|galois|

GALOIS-17-02

Systems support for Hardware Anti-ROP

AUTHORS

JASON DAGIT
Galois

SIMON WINWOOD
Galois

GETTY RITTER
Galois

JEM BERKES
Galois

ANDREW WHITE
National Security Agency

GEORGE COKER
National Security Agency

ADAM WICK
Galois

http://landhere.galois.com

Galois, Inc. 421 SW 6th Avenue, Suite 300 Portland, Oregon 97204
National Security Agency, 9800 Savage Rd., Suite 6272 Ft. George G. Meade, MD 20755-6000

Systems support for Hardware Anti-ROP

July 18, 2017

Abstract

In 2007, Shacham introduced Return-oriented Programming (ROP), a mechanism whereby an attacker can string together small snippets of existing executable code—known as *gadgets*—in order to exploit programs without injecting new bits of code. Despite numerous proposed mechanisms for mitigating their effects, ROP attacks remain a widespread attack vector for modern software systems. Research on Control-Flow Integrity (CFI) has often shown that these protections incur significant slowdown which is understood to be too costly for general-purpose use.

We investigate the design space of minor hardware extensions with potentially large performance savings and relatively few semantic changes. These hardware extensions significantly reduce the number of gadgets usable by attackers while requiring only minimal changes to existing software, and could be augmented in critical software by stronger software CFI protections.

We present a simulated hardware platform implemented as a modification of the QEMU hardware emulator that features loose-grained forward-edge CFI enforcement and fine-grained backward-edge CFI enforcement built into the operation of the instruction set, as well as modified versions of the Linux operating system and the GNU Compiler Collection (GCC) infrastructure that allow us to run a typical Linux installation with minimal changes. We show that these simple hardware extensions and the corresponding software modifications can reduce usable ROP gadgets by a significant amount, making attacks against this platform significantly more difficult. Additionally, we discuss the tradeoffs and challenges that surfaced in the course of this implementation.

This material is based upon work supported by the Maryland Procurement Office under Contract No. H98230-15-D-0035. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office.

1 Introduction

Code reuse attacks continue to be a problem, despite many different proposed solutions [2, 9, 17, 19, 20, 22]. We believe this continues to be a problem primarily for three reasons: the performance cost of checking, the cost of switching to a CFI-enabled system, and because many software-only solutions have been defeated [8, 12, 13].

We explore modest and backward compatible hardware extensions with low overhead and examine their impact on code reuse attacks. Our goal is to 1) ease adoption, 2) lower overhead, 3) offer strong basic CFI protections, and 4) provide a hardware platform that can support additional (or even application-specific) software defenses with the same enforcement guarantees as a hardware implementation, but the flexibility of software.

Our implementation is designed around a small extension to the Intel x86_64 architecture. To emulate this architecture, we modified the hardware emulator QEMU to understand our extra instructions and translate them into native code on-the-fly. We have modified GCC, the GNU C Library (glibc), and the Linux kernel to take advantage of the instructions introduced by our architectural extensions. This modified software stack runs in our modified QEMU and can successfully run a standard Linux userland environment, while flagging hand-crafted example programs with CFI violations as erroneous. Our work elucidates the end-to-end changes required to support CFI enforcement in a real system.

1.1 Threat model

We are concerned with defending against code reuse attacks. A common variant is the Return Oriented Program (ROP), in which the attacker gains control of a stored return address and uses a return instruction to jump to a location of their choosing. In addition to ROP attacks, there are also Call Oriented Programs (COP) and Jump Oriented Programs (JOP), where the attacker gains control of the address jumped to by an indirect call or jump, respectively. We will use ROP to refer generically to all three cases, except where otherwise noted.

A code-reuse attack needs the following elements in order to succeed:

1. A bug or feature that allows the attacker to inject a ROP program;
2. A bug that allows the attacker to alter the control flow of the program;
3. A set of gadgets that
   (a) are sufficient to encode the operations desired by the attacker; and
   (b) are composable into the ROP program; and
4. The addresses of these gadgets.

Mitigating ROP attacks involves removing at least one of these elements. We assume that an attacker can inject arbitrary data into a buffer in the application and force control flow to a desired location by exploiting a program error. Our goal is to reduce the number and utility of gadgets in the program, thus making item 3 more difficult to exploit. We assume that the Operating System (OS) provides Data Execution Prevention (Data Execution Prevention (DEP)); thus, we are not concerned with direct branches that have statically known destinations encoded in the instruction. Our approach does not defend against *return-to-lib* attacks, which are a variant of Call Oriented Programming that reuse entire library functions.

1.2 Design Goals

Our design goals are:

1. Maintain backwards compatibility: Require as few changes as possible to applications and the OS. We would also like our programs to run on existing hardware when CFI hardware is unavailable, at the sacrifice of CFI enforcement. We would also like to minimize (or eliminate) changes to existing instructions.

2. Low overhead: Hardware supported CFI [4, 7, 16] could offset the substantial overhead of software-only enforcement [11].

3. Offer resilience: In a resilient system, bypassing a single CFI check should not affect future CFI checks. This property is difficult to guarantee with software-only solutions that provide security only through inductive guarantees. Furthermore, injected code is also subject to CFI checks.

4. Support separate compilation: CFI systems that require whole program analysis are difficult to scale and deploy. Many also have difficulty supporting loadable modules. Supporting separate compilation makes integrating CFI protection into existing build processes simple.

5. Break composability: We aim to remove as many gadgets as possible *and* to make it hard for the attacker to launch a desirable attack with the remaining gadgets.

6. Provide a platform: A single hardware CFI system is unlikely to define the perfect policy for every program that may need to run. We aim to supply a platform rich enough to build other systems on, enabling them to have the resilience of a hardware system with the flexibility of a software system.

1.3 Outline of approach

We explore the use of specialized hardware to counter ROP attacks. Our system requires that every indirect control flow transfer instruction be dynamically followed by a special *landingpad* instruction. Furthermore, we maintain a shadow stack in hardware that is not visible to the program; the shadow stack is checked by hardware on return instructions. Landingpad instructions mark the valid destinations of indirect control flow instructions in the read-only text of the program. Calls, returns, and jumps each have a distinguished landingpad. This approach is similar to Abadi et al. [1] with three equivalence classes.

Our shadow stack provides strict control flow integrity for returns. The only valid destinations for return instructions are those in the correct position on the shadow stack and have a return landingpad. Our system constrains the possible destinations of indirect call instructions: they can only jump to call landingpads, which only occur at the start of a function. As we aim to support separate compilation (DG 4), we cannot know all of the valid destinations for indirect calls. Our treatment of indirect jumps is similar to indirect calls: indirect jumps must branch to jump landingpads. However, call landingpads are considered to be valid jump landingpads to support tail calls and dynamic linking (see Section 2.2).

Relying on hardware allows us to meet DG 2 and DG 3. The system is resilient because managing to bypass a single check has no effect on future checks. It greatly reduces the utility of many ROP gadgets (DG 5); the shadow stack makes ROP gadgets nearly useless, while call gadgets can only branch to the beginning of functions. The landingpad instructions provide a *platform* on which other enforcement policies can build, as per DG 6. Because the code immediately following a landingpad instruction is guaranteed to be executed as long as our CFI policy is followed, one can insert other, software-defined checks after landingpad instructions to build up a stronger software-enforced policy on top of our hardware-enforced coarse-grained policy.

1.4 Proposed hardware changes

We refer to our proposed architecture as the Hardware Anti-ROP Platform (HARP). We chose to extend the Intel x86_64 architecture with additional page table entries, four new hardware faults, a new register, and three new instructions:

call landingpad (`clp`): marks a valid destination for call and indirect jump instructions and updates the shadow stack.

jump landingpad (`jlp`): marks a valid destination for an indirect jump instruction.

return landingpad (`rlp`): marks a valid destination for a return instruction, checks that the current address is at the top of the shadow stack.

Our shadow stack is implemented by adding *shadow page table entries*, which are only visible to the `clp` and `rlp` instructions. The `clp` and `rlp` instructions access the shadow stack using the virtual address stored in the stack pointer register. We assume the OS has arranged to keep the physical pages of the shadow stack distinct from the rest of an application's address space. Under this approach, the application cannot see or directly manipulate the shadow stack. The CPU communicates CFI violations to the OS through the existing interrupt mechanism. We modified Linux to handle four CFI interrupts:

1. missing `clp` at destination
2. missing `jlp` at destination
3. missing `rlp` at destination
4. shadow stack check failed

We introduce one new register, *br_type*, for tracking branches. This register is set during call, indirect jump, and return instructions. This register can take one of four values: *none*, *call*, *jump*, or *ret*. Adding this register allows us to make minimal modifications to the existing jump, call, and return instructions and delay branch checking until the destination. Delaying the check allows us to minimize changes to existing instructions, as per DG 1. We give the semantics, in pseudo-microcode, of our proposed changes to existing instructions in Section 1.5 and the semantics of our new instructions in Section 1.6.

1.5 Changes to branch instructions

Our proposed changes to branch instructions should not be observable from user code. We only modify calls, indirect jumps, and returns to track the branch state. In each case, we are moving a constant value to a dedicated register.

1.5.1 call instructions

Although direct calls cannot be changed by an attacker it is important to track them so that the shadow stack is always up to date:

```
br_type = call;      /* NEW STEP: update internal CPU state to track branch type  */
/* process the call as usual: */
RSP = RSP - 8;       /* create room on stack for return address */
mem[RSP] = ret_addr; /* store return address on stack */
RIP = x;             /* change instruction pointer to start of function */
```

1.5.2 jump instructions

For jumps we only need to protect indirect jumps. Direct jumps are assumed to be in read-only pages of memory so we assume the attacker cannot change them. For simplicity we consider the case where the destination is stored in memory:

```
br_type = jump;   /* NEW STEP: update internal CPU state to track branch type */
/* process the jump as usual: */
RIP = *x;
```

1.5.3 return instructions

Similarly to calls, we track all returns:

```
br_type = ret; /* NEW STEP: update internal CPU state to track branch type */
/* Process the return as usual: */
RIP = mem[RSP];   /* goto address at the top of the stack */
RSP = RSP + 8;    /* pop the stack */
```

1.6 New instructions

We propose three new instructions: `clp`, `jlp`, `rlp`. We refer to these as landingpad instructions. If these instructions are encoded as existing NOPs, they are backward compatible and programs containing them can be executed on hardware that lacks our extensions.

```c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

Figure 1: A call to `puts`

```
main:
  clp
  push   rbp
  mov    rbp,rsp
  mov    edi,0x4005f4
  call   4003d0 <puts@plt>
  rlp
  mov    eax,0x0
  pop    rbp
  ret
  nop
```

Figure 2: Translation with landingpads

1.6.1 `clp` instruction

```c
if (br_type == call) {
  shadow_mem[RSP] = mem[RSP];  /* Save return address to shadow stack */
  br_type = none;              /* Clear the branch tracking */
}
```

1.6.2 `jlp` instruction

```c
if (br_type == jump) {
  br_type = none;          /* Clear the branch tracking */
}
```

1.6.3 `rlp` instruction

```c
if (br_type == ret) {
  /* Check the current location against shadow stack */
  /* The stack pointer has already been adjusted, so we subtract 8 to "undo" the pop */
  if ( RIP != shadow_mem[RSP-8] )
    generate shadow stack mismatch fault; /* shadow stack check failed */
  br_type = none;  /* Clear the branch tracking */
}
```

1.6.4 Checking *br_type*

The final modification to the instruction set is to check the *br_type* flag. We add this test at the start of all instructions (excluding `clp`, `rlp`, and `jlp`). It is an error for the branch flag to be set at the start of non-landingpad instructions:

```c
if (br_type != none) {
   generate fault of type br_type; /* missing clp, jlp, or rlp */
}
/* proceed as normal */
```

1.7 Example assembly code

Given our proposed changes to the instruction set, each call should now be followed by an `rlp` instruction, each function entry point should start with a `clp` instruction, and the destination of each indirect jump should have a `jlp`. For example, if we compile a simple hello world program, we would expect the following assembly code[1]:

1.8 Hardware shadow memory

Performance concerns pose a barrier to adoption of shadow stacks [11]. However, shadow stacks have a number of features which make them amenable to hardware acceleration.

First, stack accesses exhibit high temporal locality due to the last-in first-out nature of function calls. Thus, a relatively small cache could be expected to capture the majority of shadow stack accesses. Indeed, contemporary processors include such a stack for predicting the targets of returns; it may be possible to re-use or extend such support for caching shadow stack accesses and updates.

[1] The assembly code shown here is the disassembled output from our prototype compiler.

Second, a shadow stack requires only *relaxed consistency*. Because a shadow stack is thread local and is used only by the call and return landingpads, system memory needs to be updated only on thread switch, or, depending on the implementation of the shadow stack backing store, only on thread CPU migration. Additionally, a shadow stack entry is *ephemeral*: the entry need only exist until it is consumed by a return instruction. Taken together, these imply that *the majority of shadow stack operations do not need to leave the CPU*. In particular, given an application with a small call stack depth and typical return behavior, the shadow stack will result in memory traffic only on context switches.

2 Required software changes

While most applications do not need to change to support the HARP hardware, some OS and toolchain support is required. In this section, we discuss the general changes required to any OS and toolchain; in Section 3, we discuss the changes required for our proof-of-concept implementation. The changes can broadly be classified into two categories:

- Changes required to support "unusual" control flow, or control flow that does not fit into the model of a call-stack within a single address space.
- Changes required to generated assembly. This includes both changes to compiler infrastructure as well as changes to hand-written assembly or to programs which use a Just-In-Time (JIT) compilation mechanism.

2.1 Kernel changes

The addition of a shadow page table means that the OS will need to manage additional page table entries for each process. This should be a fairly conservative change for most OSs as the OS should be able to reuse much of the existing page table infrastructure. OS designers will want to allocate the shadow page table entries whenever page table entries are generated for the normal stack to reduce the number of page faults under normal stack operation.

In order to handle the `fork` system call, the OS must copy the active shadow stack to avoid a shadow stack violation when the new process returns from `fork`.

The branch tracking register (*br_type*) must also be saved and restored on context switches. Interrupts can switch control from user mode to privileged kernel mode, and this requires that *br_type* is preserved until execution resumes at its original location in the user mode program.

The OS requires new interrupt handlers to respond to hardware interrupts generated by CPU checks for landingpads and shadow stacks. The interrupt handlers should implement an appropriate policy, which may include logging or killing the offending process. Permissive policies may require the OS to adjust state so that offending processes can resume.

Finally, in OSes supporting signals, the system must ensure that the shadow stack is in a consistent state when a signal handler returns. This is a concern when signal handlers have stack frames with unusual structure or unusual instruction sequences to return to normal execution. Furthermore, the OS cannot trust any data from user code when setting up or returning from a signal handler. We discuss signal handler frame setup in more detail in Appendix A.

2.2 Compiler Toolchain

The compiler toolchain must be modified to generate the landingpad instructions. No static analysis beyond that commonly performed by C compilers is necessary. Programs are instrumented as follows:

1. Every function must start with a `clp` instruction.
2. Basic blocks targeted by indirect jumps (including jump tables) must start with a `jlp` instruction.
3. Every call that can return must be followed by an `rlp` instruction.

Compilers for languages supporting exceptions, such as C++, must emit landingpads at the start of each exception handling block. During stack unwinding, control should be transferred to exception handlers with an indirect jump. Some implementations transfer control to exception handlers with a return instruction; that will cause a shadow stack violation under HARP.

Library functions and instantiated code templates written in assembly, or containing inline assembly, must be updated to include landingpad instructions where necessary. The linker must be updated to generate PLT entries [18] with landingpad instructions. Aside from avoiding landingpad policy violations, protecting PLT entries from ROP attacks is prudent [21]. For example, the modified PLT entry in listing 1 allows a program to call `puts` from a dynamically linked C standard library in a HARP program.

```
puts@plt:
  clp
  jmp    QWORD PTR [rip+0x20055e]
  jlp
  push   0x0
  jmp    4003b0 <_init+0x20>
  nop
  nop
```

Listing 1: An instrumented PLT entry

2.3 Runtime and Standard Library Changes

Implementations of the standard libraries require special mention for two reasons: 1) they often rely heavily on inline assembly, and 2) they implement control flow primitives that do not follow a stack discipline.

Assembly functions must be manually instrumented with landingpad instructions. Many C standard libraries have efficient assembly implementations of common math and string functions. Additionally, code that runs before or after `main` is typically implemented in assembly, which must also be instrumented with landingpads. This code is often distributed as a C runtime, which may be part of either the C standard library or compiler.

Additionally, process creation must ensure that new processes have shadow page tables and an initial shadow stack. In the case of `fork`, which creates a copy of the calling process, this implies making a deep copy of the caller's shadow page table. Much of this work must take place in the kernel, but the userland components must be modified to cooperate as necessary.

The function `vfork`, which appeared in earlier versions of the POSIX standard and is still widely used in many programs, poses a problem for the shadow stack. `vfork` is a special case of `fork` which does not copy the page table; callers of `vfork` are expected to immediately call either `execve` or `_exit`. Anything else is undefined behavior. In practice, modifications to the address space made by the child of a call to `vfork` also affect the parent process, which could be an attack vector. Making `vfork` an alias for `fork` is the simplest way to ensure the integrity of the shadow stack; this behavior is allowed by the POSIX standards in which `vfork` appears, and will preserve the semantics of correct programs.

Finally, any functions that return multiple times must be modified to accommodate the shadow stack. The case of `fork` is explained above. The other major class of these functions behave like `setjmp`/`longjmp`, where `setjmp` returns normally and future calls to `longjmp` with the same context return to the matching `setjmp`. These should be modified such that the call to `setjmp` returns normally to a `clp`, while subsequent returns (through `longjmp`) "return" via an indirect jump to a `jlp` placed after the `clp`.

3 Proof-of-Concept Implementation

We have developed a proof-of-concept implementation of the system changes discussed in Section 2. In this section, we discuss our implementation choices and the ways our proof-of-concept deviates from the ideal model. We implemented our proposed hardware modifications in QEMU [5]. We encode our landingpads with reserved bit patterns based on existing NOP instructions. This choice allows programs instrumented

with landingpads to run on stock hardware, albeit without CFI enforcement. The bit patterns are based off a 4 byte NOP, which is normally encoded as `0x0f1f4000`. For landingpad instructions, we provide a non-zero value in the least significant byte. Other choices are possible, such as using a prefetch instruction [1].

Our new register, *br_type*, tracks the landingpad state between indirect control transfers and landingpad checks. The value of this register is maintained by the CPU: for instance, when the CPU executes an indirect jump instruction it sets *br_type* to *jump*. This register must also be exposed to the kernel to accommodate context switching (including interrupts). User code must not be allowed to modify this register. In our implementation, we store *br_type* in the high (reserved) 32 bits of the RFLAGS register. This allows the OS to save and restore the landingpad state simply by storing the current registers and CPU flags. As long as the OS does not truncate or clear the high bits of RFLAGS, this approach allows *br_type* preservation without extra kernel modifications.

We leverage the fact that CPU interrupt and return from interrupt (`iret`) instructions automatically push and pop RFLAGS from the stack. A similar mechanism preserves flags in the case of fast system calls (`syscall`). However, the OS is responsible for preserving registers across context switches, and must restore them before returning back to a user process. To support our approach, OS designers must be careful to store the entire 64-bit RFLAGS and not truncate it to 32-bit EFLAGS. It is also critical that all kernel entry/exit paths, including system calls and hardware exceptions, preserve the flags and not simply clear them. If the OS clears the flags or fails to properly preserve them, this can introduce opportunities for attackers to execute illegal branches (such as ROP attacks), since *br_type* will become invalid and useless for enforcing branch checks.

We implement a permissive enforcement policy: landingpad violations log a warning and then resume normal execution by setting *br_type* (inside RFLAGS) to the appropriate value. Shadow stack violations also log a warning, and then increment the instruction pointer register. When the kernel returns from the exception using `iret`, these registers are restored and user processes continue executing. This choice was made for debugging purposes: it would not be hard to modify our implementation to kill the offending process or perform some other, less permissive action.

In our proof of concept implementation, we implement the shadow stack with a hash table in QEMU, rather than implementing a shadow page table. This permitted us to explore the relevant design space with fewer OS changes. In a proper hardware implementation, we would require the OS to have more manual control over the shadow page table, including initializing shadow page table entries for new processes.

We modified GCC 4.8.1 and binutils 2.24 to emit landingpad instructions. This required straightforward modifications to the code generator and some assembly functions in `libgcc`. The linker, part of binutils, had to be modified to emit landingpad instructions in PLT entries. Our modifications to glibc primarily affected math and string functions with optimized assembly implementations. Many of these functions only required `clp`s, which were easily inserted by editing common headers used to define assembly function preludes. The dynamic linker, which is part of glibc, required similar changes.

Signal handlers are subject to unusual control flow: after a signal handler is called, the operating system expects a `sigreturn` system call to be invoked. To accommodate this, before a signal handler is called, the execution stack in the signaled process is arranged so that the handler function returns not directly to the point of execution prior to signal delivery, but to a special shim that invokes the `sigreturn` system call. Within the Linux kernel, this bit of code is called `sigtramp`, and within glibc it is given the name `__restore_rt`. With HARP, this causes a shadow stack violation, as the signal handler will return to a point in the call stack despite not having been invoked from that location. We consider two possible ways to resolve this:

- The kernel could insert the relevant shadow stack frames into the shadow stack.
- Instead of jumping to the signal handler with a synthetic stack frame that sends the function back to `__restore_rt`, the kernel could jump to `__restore_rt`, which could itself call the handler. After the handler finished, it would then return back to `__restore_rt`, which would call `sigreturn`.

We implemented the second option, as it requires fewer kernel changes and allows the kernel to be

Benchmark	Number of ROP gadgets		
	Original	HARP	Change
perlbench	129,699	1,761	98.64%
bzip2	63,541	801	98.74%
gcc	248,186	4,660	98.12%
mcf	73,127	916	98.75%
gobmk	123,364	1,362	98.90%
hmmer	81,971	1,074	98.69%
sjeng	69,264	844	98.78%
libquantum	73,086	936	98.72%
h264ref	82,296	1,032	98.75%

Figure 3: The effectiveness of HARP in reducing ROP gadgets (length 20, including libraries).

oblivious to the structure of the shadow stack. This is important for supporting multiple compilers and languages.

4 Experimental analysis

4.1 Prototype Linux system

To demonstrate our approach, we built a Linux system using the Buildroot build system with our compiler toolchain. Much like a typical Linux system, our Linux distribution includes the GNU C Library, binutils, bash, perl, GCC, and a number of common tools. Additionally, we compiled a custom Linux kernel that supports the new interrupts raised by landingpad and shadow stack violations.

This system boots under our modified x86_64 QEMU hardware emulator. The system offers a full development environment with a version of GCC that produces binaries with landingpad instructions.

4.2 Effects of HARP

One of our design goals (DG 5) is to significantly reduce the number of gadgets available to attackers. We show that HARP reduces the number of gadgets available in SPEC2006 [15] by more than 95%. We count gadgets by examining the binary image of a program as it appears after dynamic loading and performing an exhaustive search for every indirect control flow transfer instruction in the program.[2] From each indirect branch instruction, we walk backwards in the image attempting to see if there is an acceptable series of instructions before that instruction. For non-HARP programs we count gadgets ending in a return, but for HARP we ignore gadgets ending in return because using a shadow stack prevents these gadgets from being used to transition to other gadgets.

This approach is exhaustive. It finds every sequence of bytes in the program that constitute legal instructions and ends in an indirect control flow transfer. We then count these gadgets according to length: a sequence of two instructions and then an indirect branch is considered a ROP gadget of length two, for example. Note that some gadgets are counted multiple times: a ROP gadget of length two implies a ROP gadget of length one exists, as well. [3] In a normal program, all such sequences are possible ROP gadgets for an attacker to exploit. In comparison, with HARP, only instruction sequences that begin with a landingpad instruction are valid.

To evaluate the usefulness of our approach, we created two versions of each SPEC2006 benchmark: one using the standard compiler, and one using our HARP-enhanced compiler.

[2]For example, we look for the byte pattern for indirect "CALL" instructions, at any point in the image, including constants, sub-instructions, etc.

[3]As x86 contains variable width instructions and instructions can begin at any byte offset, certain byte sequences can contain unintended valid instructions. Conversely, not all byte sequences can be decoded as valid instructions. Due to potential errors caused by attempting to decode instructions at arbitrary offsets, some shorter sequences become invalid sequences when considering gadgets of greater length.

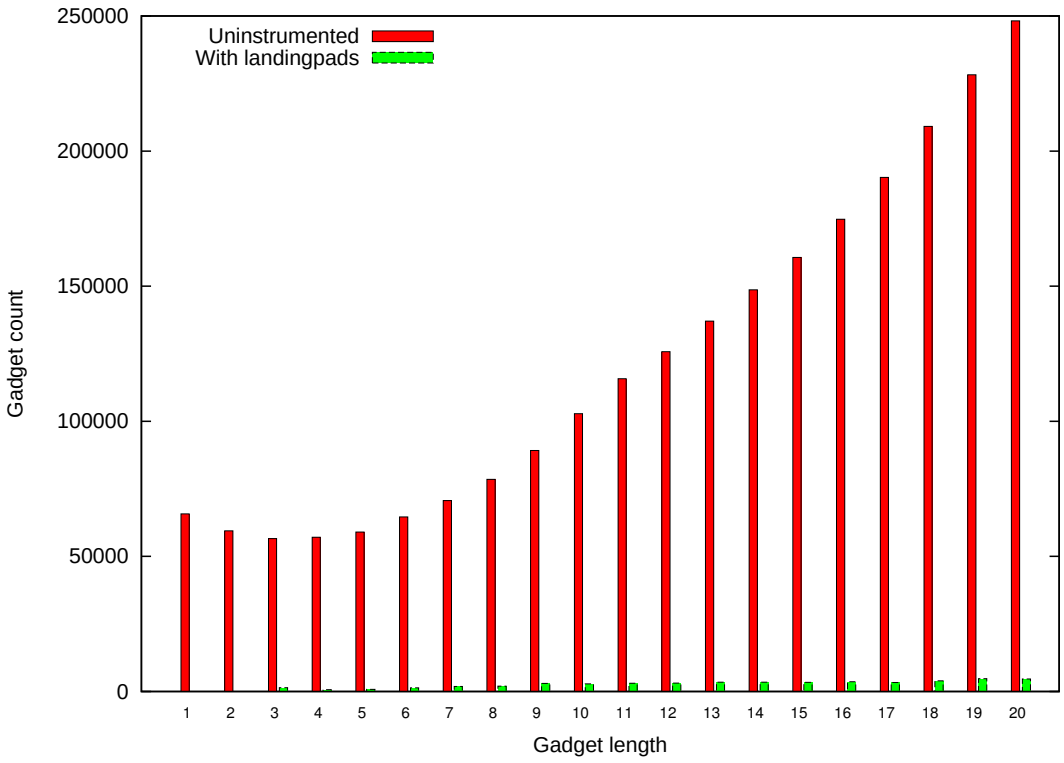

Figure 4: Reductions in ROP gadgets with ≤ 20 instructions in GCC. The dip in this graph is explained in footnote 3.

Figure 3 summarizes these results across all of the SPEC2006 benchmarks. We count ROP gadgets both in the program executable, and all of its dependencies since this combined code space is available to an attacker. HARP reduces the number of gadgets available to an attacker by over 95%.

To show the effectiveness in more detail, Figure 4 shows the ROP gadget reduction in an instrumented GCC binary. As you can see, HARP reduces the number of gadgets of all sizes. A secondary benefit of the approach is to push the attacker to use longer ROP gadgets. Since these longer gadgets will likely have side conditions that are difficult to fulfill or instructions that the attacker does not want, forcing the attacker to use longer gadgets makes crafting exploits more difficult. Other researchers have shown [14] that similar CFI approaches do not entirely remove the possibility of ROP attacks, but they make attacks more difficult.

4.3 Performance

In addition to measuring their effectiveness, we also measured the cost of using our approach on SPEC2006 benchmarks. Again, the goal of HARP is to minimize the number of useful ROP gadgets while having a minimal effect on the development or use of programs. Figure 6 show the effects on binary size and performance for programs instrumented with landingpads by our modified compiler versus an unmodified instance of the same compiler. These benchmarks were run on a quad-core i7-3520M clocked at 2.90GHz with 16GB of RAM.

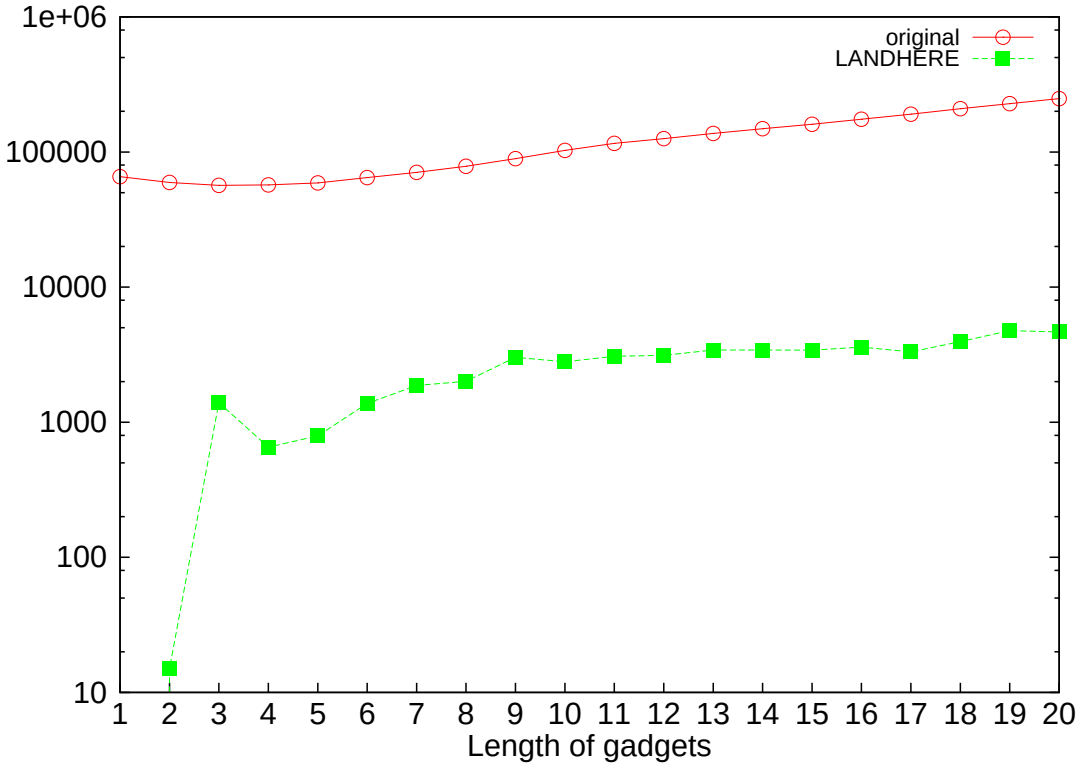

Figure 5: ROP gadgets in GCC plus libraries: original and with HARP, represented on a log scale. HARP represents a 96% reduction in ROP gadgets.

Benchmark	Size (b)			Speed		Run Time
	Original	HARP	Change	Original Std.Dev.	HARP Std.Dev	Change
perlbench	1,237,756	1,314,939	1.06	0.36%	0.23%	0.97
bzip2	72,151	75,890	1.05	0.22%	0.55%	1.01
gcc	3,710,134	3,949,782	1.06	0.91%	1.22%	1.00
mcf	22,749	25,224	1.11	5.73%	6.54%	1.02
gobmk	4,042,218	4,093,128	1.01	0.16%	0.10%	1.00
hmmer	324,601	343,668	1.06	0.08%	0.11%	1.00
sjeng	155,595	161,686	1.04	0.09%	0.13%	1.00
libquantum	59,680	64,612	1.08	0.33%	0.64%	1.01
h264ref	590,749	608,088	1.03	0.20%	0.18%	1.00

Figure 6: The cost of HARP on SPEC2006 benchmarks on binary size and performance.

Our HARP instrumentation increases binary sizes by less than 10% in most cases; we insert an additional instruction for each indirect branch target, function call, and function entry point. Furthermore, we add instructions to every PLT entry. Programs that are small, have complex control flow, or have many PLT entries grow the most. The reported run times in Figure 6 are for our instrumented binaries running on commodity hardware with no CFI enforcement. While these performance numbers do not account for CFI enforcement, they show that HARP instrumented binaries could be distributed and run with full backwards compatibility with negligible overhead.

We can only estimate the performance of a hypothetical processor implementing HARP. However, we expect that the performance overhead of landingpad instructions is likely to be low. Each landingpad instruction performs a comparison against a register and then performs a branch that should be easy to predict in the common case (i.e., when not under attack).

HARP meets DG 1, 4 and 5: backwards compatibility, supporting existing toolchains, and reducing the number and utility of gadgets. Our binaries are backwards compatible with low performance overhead. We support existing build processes by simply modifying GCC, without requiring additional tools or build steps. We also significantly reduce the number of available gadgets, while skewing their distribution towards longer and more difficult to use gadgets. We also anticipate that it meets DG 2: low overhead through hardware support.

5 Security implications

The security of CFI has been previously studied [1]; in this section we highlight a few key aspects of our approach.

5.1 Strengths

Our hardware based CFI has the advantage of *resilience*–enforcement cannot be escaped. Even if an attacker finds a way past a single CFI check, future branches are still subject to checking. Software-based enforcement mechanisms can build on hardware landingpads by enforcing security properties directly after landingpad instructions, gaining the same resilience as the hardware CFI enforcement. These derived software enforcement mechanisms can enforce more fine grained or application specific policies than are possible in hardware, but with the same guaranteed execution. Furthermore, injected code is held to the same standard. In most systems with only software enforcement, injected code would never be instrumented with enforcement checks.

Although our prototype is not suitable for benchmarking the overhead that physical hardware would incur, we believe the modest instruction changes will lend themselves to efficient implementation. As demonstrated elsewhere, CFI checks can be implemented with a small (2%) overhead [4, 16]. Additionally, there are equivalent but different design choices a hardware designer could take with our proposal. For example, our shadow memory design is not the only means of implementing a secure shadow stack.

5.2 Weaknesses

By itself our proposed hardware changes do not provide a full ROP defense, but instead it provides an important step towards that goal. Here we outline the sorts of attacks that our prototype is known weak against.

JOP & COP: We require that jumps and calls terminate at `jlp`s and `clp`s, significantly limiting their possible targets. This is a coarse grained defense for forward edges, however, and could still admit carefully-crafted attacks [13]. The coarse grained hardware enforcement provides a platform upon which different software based fine grained protections can be implemented. This two-tiered approach offers opportunities to flexibly explore defenses for forward control flow edges, which may be application-dependent.

Equivalent instructions: Our shadow stack provides fine grained checking; however, some instruction sequences are equivalent to returns, yet bypass the shadow stack check [10]. Exploiting instruction

sequences equivalent to returns would be the simplest avenue to attack our system. This could be addressed with binary rewriting and by teaching compilers to not emit these vulnerable instruction sequences.

return-to-lib: Attacks that reuse entire functions through an indirect call or jump will not be detected by our proposed hardware.

Stale shadow stack entries: Currently our shadow stack does not clear addresses after they have been consumed in a shadow stack check or otherwise made obsolete by stack unwinding. An attacker could groom the shadow stack before executing a ROP attack.

5.3 Future improvements

One of our design goals is to provide a platform on which software CFI solutions can build (DG 6). HARP provides strong protection of backward (return) edges in the call graph, but only coarse protection for forward edges. Future work should explore the types and strengths of policies that can be implemented to protect forward control flow edges. If enforcement code is placed between a `clp` (or `jlp`) and the next indirect branch, attackers cannot bypass it reliably. This could enable precise checks against a pre-computed call graph with the resilience and efficiency of a hardware solution, if such a call graph is available. Further improvements could include exploring more precise or domain-specific protection strategies built on these hardware landingpads.

6 Related Work

We are not the first to look at hardware support for CFI [7]. Budiu et al propose a modification to the Alpha architecture that adds one cfilabel instruction and modifies three branch instructions to match. Each of these instructions takes a 16 bit immediate value. When a branch is executed the immediate value at the branch and at the destination cfilabel must match. Similar to our own approach, all instructions are modified to check for illegal branching. Instead of using a shadow stack directly, their approach relies on XFI [3].

Branch Regulation [16] (BR) was proposed and implemented in a cycle accurate x86 simulator. The approach taken in BR is quite similar to our own. Calls must branch to the start of a function and starts of functions are marked with a "br annotation". Returns are checked using a hardware shadow stack. The main difference between BR and our approach is the checking for jumps. BR uses bounds checking on jumps instead of explicit jump targets. Compared to our work, where jumps may use any `clp` or `jlp` as a destination, jumps in BR are bounded to the current function, thus the attacker can use any gadgets within that range. At first this seems to be a strength of BR, but we have seen many jumps that need to exceed the bounds of the current function. For instance, symbol resolution during dynamic linking uses jumps instead of calls for performance reasons and to not disrupt the function call stack. Furthermore, when we inspect the code GCC typically emits for indirect jumps (via jump table) we see a bounds check. With BR, it is up to the OS to catch tail calls and decide if it should permit them. This would seem costly and counter to the point of tail call. The authors of BR also provide an analysis of gadget reduction with similar results to our own.

Another example of hardware support for CFI is HAFIX [4] and is also quite similar to our proposed changes, although HAFIX is designed for embedded systems. The authors introduce four new CFI instructions: CFIBR, CFIDEL, CFIRET, and CFIREC. The CFIBR instruction plays the same role as our `clp` instruction, but instead of using a stack for return addresses HAFIX maintains a set of active functions. The CFIRET instruction marks the only valid destinations for return instructions. The CFIDEL instruction occurs at the end of functions and removes a function from the active set. To handle recursive functions, HAFIX introduces a call counter and the compiler emits CFIREC instead of CFIDEL in recursive functions. The authors of HAFIX provide benchmarks for two hardware implementations of their approach. Finally, the focus of HAFIX appears to be on backward edges (returns) and little or no checking is done for forward edges (calls and jumps).

The overheads of both BR and HAFIX, both around 2%, are promising for our own work.

7 Conclusion

By simulating hardware CFI, and porting a real modern OS and applications to that hardware, we were able to demonstrate not only that doing so is within the realm of feasibility for consumer systems, and also that it can be done in a way that is backwards-compatible with current hardware. Additionally, the process of porting systems to CFI-enabled hardware revealed several corners of the design space that need special consideration, but these changes were nonetheless well-specified and localized to particular parts of the software ecosystem. We have shown that practical implementations of consumer software on similar CFI mechanisms are feasible and succeed in reducing the attack surface of resulting systems.

References

[1] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM conference on Computer and communications security. pp. 340–353. ACM (2005)

[2] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information and System Security (TISSEC) 13(1), 4 (2009)

[3] Abadi, M., Budiu, M., Erlingsson, Ú., Necula, G.C., Vrable, M.: XFI: Software guards for system address spaces. In: Symposium on Operating System Design and Implementation (OSDI) (Nov 2006)

[4] Arias, O., Davi, L., Hanreich, M., Jin, Y., Koeberl, P., Paul, D., Sadeghi, A.R., Sullivan, D.: Hafix: Hardware-assisted flow integrity extension. In: 52nd Design Automation Conference (DAC) (Jun 2015)

[5] Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track. pp. 41–46 (2005)

[6] Bosman, E., Bos, H.: Framing signals - a return to portable shellcode. In: Security and Privacy (SP), 2014 IEEE Symposium on. pp. 243–258 (May 2014)

[7] Budiu, M., Erlingsson, Ú., Abadi, M.: Architectural support for software-based protection. In: Torrellas, J. (ed.) ASID. pp. 42–51. ACM (2006), http://dblp.uni-trier.de/db/conf/asplos/asid2006.html#BudiuEA06

[8] Carlini, N., Wagner, D.: Rop is still dangerous: Breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 385–399. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[9] Castro, M., Costa, M., Martin, J.P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., Black, R.: Fast byte-granularity software fault isolation. In: ACM Symposium on Operating Systems Principles (SOSP). Association for Computing Machinery, Inc. (October 2009), http://research.microsoft.com/apps/pubs/default.aspx?id=101332

[10] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 559–572. CCS '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1866307.1866370

[11] Dang, T.H., Maniatis, P., Wagner, D.: The performance cost of shadow stacks and stack canaries. In: 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS). (April 2015)

[12] Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F.: Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 401–416. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[13] Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: Overcoming control-flow integrity. In: IEEE S&P (2014)

[14] Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G.: Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 417–432. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas

[15] Henning, J.L.: SPEC CPU2006 benchmark descriptions. SIGARCH Comput. Archit. News 34(4), 1–17 (Sep 2006), http://doi.acm.org/10.1145/1186736.1186737

[16] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Branch regulation: Low-overhead protection from code reuse attacks. In: Computer Architecture (ISCA), 2012 39th Annual International Symposium on. pp. 94–105. IEEE (2012)

[17] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 147–163. USENIX Association, Broomfield, CO (Oct 2014), http://blogs.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[18] Matz, M., Hubička, J., Jaeger, A., Mitchel, M.: System V application binary interface: Amd64 architecture processor supplement. Tech. rep. (2013), http://www.x86-64.org/documentation/abi.pdf

[19] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: Defeating return-oriented programming through gadget-less binaries. In: Proceedings of the 26th Annual Computer Security Applications Conference. pp. 49–58. ACSAC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1920261.1920269

[20] Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent rop exploit mitigation using indirect branch tracing. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 447–462. USENIX, Washington, D.C. (2013), https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas

[21] Roglia, G.F., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, 7-11 December 2009. pp. 60–69 (2009), http://dx.doi.org/10.1109/ACSAC.2009.16

[22] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. pp. 559–573. SP '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/SP.2013.44

A Signals in Linux

Enforcing CFI around signal handler stack frame setup in the Linux kernel requires caution. The `sigreturn` system call, used to return to the previous state before a signal, instructs the kernel to restore register state from a data structure called the signal frame [6]. However, the signal frame is stored in user memory and cannot be trusted. Sensitive registers, such as *br_type*, must not be restored

by `sigreturn`. Instead, *shadow memory* readable only by the kernel should be used to store sensitive register values.

A running program can be interrupted between any two instructions. It is possible for a running program to be interrupted after a call, run some other userland code such as a signal handler, and then later resume at a `clp` (at the destination of the call instruction). The `clp` instruction must take the return address from the top of the stack and store it in shadow memory. If the program was attacked between the call and `clp` instructions, then the value placed in shadow memory could have been manipulated so that a future return may be controlled by the attacker. This vulnerability could be removed by redesigning the call instruction to push the return address directly to the shadow stack, instead of delaying the shadow stack update until the `clp`. With this modification, our shadow stack should be enough to protect against even signal return oriented programming [6].