A **SECURITY PROPOSAL**
FROM THE **SYSTEMS AND SECURITY SERVICES ANALYSIS OFFICE**

NSA INFORMATION ASSURANCE
April 2015

Hardware Control Flow Integrity[1] (CFI) for an IT Ecosystem

Ever since the earliest computer systems were conceived, data has been kept in the same memory as code. This design balances many factors of IT production. However it is at the root of the "buffer overflow" exploit today, where the data that redirects control flow of a program can be intentionally corrupted by malformed input to redirect the program in ways unintended by the author of the code. The Morris worm of 1988 was an early example that demonstrated how to weaponize this weakness at scale. The Code Red virus of 2001 echoed that feat but the impact was much larger since the Internet had become significantly more engrained in the fabric of life. Subsequently, the IT industry began efforts to develop and deploy countermeasures, either in software (SW)[2] or hardware (HW)[3]. Those mitigations did not stop the adversary. They only forced him to change tactics; return oriented programming (ROP) is the latest bypass technique. Malware is still succeeding as the recent Target, Home Depot and Sony breaches have so publicly shown. In retrospect, this was driven as much by users' incessant demand for performance features as it was by industry unwillingness to apportion some of their manufacturing cycles to retool for a "security" feature that may cannibalize those performance investments. No constituency is innocent and none is guilty. The market simply failed to deliver.

Until this safety issue is fundamentally fixed, history is doomed to repeat as the trajectory of other markets converge around the same deficient IT. For example, automobiles are effectively becoming four-wheeled networked computers. The Internet of Things will computerize and network just about any ordinary object. Medical devices, power grids, and other sectors are converging similarly. The new, more physical consequences of misdirecting control flow in these emerging use cases should not be a surprise when they start to occur (e.g. exploiting the wireless interface of the car radio that must also access (and thus control) the engine speed subsystem). The good news is that there is a path to fix control flow (CF) exploitation for good. It won't materialize overnight as it was unintentionally designed wrong from the beginning, thus is significantly entrenched. The following proposal is a design path to move the ecosystem out of this dilemma into a CFI-compliant world.

Proposal

This is a notional design for a notional ecosystem to use HW to accomplish what SW alone cannot. It is a starting point for the community to begin evolving the underlying technology to a CFI compliant state in order to stop, not make more difficult, exploitation of this particular weakness. The ideas may or may not be applicable to a particular ecosystem. The reader is encouraged to pattern their specific

[1] CFI is in the spirit of "Control-Flow Integrity, Principles, Implementations and Applications" Microsoft Research.
[2] Stack and Heap canaries, Address Space Layout Randomization, Secure Development Lifecycle, Heuristics, Isolation (sandboxing or virtualization), etc
[3] No execute bit for virtual memory permission

ecosystem against this concept or propose alternatives that can meet the challenge that balances the commercial pressures and engineering practices with the hard reality of past mitigations - partial isn't good enough.

Feature 1: A method to define the intended control flow graph (CFG) of a program to HW. Conceptually pre-compute a bitmap to overlay the address space where a "1" is a legitimate branching source or destination and "0" is not. The map would be referenced using a notional Control Flow Unit (CFU) much like a memory management unit (MMU) does for virtual address resolution. The map gets loaded at the start of a running process. During a branch operation, the "branching permission" of the source or destination would be validated by the CFU in the pipeline similar to how virtual memory is checked during each instruction fetch. An expansion of this would be to have a map for each type of data driven branching (or one map with multiple types). Another way to anchor the CFG is to use new machine instructions, called "landing point" (LP) instructions. They must be the destination of any control flow branch such as function call/return or a computed jump. Otherwise, the CPU faults when a branch occurs. No matter how one decides to anchor the CFG control points into the HW, it must achieve a coarse grained, non-bypassable CFG that can be built upon to get finegrained control flow; something non-existent today.

Feature 2: A method to protect dynamic control flows - a *protected* shadow stack[4]. For any call, a copy of the return address is stored into both the regular stack and the shadow area. When a return happens, both copies are compared. If they are not equal, then the CPU faults. This provides a fine-grained CFI for *dynamic* control flow points such as return destinations that can't be pre-computed during compile time. The shadow values must be protected by HW from SW since SW bugs can undermine the soundness of the check by corrupting values after they've been stored but before being used.

Both features must be present for completeness of the security argument. Feature 1 will severely constrain (or eliminate if precision is maximal) call oriented programming (COP) and jump oriented (JOP) attack techniques. Feature 2 will *eliminate* the ROP technique and further frustrate COP/JOP attacks, perhaps to extinction.

Structural changes like this have been done before. For example, early general-purpose computing devices used SW to manage process memory separation and was able to evolve to a sound isolation mechanism using virtual memory and context switching HW. A more recent constructive disruption specifically aimed at malware was the introduction of a new virtual memory permission, write no-execute (W^X), and the now ubiquitous adoption of it by the ecosystem.

CFI can be a similar story. It will require all parties of the ecosystem to play a role (i.e. SW developers, OS vendors, HW manufacturers and users). Otherwise, the users and information owners of IT systems will continue to experience very large and substantive economic and business impacts.

[4] NSA has a related patent (i.e. "Protecting a Computer Stack" US7581089).

This proposal disrupts the existing ecosystem roughly as follows[5]. **SW**: The compiler would need to emit landing points. The linker would need to update an object header with the CFI information. The loader would need to allocate shadow memory and set CFI-mode of executable memory (derived from extended header info). The OS would need several new fault handlers for the landing point and shadow mechanisms, and it would need to manage the shadow stacks (e.g. allocation, protection, growth, context switching). The applications would need to be recompiled with the new toolchain. **HW**: Requirements for HW would include tracking branching state, four new instructions and a shadow stack architecture.

The remainder of this paper will discuss anticipated ecosystem challenges using this proposal as the archetype.

Design Considerations and Philosophy

Solutions attempting to Inject CFI compliance onto a legacy ecosystem that is ignorant of control flow constraints have always made (exploitable) compromises in order to not break that legacy world. Hence this proposal abandons legacy protection to allow flexibility to “design from scratch” to get *complete* CFI coverage (i.e. non-bypassable) of a run-time to justify that the investment is done only once. However, a backward compatible transition is still possible. Users have the option to run legacy SW on a future CFI-compliant platform or run CFI-compliant SW on non CFI-compliant HW (with no CFI in either case).

For this discussion, SW boils down to a set of units of work (i.e. functions). A compiler/linker combines them to create an application or library. Other SW re-uses these units. A stack unifies the control flow between units. There are multiple control flows within an overall control flow (i.e. threads). Code may be dynamically generated (i.e. JIT engine). This model cannot change because re-use and modularity are compelling advantages. However, the model entrenches two distinct classes of edges on the CFG. The forward edge is *static* where the destination is known before the program runs (i.e. when it was compiled or loaded). Function pointers are in this class since they still represent a known set of destinations that get primed *before* a program starts. Essentially no program intentionally loads a function pointer to jump anywhere but a function entry point. The return/backward edge is *dynamic* in that the destination is known only *after* the program starts. However, no program wants to return to anywhere but the calling unit. Exception handling kinds of control flows (e.g. Structured Exception Handling, signals) also exist within this paradigm but happen “under the covers” of the CFG expressed in the source code. Imposing an explicit CFI constraint on this implicit control flow (CF) of SW poses minimal risk since it’s enforcing the implicit agreement.

A single technique is very unlikely to materialize that can *precisely* enforce CFI and not break the above SW model. So this proposal tries to balance HW and SW to apportion the right role with the right amount of work to gain the most precision with the least amount of disruption. SW could precisely check the forward edge. However, practically speaking, the best SW could do for the return edge is check that the return site looks like a generic return. Thus, HW is unavoidable. HW is present at every

[5] NSA has rebuilt a small Linux distro with a gcc landing point modification. Further we’ve modified QEMU “HW” to be landing point aware to run the Linux distro. Based on this work, our assessment is it is a modest effort.

branch point and it cannot be turned off, thereby guaranteeing some level of coverage for every instruction executing in a run-time instance. More importantly, because the HW logic is constant and pervasive, CFI completeness can be measured and reasoned about. SW as the sole CF mitigation medium is too diverse and inconsistent to have any kind of chance of sustained validation. Finally, a proposal such as this that desires to "end this once and for all" must be future-proof to readily accommodate any unforeseen attack techniques that might render a more brittle solution worthless over time. Thus, the roles of HW and SW (HW is kept "simple, fast and immutable" and SW is "complex, slow and mutable") allows the ecosystem to "dial up" the CFI strength in the field with SW as needed without jettisoning the HW investment.

In summary, the following precepts about any CFI design are considered to avoid repeating history. 1) It must strive to *precisely* enforce a branch point to its intended destination. 2) It must validate *every* branch. This doesn't necessarily mean every point must have a check; rather the design has been validated so that a check is necessary (or not) for every branching point[6]. 3) It must have immutable, independent layers of logic enforcing the intended control flow. 4) It must not adversely impact the user. Failing any one of these will doom an implementation.

Detailed Design: Minimal Prototype (Land-here instructions + Shadow Stack)[7]

The HW will have three new (landing point) instructions that pair to a respective branching instruction (e.g. call (direct and indirect), ret and jump (indirect)). These instructions will be referred to as CLP, RLP and JLP for (C)all, (R)eturn and (J)ump Landing Point respectively. Call --> CLP, ret --> RLP and jmp indirect --> JLP.

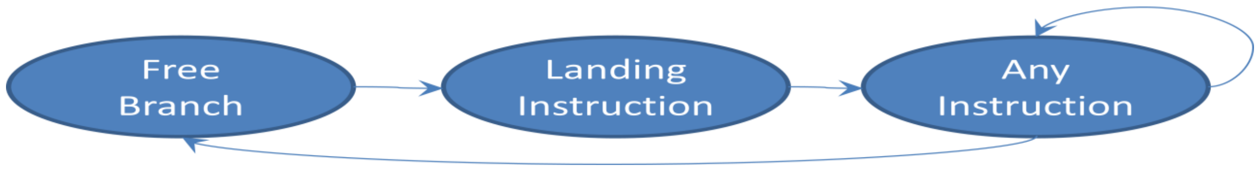


Figure 1

When "CFI-mode" is active, the HW will require the branch destination to be a corresponding landing point instruction as in Figure 1. The branch instruction will set a branching state to indicate branching is active. The landing point will then unset the branching state and the next instruction will be fetched. Any non-landing point instruction that executes when branching is active will cause a fault. This is enforced by the instruction pipeline and is non-bypassable. This approach enforces only a coarse CFG since any call site can land on *any* CLP, not necessarily the one it was supposed to go to. Likewise a return site and jump site can branch to any RLP and JLP respectively.

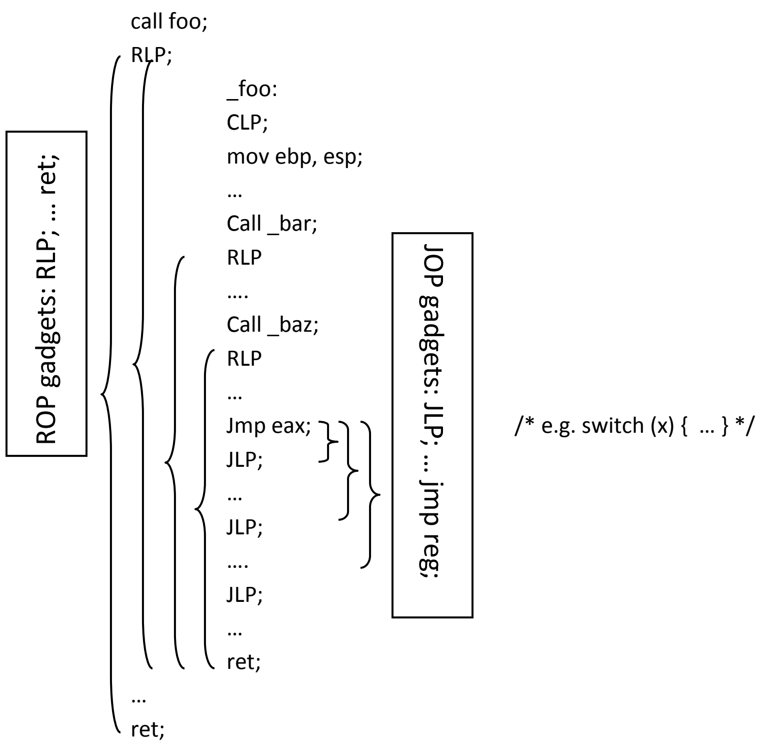
The landing point opcodes must be chosen to run on previous generation HW to allow backward compatibility. They also must be a very unique pattern within a typical executable to prevent misuse of

[6] See the SCK discussion around Figure 11 to see how short cutting these design tenets can lead to a bypass.
[7] This discussion uses x86 conventions for ease of consistency but there's no reason the ideas can't translate to other architectures such as ARM and MIPS.

un-aligned instruction streams. For example three opcodes: “0x0f,1f,40,aa”, “0x0f,1f,40,bb”, and “0x0f,1f,40,cc” perform as NOPs today could be used as LP instructions. Others are possible too. Whatever is chosen should be carefully considered by an ecosystem since their uniqueness property will need to persist forever as well.

Instrumenting SW with landing points is relatively simple. Every function entry point will now have CLP as the first instruction. An RLP instruction will follow any call-site. Every indirect jmp will have a JLP at the potential destinations. A compiler should be able to incorporate them easily. Figure 2 illustrates a disassembly of an instrumented binary:

…



```
call foo;
RLP;
        _foo:
        CLP;
        mov ebp, esp;
        …
        Call _bar;
        RLP
        ….
        Call _baz;
        RLP
        …
        Jmp eax;
        JLP;
        …
        JLP;
        ….
        JLP;
        …
        ret;
…
ret;
```

Figure 2

Instrumenting HW should also be straight forward. It is appealing but potentially nonoptimal to further encumber existing branch instructions (e.g. call/ret) with additional logic to move values to the shadow stack – call and ret are tightly integrated into the pipeline because they have been a constant since the beginning. Thus CLP and RLP are a natural place to extend “new shadow silicon” and not disrupt the old silicon if that’s appropriate for a specific architecture. This requires *both* call direct and indirect to be paired with CLP in order to protect the return address and RLP is a somewhat redundant protection mechanism for the return edge flow.

There are several challenges for a shadow stack. 1) It must keep integrity of the values stored in it or it has no enforcement capability. 2) It should impose the least amount of coordination complexity to the SW. Stacks are usually well behaved (i.e. LIFO) but occasionally they get out of order (e.g. setjump()/longjump(), thread switching, exception handling, etc). Stack synchronization is very difficult to accurately track by HW that has very little context of the overlaying SW. Requiring developer assistance to synchronization imposes an unwanted burden that he can also get wrong. 3) It must be performant.

Accomplishing protection and easy synchronization can be done by having an additional page table that is used by the CLP and RLP instructions only, in effect CR3.shadow. In this case, the real stack virtual address (e.g. esp) is (only) used by CLP and RLP to traverse CR3.shadow. The backing physical memory of that traversal can then be distinct from memory used for the normal virtual memory even though they both refer to the same virtual address. This creates a structure that will *both* protect the shadow stack integrity and track the real stack automatically, thereby maximizing the HW investment and simplifying SW integration. The reader should not get too horrified at the performance implication of coopting CR3. CR3.shadow is used as a notional concept because it has this dual property. If there's another approach that preserves compatibility with SW while maintaining equivalent protection at a lighter weight, the better. It all comes under the umbrella of a CFU.

The pseudo microcode for instructions would roughly work as follows in Figure 3 (italics are new functionality to existing instruction, a ',' is the end of a step, a '.' is the end of the instruction logic):

```
enum branching = {none, call, return, jump};
call xyz:                                   /* direct or indirect
        push return-address,                /* save the return address on real stack
        branching = call,                   /* activate CFI check
        IP = xyz.                           /* goto xyz
CLP:
        if (branching != call)
                stop.                       /* ignore and move on
        if (CR3.shadow[ESP] non-existent) then
                shadow non-exist fault.
        CR3.shadow[ESP] = [ESP],            /* save return address into shadow
        branching = none.                   /* deactivate CFI checking

ret:
        branching = return,                 /* activate CFI check
        IP = pop return-address.            /* return
RLP:
        if (branching != return)
                stop.                       /* ignore and move on
        if (CR3.shadow[ESP] non-existent) then
                shadow non-exist fault.
        if ([CR3.shadow[ESP]] != IP) then
                shadow mismatch fault.
        branching = none.                   /* deactivate CFI checking

jmp:                                        /* indirect only
        branching = jump,                   /* activate CFI check
        IP = indirect address;              /* goto indirect address
JLP:
        if (branching != jump)
```

```
                stop.                          /* ignore and move on
            branching = none.                  /* deactivate CFI checking

        SCK:                                   /* Shadow ChecK
            if ([CR3.shadow[ESP]] != [ESP]) then
                shadow mismatch fault.

        Any non-landing point instruction:     /* in Decode stage
            If (branching != none)
                landing point fault.
```

Figure 3

Intentionally, each landing instruction blithely continues if branching is not set for it. This allows for different landing points to be located at one logical point in case there's a common destination for branch-sites that use different branching to get there. For example, a function that is both indirectly jumped to and directly called would have JLP and CLP at the entry point. Notice that the branching state is never unset unless the correct landing point eventually gets executed. If the correct landing point isn't executed (thus branching remains active), a non-landing point will eventually be reached and the CPU will fault immediately (or you've run out of memory and you fault anyway…).

Since the shadow area is a page map, it can be created and destroyed either on demand or transparent to the application much like virtual memory. Thus dynamic, multi-threaded applications are easily handled. For example, the loader can deliberately allocate the CR3.shadow area before the process is started with hints from the executable header OR the memory can be allocated by a fault handler dynamically upon a first CFI access to a new stack (either a CLP or RLP).

The OS will need to be aware of several new faults: landing point fault and shadow page faults (create and mismatch). How they fault is ecosystem and platform specific. However, the current handling of page faults should provide a model. The landing point fault should include the branching type that failed. The shadow mismatch fault could pass the value stored in the shadow. On the SW side, adding landing points will require simple updates to the compile toolchain. Complex linker fix-ups will *not* be necessary as CFG-precise logic is not embedded into the HW. The loader will have some new stages to consider (i.e. pre-configuring shadow areas) but nothing it hasn't done in similar circumstances before.

This proposal can also be safely employed within the kernel, a place not immune to memory corruption exploitation.

Effect of landing points and a stack shadow on control flow exploitation

ROP techniques are the preeminent target for any countermeasure. It's assumed that successful real world gadget chains must have two or more gadgets. One-to-one precision is not possible in a C/C++ world due to OO constructs so it's (theoretically) possible to redirect a branch-site to a legal but unintended landing point that might be the uber-gadget of one. However, the uber-gadget does not exist in typical code or it would have been exposed by the hacking community by now. In practice today, typical gadget chains are 5-8 gadgets long (or longer). This reality presents several challenges to the gadget chain author: 1) he needs sufficient (safe) vocabulary embedded in the binary for his gadget chain, 2) he needs sufficient writability of data structures to control the flow between gadgets, and 3) he needs to sustain state across gadgets. Finally, existing SW countermeasures (e.g. ASLR, canaries, heuristics, etc) are ignored as adding to the strength or weakness of the proposal. They can be bypassed and thus cannot enforce the security argument.

Landing points change the geometry of a gadget chain construction significantly, making it much more brittle. Gadgets today have an arbitrary entry point and can be of arbitrary length (i.e. native gadgets). The only essential is they must terminate at a controllable free branch (e.g. ret, call/jump indirect) to get to the next gadget. The *potential* native gadget count is in effect limitless. This gives infinite flexibility to the gadget chain author for even a modest size binary. In contrast, the landing point gadget count is a very countable finite number[8]. A landing point gadget must always begin with a landing instruction and end with a free branch instruction (e.g. ret, call/jump indirect) – the gadget chain author can no longer pick an *arbitrary* location to safely start whatever code snippet is needed. As a consequence, a CFI gadget incorporates garbage code (to the malware) along with the desired code. Realistically, the potentially useful CFI gadget count will be close to zero since as the size of the gadget increases, the likelihood of crashing the application before subverting control flow increases. Unfortunately, the dramatic reduction of the gadget count does not have any bearing on gadget *content* (i.e. usefulness to the exploit). So, one can only logically conclude that a landing point CFG (significantly) reduces the potential of exploitation, but doesn't eliminate it.

Irrespective to the security effect, landing points add a bit of program safety not possible today. When the control flow has been accidentally broken due to an accidental problem, the program will trap to a handler immediately vs. run for some indeterminate number of instructions before crashing which then make it difficult to backtrack to the problem.

A shadow further restricts the geometry of a gadget chain. The shadow forces a precise pairing between a caller and a returnee. Worse (for the adversary), a *protected* shadow guarantees that precise pairing. As a result, at least roughly half of the landing point CFG has perfect CFI and ROP chaining would no longer be viable. However, it's dangerous to claim never or always in a security argument. To defeat a shadow, the pairing guarantee has to be broken. With this proposal, one cannot return to a location that is not in the shadow. One cannot return to a location that is not matched in both the real and its corresponding shadow stack slot. One cannot groom the (protected) shadow with memory corruption such as a "heap spray". The only way to populate the shadow is for the HW to issue a call, a form of control flow grooming. For example, if normal control flow has traversed the RLP gadgets needed for a chain and they conveniently remain populated in the shadow (likely not in a sequential ordering needed to run as a chain), the attack initialization needs the ability to pivot the real stack pointer to a shadow frame AND pre-populate disparate memory locations on the real stack with data for each gadget. The functionality of any RLP gadget will not only need to accomplish a specific action (e.g. load a register), it will also need to pivot the stack pointer elsewhere on the stack. Perhaps a more likely scenario, if the

[8] An initial small survey of Linux executables shows about ~95% reduction in the gadget count. Deeper analysis is planned in the near future.

shadow is not populated with any needed RLP blocks then the attacker must also stimulate the control flow to reach them. Take the following idealized code bug

```
while (getstr(str))            /* str can overwrite foo */
        …
        (*foo)();              /* calls functions that execute RLP blocks of interest */
        …
}
```

This bug would allow the attacker to target functions that contain RLP gadgets of use and populate the shadow below the current stack frame before he initiates the chain. If the code path through the function does not normally traverse the RLP gadget, the attacker will additionally need to influence the data that would induce that far away control flow. If there were more precise constraints placed around the function pointer above, as discussed in the next section, the only way control flow could be redirected arbitrarily is by influencing the data that controls conditional branching of any particular needed path.

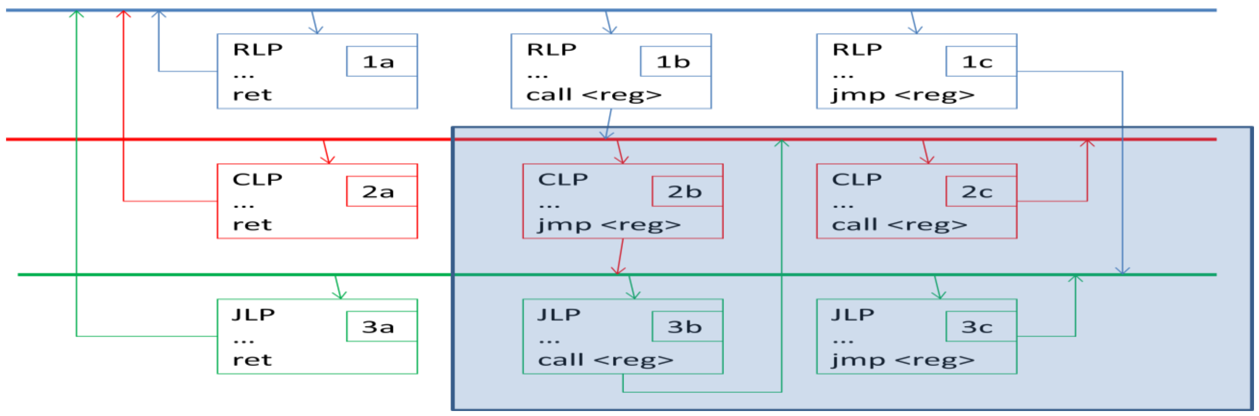
This is not a mathematical proof on the soundness of a shadow stack but it illustrates the accidental nature of exploit success when operating with a shadow. It will take a very unlikely alignment of conditions to theoretically run *one* ROP gadget. In practice, making a chain of several ROP gadgets is even more unlikely. If there is an "accidental" RLP block that can be exploited, it likely can also be remedied by compiler improvement. Thus, a protected shadow makes ROP-style chaining irrelevant.

The shadow stack has additional value to the landing point argument. If there's an accidental RLP instruction appearing somewhere in memory, control flow cannot reach it if there isn't a call opcode located immediately preceding it because a call is the only way to inject the RLP address into the shadow. This illustrates why several layers of independent enforcement are needed for any CFI design. An LP opcode that is expected to be forever unique is a hard claim to maintain alone.

The semantics of any COP gadget will be equivalent to calling a function since function entry points are the only place a CLP will exist. Then a COP gadget chain is essentially a control flow of functions that are called out of order, often with incomplete parameters. When called in this manner, the population of non-fatal, side-effect-free functions that populate registers and memory in a reliable enough way for a gadget chain to complete is small to non-existent. JOP gadgets will be similarly large, or larger.

Finally, three types of landing points vs. just one universal landing point further constrain the population of valid gadget chains. To intertwine any remaining potential COP and JOP gadgets requires an atypical transition gadget. It would take the basic block form of "JLP; … load x … call x;" and "CLP; … load x … jmp x;". These kinds of flows are uncommon. This is further complicated by the disparate location of data structures that source the indirect calls/jumps (requiring multiple overwrite stages). Many of these data structures are located in read-only locations, thereby reducing the pool of exploitable branching data. So the likely gadget chain is going to be a monoculture of JOP or COP gadgets.

The following figure illustrates the complete CFG of landing point gadget flow. The shaded area is the exploitable control flow since ROP has been eliminated as a viable option. Chaining gadgets is going to be difficult if they aren't of the form "CLP … *pop x* … call x" or "JLP… pop x … jmp x".



In summary, this proposal in its minimal form is toxic to gadget chain creation[9]. The reader is encouraged to imagine and explore this new restricted gadget world to see the difficulty that this proposal would create since much of the security claim is empirical. For example, this proposal stymies recent literature demonstrating defeats to leading edge CFI mitigation implementations:

"The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)"
"ROP is still dangerous: Breaking Modern Defenses"
"Escape From Return-Oriented Programming: Return-oriented Programming without Returns (on the x86)"
"Out of Control: Overcoming Control-Flow Integrity"
"Jump Oriented Programming: A New Class of Code-Reuse Attack"

Dialing it up

Any CFI proposal must be capable of reaching a one-to-one precision for every branch or it will leave an open path to bypass. The analysis done above is in the context of exploiting today's executables. There's no way to know what executables years from now might look like when any CFI implementation hits the market. So assume this paper underestimates that and the control flow precision of the above minimal proposal is not complete enough. Because this proposal is not solely HW based, code can be nimbly adapted where the SW toolchain can re-orient the type, number and content of each CFI block as needed. For example, using the snippet from Figure 2, the SW can wrap a label check around the landing points roughly as follows:

```
…
mov eax, $_foo;          /* get the function pointer entry */
lea ebx, [_foo-4];       /* the associated label stored at the function */
cmp ebx, 0x12345678;     /* check label can be used from this call site */
bne _terminate;
call eax;
RLP;
…
        0x12345678
        _foo:
        CLP;
        mov ebp, esp;
        …
```

[9] It also frees up resources that largely fail to stem control flow caused problems (e.g. crisis patches, monitoring systems, etc) and can be re-focused to be more effective to other aspects of cyber security.

```
                ret;
        …
```

Figure 4

One could “dial up” the performance of the original design by adding a cmp flag check into CLP instruction. Then a precise indirect call-site of Figure 3 would then look like:

```
mov eax, $_foo;            /* get the function pointer entry */
lea ebx, [_foo-4];         /* the associated label stored at the function */
cmp ebx, 0x12345678;       /* pre-sets the NE flag */
call eax → CLP             /* faults if the NE flag set */
```

Figure 5.

And a coarse indirect call-site (for the case where multiple functions with different labels are called from this site):

```
mov eax, $_foo;            /* get the function pointer entry */
cmp 0,0                    /* force the flag true */
call eax → CLP             /* never faults due to NE flag */
```

Figure 6.

And a direct call (it does not need a label check due to W^X nature of execution):

```
cmp 0,0                    /* force the flag true */
call foo → CLP             /* never faults due to NE flag */
```

Figure 7

One can even apply label checks (or some other action specific to the function) *after* a landing point instruction since the HW guarantees a landing point must be reached. This might be needed where a critical property must be maintained for the subsequent code (e.g. a function that turns off a security feature or a safety property assumption is asserted).

Once the forward edges of the CFG become as precise as the backward edges, then one has a proof of CFI effectiveness. In practice however, there will be call sites that are one-to-many (e.g. class method calls in OO languages) and thus the forward labeling will never quite be one-to-one in every case. If the data to these sites can be shown to be immutable (e.g. it came from a table in read-only memory), then the security argument remains strong. In any event, a future compiler can do many things to order the semantics *inside* of any landing point block as well, thus disrupting malware without HW upgrades.

A further use of the CR3.shadow investment would be to assist HW enforcement of all the stack instructions staying within bounds of the stack memory region. This is not critical for CFI but does make the legendary stack pivot much less feasible. Since CR3.shadow should be a mirror of the stack pages only, an additional CR3.shadow permission lookup by any stack-based instruction (e.g. pop reg; push value; move esp, xxx; etc) would fault if the stack pointer was referencing a non-existent shadow page. This now restricts any stack pivot to just the real stack for “free”, adds a little stack safety and further inflicts pain on the malware author who is now clinging to any remaining COP and JOP gadgets (e.g. CLP; … pop eax; … call eax;CLP…) that now only work within existing stack area (a much less convenient place to exploit).

Since the CR3.shadow traversal is only done by CLP and RLP and it shares the same address as the real page table, there's no way to read or write to the shadow by any other instruction in the process. However, there is no apparent reason for a debugger to see what's in the shadow area. If the shadow doesn't match the stack, the HW will fault and communicate the offending value. So this shouldn't be a problem for the ecosystem. If the scenario does present a problem, the kernel can still read and write the memory backing CR3.shadow and a new system API could satisfy this need (the paper also proposes a related instruction SCK, to check the shadow with the real stack).

How the return address is handled within HW is not fixed. The paper has shown a robust mode where the return address is written, read and checked. But a "lean" mode could be to blindly take the value in the shadow for the next instruction address. This mode would allow skipping the storing onto the real stack resulting in a performance boost. However, this implementation partially breaks debugging or any other kind of stack introspection tool (e.g. call back stack tracing).

A design knob tweak might be to allow landing points and shadow to be individually configurable for an address space. For a challenged design environment (e.g. an internet of things device that is performance constrained) one or the other technology features might be possible but not both, thus providing a significant improvement over nothing. This design knob also might be helpful if both features cannot be introduced into the ecosystem at the same time.

Adoption by the ecosystem

Achieving ideal CFI compliance (i.e. control flow *safety* for an *entire* run-time) will likely require an intermediate stage where executables have both CFI and non-CFI (i.e. legacy) code modules. It's unrealistic to expect all code will be CFI-compliant upon initial deployment of CFI HW. This will require some means for HW to differentiate modules that run in the same address space so it won't falsely fault when running non-CFI code. Thus CR3.shadow can also be employed to differentiate memory areas too (yet another use for this sunk cost of transistors!). Because there may be a long term need for some code within an address space to be CFI incompatible (e.g. legacy JIT, or extremely fast code that can't tolerate any drag, or something unpredicted), having a flexible HW mechanism to intermingle modules will be important for a long time too (ideally in small chunks, another use of the CR3.shadow mechanism!). An alternative strategy for a migration strategy is "all or nothing" which obviates design accommodations for cohabitation of CFI and non-CFI code. This alternative has its own market risks and the ecosystem could end up with an (expensive) unused feature. In any event, the ecosystem partners will need to make some choices about this that will have consequence on the eventual design.

Assuming CFI will be bi-modal within an address space, the interface between CFI and non-CFI modules must be carefully thought through, or the interface will be a wide open hole to bypass any CFI protection for that address space. Assuming legacy code will expect to work as it does today when in legacy mode, no security arguments will be made for that part of the run-time. Additionally, one can assume legacy code in a CFI ecosystem will be diminishing to zero over time which greatly reduces its potential to hijack the address space. To isolate the legacy code and keep the minimalist CFG CFI contract, then any branch in CFI code must be paired with a landing point. This presents a problem since control point anchors don't exist in legacy code so one cannot branch *in* to legacy code directly from CFI code without faulting. Bridging between worlds will require a bi-memory thunk of memory (Figure 8). Any branch *out* of legacy into CFI must be paired with a landing point. This will also require a bi-memory thunk. The goal is to minimize the ability to use the legacy as a trampoline around CFI.

Figure 8 shows a design to safely accommodate legacy code *within* a CFI run-time. For example imagine an application A that uses a critical .dll D that was written by a company U that no longer exists. The developer can re-tool A but must ship the binary with D until D's replacement can be found.

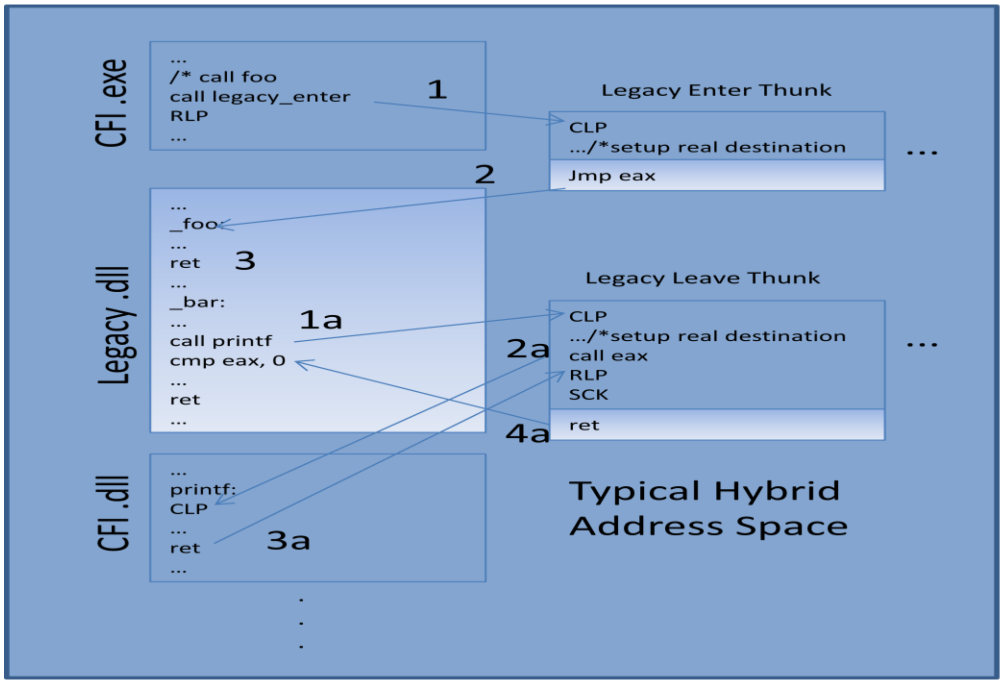
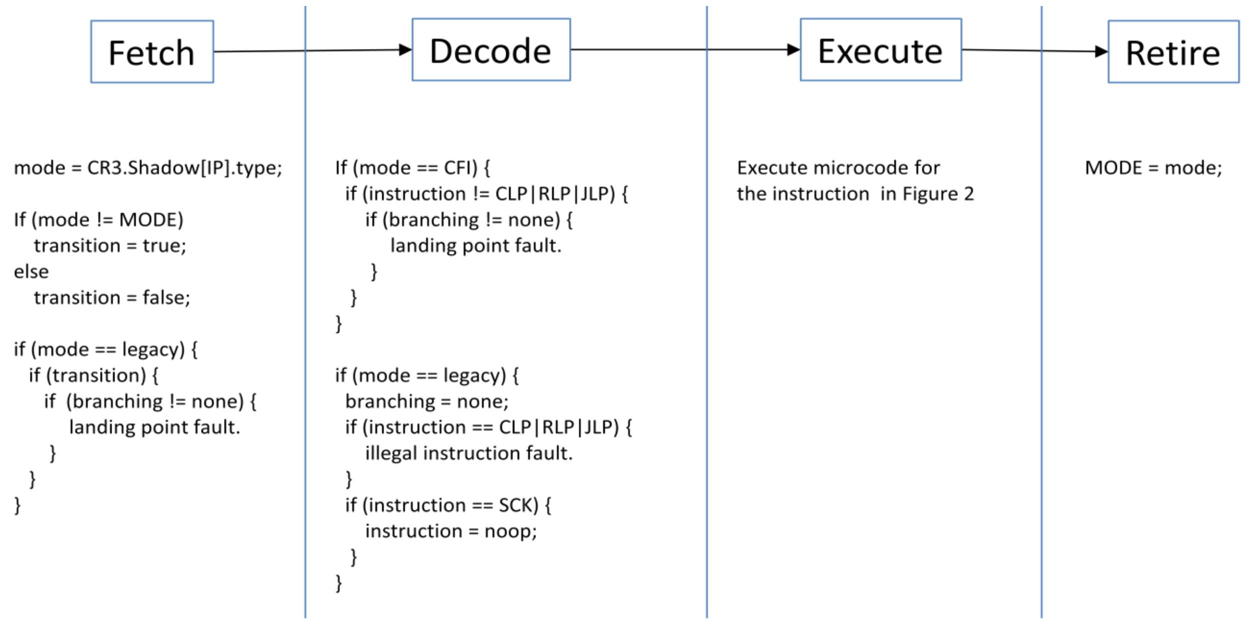


Figure 8

Figure 8 has two example flows (1-3 and 1a-4a). The gradient shaded blocks are CFI non-compliant areas in memory. The solid shaded blocks are CFI compliant. These are distinguished by a type bit in CR3.shadow. Because there are no landing points within a legacy area, they are shimmed in with the thunk object(s) as two deliberately adjacent memory areas of each type. This model takes advantage of the fact that in effect one can "drive" across that border without needing a landing point, thereby not causing a fault. However, whenever the SW "flies" across the border (i.e. using a branch) there must be a landing point, which is enforced by HW. This allows the SW to create gates which isolate the potentially toxic non-CFI code.

Steps 1, 2 and 3 illustrate a call to a legacy function foo(). The destination of the thunk has magically been done at step 1. This is simplistic in terms of what really happens for dynamic calling conventions but the example is kept this way for brevity since different ecosystems accomplish dynamic calling in their unique ways. Notice that 3 may or may not return to the RLP in CFI.exe and represents the one threat one can't control. The ret can return to another location within Legacy.dll (i.e. there was a bug). It cannot return to any other location in CFI compliant memory, the shadow prevents that. Steps 1a, 2a, 3a and 4a illustrate control flow in the opposite direction, a call from a non CFI-compliant area to a CFI-compliant area. One assumes a library object will (legally) need to do this. At 1a, the call to printf will be replaced with the address of the leave thunk and CLP will place the ultimate return address on the shadow. Note, because of the transition from legacy to CFI, the branching flag will be live. 2a is the call to the destination intended by the legacy code. That CLP will place another return address on the next slot of the real stack and the shadow. 3a must return to RLP (due to shadow). Since one still needs to return to the real caller, just take the next value off the stack that was placed there in step 1a. However, there is no guarantee it's still the same value. Since there's no way to read the shadow directly (it's opaque by design), the SCK instruction fulfills this narrow purpose. SCK is guaranteed to execute because the RLP is guaranteed to execute due to shadow enforcement. The ret at step 4a will then execute if SCK doesn't fault, correctly returning to the original caller.



```
mode = CR3.Shadow[IP].type;

If (mode != MODE)
   transition = true;
else
   transition = false;

if (mode == legacy) {
  if (transition) {
    if  (branching != none) {
        landing point fault.
    }
  }
}
```

```
If (mode == CFI) {
  if (instruction != CLP|RLP|JLP) {
     if (branching != none) {
        landing point fault.
     }
  }
}

if (mode == legacy) {
  branching = none;
  if (instruction == CLP|RLP|JLP) {
     illegal instruction fault.
  }
  if (instruction == SCK) {
     instruction = noop;
  }
}
```

Execute microcode for
the instruction in Figure 2

```
MODE = mode;
```

Figure 9: CFI HW instruction life cycle

The HW logic in Figure 9 will enforce that landing points are appropriately enforced with the bi-modal memory scheme in Figure 8. This is a notional CPU; any real CPU may have other considerations and not work in this exact fashion. There are several subtleties in Figure 9 that bear highlighting. A branching state is *always* blindly set during the Execute stage. However, the state will be ignored while in legacy mode because the Decode stage "unsets" all of it before any Execute logic occurs. This is done since the branch state could easily accumulate all three types of branching while in legacy mode. This allows for an accurate landing when transitioning to CFI mode as only the most recent free branch type will be present. Figure 9 chose a harder line while in legacy mode by faulting if any landing instruction happens to execute. Landing point opcodes could exist, although it's supposed to be unlikely. They were chosen partially because they are NOP's today so one might change Decode to Figure 10.

```
If (instruction == CLP|RLP|JLP|SCK)
        Instruction = NOP;
```

Figure 10

The ecosystem must balance the risk between branching to a very unlikely pattern of bytes appearing in legacy and breaking absolute backward compatibility. In either case the real landing point *logic* must be prevented from executing in legacy mode since it could be used to interfere with CFI correctness when the CPU bridges back and forth between CFI and non-CFI mode.

Having memory that can be reliably used in this fashion does have a side effect of creating a new category of memory that might be of use in other ways. One might imagine *little* snippets of (CFI unprotected but formally validated) logic that can only be entered and exited through predefined points (e.g. a new form of secure API??).

It's important to encourage the ecosystem to make the transition to complete CFI coverage quickly (e.g. make that JIT module emit landing points, refresh those partner .dll's, etc). Any non-CFI memory will continue to be a very rich springboard for an exploit. This is the lesson of ASLR deployment when not all the code could be relocated in an address space and the non-relocated code remained an easy exploit path. To transition to complete compliance, the ecosystem can start as opt-in with a sunset date of opt-out (much like the policy of W^X compliance). For the majority of applications, reaching this state should not present a problem.

There could be consequences to existing SW that may require a small amount of refactoring of the toolchain. Any construct within the executable that doesn't pair a calling edge with its returning edge will fail. This is by design. This construct may happen when a return occurs alone or the stack frame has been moved before the return edge occurs. For example, a pattern such as "push address; ret;". The second problem is when a location must be legitimately branched to but it doesn't have a landing point. This will fail, also by design. Setjmp()/longjmp() falls into this category. Asynchronous exceptions may fall in this category too (e.g. HW faults and signal handlers). Some types of structured exception handling may fall into this category.

Setjmp()/longjmp() illustrate the second problem. Setjmp() creates a block of control flow state to be restored by longjmp(). CFI will not adversely affect setjmp() as it's part of the normal CFG. On the other hand, CFI will break longjmp() since it "returns" to the return site immediately following setjmp() by using a jmp.[10] Typically there is only an RLP there, no JLP, since it's just a function as far as the compiler is concerned. This will (falsely) fault if jmp'ed to from longjump(). Thus some refactoring is needed to ensure a JLP is emitted into the binary. It's likely a small change to a library function or header file. This likely will also need to be dialed up. A buffer used by setjump()/longjump() will allow abrupt state change (by design). Thus it forms a potential way to link JLP gadgets if the exploit conditions allow it. For example, if setjump()/longjump() pair occurs on a program path that loops AND the attacker can influence the loop conditions AND there's a memory corruption bug of the buffer between setjump() and longjump(), then it's possible to create a gadget chain that will branch at will to *any* JLP gadget.

[10] Ubuntu 14.04 Linux libc does this for example.

Therefore, a universal setjump label must also be emitted adjacent to the setjump() JLP. This then is checked by the longjump() before it executes the jmp. This trivial change significantly increases the precision of longjump branching to only the setjump() destinations in the application (which will be very few). If longjump() "returns" to setjump() using ret, that will need to be refactored to use the indirect jmp because the value in the shadow stack is not guaranteed to match. It could have been overwritten by other function calls before longjump().

Similar challenges may lay in wait for exception handling functions that return to non-landing point locations. HW exceptions will almost always occur in these unpredictable locations and may require the process to resume at them. Any exception handler control flow that requires a restart at such a location will need to be refactored to trampoline off the OS to restart the process, an independent broker, since there won't be an anchor at that location and there's no way to branch there from within the process otherwise (by design). For example, assume process P has a HW exception X at location L. The OS catches it, stores (X, L) for P to be checked later, restarts P at the exception handler E. E calls F, calls G, etc until the handler needs to restart P at L. It then invokes the OS with L, which can be validated by the OS before it restarts P. Many exception paths will not be affected as they follow normal call and return patterns which will be anchored.

A tempting remedy is to eliminate RLP as a design feature and in its place, pair SCK with every paired ret using the assurance that a compiler can fix the problem. Figure 1 is reconfigured to show this in Figure 11.

```
…
call foo;
…
        _foo:
        CLP;
        mov ebp, esp;
        …
        Call _bar;
        ….
        Call _baz;
        …
        Jmp eax;                /* e.g. switch (x) {  … } */
        JLP;
        …
        JLP;
        ….
        JLP;
        …
        SCK;
        ret;
…
SCK;
ret;
```

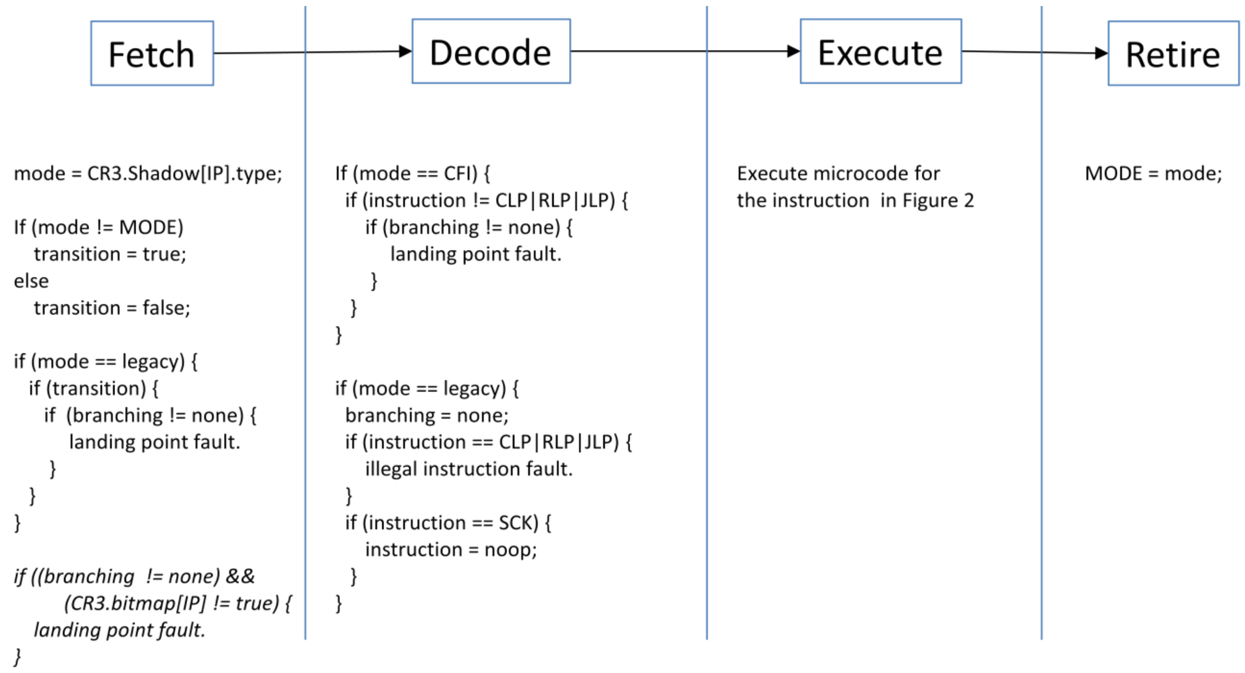
Figure 11

This increases the flexibility but demonstrates how one can break the CFI design with the best of intentions. Any return not preceded by an SCK is a potential exploit path (i.e. coverage is not complete).

The entirety of the security argument rests on the integrity of that unprotected branch and history shows that's not a good bet (i.e. SW can't adequately protect SW). Unless one can show the infallibility of the integrity at those exposed branching sites, when a site is exploited (not if), the exploit can *easily* begin ROP-ing to any other ret that isn't preceded by SCK since SCK-free gadgets will occur off-cut in many places if the aligned ones can't be harvested.

One anticipates any irregular control flows will happen in consistent ways and require one-time changes to specific parts of the toolchain. This proposal doesn't prevent soundly refactoring them to a CFI-compatible version. The integrity of data structures for exception handling is out of scope for this discussion. However, presume they are corruptible; this proposal would make exploitation ineffective.

The CFG bitmap illustrates the CFG anchoring facet of the proposal but is ignored for the minimal design as it is duplicative to landing points and without landing points, you don't get viable shadow HW. Without a shadow protecting the dynamic edge of the CFG you will never be able to get to a precise enforcement around the *entire* CFG, thereby leaving the mitigation investment at risk of being bypassed (yet again). But let's add a bitmap into the minimal CFU subsystem. It will reinforce that a landing point was intended at that location, something already safely assumed because the code is read-only and the instruction pipeline requires a landing point to follow a branch.



```
mode = CR3.Shadow[IP].type;

If (mode != MODE)
   transition = true;
else
   transition = false;

if (mode == legacy) {
  if (transition) {
    if  (branching != none) {
        landing point fault.
     }
  }
}

if ((branching  != none) &&
       (CR3.bitmap[IP] != true) {
   landing point fault.
}
```

```
If (mode == CFI) {
  if (instruction != CLP|RLP|JLP) {
     if (branching != none) {
        landing point fault.
     }
  }
}

if (mode == legacy) {
  branching = none;
  if (instruction == CLP|RLP|JLP) {
     illegal instruction fault.
  }
  if (instruction == SCK) {
     instruction = noop;
  }
}
```

```
Execute microcode for
the instruction  in Figure 2
```

```
MODE = mode;
```

Figure 12: CFI HW instruction life cycle with bitmap enforcement

Bitmap enforcement can be done in the Fetch phase (Figure 12) which faults when the bitmap is not set but branching is active (code in italics). This allows one to overlay anchor points onto legacy code that has not been made CFI compliant by recompilation (providing at least very coarse CFI protection where none exists) AND doubly enforce that landing points are intended to be where they are for CFI modules.

To overlay the bitmap when the source code is not available, one would need to *accurately* disassemble the binary to identify the branch destinations. This is unachievable in an automated way for some binary. For CFI code, every landing point would need a corresponding entry in the bitmap. A bitmap will also create an additional step for things like a JIT compiler that will need to also provide a corresponding bitmap on the fly. These are all solvable. A bitmap does not negate the need for landing points. But it begs the question, if the toolchain made the investment to emit bitmaps, why shouldn't it also emit landing points? The necessity of a bitmap is only when code cannot be recompiled, a narrow legacy case. The bitmap's unique incremental contribution to the security argument is it will further stymie code injection attacks in legacy memory (W^X already handles this attack path).

To prepare the SW base, landing points can proactively be compiled into code since they will run on legacy machines as a NOP until the new HW arrives. The shadow has no SW artifacts to complicate legacy compatibility either. In fact one could also prepare shadow sections in any object header which would then automatically be in play when the HW can support and silent in the legacy situation. This proposal has enough flexibility that shadow management can be done many different ways to accommodate particular run-time considerations or SW development schedules.

Because the shadow HW is distinct from the landing points feature, each can be deployed on its own if there are manufacturing constraints preventing both from being introduced at the same time. However, landing point instructions must come at the same time or before shadowing. Also, if they are delivered in stages, the cohabitation of non-CFI and CFI memory would not be possible until CR3.shadow is available to arbitrate. Obviously having both features available simultaneously would be far more effective for the ecosystem to adopt.

JIT is probably the most challenging SW logic areas for any CFI proposal. In this case, the engine simply needs to emit landing point instructions as part of a basic block. This will be one-time work to re-factor the engine in strategic places. Merely recompiling the engine code makes it compliant. The shadow mechanism will work automatically with a new JIT run-time. Or if unavoidable, JIT output can run in legacy memory.

The bitmap does avail the ecosystem a different adoption trajectory. First stage: the HW ships with CR3.bitmap and all code (apps and OS) eventually becomes bitmap compliant, which means it can easily become landing point compliant too. Second stage: HW ships with landing points and CR3.shadow. Note: stopping at the bitmap stage will not lead to CFI success. So the business risks of prematurely stopping at stage one vs. the incremental security benefit of a bitmap must be weighed.

The landing point and CR3.shadow design will have a performance penalty, not surprisingly. The magnitude of the penalty depends on the different ecosystems and the different designs. Whatever the magnitude, the performance cost buys very strong malware resistance properties that are not realizable any other way. It is orders of magnitude faster and more capable than an equivalent SW-only CFI. Much of the CR3.shadow access can happen in parallel with dedicated transistors so that will not have a big effect on instruction throughput. The landing point contribution to the performance impact is largely a function of how many times CLP/RLP instructions execute to do the additional shadow reads and writes.

That is code dependent and locality will play a role. They are fast instructions since it is already done fast for the existing call and ret instructions which landing points in effect are duplicating. CR3.shadow will put pressure on the cache/TLB and these effects are difficult to estimate without real HW. One would expect that the effects will be absorbed as CPU designs optimize and new types of faster, cheaper, bigger on-die memory are realized after the initial deployment. The hope is for the HW vendors to invest thought in how to make the CR3.shadow (or an equivalent) a practical reality. It is the unknown pole in the performance tent of this proposal but it solves many problems. At some point in the ecosystem refresh cycle, when non-CFI HW is unlikely to exist in large enough quantities to care, one might also remove some of the legacy SW mitigation techniques such as canaries that clearly have large performance tax, thereby increasing performance! Or perhaps the mobile ecosystem model will predominate where devices have a (relatively) short lifetime and the SW stacks are very vertical so that when the time comes, an OEM can build a lean version of the SW that isn't going to run on non-CFI devices because it's not supposed to and there won't be a market penalty for doing so. In other words deprecate to the future when it's practical.

Similarly, to tailor CFI to a given ecosystem will cost money/resources but it's not a cost that hasn't been borne by the ecosystem purveyors before for other similar scale features. Landing points are just new instructions that aren't doing anything any other instructions haven't done before so limited creative effort should be required to create them. CR3.shadow is largely a copy/paste of a subset of logic from the well worn MMU subsystem. Designing a shadow subsystem to be optimal for HW and SW will be the brunt of the HW cost and some clever innovation will likely be needed to balance the performance challenges inside the memory bus system that an additional data structure will impose. But if similar challenges were not a deterrent for similarly complex features in the past, then it doesn't need to be one for solving memory corruption. Similarly the SW ecosystem is probably most concerned with backward compatibility and minimal forking of common code. Properly chosen landing point opcodes embedded into an executable will not prevent running on legacy HW. Once the SW toolchain and core OS has integrated landing points and Shadow HW, the applications can trivially be re-compiled to be compliant. The shadow and landing points are self contained architectural elements and can be compartmented by a CFI-aware OS that can sense what kind of HW features are there and then instantiate the appropriate drivers and handlers, thereby running in a CFI-unaware state on legacy HW.

Any questions or comments can be sent to control-flow-integrity@nsa.gov