HerQules: Securing Programs via Hardware-Enforced Message Queues

Daming D. Chen
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
ddchen@cmu.edu

Wen Shih Lim
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
wmlim@alumni.cmu.edu

Mohammad Bakhshalipour
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
bakhshalipour@cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
gibbons@cs.cmu.edu

James C. Hoe
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
jhoe@cmu.edu

Bryan Parno
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
parno@cmu.edu

ABSTRACT

Many computer programs directly manipulate memory using *unsafe* pointers, which may introduce *memory safety* bugs. In response, past work has developed various runtime defenses, including memory safety checks, as well as mitigations like no-execute memory, shadow stacks, and *control-flow integrity* (CFI), which aim to prevent attackers from obtaining program control. However, software-based designs often need to update in-process runtime metadata to maximize accuracy, which is difficult to do precisely, efficiently, and securely. Hardware-based fine-grained instruction monitoring avoids this problem by maintaining metadata in special-purpose hardware, but suffers from high design complexity and requires significant microarchitectural changes.

In this paper, we present an alternative solution by adding a fast hardware-based append-only inter-process communication (IPC) primitive, named **AppendWrite**, which enables a *monitored program* to transmit a log of execution events to a *verifier* running in a different process, relying on inter-process memory protections for isolation. We show how AppendWrite can be implemented using an FPGA or in hardware at very low cost. Using this primitive, we design **HerQules** (HQ), a framework for automatically enforcing integrity-based execution policies through compiler instrumentation. HerQules reduces overhead on the critical path by decoupling program execution from policy checking via concurrency, without affecting security. We perform a case study on control-flow integrity against multiple benchmark suites, and demonstrate that HQ-CFI achieves a significant improvement in correctness, effectiveness, and performance compared to prior work.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Computer systems organization** → **Processors and memory architectures**; • **Software and its engineering** → **Operating systems**; **Message passing**.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04.
https://doi.org/10.1145/3445814.3446736

KEYWORDS

control-flow integrity, pointer integrity, memory safety, inter-process communication, compiler, kernel, shared memory, FPGA, microarchitecture

ACM Reference Format:
Daming D. Chen, Wen Shih Lim, Mohammad Bakhshalipour, Phillip B. Gibbons, James C. Hoe, and Bryan Parno. 2021. HerQules: Securing Programs via Hardware-Enforced Message Queues. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446736

1 INTRODUCTION

Many computer programs directly access memory using *unsafe* pointers, which may unintentionally dereference memory that is out-of-bounds or that has been deallocated. For example, a *buffer overflow* occurs when memory outside a referenced buffer is accessed, whereas a *use-after-free* error occurs when deallocated memory is accessed. Attackers can leverage these *memory safety* bugs to corrupt program execution, by intentionally overwriting memory that contains function pointers or program data. In response, runtime defenses have been developed to detect these bugs or mitigate corruption; e.g., by tracking allocation boundaries [10, 38, 60, 76], temporal identifiers [77], tainted values [111], or control-flow pointers [8]. However, these defenses need to maintain *runtime metadata* about memory, which must be protected from unintended access.

Past work has explored various approaches for partitioning program subcomponents, which can be used to isolate these metadata. However, software-based mechanisms impose significant overhead, reduce compatibility, or rely on hiding information. *Disjoint address spaces* [39, 70, 86] reconfigures the memory management unit (MMU) to isolate physical memory, but it adds overhead by flushing the translation lookaside buffer (TLB) on each context switch. *Software fault isolation* [109] (SFI) must mask all pointers to be effective, which requires recompilation of existing shared libraries. *Information hiding* has low overhead, but relies on randomization of program code or layout, which is vulnerable to disclosure attacks [92, 95, 97].

Hardware-based *fine-grained instruction monitoring* [9, 28, 35, 44]) takes a different approach by modifying the processor to generate, filter, and process execution events (e.g., retired instructions, function calls, memory accesses, etc.) in isolated hardware. Research

Table 1: Comparison of HerQules against hardware-based fine-grained instruction monitoring designs.

Design	Events	Recipient	Paradigm	HW Δ
FADE [44]	HW/SW	Core	Filter-Update	Big
FlexCore [35]	HW/SW	FPGA	Reconfigure	Big
Guardian Council [9]	HW/SW	μCores	Filter-Map-Reduce	Big
LBA [28]	HW/SW	Core	Filter-Update	Big
Processor Trace [6]	HW/SW	Memory	Filter-Update	–
HerQules	SW	Core	Message Passing	Small

proposals include adding a similarly-sized core [28, 44], an on-chip FPGA [35], or multiple microcontroller-sized cores (μCores) [9], as shown in Table 1. However, these designs require significant microarchitectural changes, and generate fixed hardware-defined events that may not always be useful, but nevertheless incur both energy and logic costs. For example, under FADE [44], 84–99% of all events must ultimately be discarded as irrelevant. Guardian Council [9] suffers from similar load balancing challenges, as anywhere between 2–24 μCores are needed to reduce overhead below 5%, depending on the security policy being enforced.

Recent commodity Intel processors include built-in support for Processor Trace [6] (PT), which stores hardware performance monitoring events in memory, and can support software-defined events via a `PTWRITE` instruction. However, although it has been used for instruction monitoring by past security policy enforcement designs [37, 45, 53, 71], it is not designed as a security mechanism, and thus suffers from numerous flaws. Hardware-defined events do not provide sufficient execution context, which forces these designs to reconstruct control-flow by disassembling program binaries, which adds complexity and overhead. Event packets can be lost or overwritten due to, e.g., performance monitoring interrupt skid [6], which defeats security. Tracing and decoding of events incur tremendous overhead, measuring over 500x [71] on the SPEC benchmarks. As a workaround, these designs must limit which events are monitored (e.g., monitoring only 7-10 system calls—`execve`, `mmap`, etc.), which greatly reduces effectiveness (especially since these system calls are rarely used in compute-intensive programs like SPEC benchmarks).

In this paper, we observe that prior software-based approaches cannot provide efficient isolation, whereas prior hardware-based approaches suffer from excess design complexity. Instead, we propose to augment existing hardware with a simple and fast *AppendWrite* inter-process communication (IPC) that adds *authentication* and message *integrity*, which protects metadata via existing *inter-process isolation*. We provide two implementations of AppendWrite: one using a self-contained FPGA-based message queue, and another that adds *append-only* microarchitectural memory buffers using only one ISA instruction, two registers per core, and some simple logic. Using this primitive, we design HerQules (HQ), a framework for efficiently enforcing integrity-based execution policies, which delivers log messages from a *monitored program* to a policy-enforcement *verifier program* running in a different process. For efficiency, both programs execute concurrently, and synchronize at the monitored program's system calls to prevent undesired behavior.

We describe how our system can support different execution policies, and perform a security case study on *control-flow integrity*

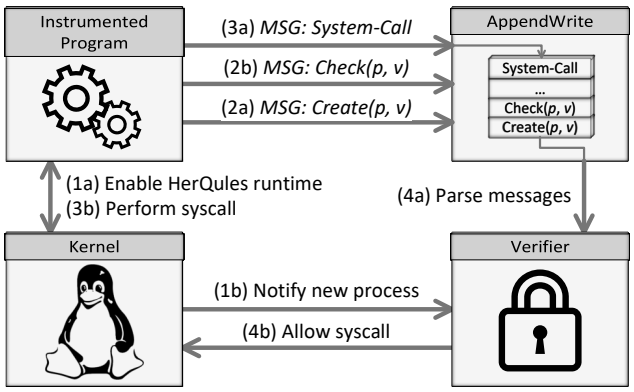

Figure 1: Overview of interactions under HerQules.

(CFI), which protects a program's execution integrity by verifying control-flow transitions at runtime. Specifically, we develop two new fine-grained *pointer integrity* designs, which are maximally precise, do not suffer from the undecidability of pointer aliasing [41, 88], and incorporate novel compiler optimizations. Compared to past work [30, 62, 63, 74], our designs maintain program correctness, preserve library compatibility, and add detection of use-after-free errors on control-flow pointers, while minimizing overhead, as shown in Table 3. We evaluate the correctness, effectiveness, and performance of our designs on the SPEC CPU2006 [55], SPEC CPU2017 [22], RIPE [90, 110], and NGINX [98] benchmarks. Our results demonstrate the successful execution of all benchmarks, and the discovery of new use-after-free bugs in the SPEC benchmarks. Our fastest design prevents all but one type of exploit with a geometric mean overhead of 14.4%, whereas our comprehensive design prevents all exploits at the cost of increased overhead, which improves over past work.

We summarize our contributions as follows:

- We observe that software-based program partitioning lacks efficient *isolation*, whereas hardware-based instruction monitoring is overly complex, and instead propose a simple new AppendWrite IPC primitive (§2.3), which we implement in both an FPGA and in hardware at very low cost (§3.1).
- We use AppendWrite to build HerQules, a framework for implementing efficient integrity-based execution policies, which executes asynchronously to maximize performance (§2.2).
- We show how HerQules supports a variety of security policies, and perform an experimental case study on control-flow integrity (§4.1), demonstrating a significant improvement in correctness, effectiveness, and performance over prior work (§5).
- We release our system as open-source at https://github.com/secure-foundations/herqules.

2 HERQULES DESIGN OVERVIEW

As a toy example, to put HerQules in context, suppose that a program wants to reliably count the number of function calls that it has made. One approach would have the program allocate an in-process global counter and increment it before every call instruction, but this counter could be corrupted by program bugs. Instead, under

our design, the compiler automatically instruments the program to send policy-relevant messages (e.g., counter increment on each function call) to a verifier running in a different process, using our AppendWrite IPC primitive. This relies on existing inter-process isolation to prevent bugs in the program from directly affecting policy-relevant state. Even if the program is corrupted immediately after sending a message, it cannot retract previously-sent messages.

Because the monitored program and the verifier execute concurrently, it takes some time for messages to be sent, received, and processed, which can impact the accuracy of our execution counter, especially in the presence of high message traffic. We can improve the accuracy of this counter by *bounding* asynchrony at system calls (§2.2), where we pause the program until the verifier has processed all in-flight messages. To determine when a paused program should resume, the kernel and verifier communicate over a separate privileged channel that is not accessible to the monitored program.

In this section, we discuss the design of AppendWrite and HerQules, deferring implementation to §3 and security policies to §4. Figure 1 highlights the four main components of HerQules, providing an overview of the interactions between the components. At compile-time, our **compiler pass** automatically instruments the program to send policy-dependent messages when policy-relevant events occur. At run-time, the monitored program enables HerQules *(1a)* during startup, causing the kernel to register it with the **verifier** *(1b)*. Subsequently, the monitored program can send messages to the verifier via AppendWrite *(2a, 2b)*. At some point, the monitored program sends a system-call message *(3a)* and performs a system call *(3b)*, where it is initially paused by our **kernel module**, until the verifier confirms no policy checks have failed *(4a, 4b)*.

2.1 Threat Model

HerQules enforces runtime execution policies that rely on *software-visible* execution events. Thus, we do not enforce policies based on microarchitectural events such as in side-channel attacks. We assume that programs begin execution in a benign state, but may contain memory safety bugs that allow adversaries to read and write arbitrary memory in the monitored process, subject to page table protections. This excludes access to processor registers, and modifications of read-only program code. We trust the microarchitecture and operating system to enforce security boundaries between user processes, and between user and kernel address spaces, which excludes side-channel leakage. We enforce arbitrary policy-defined invariants on program execution, but exclude *confidentiality* policies, because asynchronous policy enforcement, while sufficient for integrity-based policies, could allow data leakage.

2.2 Bounded Asynchronous Validation

Because monitored programs start in a benign state, messages provide a snapshot of program state at a specific point in time. This can effectively provide evidence of a future policy violation, if messages are sent *before* events of interest (e.g., function execution), and messages are guaranteed to be append-only. Even if that violation subsequently results in total program compromise, append-only messages ensure that evidence cannot be retracted.

Asynchronous messages decouple policy checking from program execution, which improves performance by minimizing critical path latency. We rely on system call synchronization to prevent compromised programs from affecting externally-visible side effects (e.g., attacking the system) before a violation is detected by the verifier. The kernel pauses system call execution until the verifier confirms that no policy checks have failed. A naive approach would require a round-trip between the kernel and verifier on every system call executed by the monitored program, which would add latency.

Instead, the monitored program sends a special System-Call message before each system call, which indicates to the verifier that all outstanding messages have been processed, and that the verifier can notify the kernel to resume system call execution. This requires instrumenting shared libraries as well, but it enables the overhead of this synchronization message to be pipelined with the overhead of the system call itself. An attacker can forge this synchronization message, but because the forgery would be transmitted after a message containing evidence of a policy violation, it has no effect. Similarly, if no synchronization message arrives within a configurable epoch, the kernel can treat it as a policy violation and terminate the monitored program.

Table 2: Comparison of existing IPC primitives, grouped by type (*top*: software-based, *center*: hardware-based, *bottom*: proposed), showing message send times.

IPC Primitive	Append Only	Async. Validation	Primary Cost	Time (ns)
Message Queue	✓	×	System Call	146
Named Pipe	✓	×	System Call	316
Socket	✓	×	System Call	346
Shared Memory	×	✓	Mem. Write	12
Light-Weight Contexts	✓	×	System Call	2010 [70]
AppendWrite-FPGA	✓	✓	Mem. Write	102
AppendWrite-μarch	✓	✓	Mem. Write	< 2

2.3 Instantiating The AppendWrite IPC Primitive

AppendWrite must guarantee message *authenticity* and *integrity*, because the monitored program may become compromised. Namely, it must ensure that all messages were sent by the monitored program, and that no messages have been modified or erased after being sent. Although the former can be provided by configuring the kernel to arbitrate creation of messaging channels, the latter requires that the IPC primitive be designed *append-only*. Messages must also have low overhead, to avoid slowing down the monitored program. As discussed below, we observe that existing software- and hardware-based primitives do not satisfy these constraints; hence, we design two hardware implementations of our primitive. AppendWrite-FPGA uses a programmable FPGA accelerator (§2.3.1), whereas AppendWrite-μarch adds append-only memory buffers to the microarchitecture (§2.3.2), which we model in software and validate in simulation (§5.3.1).

Existing IPC mechanisms either perform poorly or lack message integrity, as shown in Table 2, which includes the average runtime of a micro-benchmark that repeatedly sends messages. Primitives which require a system call (including POSIX queues, pipes, and sockets) are too slow: they cost hundreds of nanoseconds, require a

privilege transition that flushes hardware caches (c.f. kernel page-table isolation [52, 69]), and execute synchronously. Traditional optimizations, such as vectored I/O or client-side buffering, would violate integrity by buffering unsent messages in the untrusted sender. Fast IPC primitives, like shared memory, lack semantic access control, allowing writers to corrupt or erase previously-written messages. A compromised program could do so before newly-written messages have been read by the verifier.

Existing hardware-based primitives suffer from similar problems. Even the fastest disjoint address space [39, 70, 86] mechanism costs 2010 ns [70] per context switch, which would be on the critical path, and occur both to and from the verifier on each sent message. On our benchmarks (§5.4), we estimate that this would amount to a worst-case overhead of more than five hours. Certain peripherals already contain hardware first-in first-out (FIFO) queues, which we initially attempted to repurpose, but ultimately determined were unusable. For example, network interface cards (NICs) are widely deployed, and include per-port queues for packet receive/transmit. However, this requires logical or physical loopback of NIC ports, kernel bypass (§6.1) to make these resources available to user-space programs, which entails trusting a large code-base (e.g., Data Plane Development Kit [1]), and the presence of special hardware features, like an IOMMU and PCI Express (PCIe) Access Control Services.

2.3.1 Accelerator. We implement one version of AppendWrite on an FPGA accelerator, which we label AppendWrite-FPGA. It is compatible with any systems that support PCIe expansion cards, including most `x86_64` machines. However, depending on the amount of message traffic, the performance of monitored applications may be limited by the processor interconnect and PCIe bus overhead (§5.3.1).

2.3.2 Microarchitecture. We implement another version of AppendWrite by extending the instruction set architecture (ISA), which we label AppendWrite-μarch. It modifies the microarchitecture to natively support *appendable memory regions* (AMRs), which may span multiple memory pages, and may only be written to via the AppendWrite instruction by userspace programs. Other unprivileged writes to AMR memory pages must be rejected by the MMU. Two privileged per-core registers are added to the processor, which identify the virtual addresses of the next and one-past-the-end message, respectively: *AppendAddr* and *MaxAppendAddr*.

Programs on each core can use a fixed-size AppendWrite instruction to append user-defined messages to the configured AMR, by passing a pointer to a message of, e.g., 32/64/128/256-bytes. The processor automatically increments the *AppendAddr* register and copies the message to the AMR, if doing so would not exceed *MaxAppendAddr*. Otherwise, it faults to the operating system kernel, which can allocate a new buffer or reset address registers, if the AMR has been fully read.

For simplicity, our design configures AMRs using core-local registers, which do not support cross-core writers to minimize cache coherency overhead. Instead, each writer core must be assigned a unique AMR, although a single reader core can iteratively receive messages on all mapped AMRs. Although most execution policies, including control-flow integrity (§3.3), do not need cross-core message ordering, individual messages can include the value of a global counter (e.g. processor timestamp counter) if ordering is needed.

3 DETAILS ON HERQULES COMPONENTS

Below, we describe the implementation details of HerQules' four main components (Figure 1): the AppendWrite IPC primitive, compiler instrumentation, a kernel module, and a verifier process.

3.1 The AppendWrite IPC Primitive

Each message transmitted by AppendWrite is a fixed-size structure, which contains a 4-byte *operation code*, two 8-byte *operation arguments*, and on our FPGA-based implementation, an additional 4-byte *process identifier* (PID). The semantics of our operation codes and arguments are policy-dependent.

3.1.1 Accelerator. We implement AppendWrite-FPGA using a custom Accelerator Functional Unit [68, 81] (AFU) on an Intel Arria 10 [106] GX Programmable Accelerator Card (PAC), a PCIe-based FPGA. Our logic is written in SystemVerilog/Bluespec [78], interacts with the host via the Open Programmable Acceleration Engine [73] (OPAE), and synthesizes using few accelerator resources: 54k (6%) Adaptive Logic Modules and 636k (<1%) block memory bits.

Messages are decomposed into word-granularity uncached writes to memory-mapped I/O (MMIO) registers, which are reassembled by the AFU and written back to a circular buffer in the verifier on the host. To avoid address translation, this circular buffer is allocated from huge memory pages, and its physical address is pinned in memory. The AFU populates the PID field of each message using a kernel-managed PID register, which is updated on each context switch and ensures message authenticity. Operation-specific registers enable messages to be created using at most two MMIO writes.

A per-message counter is used to detect dropped messages, since the AFU lacks a back-pressure mechanism. This occupies otherwise-unused space within each cacheline-aligned memory write from the AFU to the host. The verifier checks that each message has a consecutive counter value; otherwise, the monitored program must be terminated due to violation of message integrity. In practice, we select a circular buffer size of 1 GB such that this never occurs.

3.1.2 Microarchitecture. In terms of logic, die area, and power consumption, the cost of implementing AppendWrite-μarch is extremely low. Execution of AppendWrite resembles that of normal x86 `store` instructions, except that the `store-address` micro-operation directly uses *AppendAddr* without computing an effective address (one fewer micro-operation). A few additional gates and a comparator are needed to verify that *AppendAddr* will not exceed *MaxAppendAddr*, and to bypass the TLB check for writable memory pages in the AMR. Auto-increment logic already exists for, e.g., the `REP` prefix, and can be reused for the *AppendAddr* register. These changes have negligible effect on die area and power consumption.

3.2 Compiler Instrumentation

We implement our instrumentation in Clang/LLVM [64] compiler passes, which insert runtime calls to generate policy-specific messages (§4) and System-Call messages (§2.2), while iterating through the LLVM Intermediate Representation (IR).

Programs directly perform system calls using inline assembly. If an inline assembly call contains a `syscall`, `sysenter`, or `int 0x80` instruction, we treat it as a system call, which must be preceded by a synchronization message. Our analysis uses graph dominators [65]

to find the earliest suitable point for sending such messages. Because each source file may be compiled individually, making interprocedural analysis difficult, we require this program point to be (1) on a program path that executes the system call, and (2) not succeeded by any other messages or function calls. In other words, under non-exceptional control flow, it must *dominate* the system call, be *post-dominated* by the system call, and not dominate any function calls that also dominate the system call.

Indirect system calls occur through standard library functions. As a result, we must recompile the C standard library with system call instruction enabled. Although the GNU C standard library is widely-deployed, it uses GCC-specific compiler extensions that are incompatible with Clang/LLVM, so instead we substitute the musl C standard library [42], which is compact and standards-compliant. During this process, we also statically link our runtime messaging library into the C standard library. Alternatively, our runtime library can be inlined directly into monitored programs, which reduces execution overhead at the cost of increased size. Other standard libraries, such as language libraries for C++, typically call into the C standard library, and thus do not need to be rebuilt.

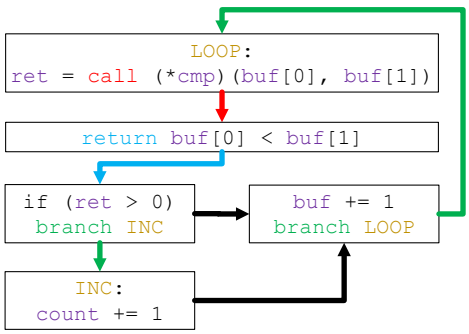

Figure 2: Control-flow graph of a small loop that counts the number of sorted (increasing) pairs in a buffer. Edges are colored, with sequential execution in black, direct forward edges in green, indirect forward edges in red, and backward edges in blue.

3.3 Kernel

We implement bounded asynchronous validation using a kernel module. To maximize compatibility, our kernel module dynamically intercepts system calls using built-in kernel mechanisms, such as *kprobes* [57] and *tracepoints* [36]. A hash table maintains kernel context for each process that has enabled HerQules, including a boolean synchronization variable and various statistics. For a given process, this boolean variable is set by the verifier upon reception of a system call synchronization message, and it is reset by our kernel module upon resumption of a system call.

Our module allocates a new kernel context for child processes upon invocation of the `fork` and `clone` system calls, but as a prototype, it does not model full POSIX program semantics, or optimize away synchronization messages for read-only system calls. Like most work [112] in this space, we do not account for shared memory mappings that may propagate updates to control-flow pointers across multiple processes, which are rarely used and do not occur in our benchmarks, but could result in false positives.

3.4 Verifier

The verifier is a user-space process which maintains a policy context for each monitored application. It receives messages from monitored programs via AppendWrite, and is notified of process events by our kernel module. Policy contexts are allocated, copied, and destroyed when a monitored process enables HerQules, executes `fork` or `clone`, and terminates, respectively. By default, monitored programs are killed upon policy violation or unexpected verifier termination, but this behavior is configurable.

4 EXECUTION POLICIES FOR HERQULES

Below, we provide a case study on control-flow integrity (§4.1), which uses separate protection mechanisms for different types of program control-flow transitions, such as forward-edge transitions (§4.1.3, §4.1.4) and backward-edge transitions (§4.1.5, §4.1.6). We also sketch designs for memory safety (§4.2) and other policies (§4.3).

4.1 Control-Flow Integrity Policy

4.1.1 Background. Control-flow integrity [8, 23] (CFI) protects the execution integrity of a program by verifying control-flow transitions. Typically, static analysis is used to identify how a program *should* execute, then runtime checks are inserted before each transition to ensure that the *actual* execution corresponds. In Figure 2, we show a small loop in the form of a control-flow graph (CFG), which identifies control-flow transitions that may need protection. Forward-edge transitions occur at `branch` and `call` instructions, and are classified as *direct* or *indirect*, based on whether the destination can be statically identified. Backward-edge transitions occur at `return` instructions.

Transition edges are protected by partitioning valid targets into sets of *equivalence classes*, and inserting checks to verify that the runtime target is indeed in the set. Because direct forward edges only have one possible target, and program code is mapped read-only to prevent modification, these edges typically do not need protection.

The effectiveness of control-flow integrity can differ based on the precision of the underlying analysis, which is used to identify valid call targets. Imprecise *coarse-grained* approaches (§6.3.1) simply partition the program based on static function attributes, including address-taken status [2, 3, 94], arity [102], and type [30, 113, 114]. Although relatively fast and widely-deployed, these approaches are vulnerable to code-reuse attacks [18, 25, 34, 47, 48] like return-oriented programming [27, 93] and jump-oriented programming [18]. *Fine-grained* approaches (§6.3.2) improve precision by updating valid call targets at runtime using execution context, e.g., by tracking call paths [37, 45, 53, 71, 79, 107], pointer values [62, 63, 74] (*pointer integrity*), or object origin [58]. These designs maintain runtime metadata to track program state, which must be guarded against memory corruption.

4.1.2 Our Design. We enforce control-flow integrity using a fine-grained *pointer integrity* (§6.3.3) policy, which protects the *values* of control-flow pointers by checking against a copy stored in the verifier via AppendWrite. Unlike other fine-grained approaches, pointer integrity is maximally precise and does not suffer from the pointer aliasing undecidability [88] problem used to defeat [41] past designs. Our approach, named HQ-CFI, differs from past work on pointer integrity, which relocate [62, 63] pointers or verify cryptographic hashes [74] within the instrumented process itself. We are also able to detect use-after-free bugs on control-flow pointers by tracking their lifetime and invalidating them upon destruction, which is not supported by prior control-flow integrity designs.

Table 3 compares our two designs against existing coarse-grained (§6.3.1) and fine-grained (§6.3.2) control-flow integrity designs. Typically, coarse-grained designs are faster but fail to prevent certain attacks, whereas fine-grained designs are more precise but impose greater overhead. As examples, for the former, we select modern Clang/LLVM CFI [30], which is included in Clang/LLVM and widely-deployed. For the latter, we select Cryptographically-Enforced CFI [74] (CCFI) and Code-Pointer Integrity [62, 63] (CPI), which are state-of-the-art pointer integrity designs.

4.1.3 Design: Forward-Edge Transitions. Although programs contain many *control-flow pointers*, some are read-only and do not need protection. For example, on Linux, ELF binaries can contain lazy relocations for imported functions from shared libraries, but we compile programs with read-only relocations and eager binding to prevent runtime changes. Similarly, read-only global variables are stored in a read-only program data section. We protect the following forward-edge control-flow pointers, if writable:

(1) *Function pointers*: Direct pointers to executable code. This includes the internal pointer stored in `jmp_buf` for *non-local gotos* via `longjmp` and `setjmp`.
(2) *Virtual method table pointers*: Indirect pointers in C++ objects that refer to a global per-class virtual method table (vtable). Although vtables contain function pointers, they are stored in read-only memory.
(3) *Virtual-method-table table pointers*: Indirect pointers in certain C++ objects that use multiple inheritance. They refer to a global per-class vtable table that stores relative offsets of individual vtables.

Our design sends messages when certain operations occur on control-flow pointers. For example, certain library functions may manipulate contiguous chunks (blocks) of memory, but because it is difficult to statically determine whether control-flow pointers are present, we notify the verifier of these events at runtime. We describe the semantics for our messages below, and defer implementation to §4.1.4.

- Pointer-Define(p,v): Initialize a pointer at address p with value v.
- Pointer-Check(p,v): Validate that the pointer at address p with current value v matches its previous definition. If not, this pointer is corrupt or a use-after-free.
- Pointer-Invalidate(p): Remove the pointer at address p.
- Pointer-Block-Copy(src,dst,sz): Copy all pointers from address range[src, src + sz) to [dst, dst + sz). These ranges may intersect, and pre-existing pointers in the destination will be invalidated. This matches the behavior of `memcpy` and `memmove`.
- Pointer-Block-Move(src,dst,sz): Move all pointers from address range [src, src + sz) to [dst, dst + sz). These ranges must not intersect, and all pre-existing pointers in the destination will be invalidated. This is an optimization for `realloc`.
- Pointer-Block-Invalidate(p,sz): Invalidate all pointers in the address range [p, p + sz). This matches the behavior of `free`.

4.1.4 Implementation: Forward-Edge Transitions. Our compiler instrumentation uses the following three components to generate runtime calls for sending control-flow pointer messages. We also enable additional devirtualization optimizations for C++, which attempt to convert indirect calls into direct calls that do not need protection. While implementing our instrumentation, we improved various built-in optimizations, fixed miscompilation bugs, and added new extension points for dynamically-loaded passes, which we have submitted for review into LLVM.

(1) *Language-Specific Annotations* (Clang built-in): Insert CFI check annotations before calling function pointers or object methods.
(2) *Initial Lowering* (LLVM): Before program optimization, insert CFI define and invalidate annotations, and convert all CFI annotations into runtime messaging calls.
(3) *Final Lowering* (LLVM/gold [100]): After program optimization, insert instrumentation on block memory operations, optimize messaging calls, and optionally inline our messaging runtime.

Initial Lowering: We examine each operation in the LLVM IR (e.g., `load`, `store`, `call`, etc), and insert runtime messaging as needed. We perform special detection of function pointers to avoid false negatives, as *type casting* allows arbitrary type conversion, and LLVM permits pointers to `struct` fields and `unions` to decay into generic pointers. Specifically, we treat any pointer as a function pointer if (1) it is ever defined from a value of function pointer type, including via pointer casts and ϕ-nodes [11, 89], or (2) other uses of its original value are ever cast to function pointer type.

Final Lowering: We perform store-to-load forwarding and message elision optimizations using our escape analysis, which is more precise than the built-in fast-but-conservative alias analysis. Then, we insert messaging on block memory operations. By default, we perform *strict subtype checking* on composite types passed into block operations using our function pointer detection scheme, which eliminates messaging on block operations that statically do not contain control-flow pointers. However, we observe that this strict checking fails on four benchmarks, which pass decayed function pointers inter-procedurally. We include a built-in allowlist that always instruments block operations in certain functions; alternatively, we could conservatively disable such subtype checking globally at the cost of increased message traffic.

We also insert an initializer function to inform the verifier of global control-flow pointers immediately after program startup. These variables are directly loaded into memory via a data section, and may be relocated by the dynamic loader during program startup. This feature is used by programs that are built position-independent or with runtime layout randomization (§6.4) enabled, which shift function addresses and the value of corresponding function pointers by a runtime offset.

Table 3: Comparison of control-flow integrity designs, grouped by precision (*top*: low, *center/bottom*: high). More • is better.

Design	Mechanism	Precision	Use-After-Free	Compatibility	Performance
Clang/LLVM CFI [30]	Language-level Types	•	×	••	•••
CCFI [74]	Cryptographic MACs	•••	×	•	•
CPI [62]	Information Hiding	••	×	•	•••
CPI [63]	Software Fault Isolation	••	×	•	•••
HQ-CFI-SfeStk	AppendWrite	••	✓	•••	•••
HQ-CFI-RetPtr	AppendWrite	•••	✓	•••	••

C++ Devirtualization: We enable three C++-specific optimization passes, which analyze the type of C++ objects to eliminate vtable loads, infer callees for virtual calls, and eliminate unused virtual functions: Virtual Pointer Invariance [82, 83], Whole Program Devirtualization [31], and Dead Virtual Function Elimination [96]. They reduce the frequency of indirect calls and associated checks.

Store-to-Load Forwarding: A field-sensitive optimization that forwards stored control-flow pointer values to dominated loads, both intra- and inter-procedurally, which reduces checks. To ensure soundness, we exclude accesses to thread-local storage, in functions that may return twice, that are atomic or volatile, or to pointers that may escape. We model inter-procedural loads by *localizing* them to the local call site on the unique call path to the remote function. Instead of passing values through intermediate callees, we create a single *canonical* remote checked load, and forward it to subsequent remote uses. To ensure correctness, we avoid mutually-recursive functions using a runtime guard, as indirect calls make static analysis difficult. While an optimized function is executing, a global boolean *guard variable* is set, and if it remains set upon a subsequent call, then the program is terminated and must be recompiled with this optimization disabled. In practice, no guards fail across all of our benchmarks (§5).

Message Elision: A field- and path-sensitive optimization that eliminates superfluous messages. This includes checks on devirtualized calls, duplicate invalidates after inlining of C++ destructors, as well as other cases that utilize graph dominators and our escape analysis. For example, if a given control-flow pointer is never checked, then it does not need to be defined or invalidated. Similarly, if multiple define messages are emitted, but intermediate values are never checked, then intermediate defines can be removed.

4.1.5 Design: Backward-Edge Transitions. We protect return pointers using two different approaches. One variant, HQ-CFI-RetPtr, sends messages as shown below, whereas our other variant, HQ-CFI-SfeStk, instead places return pointers in a safe stack (§6.3.4) that is protected by information hiding. Although faster, the safe stack is vulnerable to disclosure attacks, whereas the messaging approach is slower but invulnerable.

- Pointer-Define(p,v): See above.
- Pointer-Check-Invalidate(p,v): Performs Pointer-Check(p,v), then if successful, Pointer-Invalidate(p,v).

4.1.6 Implementation: Backward-Edge Transitions. Our compiler instrumentation checks for functions that may write to memory, are known to return, contain stack allocations, and are not always tail called. When found, we insert a runtime call to define the return address pointer in the function prologue, and we insert a runtime call to check-invalidate the pointer in the function epilogue.

4.2 Memory-Safety Policy

Memory safety ensures that all memory accesses occur within the *spatial* boundaries of the target allocation (e.g., not a buffer overflow), and that the allocation itself is *temporally* valid (e.g., not a use-after-free). This eliminates the need for mitigations, such as control-flow integrity and shadow stacks, because memory corruption cannot occur. Below, we sketch an execution policy that enforces memory safety by checking creation, access, and destruction of memory allocations.

- Allocation-Create(a,sz): Create an allocation at [a, a + sz), which cannot overlap with existing allocation(s). This matches the behavior of `malloc`, stack allocation, and read-only/global variables.
- Allocation-Check(a): Check that address a is within a valid allocation. If not, this access is out-of-bounds or use-after-free. This matches the behavior of a pointer dereference.
- Allocation-Check-Base(a1,a2): Check that addresses a1 and a2 are within the same valid allocation. If not, this access is out-of-bounds or use-after-free.
- Allocation-Extend(src,dst,sz): Extend the allocation at src to [dst, dst + sz), which cannot overlap with existing allocation(s). This matches the behavior of `realloc`.
- Allocation-Destroy(a): Destroy an allocation at a. If not present, this destruction is invalid or double. This matches the behavior of `free`.
- Allocation-Destroy-All(a,sz): Destroy all allocations within [a, a + sz). If none are present, this destruction is invalid or double. This matches the behavior of stack deallocation.

4.3 Other Policies

More generally, HerQules can enforce other execution policies for security, performance, or reliability. Examples include data-flow integrity [26], memory tagging, taint tracking, race detection, event counting, software watchdog, and redundant fault detection. These may need *message ordering* between concurrent writers, e.g., by including the value of the processor timestamp counter in each message, or *bidirectional communication* between two processor cores, e.g., by allocating one buffer for each core, and configuring each core to transmit append-only messages to the other buffer.

Table 4: Correctness of various CFI designs.

Design	Errors	False Positives	Invalid	OK
Baseline	0	0	0	48
Baseline-CCFI	2	0	2	46
Baseline-CPI	2	0	2	46
Clang/LLVM CFI	0	15	0	33
CCFI	12	29	9	19
CPI	14	0	14	34
HQ-CFI	0	0	0	48

Table 5: Successful RIPE exploits under various CFI designs, grouped by overflow origin.

Design	BSS	Data	Heap	Stack	Total
Baseline	214	234	234	272	954
Clang/LLVM CFI	60	60	60	10	190
CCFI	0	0	0	0	0
CPI	10	10	10	10	40
HQ-CFI-SfeStk	10	10	10	0	30
HQ-CFI-RetPtr	0	0	0	0	0

5 EVALUATION

We evaluate CFI designs on the RIPE [90, 110] benchmark for effectiveness (§5.2), and on the SPEC CPU2006 [55], SPEC CPU2017 [22], and NGINX web server [98] benchmarks for both correctness (§5.1) and performance (§5.3). All benchmarks were configured to use the musl C runtime library, and we patched SPEC to fix various memory safety (§5.2) and compatibility bugs. To identify the IPC primitive used by HQ-CFI, we apply the postfix -MQ for POSIX message queues (§2.3), -FPGA for the accelerator (§3.1.1), -Sim for the hardware simulator (§5.3.1), and -Model for the hardware model (§5.3.1).

We compare HQ-CFI against past designs from Table 3, which represent different design trade-offs. Since CCFI and CPI are based on Clang/LLVM 3.4.2 and 3.3.1, whereas HQ-CFI and Clang/LLVM CFI are based on 10.0.1, each design is normalized against a version-specific baseline that excludes unavailable optimizations (§4.1.4). For CPI, we disable runtime bounds checking, as we focus on pointer integrity, not spatial memory safety. Due to the prevalence of false positives amongst past designs (§5.1), we continue execution after a policy violation, except when evaluating effectiveness.

During this process, we fixed multiple correctness bugs in CCFI and CPI that crashed the compiler during compilation, as well as other bugs that defeated CPI's protections. These include an incorrect pointer mask that did not guard accesses to the safe store, a failure to redirect function pointers to the safe store, and missing updates to the safe store after `realloc` and `free`. The authors of CPI confirmed [61] that their code was a proof-of-concept prototype, and not fully robust.

5.1 Correctness

To quantify correctness, we executed all performance benchmarks under each CFI design, and checked that each benchmark produced the intended output. We summarize our results in Table 4, distinguishing between errors (crashes or hangs), false positives (no actual CFI violation), invalid results (incorrect output), and successful runs. Note that some categories are not mutually exclusive.

Both CCFI and Clang/LLVM CFI enforce pointer type matching but fail to account for type conversion from casting or decay, producing false positives on 60% and 31% of all benchmarks, respectively. For example, the `povray` benchmark defines a function pointer of type `void *(void *)`, but subsequently calls it with type `void *(pov::Object_Struct *)`, causing both CFI designs to report a violation. Although matching can be relaxed using compiler flags or a custom allowlist, these involve manual debugging.

Both CCFI and CPI cause many benchmarks to execute incorrectly, either due to design flaws or bugs introduced by compiler instrumentation, which affect 25% and 29% of all benchmarks, respectively. We note that 4% of all benchmarks also fail on both respective baselines, suggesting the presence of shared bug(s) in older versions of Clang/LLVM.

CCFI reserves eleven XMM registers to store a private cryptographic key, which breaks platform calling conventions [4] and is incompatible with existing shared libraries. As a workaround, we compile a special C runtime library that avoids registers reserved by CCFI. But, we observe reduced numerical precision and incorrect benchmark output, likely due to usage of x87 floating-point registers from increased register pressure.

CPI fails to redirect all loads and stores of each control-flow pointer to the safe store, causing infinite loops and crashing upon execution of `NULL` pointers. It also consumes significant memory–the safe store is allocated 4 TB of huge-page-backed virtual memory, and was originally evaluated with 512 GB physical memory. To avoid memory-related crashes, we eventually preallocated 16 GB physical memory for huge pages.

5.2 Effectiveness

We demonstrate effectiveness of each CFI design using the RIPE [110] test suite, which contains hundreds of buffer overflow exploits. Because all programs were compiled as 64-bit binaries, we use RIPE64 [90], a port that also adds 100 exploits. For all experiments, we disable program layout randomization, and under HerQules, we also disable enforcement of system call synchronization for `execve`, because RIPE verifies exploits directly using system calls in binary shellcode.

Table 5 shows that no exploits succeeded under CCFI and HQ-CFI-RetPtr, whereas the safe stack is vulnerable to certain exploits, as RIPE emulates disclosure attacks by using a compiler built-in to directly retrieve return pointer addresses. This affects HQ-CFI-SfeStk, Clang/LLVM CFI, and CPI, which use a safe stack; however, the Clang/LLVM implementation adds additional guard pages between the safe and unsafe stacks, which prevents 10 linear overwrite attacks. In addition, Clang/LLVM CFI is vulnerable to 160 return-to-libc code-reuse attacks due to lower design precision.

Initially, we were surprised to discover policy violations for the CPU2006 and CPU2017 benchmarks under HQ-CFI. It turns out that two `omnetpp` benchmarks suffer from use-after-free bugs caused by a subtle *static initialization order* problem, because initialization and destruction of `static` objects across compilation units occurs in undefined order. We note that this bug type has persisted despite

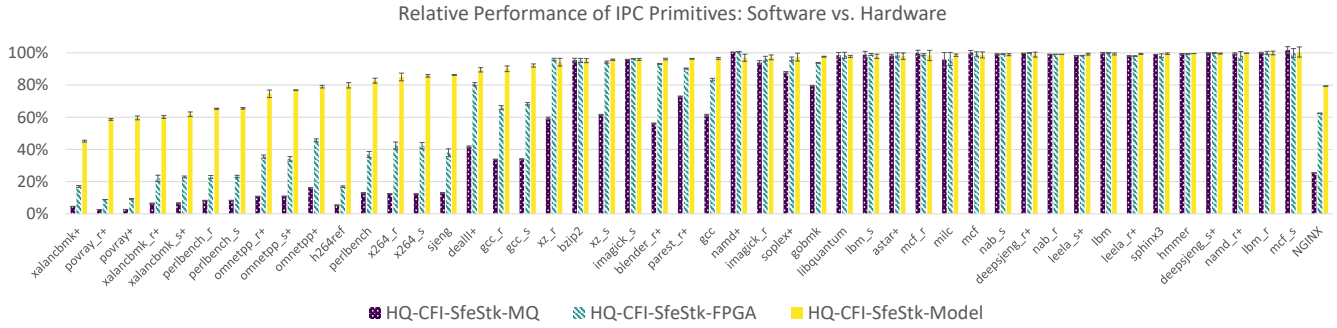

Figure 3: Relative performance of HQ-CFI using various IPC primitives, sorted on HQ-CFI-SfeStk-Model (left to right) for SPEC. Suffix '+' denotes C++.

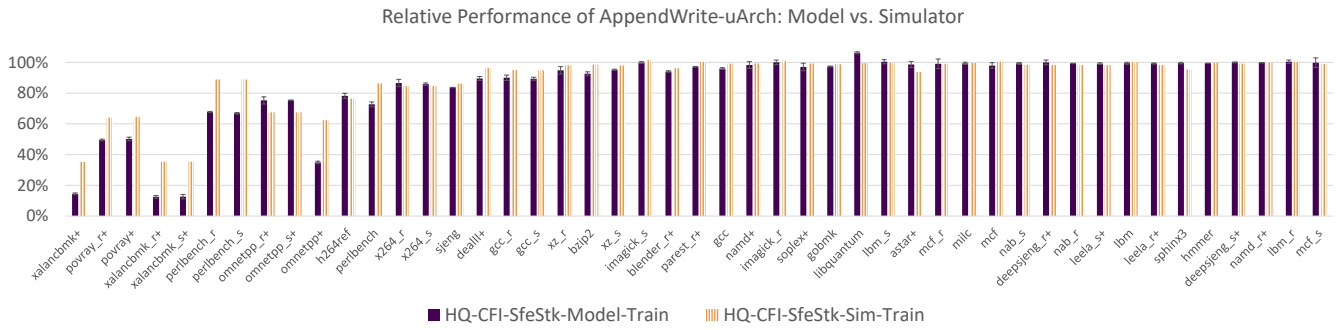

Figure 4: Relative performance of HQ-CFI using AppendWrite-μarch on the *train* input for SPEC. Suffix '+' denotes C++.

over 11 years of continuous development, as both benchmarks correspond to different versions of the OMNeT++ simulator [108]. We have reported these bugs in CPU2017 to SPEC, but no changes are planned, and CPU2006 has been retired.

5.3 Performance

On SPEC CPU2006 and CPU2017, we measure execution time of the *ref* input dataset, unless noted otherwise. On the NGINX web server, we measure request throughput using wrk [46] for 60 s. We report relative performance by computing the arithmetic mean of 3 runs, and show standard deviations using error bars.

5.3.1 IPC Primitives. Using HQ-CFI, we quantify the overhead of each IPC primitive across our performance benchmarks.

Software vs. Hardware: In Figure 3, we compare hardware-based AppendWrite against the fastest suitable software primitive from Table 2 – POSIX message queues (HQ-CFI-SfeStk-MQ). We observe that software-based IPC suffers from significant system call overhead, resulting in a geometric mean performance of only 39%. In comparison, AppendWrite-FPGA (HQ-CFI-SfeStk-FPGA) and our software-only model of AppendWrite-μarch (HQ-CFI-SfeStk-Model, described below) are much faster.

We observe that HQ-CFI-SfeStk-FPGA has a geometric mean performance of 62%, due to processor stalls caused by uncached stores and PCIe bus overhead. Writes to MMIO registers must be written out immediately from cachelines using partial writes of up to 8 bytes, which traverse the *uncore* to become transaction layer packets (TLPs) on the PCIe bus. These occupy store buffer entries until retirement and increase memory pipeline pressure by taking longer compared to cacheable writes. PCIe relies on pipelining of TLP requests with out-of-order responses to maximize bandwidth, but we must transmit each message immediately as a posted TLP *write request* with inline payload, adding bus overhead. Buffering smaller writes into 64-byte PCIe burst transactions via hardware write-combining is not currently supported by the Intel PAC.

Although HQ-CFI-SfeStk-Model should not actually be deployed because it lacks hardware enforcement of append-only messages, it does provide a lower-bound estimate of actual performance, and achieves a geometric mean of 87%. On each AppendWrite, it fetches, checks, and increments an *AppendAddr* variable in shared memory, and waits for the verifier if the message buffer is full. Because these operations are performed by software, it has higher overhead than an actual hardware implementation of AppendWrite-μarch.

Model vs. Simulator: In Figure 4, we compare the performance of two AppendWrite-μarch implementations: our software-only model (HQ-CFI-SfeStk-Model-Train) and a simulation of our actual design (HQ-CFI-SfeStk-Sim-Train). Unlike our other experiments, we use the smaller *train* SPEC dataset to allow the simulator to complete execution within a reasonable amount of time, and measure total simulated processor cycles across one run. We

execute our experiments under ZSim [91], a microarchitectural simulator with an out-of-order core model that is configured to resemble our actual processor. All benchmarks run to completion (maximum 760×10^9 instructions), but we omit NGINX because it is I/O-focused and dominated by system calls.

Our numbers show geometric mean performances of 78% and 86%, respectively, for HQ-CFI-SfeStk-Model-Train and HQ-CFI-SfeStk-Sim-Train. Actual performance of microarchitecture-based AppendWrite will be between these measurements, as HQ-CFI-SfeStk-Model incurs shared memory overhead and waits for the verifier if the buffer is full, whereas HQ-CFI-SfeStk-Sim measures userspace cycles and excludes time spent in system calls.

On HQ-CFI-SfeStk-Model, we observe a -9% change in performance between the *ref* and *train* SPEC inputs. Because *ref* is much longer and executes a different workload, the overhead of each AppendWrite instruction has less impact on benchmark execution.

5.3.2 CFI Designs. In Figure 5, we compare the performance of HQ-CFI-SfeStk-Model and HQ-CFI-RetPtr-Model against related work. We omit measurements for benchmarks that encounter errors or produce invalid output, but not if only false positives are emitted.

On SPEC, we measure geometric means of 88%, 55%, 94%, 49%, and 96%, respectively, for HQ-CFI-SfeStk-Model, HQ-CFI-RetPtr-Model, Clang/LLVM CFI, CCFI, and CPI. However, the performance of CPI and CCFI is likely skewed upwards, because we exhibit slowdowns on similar benchmarks, but of our 14 *slowest* benchmarks, CPI and CCFI crash on 5 and 9 benchmarks, respectively, and were thus excluded from the geometric means for those designs. On NGINX, we observe similar trends, measuring 79%, 62%, 97%, 78%, and 96%, respectively. Overall, on our fastest design, HQ-CFI-SfeStk-Model, we measure a geometric mean of 87.4% performance, or 14.4% overhead, across both SPEC and NGINX.

In general, these results match our expectations, as Clang/LLVM CFI trades precision for performance, CCFI uses expensive cryptography, and CPI relocates control-flow pointers to the safe store, which imposes little overhead. Nevertheless, individual benchmarks can differ; for example, HQ-CFI-SfeStk-Model beats Clang/LLVM CFI by +36% on `mcf_s`, and CPI by +7% on `sphinx3`, which we credit to our optimizations. HQ-CFI-RetPtr-Model is typically slower, with a difference of up to -72% on `gcc_s`, although other benchmarks do remain unchanged or even increase slightly, such as `namd` and `lbm_s`. Frequent execution of recursive functions, or functions with significant stack-allocated pointers, can cause this performance discrepancy. Other benchmarks perform well under all designs, because they lack significant indirect control-flow.

5.3.3 Discussion. Table 3 provides a qualitative overview of various considerations for deploying control-flow integrity, and the trade-offs made by each design. It shows that if the precision of pointer integrity with safe stack is acceptable, HQ-CFI-SfeStk-Model offers better correctness, similar performance, and includes use-after-free detection, when compared to CPI. Alternatively, for fully-precise pointer integrity, HQ-CFI-RetPtr-Model offers better correctness, significantly faster performance, and includes use-after-free detection, when compared to CCFI. Otherwise, if performance is critical, Clang/LLVM CFI is fastest and maintains program correctness, but may emit false positives from unreliable type matching.

Table 6: Size of HerQules, in approximate lines of code.

FPGA	Kernel	Compiler	IPC Interfaces	Runtime	Verifier
1250	1100	3350	900	350	750

As a research prototype, HQ-CFI is not fully optimized. Potential future improvements include modifying the kernel directly to eliminate dynamic interception overhead (§3.3), eliding synchronization for read-only system calls, and eliminating messages on block-level memory operations that do not contain control-flow pointers.

5.4 Other Metrics

Our control-flow integrity case study shows that HerQules achieves scalable policy enforcement. We observe that across our SPEC and NGINX benchmarks on a per-benchmark basis, AppendWrite is used to transmit a median of 1.4×10^3 messages per second and a geometric mean of 14 messages per second. The maximum is 53×10^3 messages per second by the `h264ref` benchmark, which achieves 77% relative performance under HQ-CFI-SfeStk-Model. For total messages, we measure a maximum of 4.76×10^9 messages by the `xalancbmk` benchmark.

In terms of memory overhead, on a per-benchmark basis, the verifier maintains a maximum of $\sim 3 \times 10^6$ entries, with a median of 285 entries and an arithmetic mean of 221×10^3 entries. Each entry is a 16-byte pointer-value pair. This includes 14 benchmarks with zero entries, which lack control-flow pointers needing protection.

In Table 6, we show a breakdown of each HerQules component in terms of lines of code. We exclude autogenerated Verilog for our FPGA and existing software-based primitives for our IPC interfaces. Most components are fairly small, with the bulk of our compiler implementation consisting of optimizations.

6 RELATED WORK

6.1 Interprocessor Communication

Many architectures include messaging primitives, which suffer from various drawbacks. On x86, interprocessor interrupts are slow, privileged, and have limited usage [19]. Intel has proposed *enqueue stores* [7], which can only send a command and authenticated identifier to memory-mapped devices. Embedded ARM devices like the Raspberry Pi typically provide a *mailbox* peripheral [72], which connects processors (e.g., CPU-GPU) and not individual cores.

6.2 Hardware Extensions

Bespoke hardware extensions run the risk of design complexity and over-specialization. Over the past decade, ARM and Intel have implemented hardware bounds checking [20, 59] (MPX), memory isolation [6, 85] (MPK/PKU), coarse-grained control-flow integrity [2, 51, 94] (BTI, CET), pointer authentication [21] (PA), and tagged memory [51] (MTE). However, their practical effectiveness and usability are questionable. MPX was removed [54] after shortcomings were identified [80]. BTI and CET implement a weak form of CFI that has been defeated (§6.3.1). MPK/PKU suffers from compatibility and scalability issues that has motivated development of software workarounds [85].

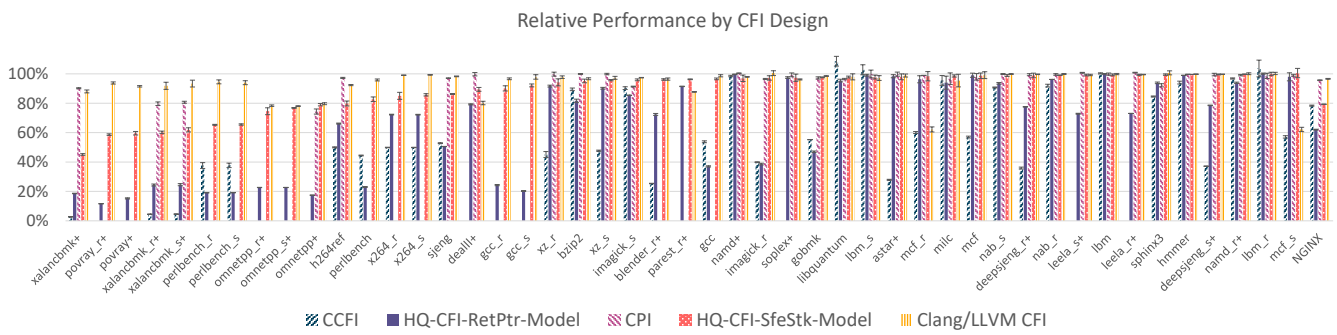

Figure 5: Relative performance of various CFI designs, sorted on HQ-CFI-SfeStk-Model (left to right). Suffix '+' denotes C++.

Few have seen broad uptake, with the exception of ARM's pointer authentication, which has been used for pointer integrity [66, 67, 75]. However, Apple's design [75] is a cryptographic approach that has lower precision than CCFI and cannot detect use-after-free, due to the difficulty of hash revocation. To maximize compatibility, it omits the address of control-flow pointers from hash computations, which allows replay attacks. As a workaround, it supports a separate *discriminator* nonce, however, it uses a constant zero discriminator for function pointers and C++ virtual table pointers. It is also not externally accessible for development or testing, and has been shown vulnerable [12, 13] to multiple flaws.

6.3 Control-Flow Integrity

6.3.1 Coarse-Grained CFI. Many existing CFI designs are coarse-grained, which approximate control flow using a limited number of equivalence classes. As a result, they have low overhead and are widely-deployed, but are vulnerable to code-reuse attacks. Some use just one class for all address-taken functions, including Microsoft's CFG [3] (MSCFG) and Intel CET's Indirect Branch Tracking [2, 94]. MSCFG has been included since Windows 8.1 [99], though it had a design flaw [17], and is to be replaced [87] with CET in future Windows 10 releases. ARM's BTI supports three [51] equivalence classes based on assembly instruction type. Others create classes based on callee arity [102] or type [113, 114], including modern Clang/LLVM CFI [30]. Google Chrome and certain Android devices [5, 103, 104] are built with modern Clang/LLVM CFI enabled, but it has a high false-positive rate (§5.3.2).

6.3.2 Fine-Grained CFI. Fine-grained designs improve precision by incorporating runtime context. However, *path-sensitive* approaches suffer from exponential explosion; they must use hardware-accelerated recording (e.g. Intel LBR [107] or PT [37, 45, 53, 71]), merge call paths [79, 107], and/or limit checks to certain system calls [37, 45, 53, 71, 107]. Alternative approaches include pointer integrity (§6.3.3) and object origin [58].

6.3.3 Pointer Integrity. Pointer integrity protects the values of control-flow pointers, rather than partitioning program callers and callees based on expected control-flow.

Code Pointer Integrity [62] (CPI) relocates backward-edge control-flow pointers to a *safe stack* (§6.3.4), and forward-edge ones to a *safe pointer store*. It also checks boundaries of relocated pointers to prevent buffer overflows from corrupting the safe store. Initially, it used information hiding to protect relocated pointers, but the safe store was defeated [40, 49] by disclosure attacks, and replaced [63] with software fault isolation, at the cost of increased overhead. Their final design has moderate overhead but breaks many programs (§5), does not implement use-after-free detection, and requires recompilation of existing libraries.

Cryptographically Enforced CFI [74] (CCFI) stores an adjacent message authentication code (MAC) hash for each control-flow pointer, and checks the hash on every load of the pointer. The MAC performs a single round of the Advanced Encryption Standard (AES) block cipher, using hardware-accelerated instructions for performance. To prevent forgery and replay attacks, the private key is stored in eleven reserved XMM registers, and a random offset is injected into each stack frame as a nonce. Their approach has tremendous overhead (§5), uses a non-standard AES construction[1], breaks platform calling conventions, does not support use-after-free detection, and emits numerous false positives.

6.3.4 Return Pointers. Some architectures, like x86, store *return pointers* on the stack, which allows for corruption. Coarse-grained CFI approaches (§6.3.1) may match call-return pairs, but are vulnerable to code-reuse attacks. Instead, *shadow stacks* [24, 33, 43, 94] place return pointers on a separate stack, ideally protected by hardware with special otherwise-inaccessible pages. Software-based approaches instead use information hiding, but Microsoft's Return Flow Guard [16] was discontinued due to information disclosure attacks. Code Pointer Integrity [62] proposed a software *safe stack* [63] instead, which stores all objects that statically may overflow. It defeats some attacks [32], and has been adopted by Clang/LLVM CFI [30], which added guard pages to detect contiguous overflows. Nevertheless, it is still vulnerable [49] to disclosure attacks (§6.4), and was thus disabled [105] for Google Chrome.

6.4 Information Hiding

Randomizing programs to hide information provides probabilistic security. Runtime randomization of program content [29, 84] or layout [14, 15, 56, 101] helps deter exploits and relocate sensitive data. But, side-channel or other disclosure attacks [50, 92, 95, 97] can defeat hiding.

[1]The official AES-128 block cipher requires 10 rounds.

7 CONCLUSION

We develop HerQules as a framework for efficiently enforcing integrity-based execution policies. By adding a simple AppendWrite IPC primitive to an FPGA-based accelerator or the microarchitecture, we are able to maintain an append-only message log of program events. We use *bounded asynchronous validation* to perform policy checking asynchronously with program execution, while preventing compromised programs from affecting externally-visible side effects. Our control-flow integrity case study demonstrates that our approach is more correct, effective, and performant than other designs, and adds detection of use-after-free bugs on control-flow pointers. In addition, HerQules supports concurrent message ordering and bidirectional communication, as well as other policies like memory safety, data-flow integrity, memory tagging, etc.

ACKNOWLEDGMENTS

This work was supported by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the Department of the Navy, Office of Naval Research under Grant No. N00014-17-1-2892, by the National Science Foundation under Grant No. CCF-2028949, and by a grant from the Alfred P. Sloan Foundation.

We would like to thank Intel Corporation for providing PAC cards, the Parallel Data Lab and its sponsors for compute resource access, Chris Fallin for helping design our microarchitecture-based IPC primitive, Joseph Melber for implementing host memory writes from the PAC card, Pratik Fegade for reviewing early drafts of this paper, Saugata Ghose for providing SPEC CPU2017, and Maverick Woo for providing feedback and managing our systems. We would also like to thank our anonymous reviewers, artifact evaluators, and our shepherd, Yannis Smaragdakis, for their feedback in improving this paper.

A ARTIFACT APPENDIX

A.1 Abstract

The instructions below will execute experiments for HQ-CFI-SfeStk-Model, HQ-CFI-RetPtr-Model, and their baseline, as well as HQ-CFI-SfeStk-Sim and its baseline, on the RIPE64, SPEC CPU2006, SPEC CPU2017, and NGINX benchmarks. This will reproduce the performance of HQ-CFI-SfeStk-Sim in Figure 4, the performance of HQ-CFI-SfeStk-Model in Figure 5, and the RIPE results for HQ-CFI-SfeStk-Model and HQ-CFI-RetPtr-Model in Table 5.

A.2 Artifact Check-List

- **Program:** SPEC CPU2006 v1.2, SPEC CPU2017 v1.0.5, NGINX v1.17.10, and RIPE64.
- **Compilation:** Build benchmarks using the pre-built modified Clang/LLVM 10.0.1 compiler and gold 1.16 linker.
- **Transformations:** Program instrumentation implemented using included LLVM transformation passes.
- **Binary:** Yes, pre-built Clang 10.0.1 for Ubuntu 20.04/18.04, pre-built musl 1.2.1, and pre-built benchmark binaries.
- **Run-time environment:** Ubuntu 20.04 with kernel 5.4.53, and Ubuntu 18.04 with kernel 4.15.70. Root is needed to load kernel module and run verifier.
- **Hardware:** For optional FPGA experiments, an Intel Programmable Acceleration Card, using OPAE 1.4.0 and Intel Acceleration Stack for Development 1.2.0 with Update 1.
- **Execution:** 350 mins for one run of CPU2006 + CPU2017 under HQ-CFI-SfeStk-Model, and 309 mins for one run of CPU2006 + CPU2017 using its baseline. 1 min for each run of NGINX.
- **Metrics:** Relative performance (execution time for SPEC, throughput for NGINX).
- **Output:** CSV file contains runtimes for each SPEC run, and throughput for each NGINX run.
- **Experiments:** See below instructions and README.md.
- **How much disk space required (approximately)?:** One build of CPU2006 + CPU2017: 7.2G.
- **How much time is needed to prepare workflow?:** If not using included VMs, per compile of CPU2006 + CPU2017: 45 mins. Per compile of NGINX: < 2 mins.
- **How much time is needed to complete experiments?:** See execution above.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** See SPEC open-source licenses. NGINX is open-source. HQ-CFI is Apache.
- **Archived (provide DOI)?:** 10.5281/zenodo.4501773.

A.3 Description

A.3.1 How to Access. We provide virtual machines (VMs) that include our compiler toolchain, runtime libraries, and experiments. Both VMs have a username/password of `user`. The *model-vm* contains pre-built binaries for HQ-CFI-SfeStk-Model and its baseline, whereas the *sim-vm* contains pre-built binaries for HQ-CFI-SfeStk-Sim and its baseline.

A.3.2 Hardware Dependencies. Similar or equivalent: at least 16 GB DDR4 memory, a Samsung 860 Pro solid-state drive, and an Intel i9-9900k CPU at 5GHz.

A.3.3 Software Dependencies. For non-simulator benchmarks, use Ubuntu 20.04. For simulator benchmarks, use Ubuntu 18.04 due to a kernel incompatibility. Disable simultaneous multithreading, and configure huge pages at boot-time, if not using our VMs.

A.3.4 Data Sets. See above for SPEC CPU2006/2017.

A.4 Experiment Workflow

A.4.1 Non-Simulator Experiments (model-vm). Move into the `herqules/build` directory. As root, insert the kernel module `kernel/hq.ko` and run the verifier `verifier/verifier`. Run the below experiments. After experiments are complete, exit the verifier and unload the kernel module.

Our pre-built experiments are stored in the following locations:

- RIPE: `/home/user/herqules/tests/ripe`
- HQ-CFI-SfeStk-Model SPEC: `/home/user/herqules/tests/llvm-test-suite/build_hq`
- HQ-CFI-SfeStk-Model NGINX: `/home/user/herqules/tests/nginx`
- Baseline SPEC: `/home/user/herqules/tests/llvm-test-suite/build_none`
- Baseline NGINX: `/home/user/herqules/tests/nginx_none`

(1) RIPE: Disable ASLR with `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`, then execute `./ripe_test.py`

`both 3 both`. Results for HQ-CFI are printed as 'Clang', whereas baseline results are printed as 'GCC'.
(2) SPEC: Execute `lit External/SPEC -time-tests -j1`, then `./scripts/combine.py tests/llvm-test-suite/build/External`. This will combine results into `External/out.csv`.
(3) NGINX: Execute `./root/sbin/nginx`, and in a separate window, `/home/user/herqules/tests/wrk/wrk -t1 http://localhost:8080`. This will print total request throughput.

To run HQ-CFI-RetPtr-Model instead of HQ-CFI-SfeStk-Model, follow the customization instructions below, rebuild affected benchmarks, then follow the execution instructions above.

A.4.2 Simulation Experiments (sim-vm). Our pre-built experiments are stored in the following locations:

- HQ-CFI-SfeStk-Sim SPEC: `/home/user/herqules/tests/llvm-test-suite/build`
- Baseline SPEC: `/home/user/herqules/tests/llvm-test-suite/build_baseline`

Copy the `./scripts/simulations/run_all.sh` script to each directory containing pre-built benchmarks, and modify the script to provide paths to the ZSim binary and simulated processor configuration file. Then, run each simulation using `./run_all.sh arg1 arg2`. `arg1` is a name for the method (e.g., baseline, HQ-CFI-SfeStk-Sim), and `arg2` is the number of experiments that will run in parallel.

A.5 Evaluation and Expected Results

We have included a copy of our experiment data in `data.xlsx`.

A.5.1 Non-Simulator Experiments. Import each `out.csv` file (for SPEC) or input the total number of requests (for NGINX) into the 'Run #1' column of their respective sheets, then clear the columns for 'Run #2' and 'Run #3'. For HQ-CFI-SfeStk-Model and HQ-CFI-RetPtr-Model, modify the 'model-ref' sheet; for their baseline, modify the 'baseline-opt-ref' sheet. Switch back to the 'benchmarks' sheet, and the corresponding column should be within ±5% of the original value. Separately, for RIPE, directly compare with the 'ripe' sheet.

A.5.2 Simulator Experiments. Use the script located at `./scripts/simulations/parse_zsim_results.py` and run `./parse_zsim_results.py arg1 arg2`. `arg1` is the path where the ZSim results of baseline experiments are located, and `arg2` is the path where HQ-CFI-SfeStk-Sim experiments are. After running the script, directly compare with the 'ss-simulator-train' sheet.

A.6 Experiment Customization

To perform other experiments, e.g. for the FPGA, build a baseline and instrumented version of each benchmark. To enable return pointer instrumentation (HQ-CFI-RetPtr), change the default value of `RunCFIRetAddr` in `llvm/cfi-finalize.cpp` and rebuild the instrumentation pass. See `scripts/envs.sh` for full configuration flags and `README.md` for full setup instructions. Note that the FPGA may require a physical machine, if PCIe passthrough to a VM is not supported.

A.7 Manual Installation

If not using our VMs, the below instructions will install HQ-CFI and build our experiments.

(1) Clone https://github.com/secure-foundations/herqules.
(2) Download the pre-built binaries for Clang/LLVM, musl, and the FPGA bitstream (if needed).
(3) For simulator experiments, follow https://github.com/ddcc/zsim and set up ZSim.
(4) Edit `./scripts/setup.sh`, and set `OPAE=1` if performing FPGA experiments. Run the script as root.
(5) Execute steps 1a and 2 under 'Standard Runtimes' in the `README.md` file, to configure the baseline environment.
(6) Execute step 1a under 'Compiler (Clang/LLVM)' in the `README.md` file, to configure our modified Clang/LLVM compiler toolchain.
(7) Execute step 3 under 'LLVM Test Suite / SPEC Benchmarks' in the `README.md` file, to fix various benchmark compatibility and memory safety bugs using our patches.
(8) For non-simulator experiments, edit `./scripts/build.sh`, and run it. Otherwise, for simulator experiments, edit `./scripts/build_simulator.sh`, and run it.

A.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] 2010. Data Plane Development Kit. https://www.dpdk.org/
[2] 2016. Control-Flow Enforcement Technology Specification. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf
[3] 2018. Control Flow Guard - Win32 Apps. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard
[4] 2018. System V Application Binary Interface: AMD64 Architecture Processor Supplement. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf
[5] 2020. Control Flow Integrity | Android Open Source Project. https://source.android.com/devices/tech/debug/cfi
[6] 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
[7] 2020. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf
[8] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 2005 ACM SIGSAC Conference on Computer and Communications Security - CCS '05*. ACM Press, 340–340. https://doi.org/10.1145/1102120.1102165
[9] Sam Ainsworth and Timothy M. Jones. 2020. The Guardian Council: Parallel Programmable Hardware Security. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 1277–1293. https://doi.org/10.1145/3373376.3378463
[10] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Montreal, Canada, 51–66. https://doi.org/10.5555/1855768.1855772
[11] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/73560.73561

[12] Brandon Azad. 2019. Project Zero: Examining Pointer Authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

[13] Brandon Azad. 2020. iOS Kernel PAC, One Year Later. https://bazad.github.io/presentations/BlackHat-USA-2020-iOS_Kernel_PAC_One_Year_Later.pdf

[14] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, USA, 8. https://doi.org/10.5555/1251353.1251361

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14 (SSYM'05)*. USENIX Association, USA, 17. https://doi.org/10.5555/1251398.1251415

[16] Joe Bialek. 2018. The Evolution of CFI Attacks and Defenses. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf

[17] Andrea Biondo, Mauro Conti, and Daniele Lain. 2018. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23318

[18] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11, Vol. 38)*. Association for Computing Machinery, 30–30. https://doi.org/10.1145/1966913.1966919

[19] Daniel P. Bovet and Marco Cesati. 2006. *Understanding the Linux Kernel: From I/O Ports to Process Management* (3. ed., covers version 2.6 ed.). O'Reilly, Beijing.

[20] Richard S Bracher. 2013. Introduction to Intel® Memory Protection Extensions. https://www.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html

[21] David Brash. 2016. Armv8-A Architecture: 2016 Additions. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-a-architecture-2016-additions

[22] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. Association for Computing Machinery, Berlin, Germany, 41–42. https://doi.org/10.1145/3185768.3185771

[23] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1 (April 2017), 16:1–16:33. https://doi.org/10.1145/3054924

[24] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. 985–999. https://doi.org/10.1109/SP.2019.00076

[25] Nicholas Carlini and David Wagner. 2014. ROP Is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, USA, 385–399. https://doi.org/10.5555/2671225.2671250

[26] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 147–160. https://doi.org/10.5555/1298455.1298470

[27] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. Association for Computing Machinery, Chicago, Illinois, USA, 559–572. https://doi.org/10.1145/1866307.1866370

[28] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. 2008. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *2008 International Symposium on Computer Architecture*. 377–388. https://doi.org/10.1109/ISCA.2008.20

[29] Frederick B. Cohen. 1993. Operating System Protection through Program Evolution. *Computers & Security* 12, 6 (Oct. 1993), 565–584. https://doi.org/10.1016/0167-4048(93)90054-9

[30] Peter Collingbourne. 2015. Control Flow Integrity Design Documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[31] Peter Collingbourne. 2016. Whole Program Devirtualization. Google, Inc. https://reviews.llvm.org/D16795

[32] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, Denver, Colorado, USA, 952–963. https://doi.org/10.1145/2810103.2813671

[33] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15*. ACM Press, 555–566. https://doi.org/10.1145/2714576.2714635

[34] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, San Diego, CA, 401–416. https://doi.org/10.5555/2671225.2671251

[35] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. 2010. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 137–148. https://doi.org/10.1109/MICRO.2010.17

[36] Mathieu Desnoyers. 2008. Using the Linux Kernel Tracepoints. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[37] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, Vancouver, BC, Canada, 131–148. https://doi.org/10.5555/3241189.3241201

[38] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[39] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 353–368. https://doi.org/10.1145/2872362.2872366

[40] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(Er): On the Effectiveness of Code Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 781–796. https://doi.org/10.1109/SP.2015.53

[41] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, 901–913. https://doi.org/10.1145/2810103.2813646

[42] Rich Felker. 2020. Musl Libc. https://musl.libc.org/

[43] Mike Frantzen and Mike Shuey. 2001. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Washington, D.C. https://doi.org/10.5555/1251327.1251332

[44] Sotiria Fytraki, Evangelos Vlachos, Onur Kocberber, Babak Falsafi, and Boris Grot. 2014. FADE: A Programmable Filtering Accelerator for Instruction-Grain Monitoring. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 108–119. https://doi.org/10.1109/HPCA.2014.6835922

[45] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. *ACM SIGARCH Computer Architecture News* 45, 1 (April 2017), 585–598. https://doi.org/10.1145/3093337.3037716

[46] Will Glozer. 2019. Wrk - a HTTP Benchmarking Tool. https://github.com/wg/wrk

[47] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 575–589. https://doi.org/10.1109/SP.2014.43

[48] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks Is Hard. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, USA, 417–432. https://doi.org/10.5555/2671225.2671252

[49] Enes Göktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX Association, USA, 105–119. https://doi.org/10.5555/3241094.3241104

[50] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*. https://doi.org/10.14722/ndss.2017.23271

[51] Matthew Gretton-Dann. 2018. Arm Architecture Armv8.5-A Announcement. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a

[52] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR Is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (Lecture Notes in Computer Science)*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[53] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY '17)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 173–184. https://doi.org/10.1145/3029806.3029830

[54] Dave Hansen. 2018. X86: Remove Intel MPX. https://lore.kernel.org/patchwork/patch/1025952/

[55] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[56] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *2012 IEEE Symposium on Security and Privacy*. 571–585. https://doi.org/10.1109/SP.2012.39

[57] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. 2005. Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt

[58] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-Sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, Santa Clara, CA, USA, 195–211. https://doi.org/10.5555/3361338.3361353

[59] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. Association for Computing Machinery, Belgrade, Serbia, 437–452. https://doi.org/10.1145/3064176.3064217

[60] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks without the Checks. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys '18)*. ACM Press, Porto, Portugal, 1–14. https://doi.org/10.1145/3190508.3190553

[61] Volodymyr Kuznetsov. 2020. Re: Resend: Code-Pointer Integrity + Software Fault Isolation.

[62] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 147–163. https://doi.org/10.5555/2685048.2685061

[63] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, and Dawn Song. 2015. Poster: Getting The Point(Er): On the Feasibility of Attacks on Code-Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 2.

[64] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[65] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems* 1, 1 (Jan. 1979), 121–141. https://doi.org/10.1145/357062.357071

[66] Hans Liljestrand, Thomas Nyman, Jan-Erik Ekberg, and N. Asokan. 2019. Authenticated Call Stack. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. ACM, New York, NY, USA, 223:1–223:2. https://doi.org/10.1145/3316781.3322469

[67] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 177–194. https://doi.org/10.5555/3361338.3361352

[68] Liu Ling, Neal Oliver, Chitlur Bhushan, Wang Qigang, Alvin Chen, Shen Wenbo, Yu Zhihong, Arthur Sheiman, Ian McCallum, Joseph Grecco, Henry Mitchel, Liu Dong, and Prabhat Gupta. 2009. High-Performance, Energy-Efficient Platforms Using in-Socket FPGA Accelerators. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '09)*. Association for Computing Machinery, Monterey, California, USA, 261–264. https://doi.org/10.1145/1508128.1508172

[69] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Baltimore, MD, USA, 973–990. https://doi.org/10.5555/3277203.3277276

[70] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Berkeley, Calif. https://doi.org/10.5555/3026877.3026882

[71] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–540. https://doi.org/10.1109/HPCA.2017.18

[72] Rintel Lubomir. 2015. Mailbox: Enable BCM2835 Mailbox Support. https://patchwork.kernel.org/patch/6342841/

[73] Enno Luebbers, Song Liu, and Michael Chu. 2017. Simplify Software Integration for FPGA Accelerators with OPAE. https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf

[74] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[75] John McCall. 2019. Pointer Authentication. https://github.com/apple/llvm-project/blob/apple/master/clang/docs/PointerAuthentication.rst

[76] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[77] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1806651.1806657

[78] R. Nikhil. 2004. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 69–70. https://doi.org/10.1109/MEMCOD.2004.1459818

[79] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, Denver, Colorado, USA, 914–926. https://doi.org/10.1145/2810103.2813644

[80] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (June 2018), 28:1–28:30. https://doi.org/10.1145/3224423

[81] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*. 80–85. https://doi.org/10.1109/ReConFig.2011.4

[82] Piotr Padlewski. 2017. Devirtualization in LLVM. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. Association for Computing Machinery, Vancouver, BC, Canada, 42–44. https://doi.org/10.1145/3135932.3135947

[83] Piotr Padlewski, Krzysztof Pszeniczny, and Richard Smith. 2020. Modeling the Invariance of Virtual Pointers in LLVM. *arXiv:2003.04228 [cs]* (Feb. 2020). arXiv:2003.04228 [cs] http://arxiv.org/abs/2003.04228

[84] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*. 601–615. https://doi.org/10.1109/SP.2012.41

[85] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19)*. USENIX Association, Renton, WA, USA, 241–254. https://doi.org/10.5555/3358807.3358829

[86] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*. 563–577. https://doi.org/10.1109/SP40000.2020.00041

[87] Hari Pulapaka. 2020. Understanding Hardware-Enforced Stack Protection. https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815

[88] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems* 16, 5 (Sept. 1994), 1467–1471. https://doi.org/10.1145/186025.186041

[89] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[90] Hubert Rosier. 2019. RIPE64. National University of Singapore. https://github.com/hrosier/ripe64

[91] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *ACM SIGARCH Computer*

Architecture News 41, 3 (June 2013), 475–486. https://doi.org/10.1145/2508148.2485963

[92] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 54–65. https://doi.org/10.1145/2660267.2660309

[93] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. Association for Computing Machinery, Alexandria, Virginia, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[94] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. Association for Computing Machinery, Phoenix, AZ, USA, 1–11. https://doi.org/10.1145/3337167.3337175

[95] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*. 574–588. https://doi.org/10.1109/SP.2013.45

[96] Oliver Stannard. 2019. Dead Virtual Function Elimination. Linaro. https://reviews.llvm.org/D63932

[97] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security (EUROSEC '09)*. Association for Computing Machinery, Nuremburg, Germany, 1–8. https://doi.org/10.1145/1519144.1519145

[98] Igor Sysoev. 2020. NGINX. Nginx, Inc.. https://www.nginx.com/

[99] Jack Tang. 2015. Exploring Control Flow Guard in Windows 10. https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

[100] Ian Lance Taylor. 2020. Gold. https://sourceware.org/binutils/

[101] The PaX Team. 2003. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt

[102] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium - SEC '14*. 941–955. https://doi.org/10.5555/2671225.2671285

[103] Sami Tolvanen. 2018. Control Flow Integrity in the Android Kernel. https://security.googleblog.com/2018/10/posted-by-sami-tolvanen-staff-software.html

[104] Sami Tolvanen. 2019. Protecting against Code Reuse in the Linux Kernel with Shadow Call Stack. https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html

[105] Vlad Tsyrklevich. 2018. 908597 - Deprecate SafeStack - Chromium. Google, Inc. https://bugs.chromium.org/p/chromium/issues/detail?id=908597

[106] Jeffrey Tyhach, Mike Hutton, Sean Atsatt, Arifur Rahman, Brad Vest, David Lewis, Martin Langhammer, Sergey Shumarayev, Tim Hoang, Allen Chan, Dong-Myung Choi, Dan Oh, Hae-Chang Lee, Jack Chui, Ket Chiew Sia, Edwin Kok, Wei-Yee Koay, and Boon-Jin Ang. 2015. Arria™ 10 Device Architecture. In *2015 IEEE Custom Integrated Circuits Conference (CICC)*. 1–8. https://doi.org/10.1109/CICC.2015.7338368

[107] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, Denver, Colorado, USA, 927–940. https://doi.org/10.1145/2810103.2813673

[108] András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools '08)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 1–10. https://doi.org/10.5555/1416222.1416290

[109] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles - SOSP '93*. ACM Press, 203–216. https://doi.org/10.1145/168619.168635

[110] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. 2011. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. Association for Computing Machinery, Orlando, Florida, USA, 41–50. https://doi.org/10.1145/2076732.2076739

[111] Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Vancouver, B.C., Canada. https://doi.org/10.5555/1267336.1267345

[112] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. 2019. CONFIRM: Evaluating Compatibility and Relevance of Control-Flow Integrity Protections for Modern Software. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, 1805–1821. https://doi.org/10.5555/3361338.3361463

[113] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*. 559–573. https://doi.org/10.1109/SP.2013.44

[114] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*. USENIX Association, Washington, D.C., 337–352. https://doi.org/10.5555/2534766.2534796