Mitigating Memory-Safety Bugs with Efficient Out-of-Process Integrity Checking

Daming Dominic Chen

CMU-CS-21-113

June 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Phillip B. Gibbons, Chair
James C. Hoe
Taesoo Kim, Georgia Institute of Technology
Bryan Parno

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Semiconductor Research Corporation (SRC) under the JUMP CONIX Research Center, by the Department of the Navy, Office of Naval Research under grant number N00014-17-1-2892, by the National Science Foundation under grant numbers CCF-2028949, CNS-1065112, CNS-1228827, and CNS-1618595, by the Alfred P. Sloan Foundation under grant number BR2013017, by the Naval Information Warfare Centers (formerly SPAWAR Systems) under grant number N660011324040, and by the Department of Defense under the National Defense Science & Engineering Graduate Fellowship Program.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords. memory safety, inter-process communication, control-flow integrity, pointer integrity, data-flow integrity, compiler, kernel, shared memory, web browser

Abstract. Computer programs written in low-level languages with manual memory management, like C and C++, can contain unintentional memory-safety bugs [175] due to developer error. Examples of these bugs include *spatial* buffer overflows, as well as *temporal* use-after-frees and double frees, which can be leveraged by attackers to exploit programs by altering their runtime behavior. Indeed, statistics from both Google Chrome [1] and Microsoft [129] show that ~70% of all security vulnerabilities in their codebases involve memory-safety bugs.

Past work, as discussed in Chapter 2, has proposed various strategies to eagerly detect or lazily mitigate such bugs. Eager approaches detect memory-safety bugs by checking pointer operations (§2.1), whereas lazy mitigations prevent exploitation by validating program data (§2.4, §2.5). To improve accuracy, mitigations may need to maintain internal state (metadata) about program execution, which must also be protected from corruption. This has been achieved using different techniques, including software-based address space partitioning (§2.2), and hardware-based fine-grained instruction monitoring (§2.3). Nevertheless, these approaches suffer from significant complexity, brittleness, or incompatibility, which reduces their efficiency and effectiveness.

In this thesis, we observe that existing mitigations are limited by their decision to maintain internal metadata within the same process. We show that augmenting hardware with a small, secure, and efficient AppendWrite inter-process communication (IPC) primitive allows metadata storage and policy checking to be performed in a separate isolated process, which improves both security and performance. We implement this design in our HerQules [42] framework, which we use to develop new approaches for *control-flow integrity* and *data-flow integrity* that are more precise than past work. We evaluate our designs on a variety of real-world programs, including multiple benchmark suites, the NGINX web server, and the Google Chromium web browser.

To my family.

Acknowledgments

I'm grateful to my advisor, Phil, for taking me on as his student. It was a chance lunch encounter on Forbes Avenue that started it all, and he has consistently supported me ever since. He granted me the freedom to pursue my own research interests, even when they have ultimately been unsuccessful. He committed to meeting with me regularly, despite scheduling conflicts. He helped revise my paper drafts, up until right before the submission deadline. I appreciate his dedication, insights, and suggestions.

I'm thankful to my previous co-advisor, Todd, for supporting and advising me. He encouraged me to persevere, suggested new research directions, and shaped my problem-solving approach.

I'm appreciative of my committee members, Bryan, James, and Taesoo, for helping me. They have set the direction of my work, collaborated on my papers, and improved the quality of my writing.

I'd like to acknowledge my collaborators, David, Manuel, Maverick, and Mohammad, for their advice and expertise; my labmates, Chris and Pratik, as well as Jonathan, Matt, Peter, Tiffany, and Rijnard, for their feedback and support; my housemates, Aurick and Evan, for putting up with me; my friends, for their friendship, board games, biking trips, cooking sessions, climbing adventures, and dance lessons; and the community, including the Parallel Data Lab, staff members, and administrative assistants.

I'm indebted to my family for their love, sacrifice, and support; my father Richard, my mother Sharon, and my sister Angela.

Contents

List of Figures

List of Listings

List of Tables

CHAPTER 1

Introduction

Computers programs are written in a variety of programming languages, which have different design paradigms. *Imperative languages* explicitly describe execution in terms of individual commands that may perform numerical computation or encode control-flow. *Declarative languages* instead express higher-level computational logic, without providing specific commands. Examples of these include languages like Java and C/C++, which are imperative, as well as Prolog and Haskell, which are declarative.

Languages can also differ in terms of their *type system*, which provides derivation rules about the validity of various language constructs. *Statically-typed languages* perform consistency checking at compile-time, when a program is converted into machine instructions, whereas *dynamically-typed languages* do so at run-time, when a program is being executed. Examples of these include languages like Fortran and OCaml, which are statically-typed, as well as JavaScript and Python, which are dynamically-typed.

Some imperative languages expose details of the underlying machine hardware, including processor registers and virtual memory. *Low-level* languages, like C and C++, require the programmer to directly manage allocated memory using *pointer* references, as opposed to *high-level* languages like Java, which executes on an abstract *virtual machine* that checks the validity of object references. Other languages, like Rust, permit manual memory management, but incorporate stronger static type properties about ownership and mutability to prevent mistakes (§2.1.1).

1.1. Memory-Safety Bugs

Perhaps due to their similarity to machine instructions, low-level imperative languages like C and C++ have found widespread use. Unfortunately, these languages are vulnerable to *memory-safety bugs*, which occur when developers access memory incorrectly. There exists three different types of these violations:

- *spatial violations*, when an access exceeds the boundaries of the intended object, e.g., *buffer overflows*;
- *temporal violations*, when an access exceeds the lifetime of the intended object, e.g., *double frees* or *use-after-frees*;
- *type violations*, when a valid access misinterprets the intended object, e.g., through incorrect *type casting*.

We show an example of each of these violations in Listing 1.1. On line 8, a buffer overflow can occur if the input data is larger than 16 B, because it is copied into the fixed-size stack-allocated buffer from line 7. On line 17, a double free can occur if the input heap-allocated data is already freed on line 13. On line 21, type confusion can occur if

```c
struct object {
    uint8_t *data;
    size_t size;
};

void spatial(struct object *obj) {
    uint8_t buffer[16];
    memcpy(buffer, obj->data, obj->size);
}

void temporal(struct object *obj) {
    if (obj->size && obj->data > obj->size) {
        free(obj->data);
        obj->size = 0;
    }

    free(obj->data);
}

size_t type_confusion(uint8_t *data) {
    return ((struct object *)data)->size;
}
```

LISTING 1.1. An example of each type of memory-safety bug: spatial, temporal, and type.

the input data is not actually of type `struct` object type, because it will be interpreted incorrectly.

Although simplified here, these bugs can be difficult to identify in practice due to various complicating factors. Examples include cross-thread accesses, where developers must also reason about concurrency, resizeable data, where developers maintain a *logical size* separate from the *allocation capacity*, and type casting, where developers explicitly reinterpret data encapsulated within a *sum* or *inherited* type.

1.1.1. Exploitation. Developers can be tempted to treat memory-safety bugs as harmless if they do not appear to impact program execution. For example, if the compiler inserts extra padding in the *stack frame*, which tracks execution of the current function, then a limited *buffer overflow* may have no impact. Similarly, if no subsequent heap allocations occur, then a *double free* may not affect program execution.

However, it is dangerous to assume that these conditions are complete and always satisfied, because they rely on *undefined behavior* that is not specified by the underlying programming language, compiler, or runtime. In practice, memory-safety bugs can have a variety of different effects, ranging from no impact, to memory leaks or unexploitable crashes, to exploitable behavior. But, these can change if the program is rebuilt for a different platform, a shared library is upgraded, or a different compiler is used.

Address	Description
rbp + $8 * (n - 5)$	Input Argument n
rbp + ...	...
rbp + 16	Input Argument 7
rbp + 8	Return Pointer
rbp	Saved rbp
rbp - 8	Local Variable 1
rbp - ...	...
rbp - $8 * m$	Local Variable m

TABLE 1.1. Stack frame under the System V AMD64 Application Binary Interface (ABI).

Offset	Description
0	PREV_INUSE Flag
1	IS_MMAPED Flag
2	NON_MAIN_ARENA Flag
3	Allocation Size (n)
8	Allocated Payload

TABLE 1.2. In-use memory chunks under the glibc allocator.

Attackers can take advantage of memory-safety bugs to deliberately execute unintended behavior, by adversarially manipulating the program using *exploits*. This is especially problematic for programs that interact directly with untrusted remote data, such as network services and web browsers, which have a significant *attack surface*.

1.1.1.1. *Buffer Overflow.* One traditional approach for doing so, known as *stack smashing* [140], takes advantage of a buffer overflow bug to corrupt the *return pointer* stored in the current stack frame. Depending on the underlying hardware platform, the processor may automatically push this pointer before every function call, in order to record the address where execution should resume after returning from the call. In Table 1.1, we show the layout of a single stack frame on x86_64 Linux, which use the rbp register to track the stack *base pointer* for the current frame. Observe that the return pointer is located above local variable allocations, as are the seventh and successive input arguments (if present). Since addresses grow upwards, overflows of local variables can corrupt these values.

If the attacker knows the memory layout of the program and can control the contents of the stack buffer, they may even be able to manipulate it to directly execute arbitrary code. This is achieved by placing machine instructions (*shellcode*) within the buffer, and carefully sizing the overflow to overwrite the return pointer with the address of the buffer.

Offset	Description
0	`PREV_INUSE` Flag
1	`IS_MMAPED` Flag
2	`NON_MAIN_ARENA` Flag
3	Allocation Size (n)
8	Forward Chunk Pointer
10	Backward Chunk Pointer
18	Forward Next-Chunk-Size Pointer (optional)
20	Backward Next-Chunk-Size Pointer (optional)
n	Allocation Size (n)

TABLE 1.3. Freed memory chunks under the glibc allocator.

1.1.1.2. *Double Free.* Another approach, known as *fastbin corruption*, takes advantage of a double free bug to corrupt the state of the system memory allocator. Although implementation-dependent, many Linux systems use the GNU C Runtime Library (glibc), which uses a derivative of the *dlmalloc* [111] memory allocator that has well-documented [56] behavior.

Individual memory allocations are modeled using *chunks*, which contain embedded metadata about the status of each allocation, as shown in Table 1.2. To compensate, the size of each user-requested allocation is increased by 8 B, which is used to record the size of the allocation, as well as various binary flags about whether this chunk follows an in-use chunk, has been allocated separately using `mmap`, or is part of a non-main heap. Large allocations that exceed a configurable threshold are satisfied directly using `mmap`, without checking existing free chunks.

When freed, the payload within each chunk is reused to store additional metadata that forms a linked list, as shown in Table 1.3. A copy of the allocation size is placed at the end of the chunk to allow iteration in either direction. These lists, known as *bins*, allow fast memory allocation and are checked as follows:

(1) *tcache*: A per-thread singly-linked list of same-size chunks, which is checked first only for an exact match.
(2) *fast bins*: Multiple singly-linked lists of same-size chunks, typically of up to 160 B, that are checked next and not coalesced with other free chunks.
(3) *unsorted bin*: A list of unsorted chunks that is checked next.
(4) *small bins*: Multiple doubly-linked lists of same-size chunks, which are accessed in insertion order and coalesced with other small chunks.
(5) *large bins*: Multiple doubly-linked lists of ranges of chunk sizes, typically of at least 1024 B, which are sorted in size order and coalesced with other large chunks. Free chunks from earlier bins are consolidated and moved when this bin is checked.

Many heap exploits [26, 145] rely on a combination of double frees and use-after-free bugs to corrupt these linked-list pointers inside the heap allocator metadata, which can allow attackers to forge chunks, overwrite arbitrary memory, and even execute arbitrary code. In Listing 1.2, we show a simplified example of a double free bug that corrupts

```
1 void doublefree_fastbin() {
2     void *a = malloc(8), *b = malloc(8), *c = malloc(8);
3     free(a);
4     free(b);
5     free(a);
6     a = malloc(8);
7     b = malloc(8);
8     c = malloc(8);
9 }
```

LISTING 1.2. A simplified example of fast bin corruption in the glibc memory allocator via a double free bug.

Offset	`Derived1`	`Derived2`
0	typeinfo for `Derived1`	typeinfo for `Derived2`
8	`Derived1::print()`	`Derived2::print(int)`

TABLE 1.4. Layout of virtual tables for `Derived1` and `Derived2`.

the state of the fast bin, by freeing the allocation stored in `a` twice (lines 3, 5). At this point, the fast bin for allocations of size 8 B contains two copies of the chunk referenced by `a` on its list. After additional allocations are performed (lines 6–8), the pointers `a` and `c` end up referring to the same chunk, which is not valid behavior. However, newer glibc implementations have incorporated more runtime checks to detect heap corruption, which is why the intervening free (line 4) is needed to evade a runtime check.

1.1.1.3. *Type Confusion.* Finally, one last approach uses a common type confusion bug to execute incorrect code. These bugs can occur anywhere an input datatype is improperly casted to another datatype without proper checking, which misinterpret data stored within the datatype. It is particularly problematic when arbitrary data is misinterpreted as a pointer, which can result in an out-of-bounds memory load/store, or even worse, as a function pointer, which can result in arbitrary code execution.

In C++, this type of bug can occur frequently in programs with complex inheritance hierarchies, because it is always valid to cast upwards from a derived to a base type, but not necessarily downwards from a base to a derived type. As a result, performing such a cast without checking can misinterpret an object of different type, which is particularly risky because C++ objects can contain a pointer to a class-specific `virtual table` that encapsulates a *virtual dispatch table* of function pointers.

We show an example of this in Listing 1.3, where a simple program allocates and refers to an object of derived type using its base type (line 19). The structure of the virtual table for both derived types is shown in Table 1.4. However, the program subsequently performs an incorrect cast (line 21), and misinterprets the object as an object of the opposing derived type. As a result, despite appearing to call the `Derived1::print()` function, the virtual table pointer for object `d2` still refers to the virtual table for `Derived2`, which is the actual function that is called, but with the incorrect type signature.

```
1  #include <iostream>
2
3  class Base { };
4
5  class Derived1 : public Base {
6  public:
7      virtual void print() {
8          std::cout << "Derived1" << std::endl;
9      }
10 };
11
12 class Derived2 : public Base {
13 public:
14     virtual void print(int x) {
15         std::cout << "Derived2: " << x << std::endl;
16     }
17 };
18
19 void vtable_confusion() {
20     Base *d2 = new Derived2();
21     static_cast<Derived1 *>(d2)->print();
22     delete d2;
23 }
24
25 int main () {vtable_confusion();}
```

LISTING 1.3. A simplified example of type confusion in C++ due to an unsafe downcast.

Pointer	Object Boundary	Lifetime Identifier	Type Signature
`buffer` (line 8)	[`&buffer`, `&buffer + 16`)	...	...
`obj->data` (line 17)	...	1	...
`data` (line 21)	...	...	`uint8_t`

TABLE 1.5. Metadata table that encodes object boundary, lifetime, and type information for vulnerable pointers from Listing 1.1.

1.2. Eager Checking

Past work (§2.1.2) has explored various approaches for eagerly detecting these memory-safety bugs. These designs generally instrument the program to maintain additional safety metadata, and insert runtime checks to verify the validity of each memory operation before it occurs. If a failure is detected, the program can be immediately terminated, or allowed to continue execution after emitting a warning message.

One common approach tracks these metadata on a per-pointer basis, by recording *provenance information* about the boundary, lifetime, and type signature of the underlying object when each pointer is created. In Table 1.5, we show example metadata for the three vulnerable pointers from Listing 1.1. This would detect the spatial violation on

Address	Value
`rbp + 8`	`&main`

TABLE 1.6. Integrity table that stores values of protected pointers from Listing 2.1.

line 8, by checking that the value of the `buffer` pointer is within range of the underlying object boundary upon dereference. Similarly, this would detect the temporal violation on line 17, by assigning a global monotonic lifetime identifier to each memory allocation, zeroing this identifier when each allocation is freed, and checking that the lifetime identifier for each pointer matches that of its underlying allocation upon dereference or free. Finally, this would also detect the type confusion on line 21, by checking that the underlying object type signature is compatible upon each type cast.

1.3. Lazy Mitigations

Unfortunately, eager checking imposes significant compatibility issues and performance overhead, because e.g., these metadata must be maintained for all pointers, and checked before every pointer dereference, deallocation, and cast operation. Lazy mitigations (§2.1.3) propose an alternative approach that allows memory-safety bugs to occur, but makes it difficult to use them to exploit programs. In effect, this trades-off precision for performance.

One common approach is *control-flow integrity* (§2.4), which protects the values of control-flow pointers. Attackers are allowed to corrupt them via memory safety or other bugs, but integrity checks are performed before every dereference to verify their integrity. Although many different designs have been proposed, *pointer integrity* (§2.4.2) can do so by storing a copy elsewhere, e.g., in a memory region protected with address-space isolation (§2.2). We show a simple example of this in Table 1.6, which depicts the protected stack frame return pointer (Table 1.1) after the function `main` calls `spatial` (Listing 1.1). Although this function contains a buffer overflow, which may be used to overwrite the protected return pointer, integrity of the pointer is checked before returning, which would detect this corruption. *Data-flow integrity* (§2.5) offers similar protections for non-control data.

1.4. Our Approach

We observe that past work on control-flow integrity and data-flow integrity suffers from two major drawbacks. First, they rely on address-space partitioning to protect internal security metadata, using techniques like information hiding and software fault isolation, which can be defeated through disclosure attacks, or are incompatible with existing dynamically-loaded code. Second, many rely on static analysis to identify authorized writes to protected data, which is inefficient and imprecise, because pointer alias analysis is fundamentally undecidable [150], can fail to distinguish between distinct fields of *product types* due to a lack of *field-sensitivity*, and may approximate correct behavior by merging runtime program states without full *context-sensitivity*.

We provide evidence to support the following **thesis statement**: *Augmenting hardware with a small, secure, and efficient AppendWrite inter-process communication (IPC) primitive decouples metadata storage and policy checking from program execution, enabling new approaches for security mitigations*. This thesis is structured as follows:

- In Chapter 2, we discuss past work on memory safety and address-space partitioning. This includes security mitigations like control-flow integrity and data-flow integrity, as well as domain-specific contributions for web browsers.
- In Chapter 3, we discuss the design of our AppendWrite IPC primitive and HerQules [42] program integrity framework, which we use to develop our control-flow integrity approach. We evaluate these on the NGINX web browser and various benchmark suites.
- In Chapter 4, we discuss HerQules-DeiFIed (in submission), our extension that supports concurrent processes and threads, as well as our language-based data-flow integrity approach, which uses lightweight annotations to identify sensitive data for automatic protection. We evaluate these on the Google Chromium web browser, and also explore using AppendWrite as a generic IPC primitive for the browser Mojo IPC framework.
- In Chapter 5, we summarize our contributions and discuss future research directions.

CHAPTER 2

Background

2.1. Memory Safety

At a high-level, past work has taken one of three different approaches in addressing memory safety: preventing, detecting, or mitigating such program bugs, which trade-off in complexity, effectiveness, and performance.

2.1.1. Prevention. Since low-level programming languages like C and C++ are inherently unsafe, prevention of memory-safety bugs is difficult without significant language changes. Early work [83, 134] proposed retrofitting language type systems with additional memory management subtypes, but saw little adoption and required programs be rewritten. More recent work has focused on incorporating mutability and ownership of resources as first-class language concepts, leading to the development of modern languages like Rust [127], which is increasing in adoption. Nevertheless, it is important to note that even Rust permits unsafe behavior, which allows accidental introduction [69, 149, 199] of memory-safety bugs.

2.1.2. Detection. Another line of work has focused on detecting [170] memory-safety bugs at runtime. *Spatial memory-safe designs* perform bounds checking on pointer computation and dereference, in order to identify out-of-bounds references and accesses. *Location-based* designs [91, 92, 162] provide lower precision, because they only ensure that pointers refer to *valid objects*, unlike *pointer-* or *object-based designs*, which ensure that pointers refer to *correct objects*. However, they can have much lower performance impact, because they only need to track memory validity and/or place adjacent guard regions for each object.

Pointer and object-based designs improve precision by tracking boundaries on a per-pointer or per-object basis, respectively. The former stores boundaries either within pointers themselves, known as *fat pointers* [83, 105, 109, 134], or by storing separate metadata [63, 65, 130] using an associative map data structure. In contrast, the latter [12, 60, 100, 155, 203] stores boundaries within each object, but may need to insert additional padding to support one-past-the-end pointers that are valid when not dereferenced. Hardware-based designs reduce software overhead by offloading bounds checking and/or storage to special-purpose hardware. However, Intel's Memory Protection Extensions [28, 104] (MPX) were removed [90] after shortcomings were identified [138]. *Type sanitizers* use a similar software-based approach to detect bad type casts at runtime, by tracking additional type metadata on a per-type [113] or per-object [64, 88, 98] basis.

Temporal memory-safe designs defend against orthogonal use-after-free and double free pointer bugs, which can occur when dynamically-allocated memory is accessed after deallocation, or deallocated twice, respectively. Proposed solutions include elimination

of dangling pointers [112, 195] to deallocated memory, or performing validity checking on pointer dereference [131]. Some designs provide both spatial and temporal memory safety, whether in software [32, 135] or in hardware [59, 132, 157, 206].

2.1.3. Mitigation. Many mitigations have been proposed to defend against various memory-safety bugs, some of which have been widely deployed and are now commonplace. Stack canaries [51] are unpredictable stack frame values that are generated at function entry and checked at function exit to detect memory corruption. Address Space Layout Randomization (ASLR) randomizes the base address of the process image at load-time, which conceals the addresses of known code and data (§2.2.1). No-execute memory [179] (NX or Data Execution Prevention) allows memory to be marked non-executable, which can prevent execution of user-supplied data as code. W^X [180] takes this one step further, by preventing memory from being both executable and writable, which makes it difficult to corrupt executable code.

2.2. Address-Space Partitioning

Security mitigations may need to maintain internal metadata, which must be isolated from the remainder of the program to prevent corruption due to latent memory-safety bugs. Past work has proposed various software-based approaches, which trade-off between overhead, compatibility, and effectiveness.

2.2.1. Information Hiding. This approach has low overhead, but relies on randomization to provide probabilistic security. Runtime randomization of program content [47, 143] or layout [21, 22, 94, 179] has been used to deter exploitation and protect sensitive data. However, side-channel [81, 160, 166, 173] or other information disclosure attacks, including leakage from uninitialized memory, have been used to defeat information hiding. In response, various countermeasures have been proposed, including runtime rerandomization [24, 192, 197], execute-only memory [17, 52, 176], and guard pages, to prevent memory probing.

2.2.2. Software-Fault Isolation. In this approach [190], the program's virtual address space is partitioned into separate logical *fault domains*, and the program is modified at compile-time to prevent untrusted cross-domain access. This allows internal metadata to be placed in one isolated domain, and the remainder of the program in another. Untrusted pointers are masked such that they cannot access the address space of the isolated domain, whereas trusted pointers are unmodified. To ensure complete coverage, the program must either prohibit dynamically-loadable modules, or all such loadable modules must be modified as well. It must also ensure that program data is not executable, and that program code is not modifiable, to prevent protections from being circumvented. Past work has used this mechanism to sandbox untrusted code, such as browser-based applications [202] or plugins [159], as well as multimedia codecs and device drivers [67].

Design	Events	Recipient	Async.	Paradigm	HW Δ
FADE [74]	HW/SW	Core	×	Filter-Update	Big
FlexCore [57]	HW/SW	FPGA	×	Filter-Update	Big
Guardian Council [11]	HW/SW	μCores	✓	Filter-Map-Reduce	Big
HerQules (§3)	SW	Memory	✓	Message Passing	Small
Intel Processor Trace [7]	HW/SW	Memory	✓	Filter-Update	–
Log-Based Architectures [45]	HW/SW	Core	✓	Filter-Update	Big

Table 2.1. Comparison of hardware-based fine-grained instruction monitoring designs.

2.2.3. Disjoint Address Spaces. Alternatively, the program can switch between disjoint virtual address spaces, instead of partitioning a single virtual address space. One approach dynamically reconfigures [66, 121] the memory management unit (MMU) to modify the virtual-to-physical page mapping directly, whereas another [104, 123, 146] relies on hardware virtualization and Second Level Address Translation (SLAT) to modify the guest-virtual to host-physical page mapping from within the guest. However, both approaches increase overhead because processor Translation Lookaside Buffer (TLB) entries must be flushed on each address space switch.

2.3. Fine-Grained Instruction Monitoring

Instead of maintaining software-based in-process metadata, hardware-based *fine-grained instruction monitoring* [11, 45, 57, 74]) modifies the processor to generate, filter, and process execution events (e.g., retired instructions, function calls, memory accesses, etc.) directly. Proposals include adding a similarly-sized core [45, 74], an on-chip FPGA [57], or multiple microcontroller-sized cores (μCores) [11], as shown in Table 2.1. 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 processing costs. For example, under FADE [74], 84–99% of all events must ultimately be discarded as irrelevant. Guardian Council [11] 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. Synchronous designs stall the processor until this processing is complete, whereas asynchronous designs enforce a system-level security boundary at e.g., system calls.

Recent Intel processors include built-in support for Processor Trace [7] (PT), which asynchronously stores processor execution events in memory. These events can be fixed-function hardware-defined events, or optionally software-defined events via the `PTWRITE` instruction. However, although it has been used for instruction monitoring by past control-flow integrity designs [61, 75, 85, 124], it is primarily designed for performance monitoring, and thus suffers from numerous flaws. Built-in hardware-defined events do not provide sufficient execution context, forcing past designs to disassemble program binaries in order to reconstruct control flow, which adds complexity and overhead. Event packets can be lost or overwritten due to, e.g., performance monitoring interrupt skid [7], which defeats security. Tracing and decoding of events incur tremendous overhead,

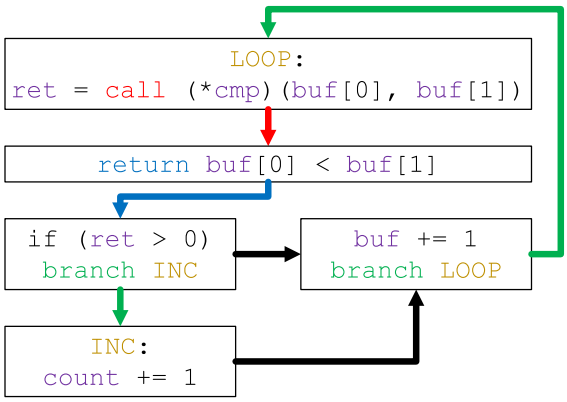

FIGURE 2.1. Control-flow graph of a small loop that counts the number of sorted (increasing) pairs in a buffer. Edges are colored, with sequential execution in **black**, direct forward edges in **green**, indirect forward edges in **red**, and backward edges in **blue**.

measuring over 500x [124] on the SPEC benchmarks. As a workaround, past designs must limit which events are monitored (e.g., monitoring only 7-10 system calls—`execve`, `mmap`, etc.), which greatly reduces effectiveness.

2.4. Control-Flow Integrity

Control-flow integrity [10, 34] (CFI) mitigates memory-safety bugs by verifying the integrity of program control-flow transitions, which can be represented using a control-flow graph (CFG). 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 that resume execution of the calling function. We show the CFG for a small loop in Figure 2.1.

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. 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 corresponding 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.

2.4.1. Coarse-Grained CFI. These designs approximate the control flow of a program using a limited number of equivalence classes. Some use just one equivalence class for all address-taken functions, including Microsoft's Control Flow Guard [5] (MSCFG) and Intel's Control Enforcement Technology's [3, 165] (CET) Indirect Branch Tracking (IBT). ARM's Branch Target Identification (BTI) supports up to three equivalence classes, based on the type of the underlying branch instruction. Others form equivalence classes

based on callee arity [181] or type [204, 205], including modern Clang/LLVM CFI [48], which redirects indirect calls through aligned jump tables for each equivalence class.

This approach is widely-deployed due to low overhead, but is vulnerable to code-reuse attacks [27, 36, 55, 76, 77], like return-oriented programming [41, 164] and jump-oriented programming [27]. MSCFG has been included since Windows 8.1 [177], though it was vulnerable to a now-patched design flaw [25], and is to be replaced [148] by CET in future Windows 10 releases. Google Chrome and certain Android devices [6, 182, 183] are built with modern Clang/LLVM CFI enabled.

2.4.2. Fine-Grained CFI. These designs improve precision by tracking valid call targets at runtime, and performing a *context-sensitive* analysis at each call site. Approaches include recording object origin [103], pointer values [42, 107, 108, 126]), or call paths.

Path-sensitive methods examine callers on the call path to each check, but since the total number of program paths grows exponentially, approximation or hardware acceleration is needed to make this approach feasible. Mechanisms used by past work include hardware-accelerated path recording (e.g., Intel Last Branch Record [186] [LBR], or Intel Processor Trace [61, 75, 85, 124] [PT]), call path merging [137, 186], and/or only checking at certain sensitive system calls (e.g., `sigaction`, `mmap`, `mprotect`, etc.) [61, 75, 85, 124, 186]. In response, attacks [71] have evolved to target the inherent undecidability [150] of pointer aliasing, which impedes the generation of an accurate control-flow graph.

2.4.2.1. *Pointer Integrity.* This is a state-of-the-art fine-grained approach that protects the *values* of sensitive pointers, rather than partitioning program callers and callees based on expected control-flow. It avoids analyzing the program control-flow graph or approximating program paths, while ensuring maximum context sensitivity.

Code Pointer Integrity [107] (CPI) places control-flow pointers in an isolated *safe pointer store* (SPS), and redirects loads/stores of the original pointer accordingly. Since control-flow pointers may be accessed indirectly through data pointers, it uses a recursive definition to identify sensitive pointers, all of which are placed in the safe store. To prevent overflows of sensitive pointers from corrupting memory, the safe store also tracks object boundaries, and checks pointer dereferences to ensure that they are in-bounds. Return pointers are protected using a separate *safe stack* (§2.4.3). Initially, information hiding (§2.2.1) was used to protect both the safe store and the safe stack, but was later shown [70, 78] vulnerable to disclosure attacks, and fixed [108] by using software fault isolation (§2.2.2) instead. We observe in §3 that their final design has moderate overhead but causes many programs to crash or hang, and cannot be easily composed with existing binaries without adding abstraction (and hence overhead) to the safe store.

Cryptographically Enforced CFI [126] (CCFI) computes a message authentication code (MAC) for each sensitive pointer, which is stored in adjacent memory and rechecked upon each pointer dereference. Sensitive pointers include control-flow pointers, as well as indirect C++ *virtual table (vtable) pointers* used to locate virtual class methods. The MAC computation applies a single round of the Advanced Encryption Standard (AES) block cipher over various properties of each sensitive pointer, including address, value, type, and class signature, using hardware-accelerated AES instructions to improve performance. To prevent forgery and replay attacks, the expanded AES key is stored in eleven reserved XMM registers, and a random offset is injected into each stack frame to

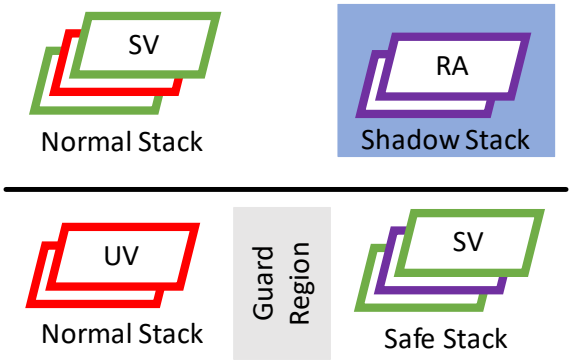

FIGURE 2.2. Comparison of shadow stacks (top) and safe stacks (bottom). Safe variables are shown in green, unsafe variables in red, and return addresses in purple.

act as a nonce. We observe in §3 that their approach has significant overhead, uses a non-standard AES construction[1], breaks calling conventions with all existing code, cannot invalidate MACs to detect use-after-free bugs, uses a fixed secret key of zero, and exhibits significant false positives.

Other related work, in both academia [117, 118] and industry [128], has implemented pointer integrity using ARM's Pointer Authentication (PA). However, Apple's design [128] is a cryptographic MAC-based approach with lower precision than CCFI. For compatibility reasons, it omits the pointer address from the MAC computation, which permits replay attacks. As a workaround, it supports a separate *discriminator* input used as a nonce, but Apple's implementation uses a constant discriminator of zero for function pointers and C++ virtual table pointers. This design is also specific to Apple's smartphones and iOS software, which are not accessible externally for development and testing, and was shown [16] vulnerable to a now-patched flaw that allowed MAC forgery.

2.4.3. Return Pointers. Many instruction set architectures (ISAs) automatically save and restore a stack-allocated *return pointer* at each function call and return, which can be corrupted. For example, in a multi-threaded program, even if a control-flow integrity check is performed before each function return, a concurrent thread could corrupt the pointer after the check but before the return, in a deliberate race condition. Many coarse-grained designs (§2.4.1) only match call-return pairs when checking return pointers, which is vulnerable to code-reuse attacks and further decreases effectiveness.

Instead, *shadow stacks* [35, 54, 73, 165] place return pointers in a separate stack, which helps prevent corruption. Software-based approaches, such as Microsoft's Return Flow Guard [23] (MSRFG), hide this stack using information hiding (§2.2.1), but are vulnerable to disclosure attacks. Instead, hardware-based approaches, such as Intel's CET, place this stack on special memory pages that are otherwise inaccessible to software.

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

```
1  void work(int *input) {
2    int v, *ptr = &v;
3    *ptr = 4; // definition 1
4    do_stuff(v);
5    *ptr = 12; // definition 2
6    *input = 38; // definition 3
7  }
```

LISTING 2.1. Sample C++ code protected by data-flow integrity.

Variable	Last Definition
v	1

TABLE 2.2. State of the reaching definitions table used by data-flow integrity at line 4 of Listing 2.1.

Safe stacks, developed as part of Code Pointer Integrity [107], propose an alternative software-based approach, by moving return pointers and objects that static analysis determines cannot overflow onto a separate stack that is protected by information hiding (§2.2.1). They can defeat some attacks [50] and are supported by the Clang/LLVM [48] compiler, which inserts a runtime guard region to detect linear overflows. Nevertheless, despite low overhead, safe stacks are still vulnerable [78] to information disclosure attacks, and were accordingly disabled [184] in Google Chrome. Figure 2.2 provides a comparison of safe and shadow stacks.

2.5. Data-flow Integrity

Memory-safety bugs can alter program behavior without changing control-flow pointers. Non-control-data attacks [38, 44, 96, 97, 158] can modify program control-flow by, e.g., altering the inputs of conditional expressions used in branches, or modifying system call arguments, which are not protected by CFI. Data-flow integrity [39] is an analogue of control-flow integrity that verifies the integrity of program data. Subsequent work has applied it to protect kernel access control data [168], or developed hardware implementations [169] with better performance.

Traditionally, static pointer alias analysis is used to generate equivalence classes that contain valid *reaching definitions* for each memory location, and a runtime definitions table (RDT) is used to track the last definition for each equivalence class. Stores update the corresponding table entry, whereas loads verify that the last definition is indeed in the corresponding set. This table must be protected from corruption, typically using software fault isolation (§2.2.2). Precision is affected by the underlying pointer analysis; as originally defined, it is intra-procedurally control-flow-sensitive and inter-procedurally context-insensitive, which can result in false negatives. In addition, both analyses are not *field-sensitive*, and cannot distinguish between different fields in a composite type.

We show a small example of this in Listing 2.1, which defines a variable v twice, on lines 3 and 5, and uses it once, by passing its value to a function call on line 4. The

program contains three definitions, which are assigned identifiers 1–3. Of those, the first two are placed within the equivalence class for variable v. The state of the reaching definitions table immediately before the call is shown in Table 2.1, where a runtime check is performed to verify that the last definition for v is within its equivalence class before proceeding with the call.

2.6. Browser Security

Past work has explored various aspects of browser security, including overall architecture [18, 151], content separation [102], extension support [37], library sandboxing [133], JavaScript bindings [30], site isolation [43, 152], and userdata protection [68], as well as accelerated runtimes like WebAssembly [87, 99, 115, 147, 193, 194]. Other work has focused on finding browser bugs using fuzzing or static analysis [31], in JavaScript engines [15, 86, 89, 95, 114, 144, 188, 191] and native code bindings [30, 62].

CHAPTER 3

Securing Programs via Hardware-Enforced Message Queues

Various runtime defenses have been developed to detect memory-safety bugs or mitigate corruption (§2.1); e.g., by tracking allocation boundaries [12, 63, 105, 130], temporal identifiers [131], tainted values [200], or control-flow pointers (§2.4). However, these defenses need to maintain *runtime metadata* about memory, which must be protected from unintended access. Past work has explored various software-based address-space partitioning mechanisms (§2.2), but they impose significant overhead, reduce compatibility, or rely on hiding information. Alternatively, hardware-based instruction monitoring approaches (§2.3) do not suffer from these drawbacks, but do impose significant hardware 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 (CFI)* (§2.4), 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 [71, 150], and incorporate novel compiler optimizations. Compared to past work [48, 107, 108, 126], 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 3.2. We evaluate the correctness, effectiveness, and performance of our designs on the SPEC CPU2006 [93], SPEC CPU2017 [33], RIPE [154, 196], and NGINX [174] 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:

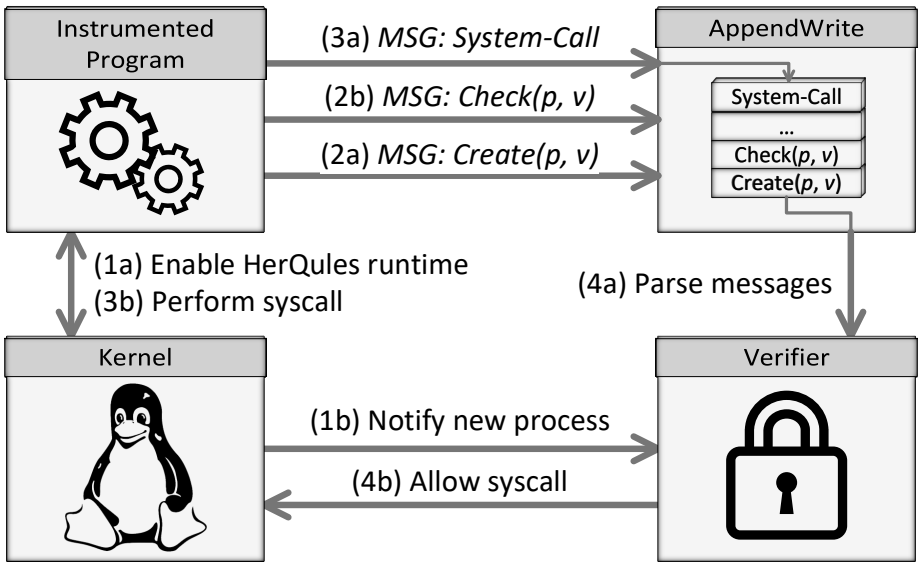

FIGURE 3.1. Overview of interactions under HERQULES

- 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 (§3.1.3), which we implement in both an FPGA and in hardware at very low cost (§3.2.1).
- We use AppendWrite to build HERQULES, a framework for implementing efficient integrity-based execution policies, which executes asynchronously to maximize performance (§3.1.2).
- We show how HERQULES supports a variety of security policies, and perform an experimental case study on control-flow integrity (§3.3.1), demonstrating a significant improvement in correctness, effectiveness, and performance over prior work (§3.4).
- We release our system as open-source at `https://github.com/secure-foundations/herqules`.

3.1. Design

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 (§3.1.2), 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.

Figure 3.1 highlights the four main components of HerQules, providing an overview of the interactions between the components.

(1) At compile-time, our **compiler pass** automatically instruments the program to send policy-dependent messages when policy-relevant events occur.
(2) 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)*.
(3) 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)*.

3.1.1. 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 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. 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.

3.1.2. 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

IPC Primitive	Append-Only	Async.	Cost	Time (ns)
Message Queue	✓	×	System Call	146
Named Pipe	✓	×	System Call	316
Socket	✓	×	System Call	346
Shared Memory	×	✓	Memory Write	12
Light-Weight Contexts	✓	×	System Call	2010 [121]
xMP	✓	×	System Call	1000 - 5000 [146]
AppendWrite-FPGA	✓	✓	Memory Write	102
AppendWrite-μarch	✓	✓	Memory Write	< 2

TABLE 3.1. Comparison of existing IPC primitives, grouped by type (*top*: software-based, *center*: hardware-based, *bottom*: proposed), showing message send times.

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.

3.1.3. 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 (§3.1.3.1), whereas AppendWrite-μarch adds append-only memory buffers to the microarchitecture (§3.1.3.2), which we model in software and validate in simulation (§3.4.3.1).

Existing IPC mechanisms either perform poorly or lack message integrity, as shown in Table 3.1, which includes the average runtime of a micro-benchmark that repeatedly sends messages. Primitives that 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 (i.e., kernel page-table isolation [84, 120]), 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 [66, 121, 146] mechanism costs 2010 ns [121] 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 (§3.4.5), 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 to make these resources available to user-space programs, which entails trusting a large code-base (e.g., Data Plane Development Kit [2]), and the presence of special hardware features, like an IOMMU and PCI Express (PCIe) Access Control Services.

3.1.3.1. *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 (§3.4.3.1).

3.1.3.2. *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 region* (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 (§3.2.3), 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.

3.2. Implementation

Below, we describe the implementation details of HerQules' four main components (Figure 3.1): the AppendWrite IPC primitive, compiler instrumentation, a kernel module, and a verifier process.

3.2.1. 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.

3.2.1.1. *Accelerator.* We implement AppendWrite-FPGA using a custom Accelerator Functional Unit [119, 139] (AFU) on an Intel Arria 10 [185] GX Programmable Accelerator Card (PAC), a PCIe-based FPGA. Our logic is written in SystemVerilog/Bluespec [136], interacts with the host via the Open Programmable Acceleration Engine [125] (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.

3.2.1.2. *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.

3.2.2. Compiler Instrumentation. We implement our instrumentation in Clang/LLVM [110] compiler passes, which insert runtime calls to generate policy-specific messages (§3.3) and System-Call messages (§3.1.2), 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 [116] to find the earliest suitable point for sending such messages. Because each source file may be compiled individually, making inter-procedural 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 [72], which is compact and standards-compliant. To support certain wrapper functions that use a separate kernel-loaded virtual dynamic shared object (vDSO), such as `getcpu` and `gettimeofday`, we manually insert two messaging calls. 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.

3.2.3. 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* [101] and *tracepoints* [58]. 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 [201] 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. For our accelerator-based IPC primitive (§3.2.1.1), this module also updates the privileged PID register upon context switch.

3.2.4. 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.

3.3. Execution Policies

Below, we provide a case study on control-flow integrity (§3.3.1), which uses separate protection mechanisms for different types of program control-flow transitions, such as forward-edge transitions (§3.3.2, §3.3.2.1) and backward-edge transitions (§3.3.2.2, §3.3.2.3). We also sketch designs for memory safety (§3.3.3) and other policies (§3.3.4).

3.3.1. Control-Flow Integrity. We enforce control-flow integrity using a fine-grained *pointer integrity* (§2.4.2.1) policy, which protects the *values* of control-flow pointers by checking against a copy stored in the verifier via AppendWrite. Unlike other fine-grained

Design	Mechanism	Precision	Use-After-Free	Compatibility	Performance
Clang/LLVM CFI [48]	Language-level Types	•	×	••	••••
CCFI [126]	Cryptographic MACs	•••	×	•	•
CPI [107]	Information Hiding	••	×	•	••••
CPI [108]	Software Fault Isolation	••	×	•	•••
HQ-CFI-SfeStk	AppendWrite	••	✓	•••	•••
HQ-CFI-RetPtr	AppendWrite	•••	✓	•••	••

TABLE 3.2. Comparison of control-flow integrity designs, grouped by precision (*top*: low, *center/bottom*: high). More • is better.

approaches, pointer integrity is maximally precise and does not suffer from the pointer aliasing undecidability [150] problem used to defeat [71] past designs. Our approach, named HQ-CFI, differs from past work on pointer integrity, which relocate [107, 108] pointers or verify cryptographic hashes [126] 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 3.2 compares our two designs against existing coarse-grained (§2.4.1) and fine-grained (§2.4.2) 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 [48], which is included in Clang/LLVM and widely-deployed. For the latter, we select Cryptographically-Enforced CFI [126] (CCFI) and Code-Pointer Integrity [107, 108] (CPI), which are state-of-the-art pointer integrity designs. Note that like past work, we do not prevent programs from creating and executing unsafe function pointers to arbitrary user input in an executable buffer, nor do we ensure that all control-flow pointers must point to valid code or data.

3.3.2. 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 §3.3.2.1.

- 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`[1].
- 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`.

3.3.2.1. *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 [178]): After program optimization, insert instrumentation on block memory operations, optimize messaging calls, and optionally inline our messaging runtime.

Our messages only support two arguments (§3.2.1), but certain messages use three (e.g., POINTER-BLOCK-COPY). Since our design currently protects user-space programs on x86_64 systems with four-level paging, we decompose the third size argument into two chunks that are stored in the upper 17 bits of the first and second pointer arguments. As an optimization, this limits manipulation of instrumented memory blocks to 16 GB in size, which is compatible with all of our benchmarks.

Initial Lowering: We examine each operation in the LLVM IR (e.g., `load`, `store`, `call`, etc), and insert runtime messaging as needed.

(1) STORE(P, V): If the stored value is a non-zero control-flow pointer (§3.3.1), send the message POINTER-DEFINE(P, V) immediately afterwards.
(2) CFI-CHECK(LOAD(P)): Send the message POINTER-CHECK(P, LOAD(P)).

[1] A memory copy that permits overlapping input/output ranges.

(3) LOAD(P): If the value of LOAD(P) is passed as a function pointer to a call or returned as a function pointer to a caller, send the message POINTER-CHECK(P, LOAD(P)) immediately before the call or return.
(4) RETURN: If this function is a complete or deleting C++ destructor, send the message POINTER-INVALIDATE(THIS) immediately before the return, where THIS corresponds to the vtable pointer at offset zero for the current object.
(5) LIFETIME-END(P): If the pointer is a control-flow pointer, send the message POINTER-INVALIDATE(P) immediately before it dies. This uses built-in lifetime analysis.

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 [14, 153], 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. This pass can execute either at compile-time (LLVM) or at link-time (gold), but by default we execute it at link-time to enable inter-procedural link-time optimizations. 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.

Instrumentation is inserted on calls to library functions, like MEMCPY, MEMMOVE, and REALLOC, that may modify function pointers inside memory blocks. We also implement special detection for memory blocks that may contain function pointers, by extending our previous function pointer detection to include composite types that contain function pointer subtypes, and checking both the input and output arguments of these library functions. However, we observe that this *strict function pointer subtype detection* fails on four benchmark functions that are inter-procedurally passed decayed pointers, which we override using an allowlist. Alternatively, strict detection can be disabled to instrument all such library calls, which is conservative and increases message traffic.

(1) MEMCPY/MEMMOVE(DST, SRC, SZ): If the destination or source may contain function pointers, and the source is not a string constant, send the message POINTER-BLOCK-COPY(DST, SRC, SZ) immediately after the copy.
(2) REALLOC(P, NSZ): Fetch the original[2] allocation size OSZ, then perform the reallocation. If it moved, send the message POINTER-BLOCK-MOVE(P, REALLOC(P, NSZ), OSZ) immediately afterwards, where NSZ is the new size.

[2]Past work behaves similarly; CPI queries the allocator for the original size, while CCFI modifies the source to recompute MACs afterwards.

(3) FREE(p): Fetch the original allocation size osz, then perform the free. Send the message POINTER-BLOCK-FREE(p, osz) immediately afterwards.

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 (§2.2.1) enabled, which shift function addresses and the value of corresponding function pointers by a runtime offset. Our instrumentation appends the address-value pair of all such global variables to a special program section, and inserts a runtime constructor function to send POINTER-DEFINE(p, v) messages on each at program startup. As an optimization, if the total number of these entries exceeds a predetermined threshold, the constructor function instead sends a special INITIALIZE-GLOBALS(o) message that contains the random offset o, which enables the verifier to directly fetch and relocate these entries using the binary.

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 [141, 142], Whole Program Devirtualization [49], and Dead Virtual Function Elimination [171]. 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 (§3.4).

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.

3.3.2.2. *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 (§2.4.3) 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).

3.3.2.3. *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.

3.3.3. 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.

3.3.4. Other Policies. More generally, HERQULES can enforce other execution policies for security, performance, or reliability. Examples include data-flow integrity [39], 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.

3.4. Evaluation

We evaluate CFI designs on the RIPE [154, 196] 1.0.5 benchmark for effectiveness (§3.4.2), and on the SPEC CPU2006 [93] 1.2, SPEC CPU2017 [33], and NGINX web server [174] 1.18.0 benchmarks for both correctness (§3.4.1) and performance (§3.4.3). All benchmarks were configured to use the musl C runtime library, and we patched SPEC to fix

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 3.3. Correctness of various CFI designs.

various memory safety (§3.4.2) and compatibility bugs. To identify the IPC primitive used by HQ-CFI, we apply the postfix -MQ for POSIX message queues (§3.1.3), -FPGA for the accelerator (§3.2.1.1), -SIM for the hardware simulator (§3.4.3.1), and -MODEL for the hardware model (§3.4.3.1).

We compare HQ-CFI against past designs from Table 3.2, 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 (§3.3.2.1). 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 (§3.4.1), 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 [106] that their code was a proof-of-concept prototype, and not fully robust.

3.4.1. 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 3.3, 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

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

Table 3.4. Successful RIPE exploits under various CFI designs, grouped by overflow origin.

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. However, 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.

3.4.2. Effectiveness. We demonstrate effectiveness of each CFI design using the RIPE [196] test suite, which contains hundreds of buffer overflow exploits. Because all programs were compiled as 64-bit binaries, we use RIPE64 [154], 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 3.4 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 over 11 years of continuous development, as both benchmarks correspond to different versions of the OMNeT++ simulator [187].

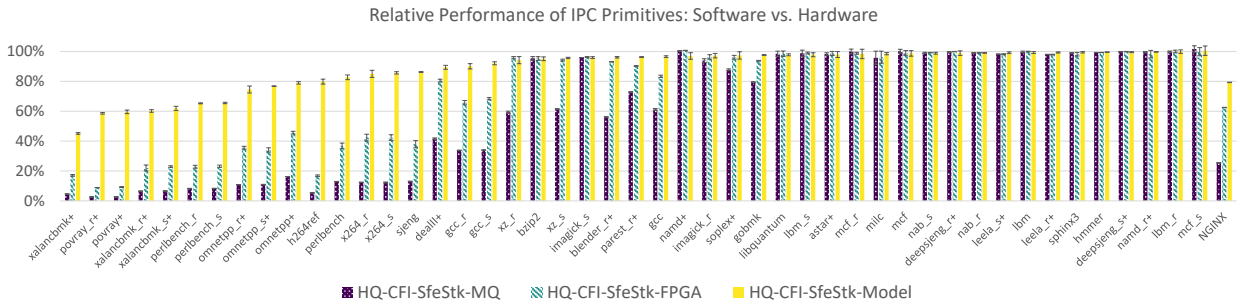

FIGURE 3.2. Relative performance of HQ-CFI using various IPC primitives, sorted on HQ-CFI-SFESTK-MODEL (left to right), on SPEC CPU2006, SPEC CPU2017, and NGINX. Suffix '+' denotes C++.

We have reported these bugs in CPU2017 to SPEC, but no changes are planned, and CPU2006 has been retired.

3.4.3. 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 [79] for 60 s. We report relative performance by computing the arithmetic mean of 3 runs, and show standard deviations using error bars.

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

Software vs. Hardware: In Figure 3.2, we compare hardware-based AppendWrite against the fastest suitable software primitive from Table 3.1 – 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

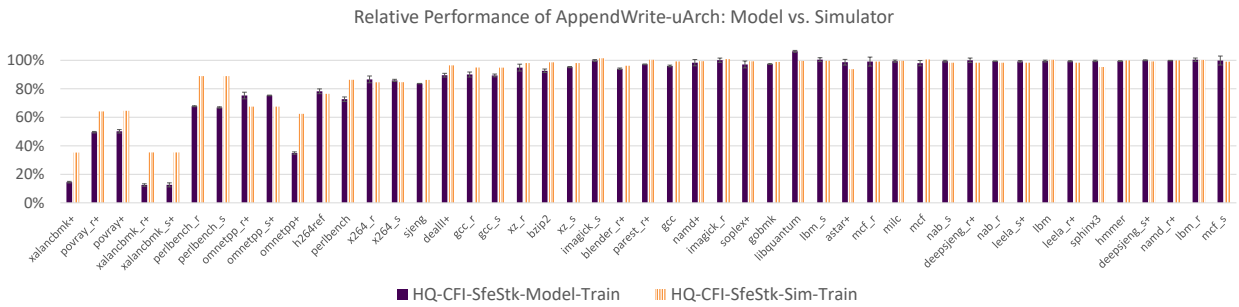

FIGURE 3.3. Relative performance of HQ-CFI using AppendWrite-μarch on the *train* input for SPEC CPU2006 and CPU2017. Suffix '+' denotes C++.

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 3.3, 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 [156], 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.

3.4.3.2. *CFI Designs.* In Figure 3.4, 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%,

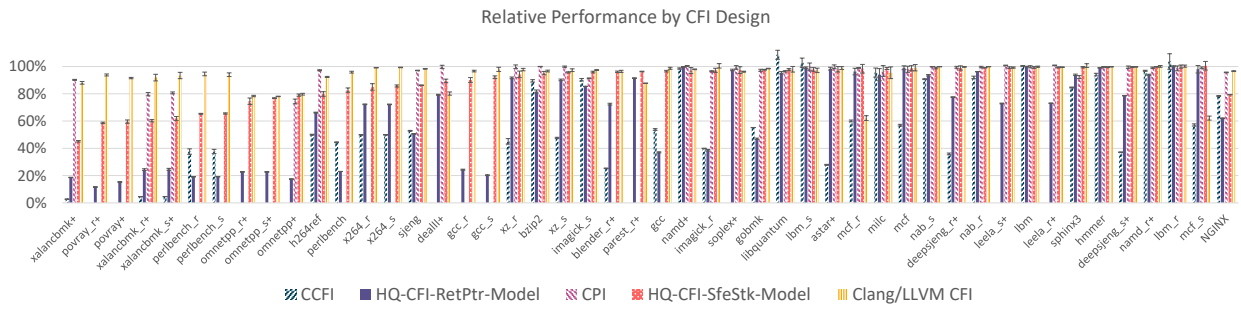

FIGURE 3.4. Relative performance of various CFI designs, sorted on HQ-CFI-SFESTK-MODEL (left to right) for SPEC CPU2006, SPEC CPU2017, and NGINX. Suffix '+' denotes C++.

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.

3.4.4. Scalability. 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 Figure 3.5, we show a breakdown of messages by type across all SPEC benchmarks for HQ-CFI-SFESTK, on a log-scale plot. Of these, 79% are POINTER-CHECK messages, and 15% are POINTER-DEFINE messages, with POINTER-INVALIDATE and POINTER-BLOCK-COPY messages amounting to 3% and 2%, respectively. All remaining messages types are negligible but non-zero, with the exception of POINTER-CHECK-INVALIDATE, which are only sent by HQ-CFI-RETPTR. A total of 60.5×10^6 messages are sent across all benchmarks.

3.4.5. Other Metrics. 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

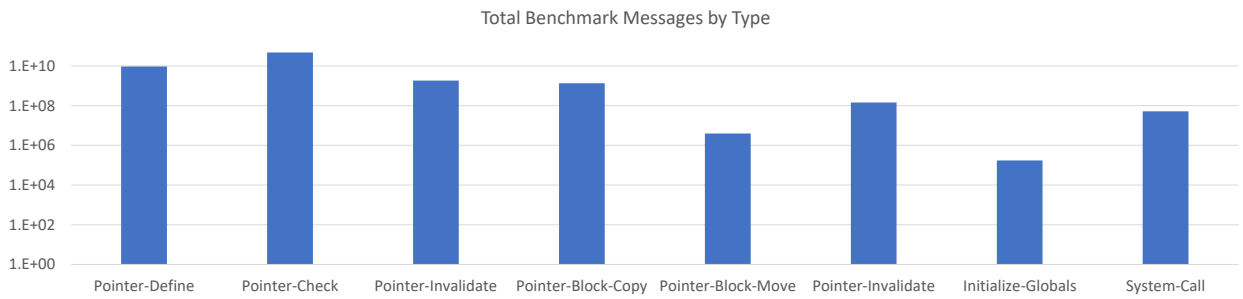

FIGURE 3.5. Total benchmark messages by type.

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

TABLE 3.5. Size of HERQULES, in approximate lines of code.

includes 14 benchmarks with zero entries, which lack control-flow pointers needing protection.

In Table 3.5, 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.

3.4.6. Discussion. Table 3.2 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.

As a research prototype, HQ-CFI is not fully optimized. Potential future improvements include modifying the kernel directly to eliminate dynamic interception overhead (§3.2.3), eliding synchronization for read-only system calls, and eliminating messages on block-level memory operations that do not contain control-flow pointers. For example, a bitvector or other data structure could be used to track the presence of control-flow pointers, which would allow certain block memory messages to be elided at runtime. Note that this does not affect the security of our design, because it only permits false positives by an attacker.

3.5. Summary

In this chapter, 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.

CHAPTER 4

Extending Program Integrity Protections to Browser Data

On mobile and desktop platforms alike, the web browser has become the most frequently-used application by billions of users around the world. Under the hood, even many seemingly-native applications are actually web-based, and rely on embedded browser web views or rendering frameworks for ease of development, e.g., Atom, Discord, Slack, etc. As a result, security of the web browser (§2.6) is pivotal in isolating sensitive local data from untrusted remote content.

However, modern web browsers are mostly written in unsafe languages like C and C++. Analysis of past security vulnerabilities in Microsoft [129] and Google Chrome [1] codebases has shown that ~70% involve memory-safety bugs. Past work has developed various mitigations for memory-safety bugs, ranging from address-space partitioning (§2.2) to control-flow integrity (§2.4). Many of these are now widely deployed, whether in hardware (e.g., Intel Control-Flow Enforcement Technology [7] [CET], ARM Branch Target Identification [82]), or in software. For example, Google Chrome on Linux is protected with Clang/LLVM [48] control-flow integrity, and development builds are tested with fuzzing and memory safety sanitizers [161, 162, 172]. Similarly, Google Chrome on Windows protects [80] return addresses with Intel CET shadow stack (§2.4.3). Nevertheless, non-control-data attacks [38, 44, 96, 97, 158] have shown that existing protections are insufficient, because memory corruption of non-control data can still affect program behavior.

Data-flow integrity [39] (DFI) mitigates these attacks by checking non-control data at runtime, much like control-flow integrity does for control data. Kenali [168] used it to protect kernel access-control data, and HDFI [169] implemented it in hardware for improved performance. Traditionally, it uses a static *reaching definitions analysis* to identify valid definitions of each variable, and dynamic instrumentation to update and check a *runtime definitions table* at affected memory stores and loads, respectively. Unfortunately, this approach (i) uses *software fault isolation* (§2.2.2) to protect the definitions table, which requires instrumenting all dynamically-loaded libraries, (ii) fails to distinguish between accesses to distinct fields of *product types* due to a lack of *field-sensitivity*, and (iii) relies on inefficient and inherently undecidable [150] pointer alias analysis that can produce false negatives. As a result, it is unclear if this approach can be effectively generalized to programs written in languages other than C, such as C++, because classes are fundamentally composite types.

In HerQules-DeiFIed, we propose an improved data-flow integrity approach that resolves these weaknesses, much like *pointer integrity* does for control-flow integrity. These improvements are twofold: first, instead of using static analysis to partition variable accesses, we track runtime *memory values*, which avoids the imprecision of pointer alias

analysis. Second, instead of storing metadata in-process using software fault isolation, we leverage emerging work on *hardware-based instruction monitoring* to transmit metadata to a separate *verifier process* that performs asynchronous policy checks. Unlike past work, our approach is the first to generalize to existing programs written in both C and C++, and includes a pre-annotated C++ standard library that provides drop-in protection for Standard Template Library (STL) data structures. We use our design to protect sensitive browser data in the open-source Google Chromium web browser, and also explore replacement of the internal Mojo IPC primitives used by both Google Chrome and Mozilla Firefox with our microarchitectural append-only ones. Our results demonstrate that HerQules-DeiFIed is feasible and has minimal overhead, with a geometric mean relative performance of 78% on browser benchmarks when used in conjunction with control-flow integrity. In comparison, prior DFI work [40] reported 104% average *overhead* on the SPEC CPU2000 benchmarks.

We summarize our contributions below:

- We develop HerQules-DeiFIed, a new DFI approach without pointer analysis that uses hardware-based instruction monitoring (§4.2.2) to generate precise runtime software-defined events when security-sensitive data is accessed (§4.2.3);
- We perform asynchronous policy checking in a separate verifier process before system call boundaries, which we extend from HerQules (§3.1.2) to support multiple processes and threads;
- We provide a simple and reusable language-based annotation mechanism that identifies sensitive data for automatic instrumentation, which includes a pre-annotated C++ standard library (§4.2.3.1);
- We protect sensitive data in the Google Chromium web browser (§4.2.4) with minimal performance and usability overhead, significantly raising the bar for browser security exploits, mitigating real Chromium security bugs (§4.4);
- We release our full system as open-source at `https://github.com/secure-foundations/herqules`, and our modifications to the Google Chromium browser at `https://github.com/ddcc/chromium`.

4.1. Motivation

Given the complexity and attack surface of web content, many web browsers rely on a layered security design. Compiler-based security mitigations like stack protection and control-flow integrity help protect unsafe code written in C and C++. Language-based checking eliminates certain memory-safety bugs, and Mozilla Firefox has begun rewriting [8] unsafe browser components into Rust, though this is still uncommon. In terms of software architecture, individual browser components are isolated [18, 152] into separate processes, which operate at different levels of trust and communicate with each other using inter-process communication (IPC). Trusted processes, such as the *browser process*, store user data and enforce content security policies, whereas untrusted processes are sandboxed and assumed to be compromised. These untrusted processes include *renderer processes* that host the JavaScript engine and interact with web content, as well as *utility processes* that interface with the GPU, network, plug-ins, storage, etc. Site isolation [152] further restricts the scope of renderer processes to specific *site origins*, which limits the

exposure of a potential compromise. As a result, achieving a full system exploit typically requires a combination of at least two different bugs: one to compromise a sandboxed process, and another to escape the sandbox.

In HerQules-DeiFIed, we significantly raise the bar for these exploits by developing a data-flow integrity design that protects *security-sensitive data* in all web browser processes. A simple compile-time annotation is used to mark data for automatic runtime protection. We identify two categories of browser data that should be protected; namely, *internal browser security data* and *private user data*, for which we supply Chromium annotations. All told, we annotate 244 different variables and 38 different source classes (§4.2.4), as the breakdown of these annotation variables and classes in Table 4.1 shows. Our runtime protection prevents attackers from corrupting browser data that is needed to exploit or expose past security bugs (§4.4.1).

4.2. Design

We design our data-flow integrity approach around a hardware-based AppendWrite primitive that transmits software-defined runtime events whenever sensitive program data are accessed, and we asynchronously check these events in a separate verifier process before system calls. We describe our threat model in §4.2.1, our system architecture in §4.2.2, our data-flow integrity protections in §4.2.3, and our browser data protections in §4.2.4, and finally our Mojo IPC improvements in §4.2.5.

4.2.1. Threat Model. Our approach protects the integrity, but not confidentiality, of sensitive program data. We assume that applications begin execution in a benign state, but may contain bugs that allow attackers to read or write arbitrary process memory, including side-channel attacks. However, writes are subject to page table permissions, which prevents modification of read-only code or data, and no modification of processor registers is permitted. We also assume that applications are already protected with control-flow integrity, and that the operating system enforces appropriate kernel security boundaries.

4.2.2. System Architecture.

4.2.2.1. *Maintaining Security Metadata.* To improve precision, security mitigations maintain metadata about program execution. For example, control-flow integrity can add context sensitivity by tracking runtime call paths [61, 75, 85, 124, 137, 186]. However, this trades off between effectiveness and performance, because metadata must also be protected from unintended access. Many past software-based approaches rely on in-process memory partitioning, using *information hiding* (§2.2.1) or *software fault isolation* (§2.2.2), but can be defeated by disclosure attacks or are incompatible with existing dynamically-loaded libraries.

Instead, emerging hardware-based instruction monitoring designs (§2.3) focus on software-defined events, which can be generated precisely where needed, and only require hardware event logging support. Event processing can also occur asynchronously before, e.g., the next system call, instead of the next instruction. These designs include the PTWRITE extension to Intel Processor Trace [7], and the AppendWrite primitive from HerQules [42] (§3.1.3). We identify the need for an abstract AppendWrite operation that

Class Name	Instances	Depth	Aggregated Datatypes
ClientSecurityState	4	0	bool, CrossOriginEmbedderPolicy, IPAddressSpace, PrivateNetworkRequestPolicy
Component	2	0	int
CrossOriginEmbedderPolicy	4	0	string, CrossOriginEmbedderPolicyValue
CrossOriginOpenerPolicy	4	0	string, CrossOriginOpenerPolicyValue
GURL	4	0	bool, string, unique_ptr, Parsed
OptionalStorageBase	1	0	bool
Parsed	10	1	bool, Parsed*, Component
StructPtr	1	0	unique_ptr
Time	1	0	int64_t
Address	7	1	string, string16, vector
CanonicalCookie	12	0	bool, string, CookiePriority, CookieSameSite, CookieSourceScheme, Time
ChildProcessSecurityPolicyImpl	13	2	bool, int, map, unique_ptr, vector, BrowserContext*, ResourceContext*, BrowsingInstanceId, FileSystemType, GURL, IsolatedOriginSource, Optional
CommandLine	5	1	map, string, vector, CommandLine*
CompromisedCredentials	4	0	string, string16, CompromiseType, Store
CookieManager	2	0	unique_ptr, CookieStore*
CookieMonster	10+	2	bool, size_t, map, multimap, set, string, unique_ptr, vector
CookieSettings	4	1	bool, set, string
CorsURLLoaderFactory	11	2	bool, int32_t, set, unique_ptr, Optional, OriginAccessList*, TrustTokenRedemptionPolicy
CredentialInfo	4	1	string16, CredentialType, Optional
CreditCard	7	0	int, string, string16, RecordType
EmailInfo	1	0	string16
Feature	2	0	char*, FeatureState
FeatureList	4	1	bool, map, string, FeatureState
IsolationInfo	4	0	bool, Optional, RedirectMode
NavigationRequest	4	2	bool, ClientSecurityStatePtr, OptInOriginIsolationEndResult, Optional, WebSandboxFlags
Origin	1	1	Optional
OriginAccessEntry	9	0	bool, uint16_t, string, CorsDomainMatchMode, CorsPortMatchMode, CorsOriginAccessMatchPriority
OriginAccessList	1	3	map, string, vector, MapType
PasswordForm	13	0	string, string16, GURL, Scheme, Store, SubmissionIndicatorEvent, Type
PhoneNumber	5	0	string16
RenderFrameImpl	2	0	bool, int
RenderViewImpl	1	0	int32_t
SchemeHostPort	3	0	uint16_t, string
SecurityState	9	2	bool, int, map, set, string, BrowsingInstanceId, CommitRequestPolicy
SiteForCookies	1	0	bool
SiteInfo	4	0	bool, GURL
SiteInstanceImpl	12	0	bool, int32_t, size_t, RenderProcessHost*, ProcessReusePolicy, AgentSchedulingGroupHost*, SiteInstanceProcessAssignment, GURL
WebUIImpl	1	0	int

TABLE 4.1. Breakdown of protected per-class data in Chromium, showing total variable instances, max template expansion depth, and aggregated datatypes. These are split between custom datatypes (top) and other classes (bottom).

can reliably append events to an otherwise-immutable memory buffer, which must satisfy the following functional properties that are not fully-supported by past work:

(1) each processor core can be configured with an event buffer;
(2) each event transmits a user-defined message of specified length;
(3) events are appended atomically; and
(4) the event buffer is otherwise immutable for the transmitting core.

4.2.2.2. *Bounding System Calls.* Execution events are transmitted using AppendWrite from *protected applications* to a separate *verifier process*, which implements our data-flow integrity policy, as shown in Figure 4.1. These events are sent before certain memory operations, and may contain evidence of data corruption. Because violations may not be detected immediately, we establish a security boundary at system calls to prevent a compromised application from performing malicious behavior. This is a common approach used by past work; e.g., for control-flow integrity [61, 75, 85, 124, 186] or taint tracking [53, 200].

However, unlike past work, our kernel module pauses execution of **all system calls** from protected applications until our verifier confirms that no checks have failed. This provides more comprehensive protection than protecting only system calls deemed to be high risk *a priori*. To avoid imposing a round-trip on each system call, we rely on *bounded asynchronous validation* from HerQules (§3.1.2), which we extend to support multiple processes and threads. Protected applications transmit a special System-Call event before system calls, which indicates to the verifier that no events remain before the next system call (step 4). Processing of this event is pipelined against the privilege transition of the system call, which minimizes overhead (steps 5–6). Applications must already be compromised to forge this event, which would be detected earlier by control-flow integrity or our data-flow integrity checks.

4.2.2.3. *Supporting Multiple Threads.* Our design must support complex applications, like web browsers, that have a multi-threaded and multi-process architecture, whereas HerQules (§3.1) does not. Within a multi-threaded process, each thread executes independently, but all threads share the same address space. One possible messaging design could allocate a unique per-thread event buffer, but this would impose significant memory overhead and require an ordering mechanism for global events. Instead, we synchronize system calls individually for each thread using a unique per-process event buffer, which provides implicit event ordering. Every System-Call event contains the identifier of the transmitting thread, so that the verifier and kernel know which thread to resume if no checks fail. However, retrieving thread identifiers typically involves a system call, which presents a recursive problem. A naive solution could place the `gettid` system call on an allowlist, and execute it before transmission of each System-Call event, but this would effectively double the program's system calls.

Instead, we observe that these identifiers are already cached within *thread-local storage* by many C runtime libraries, and retrieved by the `gettid` library function. Using this value directly would be unsafe, because the underlying address space is shared across all threads and potentially vulnerable to memory corruption. We protect cached thread identifiers with our data-flow integrity mechanism (§4.2.3), which involves transmitting

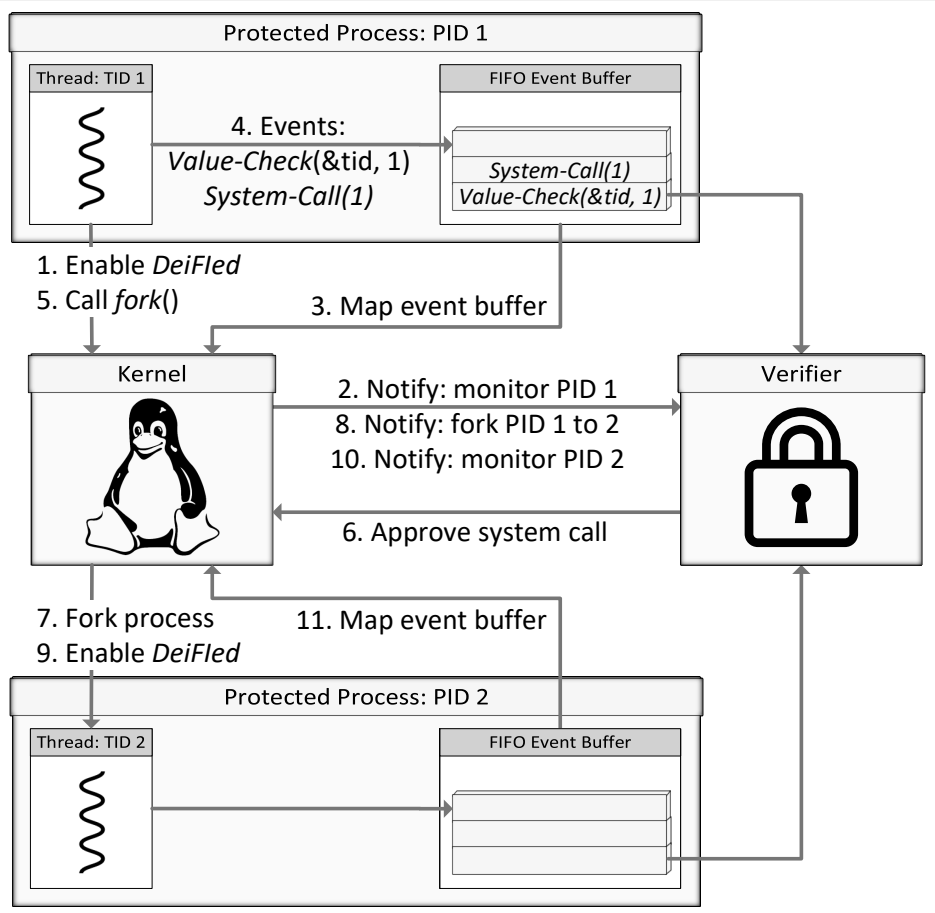

FIGURE 4.1. Overview of event processing, system call checking, and thread identifier protection under HERQULES-DEIFIED, depicting behavior at `fork`.

VALUE-DEFINE and VALUE-CHECK events immediately after each update and load, respectively. We show this design in Figure 4.1, where the verifier checks that the value `1` of tid matches the most recent VALUE-DEFINE for the `&tid` address (step 4).

One potential issue with a shared event buffer is whether concurrent AppendWrite operations could incur a *data race* or be reordered. For example, suppose that two program threads each overwrite the same protected memory location and transmit corresponding events. Partial event transmission cannot occur because the AppendWrite primitive is atomic, but event reordering is possible. Nevertheless, observe that if these protected memory operations are regular (i.e., not synchronization), then events can be reordered if and only if these operations could also be reordered–namely, iff the original program is not data-race-free. Thus, with respect to data race freedom of regular memory accesses under HERQULES-DEIFIED, **a protected program is as correct as the original program**. However, one caveat is that under the x86-TSO [163] memory consistency model, if the protected memory operations are atomic, then our instrumentation must introduce additional synchronization, e.g., using Intel Transaction Synchronization Extensions [7] (TSX) memory transactions. Otherwise, concurrent execution of two sequential atomic

memory operations on two threads may be interleaved, which could reorder event transmission.

4.2.2.4. *Supporting Multiple Processes.* On POSIX systems, applications use the *fork-join* parallelism model to spawn a duplicate copy of the current process. However, child processes of protected processes must also be protected, but event buffers cannot be shared across multiple processes. We resolve this problem by having the kernel return a unique event buffer for the caller when HerQules-DeiFIed is enabled, and by marking event buffers as non-copyable memory regions. Protected child processes must thus immediately re-enable HerQules-DeiFIed after calling `fork`, as shown in Figure 4.1 (steps 7–11), otherwise they will be suspended at system calls yet unable to transmit events. Conversely, we disable HerQules-DeiFIed when a protected process calls `execve`, because it is replaced in-situ by a new program that may not be protected.

We design our verifier program to scale with multiple concurrent protected processes and event buffers. Rather than iteratively polling on each event buffer, which would occupy an entire processor core, we utilize *fast userspace mutexes* (futexes) to wake the verifier only when an event buffer is modified. Protected processes must notify the kernel of new events by executing the `futex` system call after AppendWrite, in order to wake the verifier. To minimize protection overhead of this system call, which is also used by e.g., *pthreads* to implement synchronization primitives like mutexes and condition variables, we allowlist (§4.3.1.1) `futex` in HerQules-DeiFIed. This does not affect the security of our design, because `futex` can only be used to suspend the current thread or wake up other threads, and it already permits spurious wake-ups.

4.2.3. Data-Flow Integrity. Data-flow integrity [39] protects the value of non-control data at runtime. We do so using AppendWrite, which avoids imprecise static analysis, much like *pointer integrity* for control-flow integrity (§ 2.4). Events are transmitted whenever protected data may be loaded or stored, and a *memory corruption violation* occurs whenever loaded values do not match the last valid stored value at that address. By generating events whenever protected data goes out-of-scope, we can also detect *use-after-free violations* at subsequent accesses. If a violation is detected, we immediately terminate the corresponding process and stop processing its events, though this can be configured. We describe the semantics of our events below:

- Value-Define(a, v): Initialize a variable at address a with value v.
- Value-Check(a, v): Validate that the variable at address a with current value v matches its previous definition. If not, this variable is corrupt or a use-after-free.
- Value-Invalidate(a): Remove the variable at address a.
- Object-Define(a, sz, v...): Initialize an object at address a of size sz with corresponding values v.
- Object-Check(a, sz, v...): Validate that the object at address a of size sz with corresponding values v match its previous definition. If not, this object is corrupt or a use-after-free.
- Object-Invalidate(a, sz): Remove the object at address a of size sz.
- Block-Copy(src, dst, sz): Copy all data from address range[src, src + sz) to [dst, dst + sz). These ranges may intersect, and pre-existing data in the destination will be invalidated. This matches the behavior of `memcpy`.

```
#include <dfi>

DFI<std::map<std::string, std::pair<int, bool>>> kv1;
std::map<std::string, std::pair<int, bool>> kv2;

static_assert(std::is_same_v<decltype(kv1), std::dfi_map<std::dfi_string,
↪  std::pair<std::dfi_wrap<int>, std::dfi_wrap<bool>>>>);
```

LISTING 4.1. Two instances of a hash table data structure; one (`kv1`) annotated and protected by data-flow integrity, the other (`kv2`) is not.

- SYSTEM-CALL(TID): Allow the proximate system call from thread TID once all prior events have been processed.

4.2.3.1. *Sensitive Data.* We rely on manual annotations to identify sensitive data, and provide automatic compiler-based instrumentation to generate runtime events. Determining whether program data is security-sensitive depends on its *semantic usage*, and not necessarily its language-level characteristics, e.g., *type signature*. As a result, in large programs where full context-sensitive inter-procedural analysis is infeasible, it would be difficult to distinguish between multiple seemingly-identical variables that each may not be security-sensitive. In addition, non-trivial data structures and objects may contain internal pointers to heap-allocated data that need to be recursively protected. This resembles the *garbage collection* problem, which must identify *root references* to all live objects.

KENALI [168] used domain-specific heuristics to automatically identify access-control data in the Linux kernel, but this does not generalize to arbitrary programs in terms of precision and scalability. General-purpose applications, including web browsers, do not treat security-sensitive data uniformly, in terms of error handling and return values. KENALI also relies on a slow points-to analysis that runs in about an hour [167], and produces a 28% false positive rate on a compressed kernel of ~7 MiB, whereas a compressed uninstrumented web browser is typically an order of magnitude larger (e.g., the compressed Chromium binary measures ~81 MiB, excluding shared libraries).

We distinguish between protected and unprotected data by separating out protected instances into new language-level datatypes that contain annotations on protected internal data. Due to the need for runtime event messaging, protected and unprotected datatypes cannot share the same implementation. If pointers or references to individual instances are passed into other functions, then those functions will select the appropriate implementation based on the type signature. This ensures compatibility with object-oriented language features like class membership, inheritance hierarchies, and dynamic dispatch, which are common in real-world programs. In addition, the *one definition rule* for C++ [9] states that all definitions of the same datatype must be equivalent, because compilation units are normally translated individually, then subsequently combined and deduplicated by the linker.

To simplify the annotation process, we provide (**A**) a native wrapper class to protect primitive datatypes, (**B**) pre-annotated datatypes for the C++ *Standard Template Library* (STL), and (**C**) a pre-defined `DFI` annotation that maps unprotected datatypes to their

```
1  template <typename T>
2  class dfi_wrap final {
3    T __attribute__((annotate("dfi"))) data;
4
5    operator T&() & noexcept { return data; }
6    operator const T&() const& noexcept { return data; }
7  };
```

LISTING 4.2. Simplified wrapper class that identifies protected data for data-flow integrity instrumentation.

protected equivalent. **C** handles generic datatypes by recursing via type substitution at template expansions. **A**–**C** minimize manual effort, and HERQULES-DEIFIED automatically transforms and protects these specific annotated datatype instances.

We show a concrete example of this in Listing 4.1, which depicts two instances of a hash map data structure that maps strings to (integer, boolean) pairs. Annotations of each type are shown: `dfi_wrap` (**A**, line 6), `dfi_map` and `dfi_string` (**B**, line 6), and `DFI` (**C**, line 3). Only one instance, `kv1` (line 3), is annotated for protection, whereas `kv2` (line 4) is not. Our wrappers automatically transform the datatype of the `kv1` to be recursively protected, as shown by the datatype equivalence assertion (line 6). Observe that primitive datatypes are wrapped by our `dfi_wrap` class, whereas other STL datatypes are transformed to their protected equivalents; e.g., `dfi_map`, `dfi_string`, etc.

4.2.3.2. *Compiler Instrumentation.* Our wrapper class simply labels protected data with a special compiler attribute (line 3), as shown in Listing 4.2. This is recognized by our Clang/LLVM compiler instrumentation pass, which inspects accesses to each instance, and generates runtime calls to transmit events. We include built-in conversion operators (lines 5–6) to permit implicit data access with minimal source code change. Programs written in C can also apply this attribute directly to protected data.

- STORE(A, V): VALUE/OBJECT-DEFINE(A, V).
- LOAD(A, V): VALUE/OBJECT-CHECK(A, V).
- CALL(F, A...): VALUE/OBJECT-DEFINE(A, V) and/or VALUE/OBJECT-CHECK(A, V).
- MEMCPY(SRC, DST, SZ): BLOCK-COPY(SRC, DST, SZ).
- DESTRUCTOR(A) or LIFETIME-END(A): VALUE/OBJECT-INVALIDATE(A).
- INLINE-ASM(SYSCALL): SYSTEM-CALL(GETTID()).

Events are generated based on the type of each access to protected data, as shown for a selected example in Listing 4.3. We insert DEFINE and CHECK events immediately after STORE and LOAD instructions (lines 8, 10), respectively, and select between the VALUE and OBJECT variants based on whether the access granularity is less than or equal to the processor's word size. Since our analysis is not inter-procedural across compilation units, we conservatively generate additional events when references to protected data escape through function calls. This approach maximizes compatibility by preserving existing type signatures, but it does permit a potential race against execution of the DEFINE event after returning. At each call site, we inspect the callee attributes and type signature, to determine whether each argument is statically-known to be readable or

```
class RenderFrameImpl {
    DFI<int> bindings_ = 0;

public:
    void AllowBindings(int32_t flags) {
        // enabled_bindings_ |= flags;
        auto Arg = enabled_bindings_;
        AppendMsg(VALUE-CHECK, &enabled_bindings_, Arg);
        Arg |= flags;
        AppendMsg(VALUE-DEFINE, &enabled_bindings_, Arg);
    }
};
```

LISTING 4.3. Selected browser code that modifies protected data. Compiler instrumentation automatically inserts the register variable `Arg` and calls to AppendWrite.

writable. Then, we generate CHECK events before each may-read escape, and DEFINE events after each may-write escape.

In the event that the callee is a memory copy function (i.e., `memcpy`), we instead generate the BLOCK-COPY event immediately afterwards, in order to inform the verifier that protected data has been copied. We also check all other calls to memory copy functions, and generate similar runtime events if either the source or destination have type signatures that contain protected data. Additionally, to detect use-after-free bugs, we generate INVALIDATE events when protected data goes out-of-scope, which occurs after calls to C++ object destructors, or when the lifetime of stack-allocated data ends.

Finally, as with HERQULES (§3.2.2), we insert SYSTEM-CALL events before inline assembly statements that perform system calls.

4.2.4. Browser Protection. We apply our data-flow integrity protections to the open-source Google Chromium web browser. Certain browser information, identified as *security-sensitive program data*, is protected in all processes. Although by no means complete, we broadly classify this data into two distinct categories: (1) *security information*, which is used to enforce the browser's internal security model, and (2) *private information*, which includes user data. Both types of data are protected from corruption, though we do not protect the confidentiality of private data (§4.2.1). Below, we summarize these datatypes, which are shown in detail in Table 4.1:

- command-line arguments, which configure browser features at startup;
- feature flags, which configure browser features at runtime;
- content security policies, which includes site isolation parameters and access control lists for policies like cross-origin resource sharing (CORS), cross-origin embedder policy (COEP), and cross-origin opener policy (COOP);
- process permissions, which track sandbox capabilities for child processes;
- script bindings, which expose browser components to JavaScript, including the Mojo IPC interface (MojoJS);
- site origins, which identify unique sites via their scheme, host, and port components;

- parsed URLs, which represent canonicalized uniform resource locators (URLs);
- cookies, which store site-specific user information; and
- form data, which records identifying user information, including addresses, credit card numbers, emails, phone numbers, and passwords.

4.2.5. Generalized IPC. We also observe that AppendWrite can be used to improve browser IPC performance, and begin by describing the existing Mojo IPC framework. Both the Chrome and Firefox web browsers use Mojo to communicate between isolated processes, which may be sandboxed and unable to directly access the filesystem. Messages can be routed through intermediate nodes, and each endpoint is identified by a randomly-generated *node name* and *port name*. Message channels between endpoints can be transmitted over other channels, effectively allowing replacement of one endpoint with another. On POSIX-based systems, these are implemented using bidirectional UNIX-domain socket pairs, which use ancillary `SCM_RIGHTS` data to transmit file descriptors, and automatically close if either process terminates. Messages are received and sent using the `recvmsg` and `sendmsg` system calls, which incur overhead by copying both ordinary and ancillary message data from user-space to kernel-space and vice versa.

In HerQules-DeiFied, we replace sockets with AppendWrite, which eliminates the overhead of copying message data between kernel- and user-space, by writing it directly to user-space shared memory. However, since AppendWrite is a uni-directional IPC primitive, whereas sockets are bi-directional, our design must bootstrap bi-directional communication by performing a two-way handshake at creation. Initially, both endpoints are connected only by a uni-directional IPC channel, so the transmitter must allocate and send a transmit buffer for the receiver. Then, the receiver can initialize this buffer for its transmissions in the opposite direction.

Nevertheless, shared memory doesn't support other features used by Mojo, including transmission of message channels and detection of process termination. We extend the kernel with *file descriptor FIFO queues*, which allow transmission of arbitrary file descriptors over other file descriptors, including those that back shared memory-mapped regions. We also repurpose *robust mutexes* to detect process termination, which rely on the kernel to automatically unlock shared mutexes when the owning process dies. To support an event-based messaging abstraction, we rely on futexes to efficiently wait on multiple concurrent shared memory message buffers.

4.3. Implementation

4.3.1. System Architecture. As discussed in Section 4.2.2, HerQules-DeiFied targets platforms with an emerging AppendWrite hardware primitive that satisfies four functional properties. Currently, we are unable to implement AppendWrite directly (§3.2.1), or emulate it using PTWRITE [7], due to limitations in availability and/or functionality. The PTWRITE instruction is only commercially-available in the low-power embedded Gemini Lake microarchitecture, and Intel Processor Trace is known to suffer from event loss [7] and significant overhead [124]. Similarly, the AppendWrite primitive (§3.1.3) is not yet available in actual hardware, and does not yet permit sharing of atomic event buffers across processor cores.

In the meantime, as a work-around to enable studying the effectiveness and overheads of HerQules-DeiFied, we emulate the behavior of AppendWrite in software using *writable shared memory* over *sealed file descriptors*, which cannot be subsequently resized or remapped. Although these event buffers are not append-only, they are otherwise fully functional, and allow us to perform worst-case performance measurements. Optionally, we note that these buffers can be remapped read-only after modification using, e.g., `mremap` or Memory Protection Keys for Userspace. We use three shared memory variables to ensure mutual exclusion, identify whether the event buffer is full, and track the next write offset. On every invocation of AppendWrite, we atomically take the mutual exclusion lock, write out the current event, increment the write offset, and then atomically release the lock. If insufficient space remains in the event buffer, then the writer marks it as full and waits. When the reader reaches the end of the buffer, it resets the write offset and full flag, which allows a waiting writer to resume. In our evaluation, we provide a breakdown of our system in terms of lines of code (§4.4.3).

4.3.1.1. *Kernel Synchronization.* We implement system call synchronization using a dynamically-loadable Linux kernel module. It uses built-in kernel instrumentation mechanisms, including *kprobes* [101] and *tracepoints* [58], to intercept existing kernel functions and notify the verifier when protected threads call *clone*, *fork*, `execve`, or `exit`. A *rhashtable* keeps track of all protected threads, and records the unique system call synchronization page for each, which is shared with the verifier. Our kernel module is also responsible for handling protected processes that fail data-flow integrity checks in the verifier, with the default being to terminate them immediately (this is configurable).

Certain system calls must be handled specially in protected processes. When an interruptable system call (e.g., `read`, `write`, etc.) is re-executed after signal delivery, our kernel module automatically marks those as permitted. Other system calls are on a permanent allowlist; namely, `futex` (§4.2.2.4), as well as read-only system calls performed by the kernel virtual dynamic shared object (vDSO), which on *x86_64* platforms includes `clock_getres`, `clock_gettime`, and `gettimeofday`. This is because the vDSO is automatically loaded by the kernel, and cannot be instrumented without a complete rebuild. Permanently allowing these system calls is safe, because they are either read-only (vDSO) or have limited system effect (`futex`), even for protected applications that have not yet been detected as compromised. No System-Call event is transmitted for system calls on the allowlist, and they are automatically approved by our kernel module.

4.3.1.2. *User-Space Verifier.* We implement our verifier as a single-threaded user-space process, which maintains a data-flow integrity policy context for each protected process. This includes the per-thread shared system call synchronization page, as well as a per-process event buffer and hash table that stores protected address-value pairs. For performance and simplicity, we record protected data at word granularity, rather than tracking the granularity of individual loads and stores, then subsequently splitting and combining entries as needed (§4.3.2.2). The verifier is notified of messages from our kernel module via a synchronous signal, which are then read from a special privileged character device.

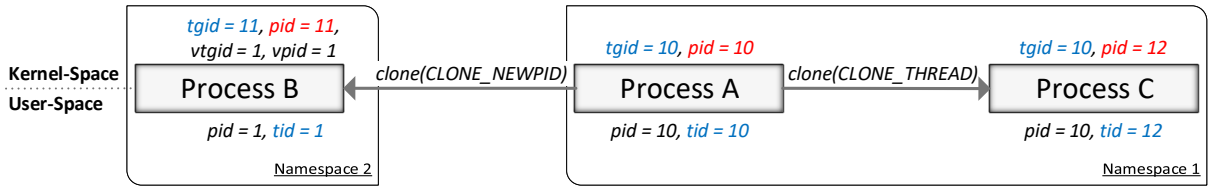

FIGURE 4.2. Process and thread identifiers in Linux, as seen from kernel-space (top) and user-space self (bottom), after Process A forks into a new namespace (left), then clones a new thread (right). Identifiers tracked by our kernel module are shown in red, whereas identifiers tracked by our verifier are shown in blue.

To avoid polling for changes to event buffers, we rely on the `FUTEX_WAIT_MULTIPLE` [20] feature of the `futex` system call to concurrently wait on multiple addresses. This mechanism has not yet been merged into the upstream Linux kernel, although it is currently being reviewed as part of the refactored `futex2` [13] interface. As a result, we must recompile our kernel to support this feature.

Due to the complexity of the kernel threading module, it is important to distinguish how threads and processes are tracked by our kernel module and user-space verifier. User-space threads are managed by the Native POSIX Thread Library, which maps threads one-to-one to light-weight kernel processes. These processes have distinct kernel-space *process identifiers* but identical *thread-group identifiers*, which correspond to user-space *thread identifiers* and *process identifiers*, respectively. We show a diagram of this in Figure 4.2.

Recent kernels support *namespaces* as a security feature, which allows scoping of global system resources within process subtrees. User-space processes can only observe namespace-local *virtual identifiers*, which may differ based on the perspective of the querying process. Since protected processes may be located in different namespaces, which may also be disjoint from that of our verifier process, these virtual identifiers may overlap and cannot be translated across namespaces in user-space. As a result, our verifier tracks processes using global process identifiers supplied by our kernel module, and threads using their virtual thread identifiers.

4.3.1.3. *Runtime Libraries.* We modify the C runtime library to instrument system calls and protect certain internal data. Typically, applications perform systems calls indirectly by calling into runtime libraries, which eventually call the C runtime library. Thus, no other existing language or runtime libraries needed to be rebuilt during our experiments. We protect the contents of the `jmp_buf` data structure used by `setjmp` and `longjmp`, as well as cached thread identifiers. Our compiler instrumentation pass automatically instruments the C runtime library, but due to compatibility issues with GCC-specific compiler extensions used by the GNU C runtime library, we instead use the musl C runtime library, which is compact and standards-compliant.

Input Type	Output Type
unsupported	`nullopt_t`
primitive, `enum`	`dfi_wrap<T>`
`string`	`dfi_string`
`map<K, V>`	`dfi_map<DFI<K>, DFI<V>>`
`pair<T1, T2>`	`pair<DFI<T1>, DFI<T2>>`
`set<T>`	`dfi_set<DFI<T>>`
`unique_ptr<T>`	`dfi_unique_ptr<DFI<T>>`
`vector<T>`	`dfi_vector<DFI<T>>`

TABLE 4.2. Simplified data-flow integrity `DFI` type mapping from input to output datatypes, showing base (top) and recursive (bottom) cases.

4.3.2. Data-Flow Integrity. We extend the `libc++` standard library for C++ to support data-flow integrity, by bundling it with our wrapper class, pre-annotated Standard Template Library data structures, and pre-defined type mapping. This includes copies of `map`, `set`, `vector`, `string`, and `unique_ptr`, which we name `dfi_map`, `dfi_set`, `dfi_vector`, etc., as well as their multi-key and unordered variants (where applicable), all of which use our wrapper class to protect their internal data.

4.3.2.1. *Type Mapping.* We show a small snippet of our type mapping in Table 4.2, which is implemented using template specialization to define base and recursive cases. Base cases include unsupported types that produce a compile error, scalar primitive types that are handled by our wrapper class, and string types that use our protected string class. Recursive cases handle templated generic types, such as the hash map type, the pair product type, the vector type, etc., by invoking our type mapping recursively on their input argument datatypes. Note that the `pair` type remains unchanged because it does not contain internal data, whereas other recursive cases are changed because they do.

4.3.2.2. *Wrapper Class.* Compared to the simplified version shown earlier in Listing 4.2, our actual wrapper class, shown partially in Listing 4.4, must ensure that protected data is word-aligned, fully-initialized at construction, and always invalidated at destruction. This alignment requirement is needed because our verifier tracks protected data at word granularity (§4.3.1.2). However, on `x86_64` systems, certain primitive datatypes, including `bool`, `char`, `short`, and `int`, are smaller and packed by the compiler with consecutive values or inaccessible padding bytes, which can cause runtime checks to fail. Our verifier rejects unaligned addresses, and memory loads that include uninitialized padding bytes will likely fail to match. We avoid these problems in our wrapper class (Listing 4.4) by imposing a minimum alignment of word-granularity for protected data (line 3), and by explicitly zeroing data during construction (line 6). But, since a minimum alignment would significantly penalize our protected `dfi_string` type, by padding out each character to word-width and negating the *small string optimization*, we instead manually insert instrumentation on the existing layout without using our wrapper class.

```
1  template <typename T>
2  class dfi_wrap final {
3    std::aligned_storage<sizeof(T), std::max(alignof(T), 8U)>::type
     ↪  __attribute__((annotate("dfi"))) data;
4
5    dfi_wrap() noexcept {
6      memset(&data, 0, sizeof(data));
7      new (&data) T;
8    }
9
10   ~dfi_wrap() {
11     if (std::is_trivially_destructible<T>::value)
12       data = {};
13   }
14 };
```

LISTING 4.4. Partial wrapper class that ensures alignment and generation of runtime events for data-flow integrity instrumentation.

Another issue that we must resolve, is that for static primitive datatypes, C++ permits the compiler to perform load-time *constant initialization* and omit empty default destructors. This prevents runtime emission of DEFINE and INVALIDATE events (Section 4.2.3) on protected data, which would result in false negatives and positives, respectively. Instead, we provide an explicit constructor (lines 5–8), and an explicit destructor (lines 10–13) that performs a dummy *default initialization* store. But, because this store is only needed to force generation of the destructor, we eliminate it inside our instrumentation pass. Unfortunately, as a side-effect, this explicit destructor also causes exit-time destruction of static data, which is unnecessary since the program is terminating. As an optimization, we manually suppress this behavior using the `no_destroy` attribute on such static data.

4.3.2.3. *Language Compatibility.* Despite our inclusion of implicit type conversion operators (lines 5–6 in Listing 4.2), certain compatibility problems do occur. According to the C++ language standard [9], the following situations can prevent implicit conversions:

- the dot operator, which cannot be overloaded to automatically invoke member functions on protected objects;
- the ternary conditional operator, which only allows the first expression to be implicitly converted to the second;
- template argument deduction, which ignores implicit type conversion;
- an existing implicit conversion, which may only be performed once; and
- ambiguous implicit conversions, which cannot be resolved to a final datatype.

Implicit type conversions are also unsuitable when protected data is exposed through mutable references, which must use the wrapped type to be updated. We manually handle all of these cases by modifying the input source code.

We believe that the C++ language could be improved to permit better integration with minimal developer overhead. In particular, we observe that transmission of e.g., DEFINE and CHECK/INVALIDATE events, allows runtime conversion of data from protected to unprotected and vice versa, respectively. This suggests that if both datatypes can be

layout compatible by sharing the same underlying memory representation, then separate implementations can be avoided by exclusively tracking protected data statically within the type system using e.g., *refinement types*. Other security mitigations, such as dynamic taint tracking [189, 200], could also benefit from these changes.

4.3.3. Browser Protection. Conveniently, the Chromium browser already builds a custom version of `libc++`, which we patch to include our data-flow integrity extensions. We also create protected variants of custom data structures used by the browser, including `GURL`, `Component`, and `Parsed`, as well as `Optional` and `StructPtr`. Most of these encapsulate different representations of parsed URL components, although some are substitutes for recent language features like `std::optional`, or provide move-only semantics for custom data structures synthesized from the Mojo IPC domain-specific language.

We also make compatibility changes to the Chromium browser by removing dependencies on non-portable or unsupported runtime library features. These include assumptions about non-standard C runtime library features, including runtime support for stack backtraces and allocator statistics, as well as the default thread stack size. Certain inline assembly statements embed systems calls within nested control-flow, which is unsupported by our compiler instrumentation and must be manually refactored out. We adjust the browser *seccomp-bpf* sandbox to permit system calls and arguments used by our data-flow integrity design and/or the musl C runtime library, including `FUTEX_WAIT_MULTIPLE` for `futex`.

4.3.4. Generalized IPC. As discussed earlier in §4.2.5, shared memory cannot transmit handles to other message channels, which is a feature used by Mojo. We develop a separate kernel module to support transmission of file descriptors by associating virtual FIFO queues with filesystem index nodes (*inode*s), and by extending the `fcntl` system call with the ability to push and pop file descriptors over these queues. We minimize overhead by supporting vectorized operations that allow multiple file descriptors to be pushed or popped in a single system call. To ensure safety, we increment the reference counter for file descriptors pushed onto the FIFO queue, and decrement it when they are popped or the underlying inode is to be destroyed.

We must also adjust the I/O thread abstraction used by the Chromium browser, which waits on file descriptor read and write events. It can call software libraries like *libevent* to perform efficient waiting using the e.g., `epoll` and/or `select` system calls, and is used by default for Mojo IPC events. However, our AppendWrite messages instead wait on shared memory events using the `futex` system call, but a single thread cannot concurrently wait on both file descriptors and futexes. We split Mojo IPC out into a separate thread, allowing both threads to wait exclusively on either event type.

4.4. Evaluation

We evaluate the effectiveness, performance, and usability of our data-flow integrity design using various experiments. This includes examining past browser exploits (§4.4.1), measuring overhead on a variety of benchmarks (§4.4.2), and quantifying the effort involved (§4.4.3). Our results show that our approach (i) mitigates real Chromium security

Issue ID	Severity	Component	Bug Type	Corrupted Data
1155426	Critical (9.6)	WebRTC	Use-after-free	MojoJS
1151865	High (8.1)	Network	Buffer overflow	MojoJS
1148749	Critical (9.6)	Autofill	Use-after-free	MojoJS
1146709	Critical (9.6)	Site Isolation	Use-after-free	MojoJS
1146675	Critical (9.6)	Storage	Use-after-free	MojoJS
1144489	Critical (9.6)	Browser UI	Buffer overflow	MojoJS
1144368	High (8.8)	Browser UI	Buffer overflow	MojoJS
1138143	High (8.8)	Clipboard	Stack overflow	MojoJS
1136078	Critical (9.6)	Payments	Use-after-free	MojoJS
1135857	High (8.8)	USB	Use-after-free	MojoJS
1135018	High (8.8)	Media Capture	Use-after-free	MojoJS
1116304	Critical (9.6)	Media Capture	Use-after-free	MojoJS
1103827	High (8.8)	Blink	Buffer overflow	MojoJS
1082105	Critical (9.6)	WebAuth	Use-after-free	MojoJS
1074706	Critical (9.6)	Browser UI	Use-after-free	MojoJS
1073015	Critical (9.6)	Browser UI	Use-after-free	MojoJS
1072983	Critical (9.6)	Storage	Use-after-free	MojoJS
1064891	Critical (9.6)	Sequence Manager	Use-after-free	MojoJS
1062091	N/A	AppManifest	Use-after-free	MojoJS, CommandLine
1059764	High (8.8)	Media Capture	Buffer overflow	MojoJS

TABLE 4.3. Chromium vulnerabilities reported in 2020 that would be mitigated by HERQULES-DEIFIED.

bugs, (ii) decreases geomean relative performance by only 14 p.p. (percentage points) on browser benchmarks, despite transmitting up to 37.4 billion events, and (iii) increases browser size by only 445.3 KiB.

We develop HERQULES-DEIFIED on Clang/LLVM 11.1.0 and musl libc 1.2.2 for Google Chromium 87.0.4280.152, and perform our experiments on a virtualized Alpine Linux 3.13 system running kernel 5.10.27, which is configured with 32 GiB memory, 6 CPU cores, and a 1920 × 1080 display resolution. We pre-allocate 16 GiB of hugepage-backed shared memory, with 256 MiB assigned to each event buffer, and 16 MiB of normal shared memory for each Mojo-AppendWrite IPC channel. Our host machine is a Ubuntu 20.04.2 system running QEMU-KVM 5.0.0 on kernel 5.8.0-53, with an Intel Core i9-9900k CPU @ 5GHz, a Samsung 860 Pro solid-state drive, and 128 GiB DDR4-2666 memory.

4.4.1. Effectiveness. Past exploits show that escaping the Chrome browser sandbox via non-control-data attacks is quite feasible. One technique [19, 29, 198] corrupts memory in the renderer process to enable internal features, like JavaScript bindings for the Mojo IPC interface (MojoJS), which expose a broader attack surface. Then, specially-crafted messages can be sent to exploit otherwise-inaccessible bugs in privileged processes. Another technique [19, 122] overwrites browser command-line flags, so that subsequent child processes execute arbitrary programs or have sandboxing disabled.

To quantify the prevalence of these bugs, we manually examine security bugs in the Chrome bug tracker that were reported in the previous calendar year and issued Common Vulnerabilities and Exposures (CVE) identifiers. Of the 235 reported security bugs in 2020, we observe that ~140 involve memory-safety bugs, and identify 20 that would be mitigated by HerQules-DeiFIed, which we summarized in Table 4.3. These are bugs that rely on corruption of protected non-control data, either as part of publicly-disclosed exploits, or as prerequisites for exposing the vulnerable attack surfaces. We are the first system to mitigate these vulnerabilities, and we manually verify the functionality of HerQules-DeiFIed by using a debugger to corrupt protected data and checking that subsequent accesses detect policy violations.

In practice, we observe that compiler-based security mitigations are more effective than they would initially appear. Many desktop applications embed web browsers using frameworks like the Chromium Embedded Framework (CEF) or Electron, but disable sandboxing for compatibility or performance reasons. Past analysis [46] has shown that this includes many popular applications, including Discord, Facebook Messenger, Keybase, Microsoft Teams, Signal, Twitch, Visual Studio Code, and WeChat. As a result, a single exploit may be sufficient to compromise the system.

4.4.2. Performance. We measure various performance aspects of HerQules-DeiFIed, including browser benchmarks and rendering latency (§4.4.2.2, §4.4.2.3), event logging scalability (§4.4.2.4), memory usage (§4.4.2.5), and binary size (§4.4.2.6). To isolate individual design components, we enable system call instrumentation (§4.2.2) independently from data-flow integrity protection (§4.2.3 and control-flow integrity protection.

Unless stated otherwise, we execute three iterations of each benchmark, reporting both the arithmetic mean measurement and standard deviation error bars. Baseline results were obtained using an uninstrumented Chromium executable, modified only for compatibility with the musl C runtime library. On each benchmark iteration, we clear the user profile directory before launching the browser.

4.4.2.1. *Browser Benchmarks.* We begin with various industry benchmarks: JetStream 2, MotionMark 1.1, Speedometer 2.0, Basemark Web 3.0, and WebXPRT 3 v2.93. These measure a combination of various browser features, including (i) throughput of JavaScript and WebAssembly [87] workloads, rendering speed of Scalable Vector Graphics (SVG) nodes, HTML Document Object Model (DOM) elements, and bitmap canvas operations, and (ii) latency of various HTML and JavaScript APIs, as well as WebGL GPU performance. We also measure the *Navigation to First Contentful Paint (FCP)* metric from the Site Isolation [152] benchmarks. This counts elapsed time between page navigation and initial rendering, on various locally-cached websites selected from the Alexa Top 50 popularity rankings, as well as on an empty webpage.

4.4.2.2. *Data-Flow Integrity.* In Figure 4.3, we compare the overhead of system call synchronization, data-flow integrity, and Clang/LLVM control-flow integrity for Chromium. System call synchronization decreases performance by 2 p.p. to 12 p.p. relative to baseline, because transmission of system call events and suspension of system calls imposes overhead. This magnitude is dependent on benchmark workload; execution of rendering benchmarks like MotionMark is mainly spent in the sandboxed GPU process, which communicates with the renderer process over Mojo IPC by making system calls. On the

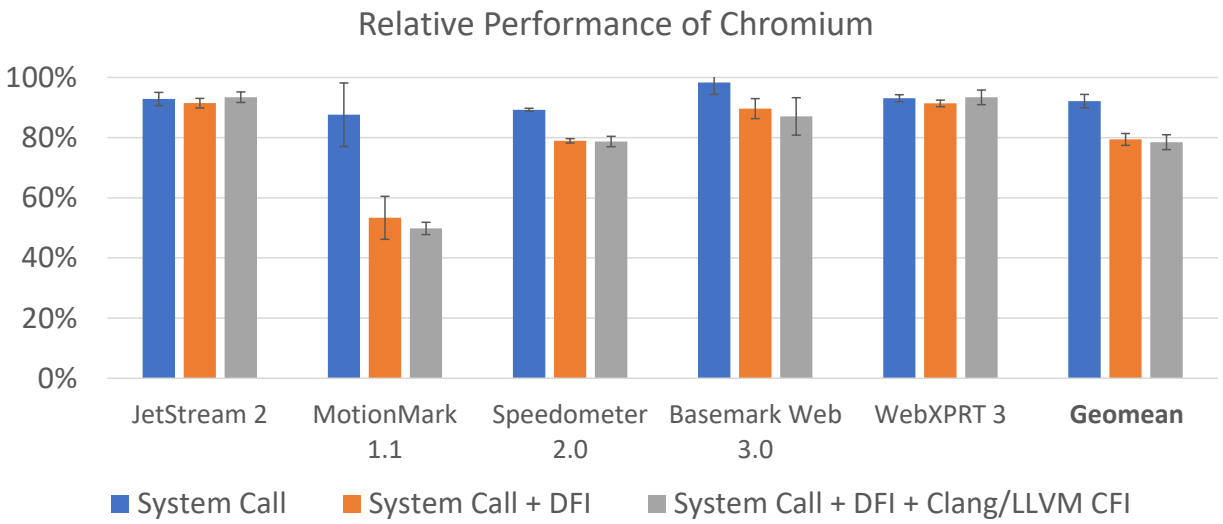

FIGURE 4.3. Relative performance of Chromium under HERQULES-DEIFIED on various browser benchmarks.

other hand, execution of compute-heavy JavaScript and WebAssembly benchmarks is mainly spent in the runtime interpreter, just-in-time compiler, or generated native code, which makes fewer system calls. Adding data-flow integrity protection further reduces performance by 2 p.p. to 35 p.p., because increased event traffic affects both the transmitting process and our verifier. Enabling built-in Clang/LLVM control-flow integrity has no measurable effect on performance, but Clang/LLVM CFI is a coarse-grained design (§2.4.3) that trades-off precision for performance, and does not protect return pointers. Overall, the geomean performance across the five benchmarks is 92% for system calls, 79% for all of HERQULES-DEIFIED, and 78% for HERQULES-DEIFIED with control-flow integrity.

In Figure 4.4, we show results for baseline, system call synchronization-only, and full-HERQULES-DEIFIED experiments. Elapsed time increases by a geometric mean of 91% from baseline to system call synchronization only, indicating that delayed system call execution has a significant effect on rendering latency. Adding data-flow integrity increases mean elapsed time by another 8%, although this effect is not consistent across all websites, and actually reverses slightly on some websites. Enabling control-flow integrity further increases mean elapsed time by 10%, but again this effect is not consistent. As shown by our confidence intervals, this metric has high standard deviation across individual site visits.

4.4.2.3. *Generalized IPC.* We perform a micro-benchmark on Mojo IPC performance with and without AppendWrite using built-in tests, and show our results in Figure 4.5. These demonstrate relative performance in transmitting a fixed number of messages between the main threads of two processes, using a variable batch size of 1, 10, or 100 messages per iteration, for up to a total of 10000 messages, and with variable message payloads of 8, 64, 512, 4096, or 65536 bytes per message. Note that in practice, Mojo can impose significant message overhead, as the actual size of a message with a 64-byte

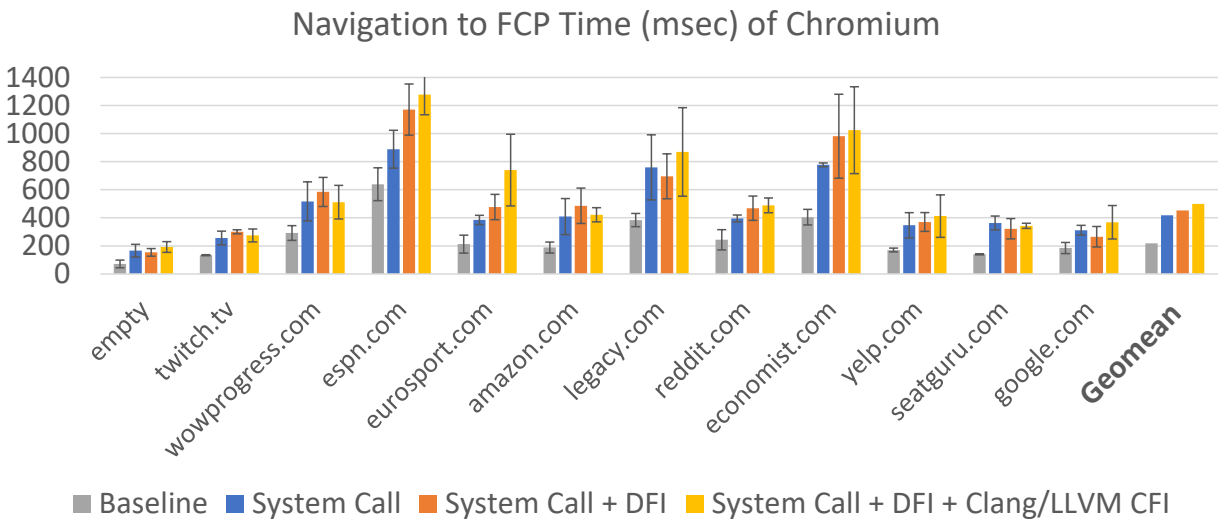

FIGURE 4.4. Relative latency of Chromium under HERQULES-DEIFIED on various popular websites.

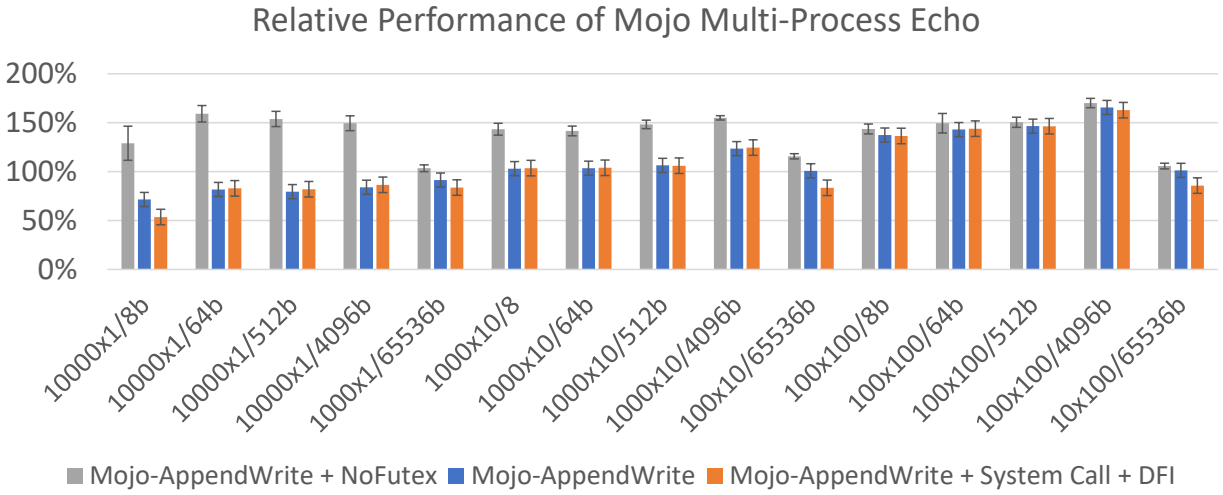

FIGURE 4.5. Relative performance of Mojo IPC, with AppendWrite, `futex`, and/or data-flow integrity.

payload on this micro-benchmark is 192 bytes. Within each batch, performance generally improves as the payload size increases, but performance at the maximum payload size can drop significantly. The break-even point appears to be around a batch size of 10 messages, after which performance generally becomes faster than baseline.

Next, we compare the overhead of Mojo-AppendWrite IPC, system call synchronization, and data-flow integrity on our browser benchmarks in Figure 4.6. Despite improvements in micro-benchmark performance, we observe that browser benchmarks suffer from a consistent performance decrease of 1 p.p. to 28 p.p.. Most messages sent over

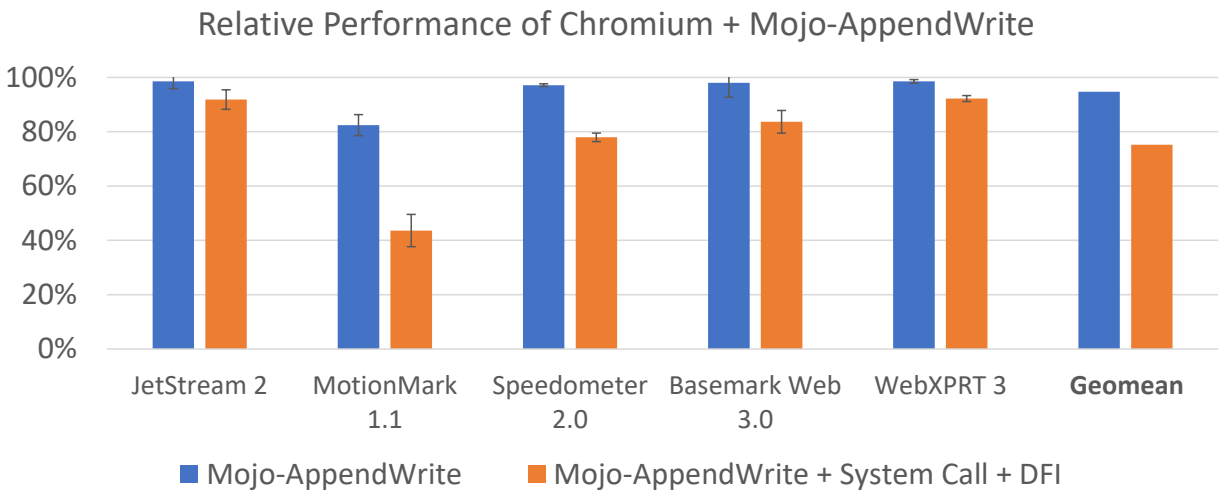

FIGURE 4.6. Relative performance of Chromium with Mojo-AppendWrite IPC under HERQULES-DEIFIED on various browser benchmarks.

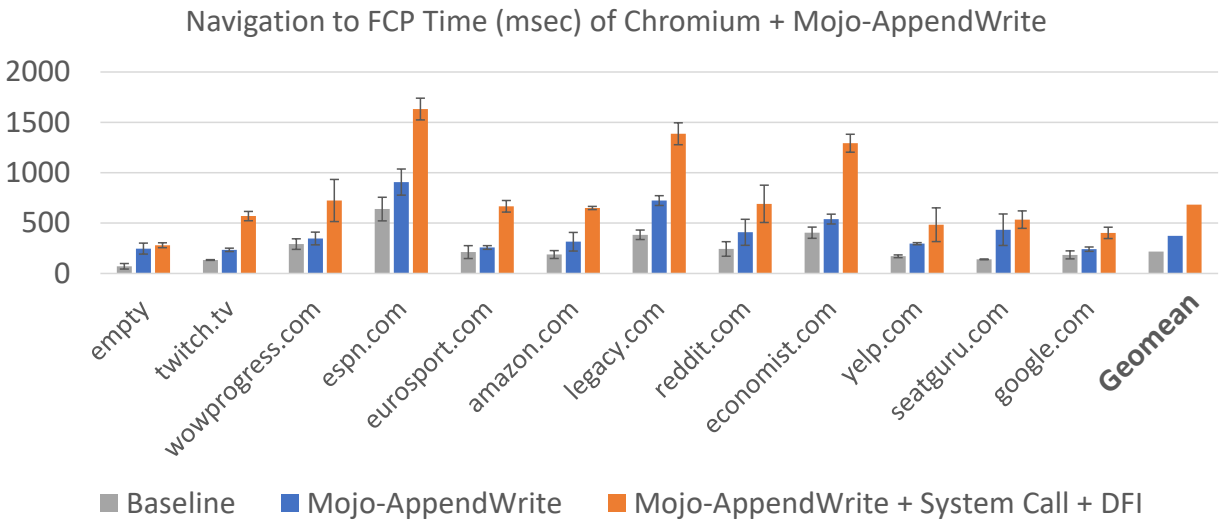

FIGURE 4.7. Relative latency of Chromium with Mojo-AppendWrite IPC under HERQULES-DEIFIED on various popular websites.

Mojo IPC messages are fairly small, which perform worse under Mojo-AppendWrite than baseline, as shown in Figure 4.5.

Finally, we show *Navigation to First Contentful Paint* metrics in Figure 4.7. We observe a mean 71% increase in elapsed time from baseline to system call synchronization, which is consistent across all loaded webpages. Similarly, we observe a further mean 83% increase in elapsed time from system call synchronization to data-flow integrity protection, again consistently across all loaded webpages.

These results are surprising, because microbenchmark improvements are inconsistent and the actual performance of AppendWrite-based Mojo IPC is consistently worse than on browser benchmarks. Nevertheless, Mojo system calls decrease under our design, because they only occur when in-transmission messages contain message channel handles, after all queued messages have been transmitted, and after completing each iteration of the event I/O loop. In comparison, under the original socket-based design, system calls occur whenever messages are received or transmitted, and after completing each iteration of the event I/O loop.

Additional experiments and performance instrumentation show that this is due to a combination of factors. First and foremost, waiting on multiple message buffers using the `futex` system call introduces significant overhead, because waiting processes are rescheduled and may not be immediately woken. In Figure 4.5, we also show results for a design that polls eagerly for new messages rather than calling `futex`, which is consistently faster than baseline. Although such a design is impractical because every Mojo thread in each browser process would busy-wait, it shows that the underlying kernel `FUTEX_WAIT_MULTIPLE` implementation could be improved. This would also increase performance of our verifier and system call synchronization, which relies on the same `futex` mechanism.

Other sources of Mojo overhead include our two-way handshake and buffer reset mechanisms. For each new node channel, two message buffers must be allocated, transmitted to both processes, and mapped in each process. However, due to the structure of Mojo's design, construction and initialization of the second buffer cannot occur until after messaging over the first buffer is available, which adds runtime overhead. Our message buffers must also block when they become full, until the corresponding reader resets the next write offset. This is problematic because we fix all Mojo-AppendWrite message buffers at 16 MB, but some will transmit significantly more messages than others. As a result, we trade-off between frequency of runtime resets and total memory overhead.

4.4.2.4. *Scalability.* Next, we measure transmitted events and verifier statistics for a single iteration of the Basemark Web 3.0 benchmark, aggregated across all browser processes. We select this benchmark because it is the longest of our browser benchmarks.

In Figure 4.8, we observe that under system call synchronization, a similar number of CHECK and SYSTEM-CALL events are processed. This is expected because each thread identifier is checked before a system call. However, the number of CHECK events is slightly higher, because the `jmp_buf` data structure used by `longjmp` and `setjmp` is also being protected.

After enabling data-flow integrity protection, both designs process a similar number of SYSTEM-CALL events, because they execute the same benchmark and data-flow integrity does not perform additional system calls. However, all other event types increase in frequency because security-sensitive browser data is now being protected. The magnitude of these changes is significant, with a minimum of 2.5x for CHECK events, to a maximum of 6721x for INVALIDATE events. Across all event types, this represents a total increase in event throughput of 2.3x, but with a geometric mean performance decrease of only 13 p.p. on browser benchmarks.

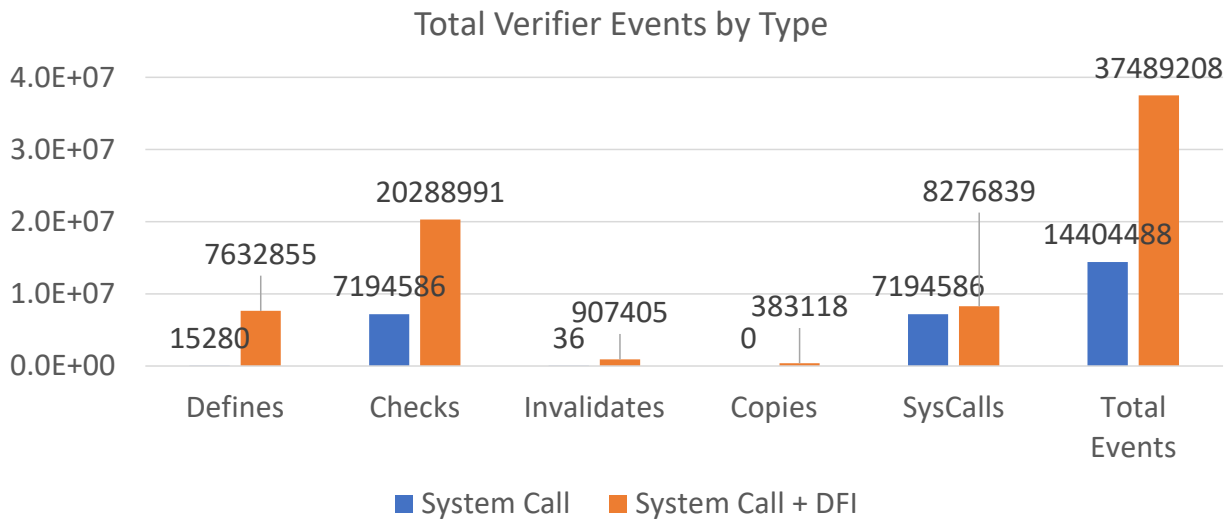

FIGURE 4.8. Processed events by type across all protected processes for Basemark Web 3.0.

	Verifier Statistic	Max	Median	Arithmean	Min
Per-Process	Event Buffer Resets	1	0	0.04	0
	Forks	26	0	0.96	0
	Max Threads	35	5	7.48	1
	Max Entries	16064	6600	7224.15	3208
	Max Entries	195052	n/a	n/a	n/a
	Protected Processes	11	n/a	n/a	n/a

TABLE 4.4. Verifier statistics for Basemark Web 3.0, with system call synchronization and data-flow integrity enabled. Certain statistics (top) are shown on a per-process basis.

In Table 4.4, we show verifier statistics from the same benchmark iteration with both system call synchronization and data-flow integrity protection. They demonstrate that our single-threaded verifier is capable of processing events from up to 11 concurrent processes, with each executing up to 35 concurrent threads. Some protected processes are short-lived, either because they terminate or because they execute unprotected programs, as shown by the increased count of `fork` system call executions. Up to 3 MiB of sensitive data is being tracked across all protected processes, as each verifier entry consists of a 16-byte address-value pair. Only one protected process filled its event buffer and needed a reset from the verifier.

4.4.2.5. *Memory Usage.* In Figure 4.9, we show total private memory usage across all browser processes on the Site Isolation benchmarks using built-in browser statistics.

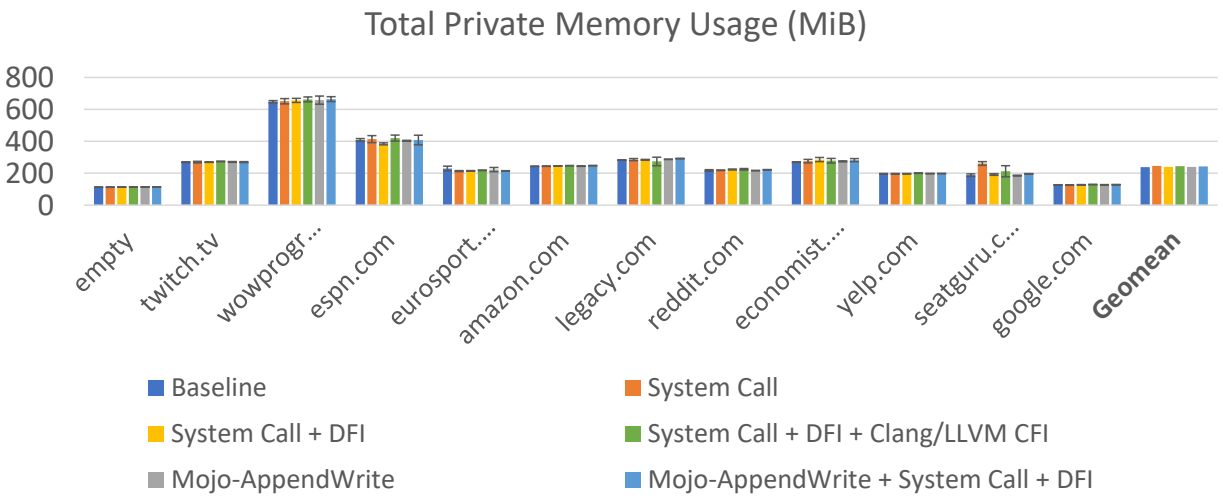

FIGURE 4.9. Total private memory usage of Chromium on various popular websites.

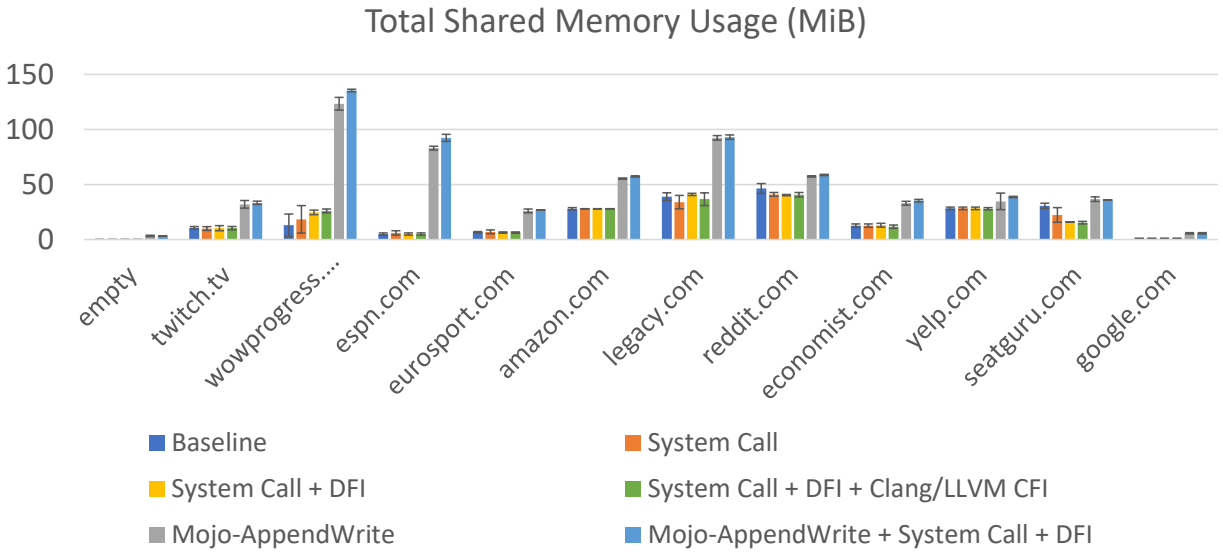

FIGURE 4.10. Total shared memory usage of Chromium on various popular websites.

These results indicate that our system call synchronization and data-flow integrity protections impose little additional private memory overhead, with a maximum difference of at most a couple megabytes.

Similarly, in Figure 4.10, we show total shared memory usage across all browser processes on the Site Isolation benchmarks using built-in browser statistics. Although fairly noisy, these results also indicate that our design has little impact on shared memory usage. Note that enabling system call synchronization and/or data-flow integrity

Baseline	+ System Call	+ DFI	+ CFI
263570400	263574224	264026416	286874416
(above)	**+ Mojo-AppendWrite**		
(above)	263572400		

TABLE 4.5. Binary size of compiled Chromium executable, with cumulative addition of features.

protection is not shown to have a significant impact, because pre-allocated hugepages used by our event buffers are not included.

We estimate that given our previous verifier statistics and per-event buffer allocations, a total of up to 2.8 GiB hugepages are consumed on the Basemark Web 3.0 benchmark. The size of these buffers is configurable, but they increase the frequency of runtime stalls and verifier resets when they become full. Enabling AppendWrite-based Mojo also increases shared memory usage, but because these message buffers are anonymous copy-on-write mappings, allocated pages are not counted until they are modified. When we re-execute this benchmark under HERQULES-DEIFIED with a 16 MiB event buffer instead, we observe an increase in total resets from 1 to 49, and a reduction in average performance from $90 \pm 3\%$ to $87 \pm 4\%$.

4.4.2.6. *Binary Size.* In Table 4.5, we measure the cumulative size of the compiled browser binary with our individual design components. Our changes have little effect on binary size: adding system call synchronization amounts to a scant 3.9 KiB increase over the 251.4 MiB uninstrumented binary, and adding data-flow integrity protection adds only an additional 441.4 KiB. In comparison, enabling control-flow integrity adds 21.8 MiB. Enabling AppendWrite-based Mojo IPC only increases the browser size by 2.0 KiB over baseline.

4.4.3. Usability. In Table 4.6, we measure the lines of code added or changed by each individual component of HERQULES-DEIFIED and Chromium, using the `cloc` tool and by hand. Of our system components, our kernel module and compiler pass comprise the bulk of our implementation, with our verifier a distant third.

As for our data-flow integrity annotations (§4.3.2), limited changes to the C++ Standard Template Library and Chromium browser are needed. Our wrapper class (§4.3.2.2), type mapping (§4.3.2.1), and protected STL string class likewise encompass few lines of code. In fact, for small datatypes, inclusion of new header files changes more lines of code than our data-flow integrity annotations themselves. However, separating out protected datatypes into distinct classes for both STL and Chromium duplicates many lines of code, most of which only needs to be automatically renamed to reflect the new datatype. The only manual changes needed are insertion of additional constructors and conversion operators to convert between protected and unprotected datatype instances.

Compatibility fixes (§4.3.2.3) in Chromium are needed to adapt to our wrapper class and protected datatypes. The majority of these changes are in test harnesses written for the GoogleTest framework, which identify expected return values using templated function calls, but do not permit implicit type conversion of input arguments due to

Component	Lines of Code
DFI Wrapper Class	137
DFI Type Mapping	326
STL DFI Annotations	107
STL DFI Classes	21227
STL String DFI Instrumentation	36
Compiler Pass	1822
FD Queue Kernel Module	289
Kernel Module	1442
Event Messaging Runtime	330
Shared Headers	531
Verifier	955
Chromium DFI Annotations	275
Chromium DFI Classes	3074
Chromium DFI Compatibility Fixes	369
Chromium DFI `no_destroy` Attribute	2165
Mojo-AppendWrite IPC	1459

TABLE 4.6. Approximate lines of code added or modified, split between data-flow integrity (top), system components (middle), and Chromium-specific changes (bottom). Unmodified Chromium is ~29.7 million lines of code.

language rules. Other changes include internal API adjustments to expose mutable references to protected datatypes, and insertion of `no_destroy` attributes on static protected data. Although not strictly necessary, this attribute allows omission of exit-time destructors, which adds unnecessary shutdown overhead. Virtually all of these insertions occur on browser feature flags (§4.2.4), of which there are thousands distributed throughout the browser codebase. We used a `sed` script to automatically append this attribute to individual feature flag instances.

4.4.4. Discussion. Our results show that our data-flow integrity approach is fully functional with limited performance overhead. Although not directly comparable, our approach imposes 28% geomean overhead to Chromium on browser benchmarks, compared to 104% average overhead on the SPEC CPU2000 benchmarks by the original inter-procedural approach [39]. By integrating our annotations into the programming language, we minimize the effort needed to deploy data-flow integrity, with the core of our annotations and compatibility fixes amounting to only 644 lines of code (Table 4.6) throughout the ~29.7 million line of code Chromium C/C++ codebase. Our pre-annotated STL datatypes simplify the deployment process, and only need manual code duplication for additional custom datatypes, which maximizes reusability for other applications.

As a research prototype, HerQules-DeiFIed is not fully optimized. System call synchronization is the key latency bottleneck. We believe the performance of system call synchronization could be significantly improved by adding side-effect-free system calls that do not need to be paused to the allowlist, or even by limiting synchronization to solely sensitive system calls (at the risk of providing less comprehensive protection). Overall system performance (overhead and latency) could also be improved by rewriting our verifier program to be multi-threaded, which would reduce the event reception and processing delay for all protected processes.

4.5. Summary

In this chapter, we present HerQules-DeiFIed, an integrity framework for real-world programs that extends HerQules. Compared to past work on data-flow integrity, our approach avoids the inaccuracy and inefficiency of pointer alias analysis. We rely on a hardware-based AppendWrite primitive and compiler instrumentation to transmit runtime events whenever security-sensitive program data are accessed, and a concurrent verifier process to perform asynchronous policy checking before system calls. We use simple language-based annotations to mark data for automatic compiler instrumentation, and include a pre-annotated C++ standard library. Our data-flow integrity approach is the first to generalize to existing programs written in both C and C++, and to protect a web browser. We demonstrate in our evaluation that our design offers effective protection, has limited overhead, and minimizes code change.

CHAPTER 5

Conclusion and Future Work

In this thesis, we have shown that adding a small, secure, and efficient AppendWrite inter-process communication (IPC) primitive has multiple benefits. We use it to develop HerQules and HerQules-DeiFIed, a framework for enforcing program integrity policies, by sending asynchronous software-defined execution events to a verifier that performs concurrent policy checking. We show that this permits new approaches for control-flow and data-flow integrity, which improve compatibility, performance, and precision over past work. Nevertheless, further improvements are possible. Below, we discuss three potential directions for future work.

5.1. Improving Other Mitigations

Although we have explored integrity protections to mitigate memory-safety bugs, we have not explored other uses for our AppendWrite primitive. With the exception of shared memory, existing IPC primitives impose overhead on program execution by performing system calls. By improving the security properties of shared memory IPC, we believe that AppendWrite can be applied to other problem domains where system calls were previously necessary. For example, the Mojo IPC framework is also used by the Mozilla Firefox web browser, but we did not benchmark our AppendWrite-based implementation there to determine whether its IPC workload differs, and how that would affect overall browser performance.

Similarly, our HerQules-DeiFIed framework could be used for other program safety mitigations, beyond control-flow and data-flow integrity. Type confusions are a category of memory safety violations that have received limited study, yet are a recurring security problem for optimized just-in-time compilers like browser JavaScript runtimes. Past work [53, 189, 200] has also proposed using runtime taint tracking to enforce information-flow security policies, but this imposes storage and processing overhead within the instrumented program. Instead, an instrumented program could simply transmit dataflow events during execution to our external verifier program, which would be responsible for tracking data movement and enforcing the underlying security policy.

5.2. Optimizing Synchronization Performance

Despite minimal decrease in overall benchmark performance under our control-flow integrity and data-flow integrity designs, system call synchronization can significantly increase rendering latency, as shown by our Chromium benchmarks. We believe that various optimizations could improve its performance.

Our Mojo-AppendWrite benchmarks show that busy-waiting for new messages instead of suspending via the `FUTEX_WAIT_MULTIPLE` feature of `futex` can significantly improve

performance. This suggests that our verifier may also be experiencing similar overhead, since it relies on the exact same feature. We believe this may be increasing message processing latency, either because the kernel scheduler delays resumption of the verifier process, or because the kernel implementation of the multiple-wait feature is suboptimal.

Similarly, our verifier implementation is only single-threaded, which can result in message processing delays when multiple protected processes are concurrently transmitting events, since events must be processed sequentially. A multi-threaded verifier that partitions protected processes among worker threads would resolve this problem, improving message processing performance.

Finally, placing *side-effect-free* system calls on an allowlist would reduce the number of systems calls that must be suspended. These system calls do not modify the system state, and thus would always be safe to execute, even by compromised processes. Not only would this reduce System-Call event traffic, but it would also decrease the total amount of time that protected processes are suspended.

5.3. Increasing Developer Usability

Two aspects of our design reduce the usability of our approach. Although we have developed an FPGA-based implementation and software-only model of AppendWrite, a lack of existing hardware support for this primitive makes it difficult to prototype or benchmark our system. An alternative construction of AppendWrite via e.g., Intel Memory Protection Keys for Userspace (MPK/PKU), or `PTWRITE` for Intel Processor Trace, could bridge this gap, by making it possible to deploy fully-functional HerQules-DeiFIed on existing hardware.

Our language-based data-flow integrity annotations impose friction on developers by requiring manual annotation, duplication, and correction of custom datatypes for compatibility. We believe that this could be improved by tracking protected data only via a compile-time type attribute, instead of actually lowering it as a first-class datatype. Runtime code to generate events would be inserted when the source abstract syntax tree is lowered to compiler intermediate representation, allowing the protected datatype to share the same underlying implementation as its unprotected equivalent, assuming that the underlying layouts are compatible and no additional padding is needed.

Bibliography

[1] Memory safety - The Chromium Projects. URL https://www.chromium.org/Home/chromium-security/memory-safety.

[2] Data Plane Development Kit, 2010. URL https://www.dpdk.org/.

[3] Control-flow Enforcement Technology Specification, June 2016. URL https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[4] System V Application Binary Interface: AMD64 Architecture Processor Supplement, Jan. 2018. URL https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[5] Control Flow Guard - Win32 apps, May 2018. URL https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[6] Control Flow Integrity | Android Open Source Project, Jan. 2020. URL https://source.android.com/devices/tech/debug/cfi.

[7] Intel® 64 and IA-32 Architectures Software Developer's Manual, May 2020. URL https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[8] Oxidation, Nov. 2020. URL https://wiki.mozilla.org/Oxidation.

[9] I. J. S. 22. ISO/IEC 14882:2017, Dec. 2017. URL https://www.iso.org/standard/68564.html.

[10] M. Abadi, M. Budiu, l. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 2005 ACM SIGSAC Conference on Computer and Communications Security - CCS '05*, pages 340–340. ACM Press, 2005. ISBN 1-59593-226-7. doi:10.1145/1102120.1102165.

[11] S. Ainsworth and T. M. Jones. 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, pages 1277–1293, Lausanne, Switzerland, Mar. 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi:10.1145/3373376.3378463.

[12] P. Akritidis, M. Costa, M. Castro, and S. Hand. 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, pages 51–66, Montreal, Canada, Aug. 2009. USENIX Association. doi:10.5555/1855768.1855772.

[13] A. Almeida. Add futex2 syscalls, Feb. 2021. URL https://lore.kernel.org/lkml/20210215152404.250281-1-andrealmeid@collabora.com/.

[14] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, Jan. 1988. Association for Computing Machinery. ISBN 978-0-89791-252-5. doi:10.1145/73560.73561.

[15] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society. ISBN 978-1-891562-55-6. doi:10.14722/ndss.2019.23412.

[16] B. Azad. Project Zero: Examining Pointer Authentication on the iPhone XS, Feb. 2019. URL https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html.

[17] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1342–1353, Scottsdale, Arizona, USA, Nov. 2014. Association for Computing Machinery. ISBN 978-1-4503-2957-6.

doi:10.1145/2660267.2660378.

[18] A. Barth, C. Reis, and C. Jackson. The Security Architecture of the Chromium Browser. Technical report, Stanford University, 2008. URL https://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf.

[19] T. Becker. Cleanly Escaping the Chrome Sandbox, Apr. 2020. URL https://theori.io/research/escaping-chrome-sandbox/.

[20] G. Bertazi. Futex: Implement mechanism to wait on any of several futexes, July 2019. URL https://lkml.org/lkml/2019/7/30/1399.

[21] S. Bhatkar, D. C. DuVarney, and R. Sekar. 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, page 8, USA, Aug. 2003. USENIX Association. doi:10.5555/1251353.1251361.

[22] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 17, USA, July 2005. USENIX Association. doi:10.5555/1251398.1251415.

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

[24] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 268–279. ACM Press, 2015. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813691.

[25] A. Biondo, M. Conti, and D. Lain. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. ISBN 978-1-891562-49-5. doi:10.14722/ndss.2018.23318.

[26] blackngel. Malloc Des-Maleficarum. *Phrack Inc.*, 13(66), Nov. 2009. URL http://phrack.org/issues/66/10.html.

[27] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, volume 38 of *ASIACCS '11*, pages 30–30. Association for Computing Machinery, 2011. ISBN 978-1-4503-0564-8. doi:10.1145/1966913.1966919.

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

[29] M. Brand. Virtually Unlimited Memory: Escaping the Chrome Sandbox, Apr. 2019. URL https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html.

[30] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and Preventing Bugs in JavaScript Bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, May 2017. doi:10.1109/SP.2017.68.

[31] F. Brown, D. Stefan, and D. Engler. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216, 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/brown.

[32] D. Bruening and Q. Zhao. Practical memory checking with Dr. Memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223, Apr. 2011. doi:10.1109/CGO.2011.5764689.

[33] J. Bucek, K.-D. Lange, and J. v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, Berlin, Germany, Apr. 2018. Association for Computing Machinery. ISBN 978-1-4503-5629-9. doi:10.1145/3185768.3185771.

[34] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1):16:1–16:33, Apr.

2017. ISSN 0360-0300. doi:10.1145/3054924.

[35] N. Burow, X. Zhang, and M. Payer. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, May 2019. doi:10.1109/SP.2019.00076.

[36] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 385–399, USA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. doi:10.5555/2671225.2671250.

[37] N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 97–111. USENIX Association, 2012. doi:10.5555/2362793.2362800.

[38] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 161–176, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-931971-23-2. doi:10.5555/2831143.2831154.

[39] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, Nov. 2006. USENIX Association. ISBN 978-1-931971-47-8. doi:10.5555/1298455.1298470.

[40] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 45–58, New York, NY, USA, Oct. 2009. Association for Computing Machinery. ISBN 978-1-60558-752-3. doi:10.1145/1629575.1629581.

[41] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, Chicago, Illinois, USA, Oct. 2010. Association for Computing Machinery. ISBN 978-1-4503-0245-6. doi:10.1145/1866307.1866370.

[42] D. D. Chen, W. S. Lim, M. Bakhshalipour, P. B. Gibbons, J. C. Hoe, and B. Parno. 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*, pages 773–788, USA, Apr. 2021. ACM. ISBN 978-1-4503-8317-2. doi:10.1145/3445814.3446736.

[43] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: Get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 227–238, New York, NY, USA, Oct. 2011. Association for Computing Machinery. ISBN 978-1-4503-0948-6. doi:10.1145/2046707.2046734.

[44] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 12, Baltimore, MD, July 2005. USENIX Association. doi:10.5555/1251398.1251410.

[45] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-Grain Program Monitoring. In *2008 International Symposium on Computer Architecture*, pages 377–388, June 2008. doi:10.1109/ISCA.2008.20.

[46] S. Codes. Applications That Run Chromium Without The Sandbox, May 2021. URL `https://github.com/sickcodes/no-sandbox`.

[47] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6): 565–584, Oct. 1993. ISSN 0167-4048. doi:10.1016/0167-4048(93)90054-9.

[48] P. Collingbourne. Control Flow Integrity Design Documentation, 2015. URL `https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html`.

[49] P. Collingbourne. Whole Program Devirtualization. Google, Inc, Feb. 2016. URL `https://reviews.llvm.org/D16795`.

[50] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. 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*, pages 952–963, Denver, Colorado, USA, 2015. ACM Press. ISBN 978-1-4503-3832-5.

doi:10.1145/2810103.2813671.

[51] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 5, San Antonio, Texas, Jan. 1998. USENIX Association. doi:10.5555/1267549.1267554.

[52] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, May 2015. doi:10.1109/SP.2015.52.

[53] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture - ISCA '07*, pages 482–482. ACM Press, 2007. ISBN 978-1-59593-706-3. doi:10.1145/1250662.1250722.

[54] T. H. Dang, P. Maniatis, and D. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15*, pages 555–566. ACM Press, 2015. ISBN 978-1-4503-3245-3. doi:10.1145/2714576.2714635.

[55] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. 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, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. doi:10.5555/2671225.2671251.

[56] D. Delorie. MallocInternals - glibc wiki, May 2019. URL `https://sourceware.org/glibc/wiki/MallocInternals`.

[57] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 137–148, Dec. 2010. doi:10.1109/MICRO.2010.17.

[58] M. Desnoyers. Using the Linux Kernel Tracepoints, 2008. URL `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`.

[59] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, Mar. 2015. Association for Computing Machinery. ISBN 978-1-4503-2835-7. doi:10.1145/2694344.2694383.

[60] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, Shanghai, China, May 2006. Association for Computing Machinery. ISBN 978-1-59593-375-1. doi:10.1145/1134285.1134309.

[61] R. Ding, C. Qian, C. Song, W. Harris, T. Kim, and W. Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 131–148, Vancouver, BC, Canada, Aug. 2017. USENIX Association. ISBN 978-1-931971-40-9. doi:10.5555/3241189.3241201.

[62] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupe, and Y. Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society. ISBN 978-1-891562-66-2. doi:10.14722/ndss.2021.24224.

[63] G. J. Duck and R. H. C. Yap. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi:10.1145/2892208.2892212.

[64] G. J. Duck and R. H. C. Yap. EffectiveSan: Type and memory error detection using dynamically typed C/C++. *ACM SIGPLAN Notices*, 53(4):181–195, June 2018. ISSN 0362-1340. doi:10.1145/3296979.3192388.

[65] G. J. Duck, R. H. C. Yap, and L. Cavallaro. Stack Bounds Protection with Low Fat Pointers. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. ISBN 978-1-891562-46-4. doi:10.14722/ndss.2017.23287.

[66] I. El Hajj, A. Merritt, G. Zellweger, D. Milojicic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan. 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, pages 353–368, New York, NY, USA, Mar. 2016. Association for Computing Machinery. ISBN 978-1-4503-4091-5. doi:10.1145/2872362.2872366.

[67] l. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, USA, Nov. 2006. USENIX Association. ISBN 978-1-931971-47-8. doi:10.5555/1298455.1298463.

[68] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. Garcia, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh. Fidelius: Protecting User Secrets from Compromised Browsers. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 264–280, May 2019. doi:10.1109/SP.2019.00036.

[69] A. N. Evans, B. Campbell, and M. L. Soffa. Is Rust Used Safely by Software Developers? *arXiv:2007.00752 [cs]*, July 2020. doi:10.1145/3377811.3380413.

[70] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy*, pages 781–796. IEEE, May 2015. ISBN 978-1-4673-6949-7. doi:10.1109/SP.2015.53.

[71] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 901–913. ACM Press, 2015. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813646.

[72] R. Felker. Musl libc, Feb. 2020. URL `https://musl.libc.org/`.

[73] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Washington, D.C., Aug. 2001. USENIX Association. doi:10.5555/1251327.1251332.

[74] S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi, and B. Grot. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 108–119, Feb. 2014. doi:10.1109/HPCA.2014.6835922.

[75] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. *ACM SIGARCH Computer Architecture News*, 45(1):585–598, Apr. 2017. ISSN 0163-5964. doi:10.1145/3093337.3037716.

[76] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, San Jose, CA, May 2014. IEEE. ISBN 978-1-4799-4686-0. doi:10.1109/SP.2014.43.

[77] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. 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, pages 417–432, USA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. doi:10.5555/2671225.2671252.

[78] E. Göktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 105–119, USA, Aug. 2016. USENIX Association. ISBN 978-1-931971-32-4. doi:10.5555/3241094.3241104.

[79] W. Glozer. Wrk - a HTTP benchmarking tool, Apr. 2019. URL `https://github.com/wg/wrk`.

[80] A. Gough. Enabling Hardware-enforced Stack Protection (cetcompat) in Chrome, May 2021. URL `https://security.googleblog.com/2021/05/enabling-hardware-enforced-stack.html`.

[81] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, Feb. 2017. doi:10.14722/ndss.2017.23271.

[82] M. Gretton-Dann. Arm Architecture Armv8.5-A Announcement, Sept. 2018. URL `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a`.

[83] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, Berlin, Germany, May 2002. Association for Computing Machinery. ISBN 978-1-58113-463-6. doi:10.1145/512529.512563.

[84] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In E. Bodden, M. Payer, and E. Athanasopoulos, editors, *Engineering Secure Software and Systems*, Lecture Notes in Computer Science, pages 161–176, Cham, 2017. Springer International Publishing. ISBN 978-3-319-62105-0. doi:10.1007/978-3-319-62105-0_11.

[85] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. 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, pages 173–184, Scottsdale, Arizona, USA, Mar. 2017. Association for Computing Machinery. ISBN 978-1-4503-4523-1. doi:10.1145/3029806.3029830.

[86] T. Guo, P. Zhang, X. Wang, and Q. Wei. GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation. In *2013 Second International Conference on Informatics Applications (ICIA)*, pages 212–215, Sept. 2013. doi:10.1109/ICoIA.2013.6650258.

[87] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN 978-1-4503-4988-8. doi:10.1145/3062341.3062363.

[88] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, Oct. 2016. Association for Computing Machinery. ISBN 978-1-4503-4139-4. doi:10.1145/2976749.2978405.

[89] H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society. ISBN 978-1-891562-55-6. doi:10.14722/ndss.2019.23263.

[90] D. Hansen. X86: Remove Intel MPX, Aug. 2018. URL `https://lore.kernel.org/patchwork/patch/1025952/`.

[91] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 135–144, New York, NY, USA, Mar. 2012. Association for Computing Machinery. ISBN 978-1-4503-1206-6. doi:10.1145/2259016.2259034.

[92] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.

[93] J. L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964. doi:10.1145/1186736.1186737.

[94] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, May 2012. doi:10.1109/SP.2012.39.

[95] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012. doi:10.5555/2362793.2362831.

[96] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 177–192, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-931971-23-2. doi:10.5555/2831143.2831155.

[97] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, San Jose, CA, May 2016. IEEE. ISBN 978-1-5090-0824-7. doi:10.1109/SP.2016.62.

[98] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2373–2387, New York, NY, USA, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-4946-8. doi:10.1145/3133956.3134062.

[99] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. Доверяй, Но Проверяй: SFI safety for native-compiled Wasm. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society. ISBN 978-1-891562-66-2. doi:10.14722/ndss.2021.24078.

[100] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*, 1997.

[101] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu. Kernel Probes (Kprobes), 2005. URL `https://www.kernel.org/doc/Documentation/kprobes.txt`.

[102] C. Kerschbaumer, T. Ritter, and F. Braun. Hardening Firefox against Injection Attacks. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 653–663, Sept. 2020. doi:10.1109/EuroSPW51379.2020.00094.

[103] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang. Origin-sensitive control flow integrity. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 195–211, Santa Clara, CA, USA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. doi:10.5555/3361338.3361353.

[104] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 437–452, Belgrade, Serbia, Apr. 2017. Association for Computing Machinery. ISBN 978-1-4503-4938-3. doi:10.1145/3064176.3064217.

[105] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida. Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth European Conference on Computer Systems*, EuroSys '18, pages 1–14, Porto, Portugal, 2018. ACM Press. ISBN 978-1-4503-5584-1. doi:10.1145/3190508.3190553.

[106] V. Kuznetsov. Re: Resend: Code-Pointer Integrity + Software Fault Isolation, July 2020.

[107] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163. USENIX Association, Oct. 2014. ISBN 978-1-931971-16-4. doi:10.5555/2685048.2685061.

[108] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, and D. Song. Poster: Getting The Point(er): On the Feasibility of Attacks on Code-Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy*, page 2. IEEE, May 2015.

[109] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security - CCS '13*, pages 721–732. ACM Press, 2013. ISBN 978-1-4503-2477-9. doi:10.1145/2508859.2516713.

[110] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, Mar. 2004. doi:10.1109/CGO.2004.1281665.

[111] D. Lea. A Memory Allocator, Apr. 2000. URL `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[112] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings 2015 Network and Distributed System Security Symposium*, pages 8–11. Internet Society, 2015. ISBN 1-891562-38-X. doi:10.14722/ndss.2015.23238.

[113] B. Lee, C. Song, T. Kim, and W. Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 81–96. USENIX Association, Aug. 2015. ISBN 978-1-939133-11-3. doi:10.5555/2831143.2831149.

[114] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630, 2020. ISBN 978-1-939133-17-5. URL `http://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung`.

[115] D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020. URL `https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann`.

[116] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, Jan. 1979. ISSN 0164-0925.

doi:10.1145/357062.357071.

[117] H. Liljestrand, T. Nyman, J.-E. Ekberg, and N. Asokan. Authenticated Call Stack. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 223:1–223:2, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6725-7. doi:10.1145/3316781.3322469.

[118] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 177–194, USA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. doi:10.5555/3361338.3361352.

[119] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, H. Mitchel, L. Dong, and P. Gupta. 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, pages 261–264, Monterey, California, USA, Feb. 2009. Association for Computing Machinery. ISBN 978-1-60558-410-2. doi:10.1145/1508128.1508172.

[120] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 973–990, Baltimore, MD, USA, Aug. 2018. USENIX Association. ISBN 978-1-931971-46-1. doi:10.5555/3277203.3277276.

[121] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. 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)*, Berkeley, Calif., 2016. USENIX Association. ISBN 978-1-931971-33-1. doi:10.5555/3026877.3026882.

[122] G. Liu and Z. Feng. The Most Secure Browser? Pwning Chrome from 2016 to 2019, Aug. 2019. URL `https://infocondb.org/con/black-hat/black-hat-usa-2019/the-most-secure-browser-pwning-chrome-from-2016-to-2019`.

[123] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619, New York, NY, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813690.

[124] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 529–540, Feb. 2017. doi:10.1109/HPCA.2017.18.

[125] E. Luebbers, S. Liu, and M. Chu. Simplify Software Integration for FPGA Accelerators with OPAE, 2017. URL `https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf`.

[126] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 941–951, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813676.

[127] N. D. Matsakis and F. S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, Oct. 2014. Association for Computing Machinery. ISBN 978-1-4503-3217-0. doi:10.1145/2663171.2663188.

[128] J. McCall. Pointer Authentication, Oct. 2019. URL `https://github.com/apple/llvm-project/blob/apple/master/clang/docs/PointerAuthentication.rst`.

[129] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape, Feb. 2019. URL `https://github.com/microsoft/MSRC-Security-Research`.

[130] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. 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, pages 245–258, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi:10.1145/1542476.1542504.

[131] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0054-4. doi:10.1145/1806651.1806657.

[132] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2. doi:10.1109/ISCA.2012.6237017.

[133] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716, 2020. ISBN 978-1-939133-17-5. URL `http://www.usenix.org/conference/usenixsecurity20/presentation/narayan`.

[134] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, May 2005. ISSN 1581134509. doi:10.1145/1065887.1065892.

[135] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007. ISSN 0362-1340. doi:10.1145/1273442.1250746.

[136] R. Nikhil. 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.*, pages 69–70, June 2004. doi:10.1109/MEMCOD.2004.1459818.

[137] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, Denver, Colorado, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813644.

[138] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. 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):28:1–28:30, June 2018. doi:10.1145/3224423.

[139] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 80–85, Nov. 2011. doi:10.1109/ReConFig.2011.4.

[140] A. One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), 1996. URL `http://phrack.org/issues/49/14.html`.

[141] P. Padlewski. 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, pages 42–44, Vancouver, BC, Canada, Oct. 2017. Association for Computing Machinery. ISBN 978-1-4503-5514-8. doi:10.1145/3135932.3135947.

[142] P. Padlewski, K. Pszeniczny, and R. Smith. Modeling the Invariance of Virtual Pointers in LLVM. *arXiv:2003.04228 [cs]*, Feb. 2020. URL `http://arxiv.org/abs/2003.04228`.

[143] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615, May 2012. doi:10.1109/SP.2012.41.

[144] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, May 2020. doi:10.1109/SP40000.2020.00067.

[145] P. Phantasmagoria. The Malloc Maleficarum, Oct. 2005. URL `https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`.

[146] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577, May 2020. doi:10.1109/SP40000.2020.00041.

[147] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274, May 2019. doi:10.1109/SP.2019.00064.

[148] H. Pulapaka. Understanding Hardware-enforced Stack Protection, Mar. 2020. URL `https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815`.

[149] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 763–779, New York, NY, USA, June 2020. Association for Computing Machinery. ISBN 978-1-4503-7613-6. doi:10.1145/3385412.3386036.

[150] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994. ISSN 0164-0925. doi:10.1145/186025.186041.

[151] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 219–232, New York, NY, USA, Apr. 2009. Association for Computing Machinery. ISBN 978-1-60558-482-9. doi:10.1145/1519065.1519090.

[152] C. Reis, A. Moshchuk, and N. Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1661–1678, USA, Aug. 2019. USENIX Association. ISBN 978-1-939133-06-9. doi:10.5555/3361338.3361454.

[153] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, Jan. 1988. Association for Computing Machinery. ISBN 978-0-89791-252-5. doi:10.1145/73560.73562.

[154] H. Rosier. RIPE64. National University of Singapore, May 2019. URL `https://github.com/hrosier/ripe64`.

[155] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, NDSS '04, pages 159–169, 2004.

[156] D. Sanchez and C. Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 41(3):475–486, June 2013. ISSN 0163-5964. doi:10.1145/2508148.2485963.

[157] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan. Practical Byte-Granular Memory Blacklisting using Califorms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 558–571, New York, NY, USA, Oct. 2019. Association for Computing Machinery. ISBN 978-1-4503-6938-1. doi:10.1145/3352460.3358299.

[158] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, May 2015. doi:10.1109/SP.2015.51.

[159] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. J. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Conference on Security - SEC '10*, USENIX Security'10, pages 1–12. USENIX Association, Aug. 2010. ISBN 8887666655554. doi:10.5555/1929820.1929822.

[160] J. Seibert, H. Okhravi, and E. Söderström. 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, pages 54–65, Scottsdale, Arizona, USA, Nov. 2014. Association for Computing Machinery. ISBN 978-1-4503-2957-6. doi:10.1145/2660267.2660309.

[161] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. doi:10.1145/1791194.1791203.

[162] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. doi:10.5555/2342821.2342849.

[163] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. ISSN 0001-0782. doi:10.1145/1785414.1785443.

[164] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, Alexandria, Virginia, USA, Oct. 2007. Association for Computing Machinery. ISBN 978-1-59593-703-2. doi:10.1145/1315245.1315313.

[165] V. Shanbhogue, D. Gupta, and R. Sahita. 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, pages 1–11, Phoenix, AZ, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-7226-8. doi:10.1145/3337167.3337175.

[166] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588, May 2013. doi:10.1109/SP.2013.45.

[167] C. Song. Re: Kernel DFI Paper, May 2021.

[168] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016. Internet Society. ISBN 978-1-891562-41-9. doi:10.14722/ndss.2016.23218.

[169] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, May 2016. ISBN 978-1-5090-0824-7. doi:10.1109/SP.2016.9.

[170] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, May 2019. doi:10.1109/SP.2019.00010.

[171] O. Stannard. Dead Virtual Function Elimination. Linaro, June 2019. URL `https://reviews.llvm.org/D63932`.

[172] E. Stepanov and K. Serebryany. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 46–55, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. doi:10.5555/2738600.2738607.

[173] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, Nuremburg, Germany, Mar. 2009. Association for Computing Machinery. ISBN 978-1-60558-472-0. doi:10.1145/1519144.1519145.

[174] I. Sysoev. NGINX. Nginx, Inc., Apr. 2020. URL `https://www.nginx.com/`.

[175] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013. doi:10.1109/SP.2013.13.

[176] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 256–267, Denver, Colorado, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813685.

[177] J. Tang. Exploring Control Flow Guard in Windows 10, 2015. URL `https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf`.

[178] I. L. Taylor. Gold, Feb. 2020. URL `https://sourceware.org/binutils/`.

[179] T. P. Team. Address Space Layout Randomization, Mar. 2003. URL `https://pax.grsecurity.net/docs/aslr.txt`.

[180] T. P. Team. NOEXEC, May 2003. URL `https://pax.grsecurity.net/docs/noexec.txt`.

[181] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, l. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium - SEC '14*, pages 941–955, 2014. ISBN 978-1-931971-15-7. doi:10.5555/2671225.2671285.

[182] S. Tolvanen. Control Flow Integrity in the Android kernel, Oct. 2018. URL `https://security.googleblog.com/2018/10/posted-by-sami-tolvanen-staff-software.html`.

[183] S. Tolvanen. Protecting against code reuse in the Linux kernel with Shadow Call Stack, Oct. 2019. URL `https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html`.

[184] V. Tsyrklevich. 908597 - Deprecate SafeStack - chromium. Google, Inc, Nov. 2018. URL `https://bugs.chromium.org/p/chromium/issues/detail?id=908597`.

[185] J. Tyhach, M. Hutton, S. Atsatt, A. Rahman, B. Vest, D. Lewis, M. Langhammer, S. Shumarayev, T. Hoang, A. Chan, D.-M. Choi, D. Oh, H.-C. Lee, J. Chui, K. C. Sia, E. Kok, W.-Y. Koay, and B.-J.

Ang. Arria™ 10 device architecture. In *2015 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, Sept. 2015. doi:10.1109/CICC.2015.7338368.

[186] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, Denver, Colorado, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi:10.1145/2810103.2813673.

[187] A. Varga and R. Hornig. 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, pages 1–10, Brussels, BEL, Mar. 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). ISBN 978-963-9799-20-2. doi:10.5555/1416222.1416290.

[188] S. Veggalam, S. Rawat, I. Haller, and H. Bos. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, editors, *Computer Security – ESORICS 2016*, Lecture Notes in Computer Science, pages 581–601, Cham, 2016. Springer International Publishing. ISBN 978-3-319-45744-4. doi:10.1007/978-3-319-45744-4_29.

[189] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184. IEEE, Feb. 2008. ISBN 978-1-4244-2070-4. doi:10.1109/HPCA.2008.4658637.

[190] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles - SOSP '93*, pages 203–216. ACM Press, 1993. ISBN 0-89791-632-8. doi:10.1145/168619.168635.

[191] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, May 2017. doi:10.1109/SP.2017.23.

[192] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security - CCS '12*, pages 157–157. ACM Press, 2012. ISBN 978-1-4503-1651-4. doi:10.1145/2382196.2382216.

[193] C. Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 53–65, New York, NY, USA, Jan. 2018. Association for Computing Machinery. ISBN 978-1-4503-5586-5. doi:10.1145/3167082.

[194] C. Watt, A. Rossberg, and J. Pichon-Pharabod. Weakening WebAssembly. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):133:1–133:28, Oct. 2019. doi:10.1145/3360559.

[195] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. Tomasz Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. W. Moore, P. G. Neumann, and R. N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, May 2020. doi:10.1109/SP40000.2020.00098.

[196] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. RIPE: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, Orlando, Florida, USA, Dec. 2011. Association for Computing Machinery. ISBN 978-1-4503-0672-0. doi:10.1145/2076732.2076739.

[197] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 367–382, Savannah, GA, USA, Nov. 2016. USENIX Association. ISBN 978-1-931971-33-1. doi:10.5555/3026877.3026906.

[198] N. Williamson. Exploiting Chrome IPC, Nov. 2018. URL `https://powerofcommunity.net/poc2018/ned.pdf`.

[199] H. Xu, Z. Chen, M. Sun, and Y. Zhou. Memory-Safety Challenge Considered Solved? An Empirical Study with All Rust CVEs. *arXiv:2003.03296 [cs]*, May 2020. URL `http://arxiv.org/abs/2003.03296`.

[200] W. Xu, S. Bhatkar, and R. Sekar. 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, Vancouver, B.C., Canada, July 2006. USENIX Association. doi:10.5555/1267336.1267345.

[201] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin. 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, pages 1805–1821. USENIX Association, Aug. 2019. ISBN 978-1-939133-06-9. doi:10.5555/3361338.3361463.

[202] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, May 2009. ISBN 978-0-7695-3633-0. doi:10.1109/SP.2009.25.

[203] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, Apr. 2010. Association for Computing Machinery. ISBN 978-1-60558-936-7. doi:10.1145/1755688.1755707.

[204] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, May 2013. doi:10.1109/SP.2013.44.

[205] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 337–352, Washington, D.C., Aug. 2013. USENIX Association. ISBN 978-1-931971-03-4. doi:10.5555/2534766.2534796.

[206] T. Zhang, D. Lee, and C. Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 631–644, New York, NY, USA, Apr. 2019. Association for Computing Machinery. ISBN 978-1-4503-6240-5. doi:10.1145/3297858.3304017.