CCFI: Cryptographically Enforced Control Flow Integrity

Ali José Mashtizadeh
Stanford University
mashti@cs.stanford.edu

Andrea Bittau
Stanford University
bittau@cs.stanford.edu

Dan Boneh
Stanford University
dabo@cs.stanford.edu

David Mazières
Stanford University

ABSTRACT

Control flow integrity (CFI) restricts jumps and branches within a program to prevent attackers from executing arbitrary code in vulnerable programs. However, traditional CFI still offers attackers too much freedom to chose between valid jump targets, as seen in recent attacks [5, 9, 11].

We present a new approach to CFI based on cryptographic message authentication codes (MACs). Our approach, called cryptographic CFI (CCFI), uses MACs to protect control flow elements such as return addresses, function pointers, and vtable pointers. Through dynamic checks, CCFI enables much finer-grained classification of sensitive pointers than previous approaches, thwarting all known attacks and resisting even attackers with arbitrary access to program memory.

We implemented CCFI in Clang/LLVM, taking advantage of recently available cryptographic CPU instructions (AES-NI). We evaluate our system on several large software packages (including nginx, Apache and memcache) as well as all their dependencies. The cost of protection ranges from a 3–18% decrease in server request rate. We also expect this overhead to shrink as Intel improves the performance AES-NI.

Categories and Subject Descriptors

D.3.4 [**Programming Lnaguages**]: Processors; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Languages

Keywords

Control Flow Integrity; Return Oriented Programming; Vulnerabilities

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813676 .

1. INTRODUCTION

In recent years, sophisticated attacks on software vulnerabilities show that deployed protection mechanisms can be bypassed (e.g., [3, 4, 14, 19] and many others). The weakness in many deployed defenses is that they focus on patching specific attack techniques rather than addressing the fundamental problem. For example, stack canaries [8] assume a stack overflow; non-executable memory [24] assumes code injection; and address space layout randomization [18] assumes that information cannot be leaked. These defenses can be circumvented, for example, by overflowing the heap, executing a chain of existing code fragments using return-oriented programming, and leaking a pointer. We need a more principled approach to defense.

Exploits often work by hijacking the program's control flow to execute unintended code, for example, to start a shell. Indeed, all the attacks mentioned above work by hijacking control flow and all the defenses mentioned try to prevent specific approaches to control flow hijacking.

Control Flow Integrity

A principled solution, called *control flow integrity* (CFI) [2], prevents an attacker from arbitrarily modifying the target of indirect jumps (e.g., return addresses and function pointers). Ensuring control flow integrity would prevent all attacks based on control flow hijacking, which includes all the sophisticated attacks listed above.

However, practical implementations of CFI are insecure [5,9,11] for two main reasons. First, CFI uses static analysis to determine the target of a pointer, which is not always precise and leads overly permissive checks. Second, implementations seek to minimize runtime checks for performance reasons.

In practice, existing CFI systems are very coarse-grained; they group function pointers and return addresses into just two different classes, preventing swaps between the two. While a function pointer cannot be replaced with a return address, these systems allow attackers to swap any two return addresses (or function pointers). Attacks against existing systems exploit the coarse granularity of those systems to break CFI. The most advanced system today groups function pointers based on the number of arguments (Arity) [21]. Existing systems do not provide fine-grained classification because they are statically classifying pointers at compile time based on language characteristics.

This paper introduces a dynamic approach to control flow integrity that allows our compiler to efficiently encode and modify pointer classification at runtime. We associate metadata with all control flow pointers that prevent attackers from arbitrarily swapping or modifying pointers. Unlike previous approaches we can naturally support language constructs such as type casting without compatibility issues or overly permissive checks. We can also clas-

sify pointers based on purely runtime properties such as the address they are stored at.

We show that on modern processors fine-grained control flow integrity can be efficiently achieved using cryptography. CCFI identifies all objects that would affect a program's control flow (e.g., return addresses and function pointers) and computes a message authentication code (MAC) of such objects each time they are stored in memory. This MAC is stored along with the object and checked every time the value is loaded from memory. The random secret key used for computing these MACs is stored in dedicated registers so that it can never leak by a memory disclosure bug. By checking the MAC of every control-flow element before using it, the system prevents the attacker from writing arbitrary addresses to hijack execution.

Cryptographic CFI

Cryptographic CFI is a general technique for building control flow integrity systems. Existing systems can be implemented using our technique by mapping pointer classes generated by their classifier into an integer tag that is included in the MAC computation. All existing systems can make use of our approach and could extend themselves to benefit from Cryptographic CFI's ability to reclassify pointers at runtime. This allows classifiers to express program behavior that cannot be determined statically. There are three or fewer classes in most existing systems (except FECFI, which categorizes by number of arguments leading to an effective limit around eight), while CCFI supports up to 2^{80} as explained in Section 4.1.

Our approach enables new classifications of pointers that are not possible with static approaches. First, the ability to reclassify function pointers at runtime to support type based classification. In existing systems, the static analysis is not sophisticated enough to determine what pointers will be type casted; hence, a pointer cannot be properly associated with the classes of all the types it may be cast to. CCFI is able to explicitly verify and recompute the MAC with a new class each time a pointer is casted.

Second, our classification additionally incorporates runtime characteristics, notably the address at which a pointer resides, which prevents swapping two valid pointers of the same type. Categorizing by address protects return pointers, which have no type signature. Static approaches have no equivalent to runtime characteristics that they can use for return pointers. In particular, existing CFI systems lump all return pointers into a single class, meaning they are all interchangeable.

To argue security, we assume a very powerful adversary: that has arbitrary read access to all of memory and write access to all writable control-flow elements in memory (e.g., return addresses and function pointers). CCFI binds a pointer to a set of criteria such that a powerful adversary cannot hijack control flow and cause unintended code to run. An attacker who overwrites a control flow element will crash the program because of a MAC failure.

The use of MACs in CCFI gives us several useful advantages over traditional CFI approaches using static tables with runtime checks. First, attackers cannot generate MACs without the secret key, which is unlike static approaches where attackers can call any valid target. Second, we can group pointers based on runtime characteristics, such as binding a return pointer to the address it is stored at. Furthermore, these new approaches are compatible with static and dynamic libraries. Third, MACs are flexible enough to allow us to also group pointers based on compile time grouping, e.g., type information, or return address versus function pointers. Finally, AES-NI instructions on Intel x86 makes it possible to compute cryptographic MACs with low overhead. Our MAC function takes 92 cycles in the original implementation of AES-NI and has decreased in cost with newer processors.

Implementation

We implemented CCFI in LLVM [16] for x86-64 and recompiled SPEC2006, three web servers (Apache, Nginx, Lighttpd), two cache servers (memcached, redis) and all of their 21 dependencies. Only three packages (Apache's libapr, Nginx, and perlbench) required code changes for compatibility, where manual MAC computations had to be inserted, with at most a few lines of code changed. The implementation is generalizable to any architecture that supports a fast cryptographic MAC function. The web servers showed a reduced request throughput rate of only 3–18% when serving a static file. There is a 38% overhead when running with SSL, which contends for AES-NI register use. Results show that CCFI is practical and provides better security than existing approaches.

2. BACKGROUND

Software vulnerabilities take all shapes and forms. The classic example is a stack buffer overflow, where the lack of bounds checking lets an attacker corrupt a return address on the stack causing execution to jump to an arbitrary location. Another example is sending data beyond the end of a buffer, possibly leaking sensitive information. Yet another example is forgetting to include important authentication steps in the program's logic. We classify software vulnerabilities and attacks as follows:

1. *Control flow attacks* result in the attacker being able to execute arbitrary code. These are the most common exploits, and they typically yield a remote shell.

2. *Data flow attacks* result in the attacker being able to read or write program memory, not necessarily leading to arbitrary code execution. OpenSSL's heartbleed bug [17] is an example, where the attacker may be able to read the server's private key from memory.

3. *Logic errors* result in the attacker being able to skip checks. For example, Apple's goto fail [15] bug did not properly check SSL certificates allowing attackers to mount man-in-the-middle attacks.

Our work focuses on the first class of attacks only. It is, however, the most prevalent class of attack today and the most powerful, because it allows the attacker to achieve everything that other attack classes do. Running arbitrary code lets an attacker disclose memory to leak SSL keys or jump past any checks. Conversely, a data flow bug that merely discloses memory (though still catastrophic) cannot be alone used to execute a remote shell on the system.

Control Flow Integrity.

CFI [2] is a technique where static analysis determines where an indirect jump can land. Runtime checks are added to enforce that the jump lands only to the valid locations determined by static analysis. For example, suppose that the pointer analysis determines that a function pointer can only point to `read` or `write`. The program will detect attempts to call a different function, e.g., `execve`, and terminate just before calling the function. However, the attacker can still swap a `read` for a `write`, which may be enough to conduct an attack.

In practice, static analysis has limits; there are cases where it cannot determine all possible values of a function pointer. In this case, the set of valid locations can be any function whose address has

been taken. Worse, practical CFI implementations split valid locations into only two sets: function pointers can jump to any function whose address has been taken, and return instructions can return to any return site. These loose implementations are not enough as an attacker can swap return sites to eventually execute arbitrary code and break out of CFI [5, 9, 11].

Our approach is tackling CFI dynamically at runtime with the use of cryptographic primitives. Every control flow pointer has a cryptographic MAC associated with it that enforces several runtime and language properties. When first stored in memory a pointer is secured via a MAC, and validated before use. This approach does not require static analysis and so does not inherit any of its limitations. Furthermore, this approach allows us to express much richer CFI constructions where the classification of the pointer can change at runtime.

3. THREAT MODEL

Many security systems today (e.g., stack canaries, ASLR) assume the attacker cannot read memory. An attacker who can read arbitrary memory can easily defeat these defenses as demonstrated in several recent papers [3,14]. The BROP attack specifically shows that a buffer overflow can be used as an information leak bug to defeat these systems.

In this paper we assume a powerful attacker who has the ability to read arbitrary areas of memory and overwrite all writable control-flow elements in memory (e.g., return addresses and function pointers). However, the attacker is unable to write to executable memory (marked read-only) or read the value of special registers our compiler reserves. These are reasonable assumptions that accurately model modern control hijacking attacks.

CCFI is focused on protecting user-level programs such as web servers, and we do not address protecting the kernel in this paper. We assume that the kernel does not save any user-level registers—at least the ones that are used to store the key—during context switches in user accessible memory. This is true of all major modern operating systems that we are aware of. Custom user-level threading libraries may also require changes to ensure these registers are not saved.

4. DESIGN

Cryptographic CFI is a compiler enhancement that protects writable memory that affect the program's control flow. Specifically, it protects:

- Return addresses and frame pointers.
- Function pointers.
- vtable pointers. The vtable is a (read-only) function pointer table used by C++ to invoke virtual methods of a class.
- Exception handlers.

CCFI protection is achieved using a MAC. Each time a control-flow object is stored, its MAC is computed, and on each load, its MAC is verified. The MAC is stored alongside the object. Attackers cannot overwrite control objects because they do not posses the MAC key needed to produce a valid MAC for the object. The MAC key is randomly generated at program start, and stored in registers the CCFI compiler reserves (in x86-64 we use XMM5–XMM15).

Security relies on two assumptions: (1) Code never leaks the key into memory because the compiler enforces that no code ever touches the reserved registers. (2) Attackers cannot execute (misaligned) code that accesses these registers because they would have to break control flow in the first place.

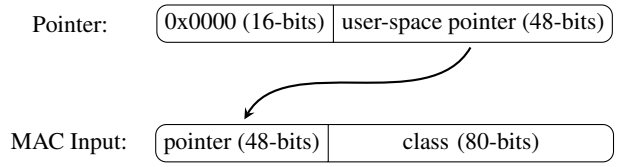

Figure 1: Shows the encoding of user-space pointers in the x86-64 architecture. Pointers are safely truncated to 48-bits and concatenated with the pointer class to form a 128-bit input to the MAC.

The remaining security concern is how attackers can reuse a pointer/MAC in a form of replay attack. For example, an attacker must not be able to swap two function pointers by reading a function pointer and its valid MAC to replace a different function pointer/MAC. To combat replay attacks we must define what is included in the MAC, which determines the security properties CCFI provides.

4.1 MAC Function

Our MAC is implemented as a single block of AES applied to the input data. More precisely, the concatenation of the control pointer and *pointer class* is encrypted with a secret key using AES-128. A *class* is a group of pointers invoked at one or more indirect branches. In the original CFI work, there were two classes: function pointers and return pointers. In our scheme this can easily be represented by a one or zero. Any modern CFI system that we know about can be implemented using MACs, but CCFI can extend classes to include runtime properties.

Pointers in the x86-64 architecture have 48 bits of significance, which is sign extended to 64 bits. Figure 1 shows an example of a user-space pointer sign extended with zeros. We can safely truncate pointers to 48 bits. As shown in the figure we concatenate a 48-bit `pointer` with its 80-bit `class` to form a 128-bit AES block, which is then encrypted using AES with a 128 bit key. Two bits of the `class` define the type of pointer being protected to ensure domain separation.

Our implementation defines the class as follows:

$$class := \begin{cases} \{0, \textit{hash of type, address}\} & \text{Function pointers} \\ \{1, \textit{frame address}\} & \text{Return addresses} \\ \{2, \textit{method ptr., address}\} & \text{Method pointers} \\ \{3, \textit{address}\} & \text{vtable pointers} \end{cases}$$

Function Pointers

Function pointers use the address of the `pointer`, 30-bit hash of type signature, and two bits for the class. The type signature's hash function is computed by the compiler and thus has no performance impact at runtime. It is deterministic so that pointers can be shared across dynamic libraries, and has few collisions to provide fine-grained classification. The type hash is based on LLVM's type system and is coarser than C/C++ types.

Return Pointers

Return pointers use the old frame pointer as the class. In each function prologue we MAC both the saved return and frame pointers, and in the epilogue we verify the MAC before returning. The MAC is stored in a stack slot reserved by the compiler. For functions that do not call any others, we have a *leaf optimization* that skips MAC computation and instead stores a copy of the return address in a register.

C++ Method Pointers

C++ method pointers consist of two 64-bit values. When the first value is even, the second is the address of a static method. Otherwise, the second value is an index into a vtable. The class is the concatenation of the first field and the address where the pointer is stored. The second field is used as the 48-bit `pointer` contained in the MAC.

C++ vtable Pointers

C++ vtable pointers, which point to read-only memory containing function pointers, are protected using the address they are stored at as the class. The vtables themselves need no protection as they are read-only. We similarly protect vtable tables (VTTs), which are used to find virtual base classes.

Rationale

Domain separation between all these pointer types prevents pointers from accidentally aliasing to each other in ways useful to attackers. For example attacks cannot use a function pointer as a return address and vice versa.

Including the pointer address in the MAC data ensures that an attacker cannot swap a pointer stored in one memory address with a pointer stored in a different memory address. However, the attacker can still replace a pointer stored at location x at time t with a pointer stored at the same location x at time $t' < t$. We refer to this as a *replay attack* and discuss defenses against it in Section 4.6.

Including the type signature ensures that pointers can only be swapped if they contain the same signature. In C++ we protect method pointers and vtable pointers, but often C programs place function pointers in structures. As structure pointers are not protected in our current implementation, attackers could use structure pointers to indirectly execute a given function pointer. The inclusion of type signatures in the class limits them to executing function pointers of the same type signature.

4.2 Runtime versus Static Classification

While any existing static CFI system can be implemented using MACs, runtime classifications offer several benefits. Current CFI systems must determine a fixed set of functions within a class and these are stored as read-only tables, otherwise an attacker could attack the tables used to verify control flow integrity. CCFI can be based on runtime properties. For example, we include the address where a pointer is stored in the MAC to prevent swapping function pointers. This is not possible to determine at compile time.

Another benefit is that even within a particular class, an attacker may only witness a subset of all possible values a function pointer can take. The unobserved pointer/MAC values are either never generated by a particular run of a program, or an attacker could not observe them. Even with full access to memory, the attacker cannot call the remaining functions because the attacker does not have valid MACs for them.

4.3 Architecture Compatibility

Our approach generalizes to any hardware supporting fast MACs. In the Intel x86-64 architecture we use the AES-NI instructions [12] to minimize performance impact of these computations. Hardware support for AES, SHA-1, and SHA-256, which enables efficient implementations of CCFI, is available on many architectures [23] including ARM 64-bit, SPARC, and PowerPC. On other architectures we may need to pack the inputs to the pointer class differently, but otherwise the design need not change.

4.4 Other Control Flow Protections

There are other less obvious control flow structures that must be protected for a complete solution. These are the the global offset table (GOT) and global destructors (`.dtors`). The GOT is used for dynamic linking and filled by the loader with the addresses of external library functions. Global destructors (like global constructors) are function pointers registered at program load time and executed at program termination.

To protect these we use an existing mechanism, RELRO [6], which computes relocations at program load time and marks the GOT and `.dtors` read-only. This prevents the attacker from tampering with these sensitive pointers.

4.5 Program Compatibility

We observed a few compatibility issues depending on the classification scheme. None of these compatibility issues were security holes, but they cause MAC failures that terminate the program.

Address based classification binds function pointers to their storage locations, which can interact poorly with `memcpy`. In several C programs, structures containing function pointers were converted to a `void*` and length, then copied with `memcpy`. In such cases, our system cannot determine that MACs need to be verified and recomputed without developer assistance, and hence the program crashes upon use of a copied pointer. We observed this in Apache and Nginx, both of which have a dynamic array implementation that treats structures containing function pointers as opaque objects (cast to void).

However, our system does properly handle `memcpy` of typed structure pointers, as we know for any given structure whether it contains function pointers that must be verified and reMACed. The problem with Apache and Nginx is that structure pointers are cast to `void*` before they are copied and subsequently used. A similar problem can arise with `realloc`.

C++ templates do not suffer from these problems as type information is available to our compiler pass. Language runtime pointers, such as return addresses and vtable pointers, do not have these issues as they are not exposed to the programmer.

Our type based classification may also encounter compatibility issues, but for several reasons this is not a common problem in practice. First, we can recompute the type based class when there is an explicit cast (not possible with static CFI approaches). Second, we use the type system understood by LLVM that is coarser grained than the C/C++ type system.

The C standard limits what function pointer casts are valid. Unfortunately in the newer standards it states that casting a function pointer to a `void*` then back to function pointer before use is valid. This obviously means we will not track the pointer when it is not a function pointer. Rather than generate a MAC when the pointer is cast back (a security problem), we do not support this behavior to avoid this issue.

We wrote a simple static analyzer to help programmers find cases where type information is lost for function pointers, in which case either a code modification or manual MAC checks may be necessary. Developers can use the `checkptr` and `macptr` primitives to manually fix particular instances. The Nginx compatibility issue was in fact found by our tool.

4.6 CCFI Limitations

In CCFI, we compute the MAC on a pointer and its class. The class acts as a (naïve) nonce. However, it is still possible to replace the current pointer at location x with an old pointer previously stored at location x and potentially disrupt control flow. This is a

form of replay attack, where the attacker reuses a signed message at a later date because it lacks a unique nonce.

Replay attacks are a concern for heap and stack addresses because the address does not have a fixed meaning for the duration of the program's execution. A program may reuse a stack address to store pointers that are never meant to be exchanged with one another, but when both pointer's classes are equivalent the attacker can replace the pointer/MAC. This problem does not exist with globals, because they exist at a unique address with a single meaning to the program.

One approach to limit these attacks is to add randomness to our nonce, which in this case is the address. Unlike ASLR that randomizes base addresses of binaries and does not rerandomize forking services, we will inject entropy at a finer granularity. For every stack frame we will randomly allocate stack space at runtime. For every heap allocation we will randomize the objects location. Together these techniques make it more difficult for the attacker to control when two function pointers align, and can be substituted for one another. This is a trade-off of memory overhead versus entropy injected. We inject 4-bits of entropy per allocation, the same value OpenBSD uses for heap randomization.

5. IMPLEMENTATION

Our system is a C/C++ compiler built on the Clang/LLVM compiler framework and supports x86_64, tested on FreeBSD. Any application wishing to be hardened with CCFI must be recompiled along with all of its dependencies. We provide a command-line compatible wrapper to clang/clang++ that adds our compiler passes and links the runtime libraries.

Internally, the implementation consists of the following major components:

- **LLVM Target:** ABI changes to reserve registers to ensure the compiler never leaks the key. Implements stack protection into the target specific code.
- **Compiler Intrinsics:** `macptr` and `checkptr` compiler intrinsics implemented as machine specific code and made available to the C language.
- **Pointer Protection:** High-level LLVM pass that identifies function pointers and vtable pointers and inserts the MAC computation and verification code. A runtime library provides error reporting and handles initializing global function pointers.
- **Stack randomization:** An LLVM function pass that randomly offsets the stack pointer on each call to reduce replay attacks.
- **Modifications to libc:** We modified `malloc` to randomize heap layout. The `sigaction` system call was patched to verify the pointer and MAC the pointer returned by the kernel.
- **Static analysis tool:** Finds any possible code that casts function pointers to/from non-function pointers, which will break CCFI.

5.1 ABI Changes

We implemented our MAC using the AES-NI instructions on x86. These instructions take their arguments in XMM registers. A 128-bit AES key expands to 11 128-bit values, requiring 11 XMM registers (each 128-bits wide) to hold the key.

Registers	SysV ABI	CCFI ABI
`xmm0-xmm3`	Arguments	Arguments
`xmm4`	Arguments	Temporary†
`xmm5-xmm7`	Arguments	Expanded Key
`xmm8-xmm15`	Temporary	Expanded Key
`xmm16-xmm31`	Temporary (Available in AVX-512)	

Table 1: Shows XMM register usage in the AMD64 SysV ABI and our modified ABI. The general purpose registers, float point stack and MMX registers remain unchanged in the new ABI. The changes only affect the XMM registers, which are used for floating point and vector operations. †The leaf function optimization uses XMM4 to store the return instruction pointer and frame pointer.

XMM registers are used for floating point and vector operations in the AMD64 ABI specification. We must therefore reserve 11 of these registers to hold our expanded key. An additional scratch XMM register is needed while computing the AES rounds. This register must not be used for argument passing or it would be clobbered during our AES computation during the function's prologue. It can, however, be used in the function's body as a temporary. Table 1 summarizes the ABI modifications we made for CCFI.

These changes limit the compiler's ability to keep more variables in the registers and thus can have a substantial impact on performance. Code that does not use floating point or vector math operations should not notice a performance impact from this change. Furthermore, the AVX-512 extensions to x86-64 architecture will double the number of XMM registers to 32. Some programs use the XMMs also for copying memory, zeroing memory, and similar operations. These tasks typically require only one or two XMM registers.

5.2 Memory Randomization

To prevent alignment of control pointers and MACs that make replay attacks possible, each memory and stack frame allocation is randomized. The basic idea is to add a random offset to each `malloc` and stack frame. There is a trade-off between how much virtual memory to waste and how much entropy to add. OpenBSD already implements this for `malloc` and we use their same entropy parameter of four bits.

We implemented randomized allocations in FreeBSD's libc `malloc` by adding a random offset to each allocated chunk. Unlike OpenBSD, we are not allowed to store a randomness source in memory because the attacker can modify this as per our threat model. We use Intel's random instruction [1] to generate the random offsets. The attacker would have to align memory layout between reading a useful pointer/MAC and replaying it elsewhere in the program. For, testing purposes on machines that lack this instruction we use the low order bits of the cycle counter (similar cost to RDRAND).

Stack randomization is implemented similarly. On each function prologue, we `alloca` a random size which has the effect of padding the stack frame by a random value.

5.3 Stack Protection

Our stack protection mechanism allocates a local variable to store the MAC of return address and frame pointer. The prologue of a function generates the MAC and stores it. The epilogue must recompute the MAC, and compare it, and crash the program if the MAC does not match. In the event of a bad MAC, it crashes the program by storing zeros in the return address and (if there is one) frame pointer, which saves a few instructions and avoids a branch.

Our leaf optimization skips stack pointer MAC creation and check-

ing for functions that do not call other functions. Instead, we store the instruction pointer and frame pointer in XMM4. The cost of this is minimal on the order of a few cycles. Since leaf functions do not make any calls we can safely rely on a register to store the value.

5.4 Compiler Intrinsics

In the LLVM backend we expose two intrinsics `checkptr` and `macptr` that compute or verify a MAC. While their implementation is machine specific depending on the availability of cryptographic instructions, these are portable and reusable intrinsics. Both programmers and our function pointer protection pass can make use of these intrinsics. These primitives depend on our ABI changes and never leak any part of the expanded key onto the stack.

Ideally, we would increase the size of function pointers to contain both MACs and addresses for pointers stored in memory. (A register can safely contain only a target address without a MAC.) Such wide pointers are conceptually simple, but complicate matters by changing the size of structures. While not ideal, preserving the size of function pointers significantly simplified incremental deployment and testing, for instance allowing us to test code without recompiling every single library.

Our implementation instead keeps function pointers unchanged and relies on a large, in-memory hash table to associate MACs with function pointers. The hash table causes additional performance overhead due to the computation of the hash function and additional TLB and cache misses. We also occasionally have hash collisions that require executing a slow path or resizing the table. The pointer value, pointer's address, type, and type signature are also stored in the hash table for debugging and used by `ccfi_memcpy` (described in Section 5.6).

5.5 Pointer Protection

Pointer protection is implemented as an LLVM module pass which does two things. First, for each basic block it finds loads and stores of function pointers, adding calls to `checkptr` and `macptr`. Care must be taken to recursively walk every structure, array and vector so that nested function pointers are found. When structures containing function pointers are copied using assignment or `memcpy` we must verify and recompute the MAC.

Second, the pass creates constructor functions that MAC all global function pointers on startup. This ensures that loading a global function pointer does not result in a MAC failure. Globally defined C++ classes do not require this, because the C++ constructor will be called, which computes the MAC.

Some system calls take or return function pointers. No special handling is needed when these pointers reside in registers, as the compiler already checks function pointers when they are loaded into registers. However, some system calls, such as `sigaction`, exchange structures containing function pointers. Instead of modifying the kernel, we modified libc to check pointers in argument structures and MAC those in return structures.

Runtime

The runtime mostly provides common functions to limit binary size bloat. We have a constructor function that is executed on program launch to allocate a memory region for MAC storage (our hash table). A global MAC helper function reduces the instantiations of the `macptr` intrinsics inside constructors. Lastly, our MAC failure reporting function helps with debugging and identifying whether it might be a program issue (e.g., missing MAC on untyped function pointer copy) or attack.

5.6 Static Analysis Tool

We wrote a static analysis pass for Clang's static analyzer to help developers find any code which may circumvent the automatic MACing of function pointers and therefore cause bogus MAC failures. It detects and flags the following cases:

- A `memcpy` where both supplied arguments (before casting) are of type `void*`. These could be pointers to data structures containing function pointers cast to `void*` in another object file. In our test applications, so far we found this to be the only case where we miss function pointer copies.
- Any place where a function pointer is cast to a non-function pointer type e.g., unsigned long, or `void*`.
- Any place where a non-function pointer type is cast back to a function pointer.

We also provide users with a utility function, `ccfi_memcpy`, for debugging MAC failures due to `memcpy`ing untyped function pointers. This function iterates through the hash table to identify existing pointers, and will verify and recompute the MAC for any pointers in the region being copied. We used this in Nginx and Apache's libapr for example, where function pointers were being `memcpy`ied without type information and the MAC had to be recomputed. The `ccfi_memcpy` function is intended only for development and testing, because it may allow an attacker to move a pointer and MAC between two addresses depending on how it is used.

6. SECURITY DISCUSSION

Our security discussion begins with an analysis of a running Lighttpd instance. Then we will focus on three security concerns we have and possible future improvements.

6.1 Analysis of Lighttpd Run

As an example, we will consider a run of Lighttpd compiled with CCFI. We found that the program used 47 unique function pointers. CCFI uniquely classified all of them except for function pointers intentionally interchanged by the programmer. By contrast, the state-of-the-art compile-time CFI separates these into only 8 classes [21]. CCFI also uniquely classified half of all return addresses using the address method; the remaining ones were aliased into small sets. By contrast, existing systems classify all return pointers identically, a coarse granularity that has allowed attacks blocked by CCFI.

6.2 Address Aliasing/Replay Attacks

The weakness of address-based classification is that it allows old pointers to be replayed at the same address. For example, imagine that a sensitive pointer is stored on the stack and the attacker uses an information leak to observe both the pointer value and its MAC. At a later time, the same stack location contains a different pointer. The attacker can successfully substitute the old pointer for the new one by overwriting both the pointer and the MAC. This attack is not detected because of address aliasing.

Several design choices address this issue. First, domain separation between our different pointer categories—e.g. function pointer vs. return address vs. vtable pointer—restricts aliasing to pointers of the same category. Second, function pointer type signatures prevent these sorts of attacks between function pointers with different type signatures. Lastly, heap and stack randomization tries to keep these alignments outside of the attacker's control.

A future improvement is to use a slab allocation scheme, rather than libc's heap, to isolate fixed-length structures of known types.

This will prevent aliasing of function pointers contained with different structures. Similar benefits can be provided for sensitive pointers stored on the stack through segmented stacks.

6.3 Indirectly Referenced Pointers

C programmers implement object oriented code by placing function pointers inside a structure. While this is a good design practice, unlike C++ where vtable pointers are easily identifiable, our CCFI pass cannot always identify sensitive structure pointers. An attacker may try to control a pointer to a structure containing function pointers, thus defeating address-based classification.

Our main defense against this attack in the current design is type-based classification of function pointers. The attacker can only call a function with a compatible type signature.

A future improvement is to protect all pointers using CCFI. Analysis could also be used to identify all *sensitive* structure pointers that indirectly point to control flow pointers, and only protect this subset of structure pointers. This approach has been implemented by CPI [13].

6.4 Data Flow Attacks

Assuming an ideal CFI implementation, program data can still modify control flow, such as an index to a switch statement. These attacks are outside the scope of CFI. CCFI offers an interesting option where future compiler passes could reuse the `macptr` and `checkptr` intrinsics to protect indices into function tables and critical conditional values.

Such a compiler pass could protect sensitive variables that are annotated as such by developers. For example, to protect a global, which specifies whether or not a connection has been authenticated, the pass will call `macptr` when setting the value. When reading the value it uses `checkptr` to verify it. Attackers can only set the variable to values that have been previously observed. In this example, attackers cannot overwrite the variable to skip authentication unless they have previously authenticated themselves. The pass should also ensure domain separation with CCFI's values.

7. EVALUATION

We evaluate two aspects:

1. Do applications break? When copying function pointers, they must be reMACed. This will not occur automatically if a function pointer is cast to a non-function pointer type.

2. What is the overhead of the MAC computations and checks?

All performance benchmarks were conducted on a computer running FreeBSD 10.1 powered by dual Intel Xeon E5620 processors running at 2.4 GHz with four cores each. The machine had 48 GBs of RAM and an Intel SSD. An identical machine running Ubuntu Linux was connected via gigabit Ethernet to launch the network benchmarks.

7.1 Application Compatibility

We compiled 21 libraries, 5 servers, and SPEC CINT2006 using CCFI. Out of these, we only had to modify two lines in libapr, a single line in nginx, and a few lines in perlbench (part of CINT2006), all of which copied function pointers with `memcpy`, breaking our MAC. In all cases, the programs crashed upon initialization due to a null MAC.

We ran our static analysis on nginx and it pointed out three possibly incompatible calls to `memcpy`. Two were in a variable sized array implementation which would `memcpy` its elements to a new buffer when resizing. The third was in a resolver code. Sixteen calls to function pointers being cast to void types were spotted. All of these were calls to push function pointers into the array implementation containing the `memcpy`. This information directly pointed us to the problematic `memcpy`. Interestingly, libapr had the same exact problem. A custom array implementation was used to store function pointers in non-function pointer typed memory.

OpenSSL makes extensive use of function pointers and we were able to run it unmodified. We disabled the hand written optimized assembly that used our reserved XMM registers. We could have modified the assembly code to either not use the reserved registers, or save and restore our key in the top half of the YMM registers. Both solutions require a lot of engineering effort.

Operation	Baseline	Ptr Prot.	CCFI
Func. call	7	-	70
Fptr. call	7	50	153
Mthd. call	8	53	156
Vptr. call	17	60	164

Table 2: Shows the round-trip function call and return for a noop function in cycles. The baseline numbers include no protection using an unmodified compiler. CCFI without stack protection shows the overhead when only function pointer protection is enabled. The CCFI column shows the results with stack and function pointer protection enabled.

7.2 Microbenchmarks

Our system proposes to compute AES on every call, return and indirect branch. This seems like a high price to pay but the key to making this practical is the low latency offered by the AES-NI instructions.

The intrinsic functions that perform MAC computation and verification have a latency of approximately 92 cycles. Averaged over many iterations both operations cost approximately 40 cycles because of pipelining. Our experiments used an Intel Westmere processor, the first microarchitecture with AES-NI support, on which the latency is 8 cycles per round. Latency decreases to 7 cycles on Haswell and is expected to improve further in the Skylake microarchitecture.

Table 2 examines how the MAC computation time affects function call and return times in cycles. This is our worst case performance because the function does not do any work. This is measured in a loop to represent the cycles added to a program execution rather than end-to-end latency.

Stack protection adds approximately 63 cycles to the function. This value is less than the MAC computation time because processor pipelining allows both MAC creation and verification to occur simultaneously until the epilogue must compare the results. Any function performing a significant amount of computation will mask our fixed overhead of 70 cycles.

The function pointer call latency is listed in the second row. We see that function pointer protection costs an additional 43 cycles. With both pointer and stack protection enabled, we measured 153 cycles.

Finally, two C++ call benchmarks: a non-virtual method pointer call and a virtual method pointer call are shown. Calling method pointers is more expensive in C++ as the compiler lowers the call into a conditional that either calls the vtable entry if it is virtual otherwise calls the pointer directly. Virtual calls are the most expensive because of the extra vtable access.

CCFI adds a fixed overhead ranging from 70–164 cycles to function calls. Any function doing significant work will amortize this

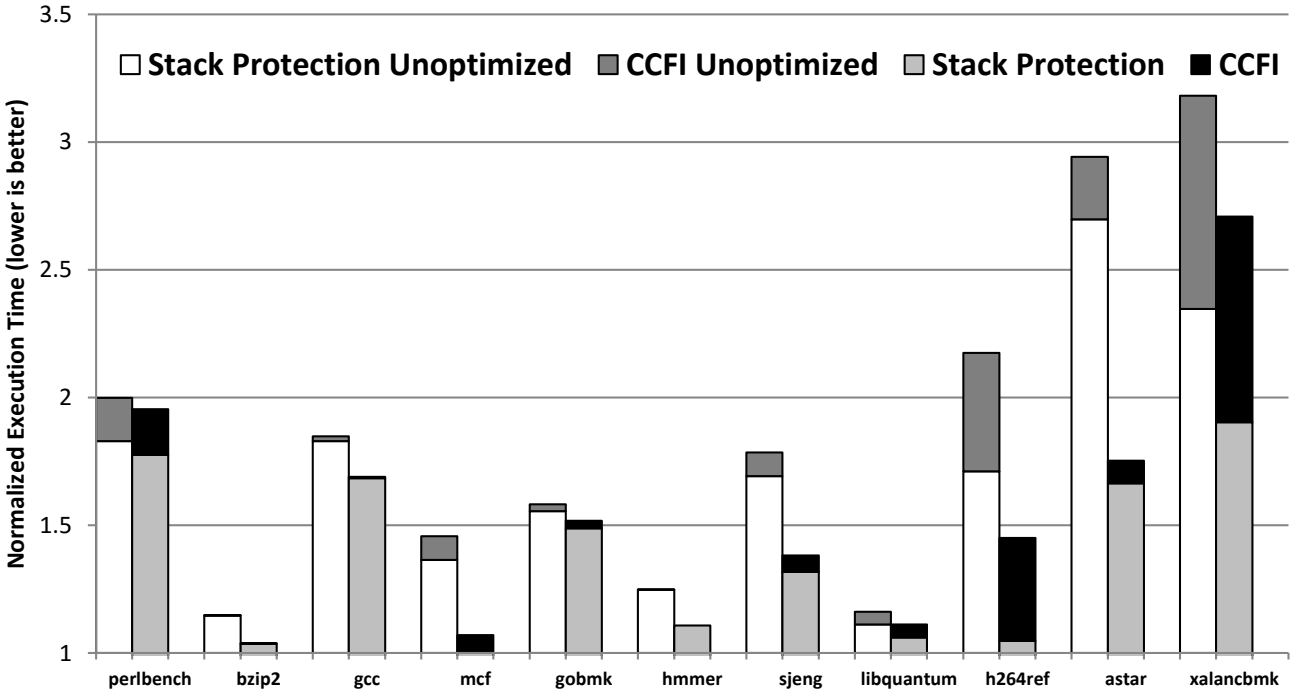

Figure 2: Shows the SPEC2006 results. The left bar is unoptimized CCFI and the right bar is optimized. Results are normalized to baseline execution (of 1x). The bars are stacked to show the overhead breakdown between stack protection and function pointer protection.

fixed latency. As a reference point, a single cache miss is 300 cycles on modern machines. Larger functions enable the processor to take advantage of instruction reordering and speculative execution to hide some of this latency. The instruction reordering optimization explains the non-linearity visible in this table. We evaluate application benchmarks next to measure the overall effect.

7.3 SPEC2006 Benchmarks

Figure 2 shows the normalized execution time relative to a run with an unmodified Clang 3.3 compiler for the SPEC CPU2006 integer benchmarks. All benchmarks worked both with Clang and CCFI, with no changes to the benchmark source code. We also measured the overhead of the ABI changes alone to measure the impact of register pressure, but the results were negligible so we did not plot it. All the overhead comes from stack and function pointer protection.

We show the results of SPEC for full protection with and without the leaf optimization for stack frames. The stack protection overhead appears as the lower half of the bar in each of the two cases. We measured an average of 52% overhead for all benchmarks, and 23% overhead for the benchmarks written in C.

Function pointer protection overhead becomes more apparent in the C++ benchmarks that we have measured. This is because inheritance depends on vtable pointers that must be protected. The C code has few hot paths containing function pointers thus we see a smaller performance difference between stack protection alone and full protection.

The omnetpp benchmark broke when porting our system from Clang/LLVM 3.2 to 3.3 and FreeBSD 10.0 to 10.1. It originally did not require any modification, but in our current build environment it crashes even without CCFI. In the original implementation, omnetpp had a 5x overhead that was reduced to 3.5x with the leaf optimization enabled.

7.4 Stack Leaf Optimization Gains

Our stack protection cost dominates in small functions. The effect worsens when such functions are called frequently. To better understand this behavior, we examined our worst and best cases from SPEC (omnetpp and bzip). Figures 3 and 4 show the approximate total cost per function (instruction_count $\times$ number_of_calls) for omnetpp and bzip2, when using different compilers. Each curve is sorted by function cost. The gap between the top curve (stack protection) and the bottom curve (vanilla compiler) shows the overhead of stack protection. The middle curve approximates the cost graph with the leaf optimization.

In the omnetpp case, there are many frequent calls to smaller functions (typical in C++) leading to higher overhead. This is indicated by the middle curve that hugs the unoptimized curve on the left side of the graph (costly functions). On the remainder of the graph however the optimization pays back as it sits between the baseline and unoptimized curve. Our leaf optimization reduces a 5x overhead to 3.5x.

C code represented by bzip2 calls larger functions, which is indicated in the graph by all three lines nearly overlapping. The optimization has a smaller impact on overall performance as stack protection contributes to a smaller percentage of a functions execution time.

7.5 Applications

We compiled a number of high performance servers and their dependencies with CCFI. Table 3 shows the request rate when comparing a vanilla build of the system compared to CCFI. We used default settings for all servers and the ApacheBench benchmarking tool. In the HTTP case, there is a 3–18% overhead depending on the server used.

In the HTTPS case, performance drops by 38% for two reasons.

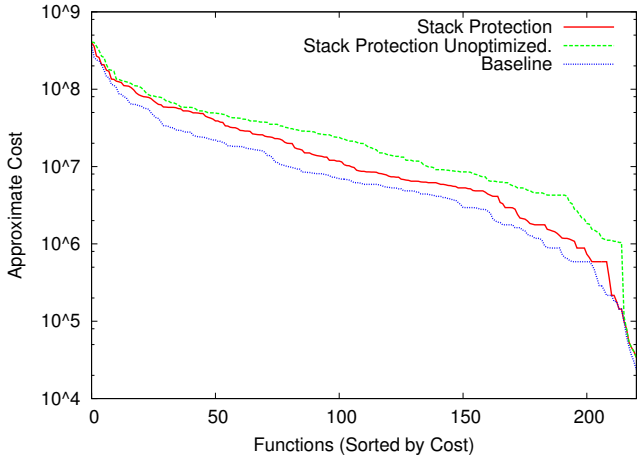

Figure 3: Shows the approximate performance cost of the top 220 functions for omnetpp, a C++ benchmark, with vanilla Clang, and stack protection with and without the leaf optimization. We have many very high frequency smaller sized functions that appear in the graph as the large gap between the vanilla and unoptimized lines. Our optimization reduces almost half the cost as shown by optimized line which is bounded by the other two.

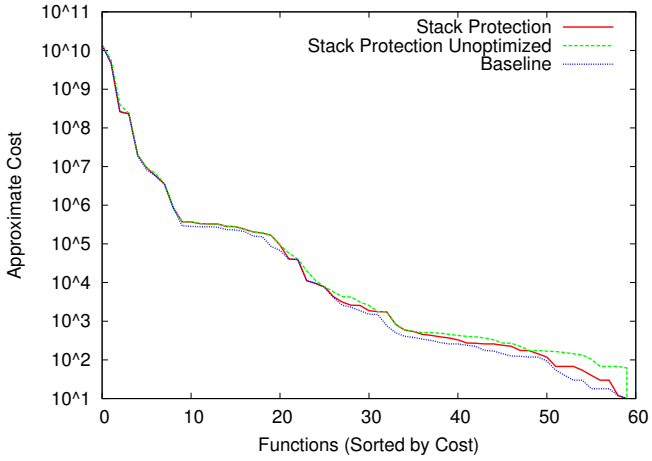

Figure 4: Shows the approximate performance cost of the top 60 functions for bzip2, a C benchmark, with vanilla Clang, and stack protection with and without the leaf optimization. The lines almost completely overlap except for the lowest cost functions (far right). The low cost functions are executed few times and are not very long thus showing more dramatically the cost of stack protection.

First, we disabled the optimized assembly code in OpenSSL which used XMM registers 5–15. Second, all the intensive vector code felt the XMM register pressure. Although we disabled OpenSSL's AES-NI implementation, we used FreeBSD's cryptodev kernel AES-NI implementation for high speed AES. Using FreeBSD's cryptodev device adds well over 1000 cycles of overhead (system call cost) that is amortized for large messages (anything over 128 bytes will break even).

We did not pursue the task of modifying the large hand written assembly code base of OpenSSL. For evaluation purposes however, we did evaluate two integration strategies that can be used on AES-NI code. We did this as a proof of concept, outside of OpenSSL, using FreeBSD's C implementation of AES-NI that is compatible with our CCFI compiler without modification. The first integration strategy is to avoid using the reserved registers, which is automatically done when compiling the C implementation with our compiler. The other one is to save and restore the key schedule into the top half of the YMM registers that are not used by OpenSSL's hand written assembly. This can reduce the cost to about 8 cycles for saving and restoring CCFI's keys.

Table 4 shows the performance cost of encrypting a 4 KiB block of plaintext using AES-NI in CBC or XTS modes. The CCFI unoptimized and optimized columns show cost of using AES-NI while avoiding the reserved registers, which we expect to decrease when Intel doubles the number of total XMM registers with the AVX-512 extension. The CCFI swap column demonstrates that by saving and restoring the key into the top half of the YMMs we can encrypt at virtually zero overhead. This is a microbenchmark so it does not include the costs associated with function pointer protection that exists in OpenSSL, and not in FreeBSD's code. Most of those costs are small constant overheads per call.

We measured the performance of two additional servers, memcached and redis, shown in Table 5. We used the mutilate tool to benchmark memcached, and redis's own benchmark tool for redis. The performance degradation is between 3–18%.

These results are promising for securing network servers where most of the overhead comes from IO or complex application code.

Configuration	Baseline	CCFI
Nginx (https)	207	128
Nginx	16482	14103
Lighttpd	22714	18516
Apache	25305	24537

Table 3: Webserver request throughput.

Mode	Baseline	CCFI unopt.	CCFI opt.	CCFI swap
CBC	17968	26709	18038	18030
XTS	27673	59810	30294	27729

Table 4: AES-128 CBC and XTS Encryption time in cycles for a 4 KiB block microbenchmark based on FreeBSD's AES-NI code that is written in C. This is intended to demonstrate the performance costs associated with using AES-NI inside user-level once we patch OpenSSL to avoid the reserved XMM registers. The CCFI swap column using the YMM register swapping trick to save and restore the key securely.

Configuration	Baseline	CCFI
memcached	283403	276006
redis	107527	88496

Table 5: Cache server request throughput.

8. RELATED WORK

Modern operating systems in conjunction with compilers implement several security features. Address space layout randomization (ASLR) [20] randomizes the in memory location of code to make attacks against known binaries difficult. In addition, most compilers including support for stack cookies that attempt to detect stack smashing attacks [8]. These systems both require recompilation of software and have been circumvented by attackers for years in 32-bit systems. The BROP attack [3] showed that a generalized attack was practical even on 64-bit systems without knowledge of the binary. While these solutions raise the attack's complexity, they offer no principled security.

After the initial CFI implementation [2] was introduced by Abadi et al. there are now many CFI systems built on static analysis techniques to achieve security. All these systems classify pointers into several categories such as call-sites and function pointers. Arguably the most secure of these is CCFIR [25] that only classifies pointers into three categories. This along with the difficulty of achieving compatibility within the limits of static analysis has lead to practical attacks on all known CFI systems [9, 11]. Cryptographic CFI offers the first new approach to CFI since the original paper, because all previous approaches depend on generating static tables at compile and/or load time. CCFI is the only approach that can securely build tables based on runtime parameters such as addresses where pointers are stored and function type information (casting makes this incompatible with static approaches). Unlike existing CFI systems, CCFI requires binaries and libraries to be recompiled, as existing libraries may leak our key or destroy it.

Forward Edge CFI [21] classifies pointers by the number of arguments for C code (Arity) or into a single class (Single). Not only is Arity mode insufficient, it can be incompatible with runtime casting of functions into function pointers with fewer arguments. The FECFI paper states while they support using function type signature based classification they found real code was not compatible. The main reason for this is C/C++ programmers often cast function pointers to tweak types or eliminate unused arguments. Unfortunately FECFI offers no protection for return pointers and falls back to existing mechanisms that are known to be weak. As with previous CFI systems, FECFI does not have the runtime benefits that CCFI does.

Another very related work is PointGuard [7]. The PointGuard system exclusive-or's all function pointers with a random value chosen at startup. In a way this can be thought of like pointer encryption except it assumes that attackers will only read or modify a single pointer. Once an attacker has read several pointers the secret exclusive-or value can be computed. Cryptographically secure encryption (or MAC'ing) by itself provides little security as functions can be swapped. CCFI's improvement over PointGuard is realizing the connection between inputs to a MAC and CFI. Another problem is that modifying pointers in-place meant that a lot more program/library changes are required. Pointers had to be manually decrypted/encrypted when issuing system calls.

Several systems use memory protection hardware to protect the return stack such as shadow stack. The StackGhost system relied on register windows and OS support on the SPARC architecture to provide stack smashing protection [10]. StackShield implemented a shadow stack using the data segment so that it would not be susceptible to stack smashing attacks [22]. These systems do not protect local function pointers stored on the stack. Some shadow stack implementations on x86 use segmentation to isolate the shadow stack, such that an attacker could not overwrite it without the use of a special instruction prefix. This CPU feature is not supported by any popular architecture today including x86-64 and thus an attacker with a stronger threat model could attack the shadow stack.

CPI [13] is a system that provides strong protection through the use of segmentation. On x86-32 they use segmentation to prevent normal code from reading or writing to a special segment that stores sensitive pointers. To support x86-64 they use segment base addresses that still remain from x86-32, and rely on address randomization to hide the location. If an attacker determines the base address of the region then all sensitive pointers are vulnerable. This is because x86-64 does not offer segmentation protection. Without segmentation we cannot see how any security guarantees can be made with the CPI approach.

kBouncer uses hardware performance counters on x86 to record the last 16 return addresses, and has the operating system verify the stack within the system call handler. Attacks on kBouncer show one can prevent detection of ROP attacks by producing 16 valid calls before executing a system call [5].

9. CONCLUSION

We showed that cryptographic control flow integrity is a viable approach to protecting program control flow on modern processors. We provide the finest-grained classification of existing CFI systems. Our system ensures that an attacker who has arbitrary read/write access to memory cannot arbitrarily modify control flow data, such as return addresses and function pointers, without being detected. While attackers can cause the program to crash, they cannot alter control flow to execute code of their choice.

CCFI can classify pointers based on dynamic runtime characteristics such as address and type, both of which are not possible with previous approaches. The approach is general enough to support any compile time classification as well. Ensuring that CCFI can always provide finer-grained classification than any static approach to CFI.

We experimented with our CCFI system on a number of large software packages. In all cases the packages compiled with no problems after changing at most a few lines of code in each package.

Clearly a cryptographic system that provides strong control flow protection must incur some performance cost. Through optimization and by using hardware AES available in modern processors we achieved between 3–18% slowdown over the unprotected system. In many environments this is a worthwhile trade off given the strong protection it provides. This work shows how to protect control flow structures, but does not protect other data in memory. By implementing `checkptr`/`macptr` in hardware using asynchronous exceptions a processor vendor could help nearly hide all of this cost.

Source code is available at `http://ccfi.scs.stanford.edu`.

Acknowledgements

We would like to thank the CCS program committee. We thank Tal Garfinkel, Amit Levy, and Edward Yang for their helpful comments. This work was funded by the NSF, DARPA CRASH, DARPA PROCEED, and a grant from ONR.

10. REFERENCES

[1] Intel Digital Random Generator (DRNG), August 2012. http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, Nov. 2009.

[3] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. BROP: Blind Return Oriented Programming. In *Proc of IEEE Security Privacy 2014*, Jun 2014.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of CCS 2008*, pages 27–38. ACM Press, Oct. 2008.

[5] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, 2014. USENIX Association.

[6] K. Cook. hardening-check - check binaries for security hardening features. http://manpages.ubuntu.com/manpages/lucid/man1/hardening-check.1.html.

[7] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proc. of the 12th Usenix Security Symposium*, 2003.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of Usenix Security*, 1998.

[9] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium*, San Diego, CA, 2014. USENIX Association.

[10] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *In Proceedings of the 10th USENIX Security Symposium*, pages 55–66, 2001.

[11] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proc of IEEE Security Privacy 2014*, Jun 2014.

[12] S. Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. Number 323641-001. Intel Corporation, May 2010.

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[14] M. Labes. MWR Labs Pwn2Own 2013 Write-up - Webkit Exploit. https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/.

[15] A. Langley. Apple's SSL/TLS bug, 2014. www.imperialviolet.org/2014/02/22/applebug.html.

[16] C. Latner. The LLVM Compiler Infrastructure. http://llvm.org/.

[17] N. Mehta. The Heartbleed Bug, 2014. heartbleed.com.

[18] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proc. of the 11'th ACM conference on Computer and Communications Security (CCS)*, pages 298–307, 2004.

[19] A. Sotirov. Heap Feng Shui in Javascript. Blackhat Europe 2007, 2007.

[20] P. Team. PaX address space layout randomization (ASLR), 2014. http://pax.grsecurity.net/docs/aslr.txt.

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium*. USENIX Association, 2014.

[22] Vendicator. StackShield. http://www.angelfire.com/sk/stackshield/.

[23] Wikipedia Foundation, Inc. AES instruction set. http://en.wikipedia.org/wiki/AES_instruction_set, Apr. 2014.

[24] Wikipedia Foundation, Inc. NX bit. http://en.wikipedia.org/wiki/NX_bit, Apr. 2014.

[25] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 559–573, 2013.