promising results. This responsibility may fall to the developer and have difficulties with adoption rates as such. However, even a simple system of marking a function pointer to denote that it should never call library functions could prevent a great deal of attacks that programs are vulnerable to in spite of CFI and shadow stack solutions. We believe the overhead incurred for such a solution would be negligible.

8. Conclusion

In this paper we have researched the current state of CFI and shadow stack attacks, and analyzed if those attacks would be prevented by the NSA's proposal. We defined a gadget discovery algorithm for the NSA's proposal, and performed analysis on the remaining set finding little of value. Most attacks are mitigated or weakened. However, *Control Flow Bending* [6] and *Control Jujutsu* [16] are promising routes for attackers because they obey CFI landing sites and make use of software vulnerabilities that lead to retargeting indirect function calls. Finally, we demonstrate this attack in our own proof of exploit in a minimal program according to the BET [6] defined by Carlini et al. Our results show that the proposal of CFI and shadow stack by the NSA greatly reduces the capabilities of attackers, but new and novel attacks are still possible.

Acknowledgments

The authors would like to thank our professor, Dr. Patrick Tague, along with Melissa Dark and Purdue University, for fostering the work of the INSuRE program. As well as Andy White of the National Security Agency for excellent insights into CFI and willingness to share his expertise as our project technical director.

References

[1] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, November 1996. [Online]. Available: http://www.phrack.com/issues.html?issue=49\&id=14

[2] J. C. Foster, V. Osipov, N. Bhalla, N. Heinen, and D. Aitel, "Chapter 6 - heap corruption," in *Buffer Overflow Attacks*, J. C. Foster, V. Osipov, N. Bhalla, N. Heinen, and D. Aitel, Eds. Burlington: Syngress, 2005, pp. 229 – 271. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781932266672500463

[3] S. Designer, "lpr libc return exploit," http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[5] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 575–589. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.43

[6] C. Nicholas, B. Antonio, P. Mathias, W. David, and G. Thomas R., "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[7] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[8] Z. Wang, R. Cheng, and D. Gao, "Revisiting address space randomization," in *Proceedings of the 13th International Conference on Information Security and Cryptology*, ser. ICISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 207–221. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041036.2041054

[9] X. Chen, "Aslr bypass apocalypse in recent zero-day exploits," *FireEye*, 2013. [Online]. Available: https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html

[10] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[11] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267549.1267554

[12] Vendicator, "Stackshield," http://www.angelfire.com/sk/stackshield/, 2000.

[13] G. Richarte, "Four different tricks to bypass stackshield and stackguard protection," *World Wide Web*, vol. 1, 2002.

[14] NSA, "Control flow integrity," https://github.com/iadgov/Control-Flow-Integrity, 2015.

[15] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 555–566. [Online]. Available: http://doi.acm.org/10.1145/2714576.2714635

[16] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, and H. Okhravi, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *CCS15 Denver, Colorado, USA.*, Oct. 2015.

[17] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *22nd ACM Conference on Computer and Communications Security (CCS)*, Oct. 2015.

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proceedings of the USENIX Security Symposium*, 2011.

[19] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[20] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, "Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[21] rix, "Smashing c++ vptrs," *Phrack magazine*, vol. 10, no. 56, 2000.

[22] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *USENIX Security Symposium*, 2014.

[23] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[24] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: size does matter in turing-complete return-oriented programming," in *Proceedings of the 6th USENIX conference on Offensive Technologies*. USENIX Association, 2012, pp. 7–7.