Thesis Proposal

Avoiding and Measuring Memory Safety Bugs

Daming Dominic Chen

October 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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.

Contents

CHAPTER 1

Introduction

Many computer programs written in *unsafe* languages like C and C++ perform low-level memory operations involving pointers, which may accidentally introduce memory safety bugs [142] due to developer error. Common examples of these bugs include buffer overflows, use-after-frees, and double frees, which can all be used by attackers to exploit programs. Indeed, statistics from both Google Chrome [1] and Microsoft [105] have shown that ~70% of all security vulnerabilities in their products involve memory safety bugs. Recent research [61, 122, 162] has also demonstrated that these bugs can affect programs written in safe languages like Rust that may contain unsafe code.

Past work has proposed various strategies to detect or mitigate such bugs. These include adding runtime checks (§2.1), randomizing program layout to hide data (§2.2), and validating program execution against models of expected program behavior (§2.3, §2.4). Nevertheless, many of these proposals suffer from high runtime overhead, brittle designs, and/or imprecise analyses, which limits their efficiency and effectiveness.

Past work has also developed various methods for measuring the impact of these and other security bugs. Of particular interest are embedded devices, which are widely-deployed and occupy a privileged network position, yet are rarely-updated and riddled with security bugs. One approach is internet-scale scanning (§2.5), which requires a feasible network search space, and that remote hosts be both online and remotely-accessible. Another is firmware analysis (§2.6), but which may not accurately model runtime interactions involving multiple programs and scripts.

This thesis addresses these problems as follows: *First, we show that emulation can be used to automatically measure the impact of software vulnerabilities in embedded devices. Second, we develop scalable protections for memory safety bugs in common software platforms, and quantify our improvements in terms of correctness, effectiveness, and performance.* We provide an overview of our contributions and proposed work below:

(1) We build FIRMADYNE [32], a system that enables large-scale dynamic analysis of Linux-based embedded devices, and use it to measure the prevalence of various vulnerabilities–including memory safety bugs–across our dataset, as described in §3.
(2) We develop multiple mechanisms for mitigating memory safety bugs by protecting program integrity. In §4, we extend control-flow integrity to WebAssembly[1], and in §5, we design HERQULES, an integrity framework that leverages hardware-enforced message queues (under submission).
(3) We propose to extend integrity protections to program memory safety, and outline a timeline for this thesis, in §6.

[1] `https://reviews.llvm.org/D87258`

CHAPTER 2

Background

2.1. Memory Safety

Early work proposed various mitigation strategies to defend against memory access bugs, either by preventing risky behavior or checking for overflows. These include stack canaries [41], no-execute memory [146] (NX or Data Execution Prevention), and W^X [147], all of which are now commonplace.

Subsequent work has focused directly on detecting [138] these memory access bugs at runtime. Many *spatial memory safety* approaches add bounds checking to detect out-of-bounds accesses. Both pointer- and object-based designs ensure that each pointer or access refers to the correct object, whereas location-based designs only ensure that each access refers to a valid object. Pointer-based designs track boundaries either by storing them within the pointers themselves, known as *fat pointers* [74, 89, 93, 112], or by storing metadata separately [55, 56, 109] using an associative data structure. However, certain work [74, 112] relies on modifications to the language type system, which may require programs be rewritten. Object-based designs [11, 52, 87, 127, 164] instead store boundaries in each object, but may need to insert additional padding to support one-past-the-end pointers that are valid when not dereferenced. Location-based approaches [78, 79, 132] track memory validity and/or place guard regions adjacent to each object. Alternative hardware-based designs [50, 116, 161] reduce overhead of software-based designs by offloading bounds checking and/or storage.

Temporal memory safety defends against orthogonal use-after-free and double free bugs, which occur when dynamically-allocated memory is accessed after being freed, or memory is freed twice, respectively. Proposed solutions include eliminating dangling pointers [94, 158] to freed memory, and checking memory validity upon dereference [110]. Some designs provide both spatial and temporal memory safety, whether in hardware [51, 111, 128, 169] or in software [24, 113] via dynamic binary instrumentation.

2.2. Information Hiding

Another mitigation strategy is information hiding, which provides probabilistic security. Runtime randomization of program content [36, 119] or layout [16, 17, 83, 146] has been used to deter exploitation and protect sensitive data. Common strategies include Address Space Layout Randomization; however, side-channel [72, 131, 135, 140] or other information disclosure attacks have been used to defeat information hiding. In response, various countermeasures have been proposed, including rerandomization [19, 155, 160], execute-only memory [13, 42, 143], and guard pages.

2.3. Control-Flow Integrity

Control-flow integrity [7, 26] (CFI) mitigates certain memory safety bugs by ensuring the integrity of transition edges on the program control-flow graph (CFG). Forward edges occur at `branch`, `jump`, or `call` instructions, and depending on whether the destination is statically identifiable, can either be *direct* or *indirect*. Backward edges occur at `return` instructions that resume execution in the calling function.

2.3.1. Coarse-Grained CFI. Coarse-grained designs approximate the control flow of a program using a limited number of equivalence classes. Some use one equivalence class for all address-taken functions, including Microsoft's Control Flow Guard [4] (MSCFG) and the Indirect Branch Tracking (IBT) component of Intel's Control Enforcement Technology [2, 134] (CET). Others form equivalence classes based on call target type [166, 168] or callee arity [148]. Modern Clang/LLVM CFI [37] uses the callee's language-level type, generating checks using bitmasks against aligned jump tables for each equivalence class. We observe in §5 that it has low overhead, but exhibits false positives, and requires recompiling with link-time optimization, like many other designs, which can increase compilation time. By default, casted pointer types are strictly matched, which increases false positives, although this behavior can be altered using additional compiler flags or a manual allowlist, which were not tested.

Coarse-grained approaches are the most widely-deployed due to their low overhead, but are vulnerable to code-reuse attacks [21, 28, 46, 68, 69], like return-oriented programming [31, 133] and jump-oriented programming [21]. MSCFG has been included since Windows 8.1 [144], though it was vulnerable to a now-patched design flaw [20], and is to be replaced [121] by CET in future Windows 10 releases. Google Chrome and certain Android devices [5, 149, 150] are built with modern Clang/LLVM CFI. ARM has added Branch Target Identification as an optional processor feature since ARMv8.5-A [73], which supports three equivalence classes.

2.3.2. Fine-Grained CFI. Fine-grained designs improve control flow precision by incorporating a *context-sensitive* analysis at each call site. These include object origin [88], pointer integrity (§2.3.2.1), and call paths.

Path-sensitive methods examine each potential caller on the call path to a given function, but since the total number of paths grows exponentially, some approximation or hardware acceleration is still necessary. Various mechanisms used by past work include hardware-accelerated path recording (e.g. Intel Last Branch Records [152], or Processor Trace [53, 75, 100]), call path merging [114, 152], and/or limiting checks to certain sensitive system calls (e.g. `sigaction`, `mmap`, `mprotect`, etc.) [53, 152]. In response, attacks [63] have evolved to target the inherent undecidability [123] of pointer aliasing.

2.3.2.1. *Pointer Integrity. Pointer integrity* is a state-of-the-art fine-grained approach that protects the *values* of sensitive pointers, which avoids approximation and ensures maximum context sensitivity by using a singleton equivalence set for each call site.

Code Pointer Integrity [91] (CPI) uses a recursive definition to identify sensitive pointers that can include certain data pointers, and places them in a separate *safe pointer store* (SPS). It also tracks and checks object boundaries to prevent buffer overflows from corrupting adjacent store entries. To protect return pointers, it places them on a *safe*

stack (§2.3.3). Initially, information hiding was used to protect both the safe store and the safe stack, but such hiding was shown [62, 70] vulnerable to disclosure attacks, and later replaced [92] with software fault isolation [154] for the SPS, at the cost of additional overhead. We observe in §5) that their final design has moderate overhead but causes many programs to crash or hang, does not implement proposed temporal safety checks, requires recompiling with LTO, and cannot be easily composed with existing binaries without adding abstraction (and hence overhead) to the SPS.

Cryptographically Enforced CFI [102] (CCFI) computes a message authentication code (MAC) for each sensitive pointer, which is stored in memory and rechecked upon use. The MAC uses a single round of the Advanced Encryption Standard (AES) block cipher, and includes various pointer properties, including its address, value, and certain metadata (e.g. class, language type). Hardware-accelerated AES instructions are used 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 act as a nonce. We observe in §5 that their approach has significant overhead, uses a non-standard AES construction[1], requires recompilation with LTO, 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 [97, 98] and industry [104], has used ARM's Pointer Authentication (PA) to implement pointer integrity. However, Apple's design [104] is a cryptographic MAC-based approach that provides even lower precision than CCFI, and also lacks use-after-free detection. For compatibility reasons, it omits the address of the sensitive pointer in the MAC computation, which allows replay attacks. As a workaround, it supports a separate *discriminator* to be used as a nonce, however their implementation uses a *constant* discriminator of zero for function pointers and C++ virtual table pointers. It is also specific to Apple's software and hardware, which are not accessible externally for development and testing, and was shown [12] vulnerable to a now-patched flaw that allowed MAC forgery.

2.3.3. Return Pointers. When a function is called, some architectures save *return pointers* on the stack, which can allow corruption and race attacks in multi-threaded programs. Coarse-grained approaches (§2.3.1), which many only check that e.g. the return address is preceded by a call instruction, are vulnerable to code-reuse attacks.

Subsequent works use *shadow stacks* [27, 45, 64, 134], which improve protecting by placing return pointers on a separate stack. Software-based approaches like Microsoft's Return Flow Guard [18] (MSRFG) protect this stack with information hiding, but are vulnerable to side channel or information disclosure attacks. Hardware-based approaches, including the shadow stack component of CET, solve these problems using special memory pages that are otherwise inaccessible.

Code Pointer Integrity [91] proposed a software *safe stack* [92], containing all objects that may overflow. It defeats some attacks [39] and has been adopted by Clang/LLVM [37], which adds guard pages to detect overflow. Despite low overhead, safe stack is vulnerable [70] to information disclosure attacks, and was disabled [151] in Google Chrome for this reason.

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

2.4. Data-flow Integrity

Non-control-data attacks [29, 33, 84, 85, 129] utilize memory safety bugs to modify program behavior without directly altering control flow. For example, changes to program data can indirectly influence evaluation of conditional expressions, or affect the semantics of system call arguments, all of which are not protected by CFI.

Data-flow integrity [30] is an analogue of control-flow integrity that computes static reaching definitions for each memory location using pointer analysis, and checks each read at runtime to ensure that the last definition was a member of the corresponding equivalence class. To protect the runtime definitions table, it uses software fault isolation, which requires instrumenting all memory writes. For intra-procedural analysis, it is flow-sensitive, and for inter-procedural analysis, it is context-insensitive, but neither are *field-sensitive*, which renders it unable to distinguish between different fields within a composite type. Subsequent work has applied this technique to protect kernel access control data [136], or developed [137] hardware implementations with better performance.

2.5. Internet-Scale Scanning

Internet-scale scanning non-invasively queries remote hosts for metadata, to determine if they may be affected by a given security vulnerability. However, scaling to IPv6 networks is challenging [108] due to the increased address space, and scanning requires that remote hosts be both online and remotely-accessible. Using tools like Censys [59], Masscan [71], Nmap [101], and ZMap [57], researchers have identified embedded devices with default access credentials [43], hosts that use vulnerable cryptographic keys [81], popular sites vulnerable to Heartbleed [58] and LogJam [8], and misissued TLS certificates [90].

2.6. Firmware Analysis

A variety of techniques have been used for firmware analysis. One common large-scale approach is static analysis, which examines the contents of firmware images without emulating or executing any programs. Past work has identified hardcoded SSL [80], SSH [107], and other [14] private keys, as well as backdoors and other security vulnerabilities [40] in commodity firmware.

Another technique is dynamic analysis; although it is more precise, it is less commonly used because it may require [165] analysts to obtain the physical hardware and manually interface with a debugging port on the device. Other techniques combine both using hybrid emulation [96], or symbolic execution [47].

Manual analysis has also aided in the discovery of specific vulnerabilities that affect various classes of embedded devices. These include LaserJet printers [44], cellular basebands [157], remote server management [22], and USB-attached peripherals [103, 115].

CHAPTER 3

Scalable dynamic analysis for Linux-based embedded devices

With the proliferation of the "Internet of Things", an increasing number of embedded devices are being connected to the Internet at an alarming rate. Commodity networking equipment such as routers and network-attached storage boxes are joined by IP cameras, thermostats, or even remotely-controllable power outlets. Nevertheless, many of these devices are controlled by vendor and chipset-specific firmware that is rarely, if ever, updated to address security vulnerabilities affecting these devices. Unfortunately, the poor security practices of these device vendors are only further exacerbated by the privileged network position that many of these devices occupy. For example, a wireless router is frequently the *first and only* line of defense between a user's computing equipment (e.g., laptops, mobile phones, and tablets) and the Internet. Since most vendors have not taken any initiative to improve the security of their devices, millions of home and small business networks are left vulnerable to both known and unknown threats.

To overcome these shortcomings, we develop FIRMADYNE, which leverages software-based full system emulation to enable large-scale automated dynamic analysis of commodity Linux-based embedded firmware. Our approach does not require network scanning (§2.5) or physical hardware (§2.6), allowing it to scale with additional computational resources. Additionally, our emulation approach enables dynamic analysis regardless of the underlying programming language, and provides actionable results in the form of a successful exploit for any given vulnerability. In doing so, we address a number of challenges that are characteristic of embedded devices, such as the presence of various hardware-specific peripherals, storage of persistent configuration in non-volatile memory (NVRAM), and dynamically-generated configuration files.

3.1. Design

3.1.1. Motivation. Dynamic analysis for embedded firmware can be performed at different levels of the abstraction hierarchy. In this section, we discuss a selection of potential vantage points, illustrate challenges and shortcomings for each, and argue why full system emulation is the most promising approach.

3.1.1.1. *Application-Level.* Perhaps the most straightforward approach is to statically extract application data, and execute it natively using a similar application. For example, it is possible to extract webpages served by a embedded web server, and host the same content using a regular web server like Apache. Unfortunately, this approach has multiple drawbacks that are incompatible with our design goal of creating a generic platform for dynamic analysis of embedded firmware.

An analysis of the firmware images in our dataset shows that many of these contain webpages which rely on non-standard extensions to server-side scripting languages

for access to hardware-specific functionality, such as NVRAM values. For example, hundreds of images in our dataset make use of the custom functions `get_conf()` in `PHP` and `nvram_get()` in `ASP.NET` to obtain device configuration values. However, this functionality is a custom addition that is not supported by their upstream open-source counterparts.

Similarly, other firmware images do not store webpages on the filesystem, but instead embed their content within the binary of a custom web server. Finally, such an approach can only detect vulnerabilities within the application-specific data (e.g., command injection vulnerabilities in `PHP` files), but not those within the application binary or other system components.

3.1.1.2. *Process-Level.* Another approach is to emulate the behavior of individual processes within the context of the original filesystem. This can be achieved by executing `QEMU` in user-mode as a single process emulator, constrained using `chroot` to the original filesystem. Thus, one could simply launch the original web server from the firmware image in `QEMU` to host the router web interface.

Unfortunately, this approach only partially obviates the concerns mentioned above. While an application would be able to execute within the context of the filesystem, specific hardware peripherals (e.g., NVRAM) are still unavailable. As a result, when an application attempts to access the NVRAM peripheral via `/dev/nvram`, it will likely terminate in error. Without precise knowledge of the desored system environment, the host environment can inadvertently affect dynamic analysis of individual processes by altering program execution.

Minor differences in execution environment can have a significant effect on program behavior. For example, the `alphafs` web server, used by multiple firmware images, verifies hardware-specific product and vendor IDs before accessing NVRAM. If these values are not present at predetermined physical memory addresses, the web server ceases operation and terminates with an error message. To this end, the web server uses the `mmap()` system call to access memory via `/dev/mem`, and checks specific offsets for the `ProductID` and `VendorID` of supported EEPROM chips, all of which is difficult to support in a user-mode emulator.

Similarly, due to limited write cycles on the primary storage device, many firmware images mount a temporary memory-backed filesystem at boot for volatile data. This filesystem is mounted and generated dynamically. As a result, the directories `/dev/` and `/etc/` may be symbolic links to subdirectories within the temporary filesystem, thus appearing broken when examined statically. For example, the firmware for the D-Link DIR-865L wireless router uses a startup script to populate configuration for applications, including the `lighttpd` web server. As a result, simple dynamic emulation of the `lighttpd` binary will fail, even with the original base filesystem in place.

3.1.1.3. *System-Level.* In comparison, a system-level emulation approach is able to overcome these aforementioned challenges. Interfaces for hardware peripherals will be present, allowing their functionality to be gracefully emulated. Accurate emulation of the full environment permits dynamically-generated data to be created in the same manner as on the real device. All processes launched by the system can be analyzed, including various daemons responsible for protocols such as HTTP, FTP, and Telnet.

For these reasons, we explicitly chose full system emulation as the basis for FIRMADYNE. By leveraging the built-in abstraction provided by the Linux kernel, we replace the existing kernel with our modified kernel, which is specifically designed and instrumented for our emulation environment. Then, in conjunction with a custom user-space NVRAM implementation, we boot the extracted filesystem and our pre-built kernel within the `QEMU` full system emulator. Otherwise, booting the original kernel would likely result in a fatal execution crash, since it only supports a specific hardware platform.

Our results (see §3.2) show that this approach is successful for initial emulation of over 96.6% of all Linux-based firmware images in our dataset. This is likely due to the stable and consistent interface between user-space and kernel on Linux systems, with the exception of custom `IOCTL`'s introduced by vendor-specific kernel modules. In fact, Linux kernel developers will revert kernel changes that break backwards-compatibility for user-space applications; for example, programs built for pre-0.9 (pre-1992) kernels will still function correctly even on the latest kernel releases.[1]

However, this does not hold for kernel modules; indeed, one of the drawbacks of our current implementation is that we cannot load out-of-tree kernel modules located on the filesystem. Nevertheless, our dataset shows that such support is generally not necessary, as we use newer kernels that include common functionality built-in. For example, older 2.4-series mainline kernels lacked `netfilter` connection tracking and NAT support for various application protocols, which became available in-tree around kernel version 2.6.20. As a result, more than 99% of all out-of-tree kernel modules are not useful for our system, as 58.8% are used to implement various networking protocols and filtering mechanisms, and 12.7% provide support for specific hardware peripherals. In comparison, the third-party NetUSB kernel module, which was identified to contain a remotely-exploitable buffer overflow vulnerability, comprises less than 0.2% of all kernel modules from our dataset.

3.1.2. Components. Our system, FIRMADYNE, consists of four major components, which are depicted in Fig. 1 and described below.

(1) **Acquisition**: A custom web crawler automatically downloads firmware images from vendor websites for analysis, and stores them in a database. We manually wrote smart parsers for the website of each vendor to track important metadata such as vendor, product name, release date, version number, etc., and to distinguish firmware images from undesirable binaries; e.g. drivers, configuration utilities, etc. We also supplemented this with firmware images mirrored from the FTP websites of certain vendors, which can include beta and test images that are not widely released, as well as other firmware images that were manually downloaded due to vendor websites that were difficult to crawl automatically.
(2) **Extraction**: A custom-written extraction utility separates the kernel and root filesystem from within each firmware image, and stores them in normalized form in our firmware repository.

[1] `https://www.kernel.org/doc/Documentation/stable_api_nonsense.txt`

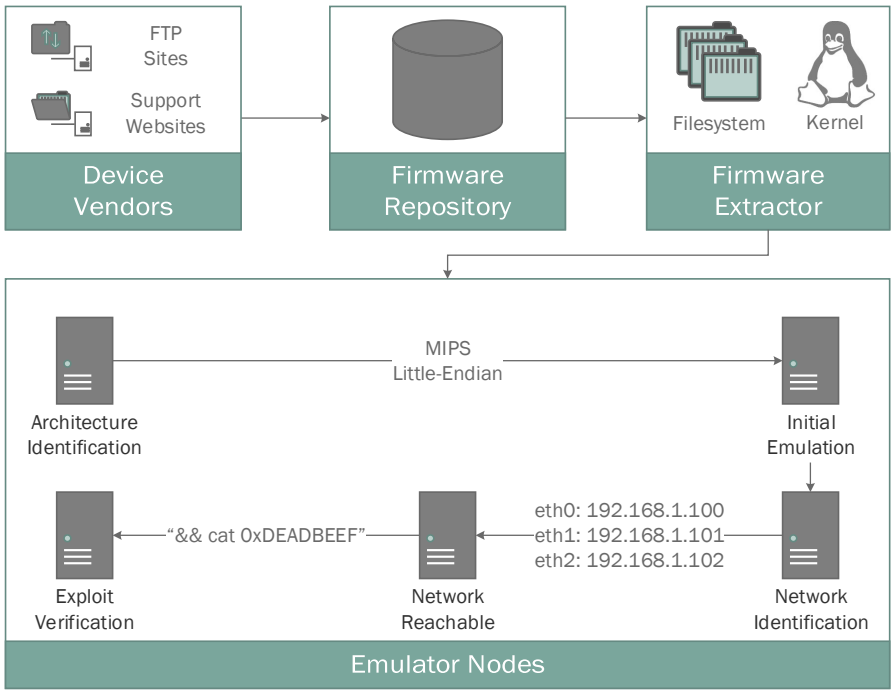

FIGURE 1. Architectural diagram of FIRMADYNE showing the emulation life-cycle for an example firmware image, as described in §3.1.2.

(3) **Emulation**: The root filesystem is analyzed to determine the target architecture and endianness of binaries. Then, a custom library is inserted to emulate the NVRAM peripheral, and the firmware is initially emulated using a pre-built Linux kernel within the QEMU [15] full system emulator, all configured with a matching architecture, endianness, and word-width. Currently, we support the ARM little-endian, MIPS little-endian, and MIPS big-endian platforms, which comprise 90.8% of our dataset (§3.2.1.1). During initial emulation, we intercept system calls to the networking subsystem, which allows us to infer the desired network configuration. Finally, we re-emulate the firmware with its desired configuration, and verify that it is accessible over the network.

(4) **Analysis**: To illustrate the versatility of our platform, we have developed three vulnerability detection passes, which are able to assist with finding and verifying vulnerabilities. They include analyses for publicly-accessible webpages and SNMP information, which may be vulnerable to information disclosure, buffer overflow, or command injection vulnerabilities, as well as an exploit checker for both known and previously-unknown vulnerabilities.

3.2. Evaluation

In this section, we evaluate our implementation of FIRMADYNE. First, we examine the composition of our input dataset, and analyze its effect on the emulation fidelity at every stage in the emulation pipeline. Second, we demonstrate how we leveraged our

Arch.-Endian	TILE-LE x86-64-LE	ARC-LE Unknown	M68k-BE ARM-LE	x86-LE MIPS-BE	MIPS64-BE MIPS-LE	PPC-BE	ARM-BE Total
# Image(s)	1	10	10	31	50	84	102
	147	439	843	3,137	4,632		9,486

TABLE 1. Breakdown of firmware images by architecture, based on binary fingerprinting of extracted root filesystems.

system to identify 14 previously-unknown vulnerabilities within the collected firmware samples. Using proof-of-concept exploits that we developed, we use our system to assess the prevalence and impact of each on our dataset. Finally, we demonstrate the analysis flexibility of our system by supplementing it with 60 known exploits, mostly from the Metasploit Framework [106], to assess prevalence and impact.

It is important to note that the distribution of firmware images across product lines and device vendors is not uniform, and thus may skew interpretation of the results. In particular, although we attempt to scrape metadata about the model number and version number of each firmware image, this information is not always available, nor is it present in a format that can easily establish a temporal ordering. For example, vendors may re-release a given product with different hardware, or release a product with different hardware or firmware in each region. Similarly, there is not a direct one-to-one correspondence between firmware images and products. For example, some vendors, such as Mikrotik, distribute a single firmware image for each hardware architecture, whereas other vendors, such as OpenWRT, distribute different binary releases of the same firmware image using various encapsulation formats. Given two different firmware binaries, this raises the question of how functionally identical they may be, which we do not address. Nevertheless, we attempt to provide a lower-bound on the number of affected products, where possible.

3.2.1. Statistics.

3.2.1.1. *Architectures.* For all firmware images with extracted root filesystems, we were able to identify its architecture by examining the format header of the `busybox` binary on the system, or alternatively binaries in `/sbin/`.

Table 1 shows that the majority of our firmware images are 32-bit MIPS (both big-endian and little-endian), which constitute approximately 79.4%. The next most popular architecture type is 32-bit little-endian ARM, which constitutes approximately 8.9%. Combined, these two architectures constitute 90.8% of all firmware images, with the remainder forming the little-tail of this distribution.

3.2.1.2. *Emulation Progress.* As shown in Fig. 2, of the 8,617 extracted firmware images for which we identified an architecture, our system initially emulated 96.6% (8,591) successfully. The failures can be attributed to a number of causes, including the lack of an `init` binary in a standard location (`/bin/init`, `/etc/init`, or `/sbin/init`), or an unbootable filesystem. For example, certain images containing Ralink chipsets are known to name their `init` binary `ralink_init`, which we currently do not support. Likewise, extraction failures can also affect success of the initial emulation. Since we only extract the first UNIX-like filesystem from firmware images that contain multiple filesystems,

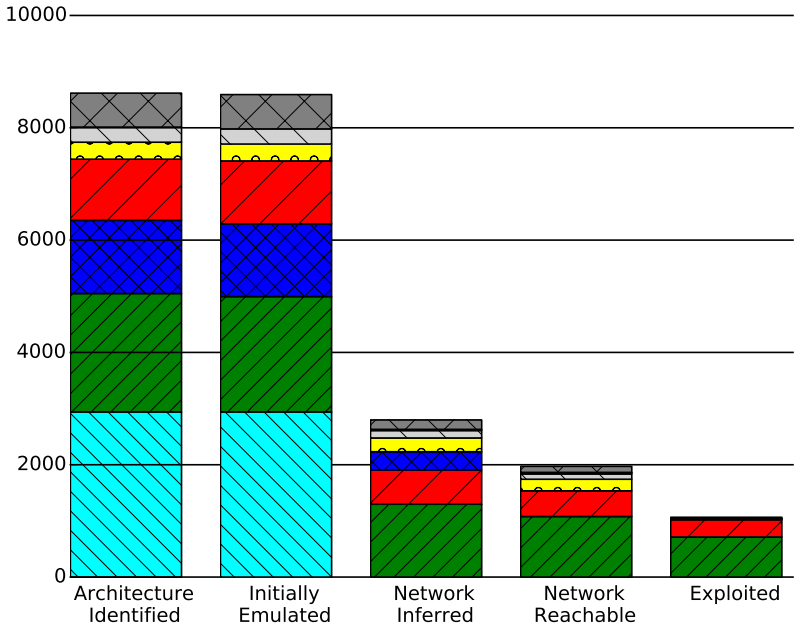

FIGURE 2. Breakdown of firmware images by emulation progress, colored by vendor.

it is likely that only part of the filesystem has been extracted, leading to a boot failure. Reassembling such systems into a single filesystem is not straightforward because each filesystem can potentially be mounted on top of another at arbitrary locations.

Of the 8,591 firmware images that entered the "learning" phase, only 32.3% (2,797) had their networking configuration successfully inferred. We believe that this decrease occurred due to failures in the boot process while attempting to infer the network configuration. Problems with NVRAM emulation are a significant contributor to these failures. For example, some routers may not initialize correctly if our NVRAM library was not able to override the built-in implementation, if insufficient default NVRAM values were loaded by our library, or if the built-in NVRAM implementation has different function semantics. These manifest as various crashes or hangs during the boot process, especially if memory or string manipulation functions (`memcpy()`, `strcpy()`, etc.) are called on `NULL` values returned for nonexistent keys. Additionally, it is also possible that some images do not use a NVRAM hardware peripheral, but instead write configuration values directly to a MTD partition, which we may not successfully emulate.

Other potential sources of networking failures include different naming conventions for networking devices. For example, devices that utilize Atheros or Ralink chipsets may expect platform networking devices to be named similarly to `ath0` or `ra0`, respectively, instead of the generic `eth0`. Likewise, other devices may expect the presence of a wireless networking interface such as `wlan0`, and fail otherwise. In addition, since our ARM little-endian emulation platform currently supports only up to one emulated Ethernet device, this may prevent some firmware images from configuring networking correctly.

Exploit ID	# Images	# Products	Affected Vendor(s)
47	282	16	21, 22, 37
56	169	14	16, 21, 35
64	169	27	12, 21, 37
45	136	13	21
43	88	10	12
202	49	11	12, 16, 21, 36, 37, 42
207	35	6	21
60	31	9	7, 12, 19, 21, 37
205	16	5	21
206	14	4	21
203	13	5	12
59	9	N/A	12
200	8	1	21
201	7	1	21
210	7	2	12
4	6	N/A	12
24	5	1	19, 42
213	4	1	21
214	4	1	21
39	3	N/A	12
209	3	1	12
212	3	1	21
61	2	1	42
204	1	N/A	21
211	1	1	21

TABLE 2. Breakdown of exploits by number of affected firmware images, number of affected products, and affected vendor(s). Note: `N/A` indicates that we do not have sufficient metadata to compute a lower-bound on affected products.

Only 70.8% (1,971) of the 2,797 images with an inferred network configuration are actually reachable from the network using `ping`. This may be caused by firewall rules on the emulated guest that filter ICMP echo requests, resulting in false negatives, or various other network configuration issues. For example, our system may have mistakenly assigned the host TAP interface in `QEMU` to the WAN interface of the emulated device instead of a LAN interface, or identified the default IP address of the WAN interface instead of the LAN interface. Similarly, firmware may change the MAC address of the emulated network device after it has booted, resulting in stale ARP cache entries and a machine that appears unreachable.

Surprisingly, our results show that 43% (846 out of 1,971 firmware images) of the network reachable firmware images are vulnerable to at least one exploit. We discuss this further in §3.2.2, where we give a breakdown by exploit.

3.2.2. Vulnerabilities. In Table 2, we provide a breakdown of all successful exploits by exploit. Exploits in the range #0–#100 are sourced from the Metasploit Framework, whereas most exploits greater than or equal to #200 are previously-unknown vulnerabilities for which we developed proof-of-concepts (POC's). This excludes #202, which is a known vulnerability but not sourced from the Metasploit Framework. Each of these previously-unknown vulnerabilities has been reported to the respective vendor, following the policies of responsible disclosure. We discuss a few specific vulnerabilities below in greater detail as case studies.

3.2.2.1. *Command Injection (#200, #201, #204–#206, #208).* While analyzing the aggregate results of our automated accessible webpages analysis, we discovered six previously-unpublished command injection vulnerabilities that affect 24 firmware images for wireless routers and access points manufactured by Netgear. All six vulnerabilities were within PHP server-side scripts that provided debugging functionality but appeared to be accidentally included within production firmware releases. In particular, five of these were used to change system parameters such as the MAC address of the WLAN adapter, and the region of the firmware image (e.g., World Wide [WW], United States [US], or Japan [JP]). The remaining one was used to write manufacturing data such as MAC address, serial number, or hardware version into flash memory. Our manual analysis of the PHP source code revealed that all were straightforward command injection vulnerabilities through the `$_REQUEST` super-global and unsafe use of the `exec()` function. After discovering these potential vulnerabilities, we leveraged FIRMADYNE to automatically verify their exploitability across our entire dataset.

3.2.2.2. *Buffer Overflow (#203).* Another new vulnerability that we manually discovered, using the results of our automated accessible webpages analysis, was a buffer overflow vulnerability within firmware images for certain D-Link routers. To implement user authentication, the webserver sets a client-side cookie labeled `dlink_uid` to a unique value that is associated with each authenticated user. Instead of verifying the value of this cookie within the server-side scripting language of the webpage, this authentication functionality was actually hard-coded within the webserver, which uses the standard library functions `strstr()`, `strlen()`, and `memcpy()` to copy the value of the cookie. As a result, we were able to set the value of this cookie to an overly-long value to cause the webserver to crash at `0x41414141`, another poisoned argument that we monitor for.

3.2.2.3. *Information Disclosure (#207, #209–#214).* Using the automated webpage analysis, we also discovered seven new information disclosure vulnerabilities across our dataset that affect 51 firmware images for various routers manufactured by both D-Link and Netgear. One of these (#207) was within an unprotected webpage that provides diagnostic information for the router, including the WPS PIN and passphrases for all locally-broadcast wireless networks.

The remaining six vulnerabilities (#209–#214) were within the Simple Network Management Protocol (SNMP) daemon of both manufacturers. This feature was enabled by default likely because these routers were targeted towards small businesses rather than home users. To interpret results obtained from SNMP queries, one needs access to a Management Information Base (MIB) file that describes the semantics of each individual

object (OID) field. As discussed in §3.1.2, our crawlers record links to MIB files in the collected metadata, enabling manual verification of the obtained results.

Our automated exploit verification showed that these firmware images would respond to unauthenticated SNMP v2c queries for the `public` and `private` communities, and return values for the OID's that contain web-based access credentials for all users on the device, and wireless credentials for all locally-broadcast wireless networks.

CHAPTER 4

Protecting WebAssembly with control-flow integrity

The fundamental design of WebAssembly (§4.1) includes multiple security features. A separate execution stack ensures that programs cannot overwrite return pointers, and strict function typing ensures that indirect calls must target a function of corresponding type, all of which help prevent manipulation of control-flow. Linear memory ensures that all memory accesses are contained within a given memory region, by computing addresses with greater bit-width to prevent integer overflows, and automatically bounds checking esulting addresses. Separation of linear memory from the program's code and execution stack, as well as the runtime's internal data structures, limits the scope of memory safety bugs to the program's own data. Well-defined language semantics enable creation of a soundness proof [156] verifying progress and preservation of the mechanized specification, which allows cross-validation of runtime implementations.

Nevertheless, there is still room for improvement. Past work [95] has shown that the coarse type system allows multiple source language types to alias into the same fundamental types in WebAssembly, which weakens type-based protections for indirect calls. Likewise, they have also shown that overflows within linear memory can be used to overwrite other program segments. This is because vector data that cannot be converted into natively-typed variables, such as arrays, lists, or strings, are stored in linear memory, which is subdivided by the compiler for the program's heap, stack, and data.

Due to a lack of automatic bounds checking for individual variables or subfields in linear memory, overflows of automatically-generated stack frames or metadata corruption attacks against the default heap allocation implementation can be used to overwrite other program segments. Additionally, since linear memory is mutable and no fine-grained page protections are available, these overflows can overwrite "constant" data. However, support for multiple linear memories has already been proposed [126], which would mitigate many of these issues by allowing heap, stack, and data to be placed in disjoint linear memories. Other past work [54] has also proposed segment memory that can only be addressed using handles, which act as a replacement for pointers.

We propose and implement coarse-grained control-flow integrity for WebAssembly, which if enabled, would have prevented the proposed remote code execution attack [95]. Their attack corrupts the metadata of the heap allocator to overwrite an indirect function call reference in linear memory, in order to call a different function that executes an arbitrary command. In conjunction with existing protections for the execution stack, our control-flow integrity design raises the bar for attackers by mitigating code reuse or other control-flow attacks.

4.1. Background

In the past few years, major browser vendors have collaboratively developed a low-level bytecode language to replace JavaScript for high-performance web applications, known as WebAssembly [76]. As the spiritual successor of Google's Native Client [130, 163], it provides a safe runtime environment that is compatible with existing applications written in high-level languages (e.g. C, C++, Rust, etc.), while supporting direct compilation into native machine code irrespective of browser, architecture, or operating system. We defer discussion of WebAssembly's security properties to §4. Recent efforts have focused on extending WebAssembly with a system interface [67] (WASI) to better support non-web *embeddings*, and native representation of opaque reference types [125] to enable pass-through of JavaScript values.

Early benchmarks from both Chrome's V8 and Firefox's SpiderMonkey JavaScript engines have shown [76] that WebAssembly applications are smaller and faster, on average, than their JavaScript asm.js equivalents, with typical performance within 10% of native. Subsequent benchmarks on the SPEC CPU2006 and CPU2017 suites have shown [86] a mean slowdown of up to 1.55x, with a peak of up to 2.5x, compared to native.

WebAssembly [6] is a stack-based bytecode language without explicit registers, where return values and arguments for instructions are implicitly pushed to and popped from an operand stack. Currently, all values must correspond to one of four fundamental types; *integers* and *IEEE-754 floating-point values*, each of which can be 32- or 64-bit. Each binary is represented as a *module*, which is composed of *functions*, *globals*, *linear memories*, and *tables*:

- Functions: Encapsulate a sequence of WebAssembly instructions with a function type, which takes a sequence of input arguments and returns a sequence of output results. Control-flow must be structured and well-nested, such that each basic block and loop are labeled with `block/loop` and `end` instructions. Branch instructions, such as `br` (unconditional), `br_if` (conditional), and `br_table` (jump table), must specify the relative depth of an encapsulating block or loop that immediately precedes the target.

  At runtime, a fixed set of local variables for each function are zero-initialized, and can be accessed using special `set_local` and `get_local` instructions. Function calls utilize an execution call stack that is not directly accessible by the program, and is thus effectively a shadow stack.
- Globals: Allow certain variables to be shared across all functions in the module. Each global variable is either mutable or immutable, and initialized using a constant expression. Special `get_global` and `set_global` instructions provide access to individual global variables.
- Linear Memories: Each memory represents a sequence of 64 kB pages that are mutable, contiguous, and resizeable. Individual bytes within a memory can be accessed in little-endian format using `load` and `store` instructions with corresponding bit-length, and will automatically trap (abort execution) if out-of-bounds. Currently, at most one such memory is supported.
- Tables: An indexed list of values with the same type. Currently, this can only be used to store untyped function references for the `call_indirect` indirect call instruction, and at most one such table is supported. The indirect call instruction must also

```
1 (module
2   (type $out-i64 (func (result i64)))
3
4   (func $zero-i64 (type $out-i64) (i64.const 0x0))
5   (func $one-i64 (type $out-i64) (i64.const 0x1))
6
7   (table $itable funcref
8     (elem (i32.const 1) $zero-i64 $one-i64)
9   )
10  (table $itable-one (type $out-i64)
11    (elem (i32.const 1) $one-i64)
12  )
13
14  (func $call-baseline (param $idx i32) (result i64)
15    (call_indirect (type $out-i64) (local.get $idx))
16  )
17  (func $call-single (param $idx i32) (result i64)
18    (block $cont
19      (br_if $cont (i32.eq (local.get $idx) (i32.const 1)))
20      (unreachable)
21    )
22    (call_indirect (type $out-i64) (local.get $idx))
23  )
24  (func $call-multiple (param $idx i32) (result i64)
25    (call_indirect $itable-zero (local.get $idx))
26  )
27 )
```

Listing 1: Comparison of indirect calls in WebAssembly, under baseline, single-table CFI, and multiple-table CFI.

specify an expected function type, which is checked at runtime against the type of the referenced function. This can be viewed as a very coarse-grained form of control-flow integrity that is enabled by default, but to avoid confusion, we use control-flow integrity in this context to refer to additional explicit checks.

4.2. Design

Our approach extends the compiler toolchain to support modern Clang/LLVM CFI [37], which partitions indirect call targets into equivalence classes based on their source language type (§2.3.1). By default, the compiler populates the indirect call table with every function that is address-taken in the program. As an example, consider the simple program shown in Listing 1. In the baseline version (line 14), it performs a normal indirect call to a function in the default table (line 7), which is checked at runtime against the specified named type `out-i64`. Note that the table starts at index 1 (line 8), to avoid aliasing a valid index with a `NULL` pointer, and that because the table contains untyped function references (`funcref`), the indirect call must explicitly specify a function type, which here corresponds to a function that takes no parameters and returns a single 64-bit integer value (line 2).

Under control-flow integrity, additional checks occur before each indirect call. In our *single table design*, we pre-assign monotonic table indexes such that members of each equivalence class are placed consecutively, and we insert a range check at each indirect call site on the runtime index. Should the check fail, a trap instruction that terminates the program is executed. As shown in Listing 1, an explicit control-flow integrity check is inserted before the indirect call (lines 19–22), which validates that the call index equals one, corresponding to the `one-i64` function. If this check fails, instead of continuing with the indirect call, the program terminates by executing a trap instruction (line 20).

In our *multiple table design*, we improve performance by placing each equivalence class in a separate typed table, which leverages existing bounds checks on runtime table indexes and eliminates explicit range checks. This requires support for multiple tables, which has been subsumed into the reference types proposal [125], and augments the `call_indirect` instruction with an additional argument to specify the referenced indirect table. As shown in Listing 1, a separate typed table (line 10) is explicitly referenced by the indirect call (line 25). Compared to the single table design, this version improves performance by eliding the runtime index check (line 19) with an implicit load-time check for homogeneous function types in the table.

However, this design also has some additional challenges; namely avoiding index aliasing between multiple tables, and tagging each indirect call site with the correct table. If indexes are allowed to overlap across multiple tables, then false negative checks may occur, because automatic bounds checking cannot distinguish between e.g. index 1 from one table and index 1 from another table, since they are both valid indexes. As for tagging, indirect calls are generated in the compiler frontend, and control-flow integrity checks are lowered later at link-time, but with multiple tables, the indirect call site also needs to know its corresponding table. Currently, we use level-ordered search to find and fix-up the corresponding call site, but the frontend should instead emit a direct link in the IR.

4.3. Evaluation

To evaluate the performance of our control-flow integrity designs, we implemented them in the compiler toolchain (binaryen, Clang/LLVM), and added support for multiple tables in Chrome's V8 JavaScript engine. We compiled and measured the performance of the SPEC CPU2000 benchmark applications, taking the arithmetic mean of three runs for each, as shown in Figure 1. Since we used an early version of the compiler toolchain and V8, not all of the SPEC benchmarks were compatible with WebAssembly due to unrelated compiler bugs, and the `vortex` benchmark encountered false positive failures due to strict type matching (§2.3.1). Our results demonstrate that coarse-grained control-flow integrity has minimal performance impact, with a geometric mean of 99% relative performance for both designs, and a minimum of 98% and 96% for our single and multiple table designs, respectively. We believe that the slight decrease in performance with multiple tables was due to an unoptimized implementation, as our code was experimental.

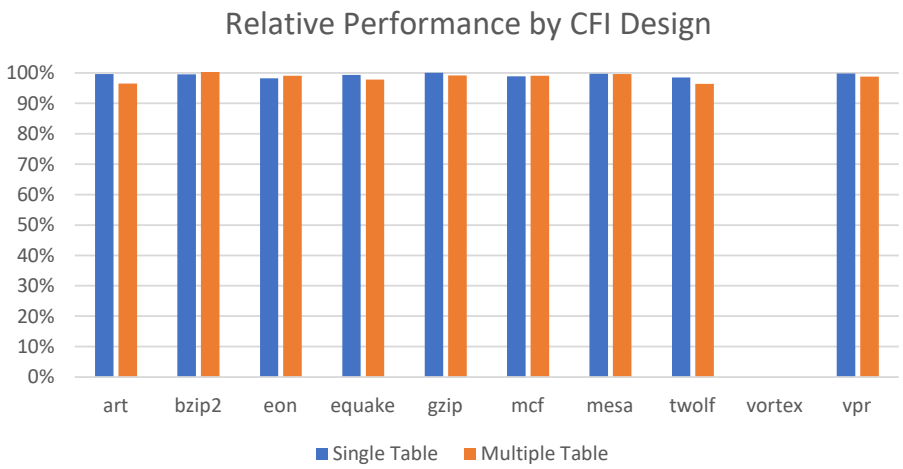

FIGURE 1. Relative performance of various SPEC CPU2000 benchmarks in V8.

CHAPTER 5

Ensuring program integrity via hardware-enforced message queues

Existing program protections rely on a variety of *intra-process isolation* techniques, such as memory segmentation, disjoint address spaces [60, 99, 120], software fault isolation [154] (SFI), and information hiding (§2.2), to provide security guarantees by partitioning the program. However, these software-based isolation mechanisms suffer from compatibility issues, have significant overhead, or rely on security through obscurity. Software fault isolation must mask all program pointers to limit their access, which has high cost and is incompatible with existing libraries. Information hiding has low overhead, but relies on randomization of program code or layout to discourage attackers, which has been shown vulnerable to disclosure attacks [131, 135, 140].

A hardware-based alternative is fine-grained instruction monitoring [9, 34, 48, 65]), which modifies the processor to export a stream of fixed execution events to a special-purpose processor for analysis. These events correspond to state changes, such as retired instructions, function calls, or memory accesses, and past work has proposed sending them to a similarly-sized core [34, 65], multiple microcontroller-sized cores [9], or an on-chip FPGA [48]. However, these approaches require significant microarchitectural changes, generate a fixed set of hardware events, and impose filtering overhead on undesired events. For example, under FADE [65], regardless of security policy, 84%–99% of all events are irrelevant and must be discarded. Similarly, under Guardian Council, different security policies can require anywhere between 2–24 processing elements to reduce monitoring overhead below 5%, which results in idle cores.

Instead, we observe that a simple and fast AppendWrite inter-process communication (IPC) primitive can provide this isolation without the drawbacks of past work by leveraging existing *inter-process isolation*. It ensures both *authentication* and *integrity* for events delivered from a monitored program to a separate verifier process. We implement two variants of AppendWrite; one with no microarchitectural changes using an FPGA, and another that adds one instruction to the microarchitecture to reduce overhead. Using this AppendWrite primitive, we design HerQules, a framework for implementing efficient integrity-based security policies, which uses concurrent execution to offload the overhead of policy enforcement to the verifier process. Our case study on control-flow integrity against a benchmark program suite demonstrates that HerQules achieves a significant improvement in correctness, effectiveness, and performance over prior work.

5.1. Design

We assume a strong threat model that allows adversaries to read and write arbitrary process memory, subject to page table protections. This excludes access to processor registers and modification of program code, which is typically read-only. We trust the

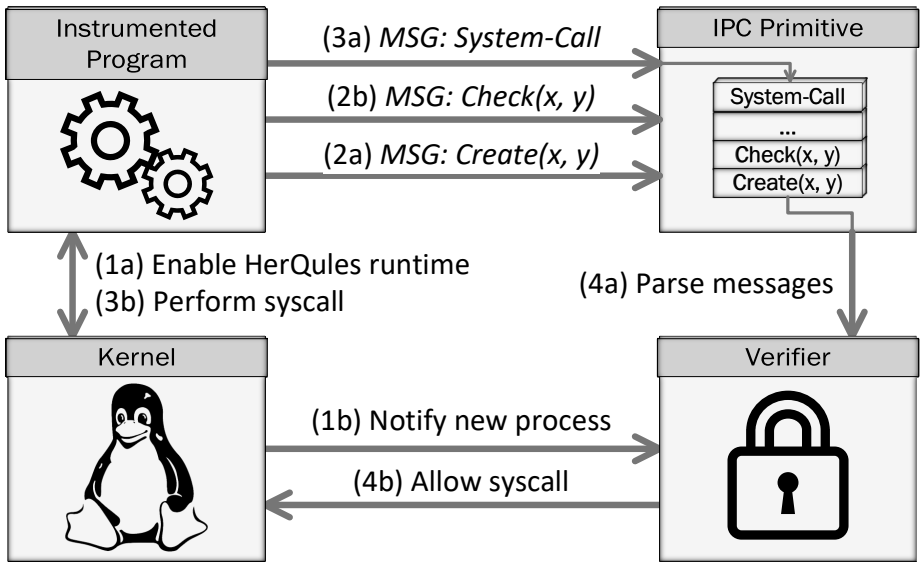

FIGURE 1. HERQULES Overview

processor microarchitecture and operating system to enforce boundaries between independent user-space programs, and between user-space and kernel-space.

Under HERQULES, an instrumented program communicates *asynchronously* with an external verifier over an IPC channel, sending messages before certain security-relevant events (e.g., indirect control transfers) occur. Our key insight is that because the instrumented program begins execution in a benign state, it will send a message demonstrating a policy violation *before* it occurs. Even if the violation results in a total program compromise, the append-only IPC ensures that the compromised program cannot retract evidence of the violation. To prevent the program from effecting externally-visible side effects (e.g., leaking secrets, or attacking the rest of the system) before the verifier detects the compromises, we *bound* asynchrony at system calls, by pausing execution until the verifier has processed all in-flight messages.

Below, we highlight the four main components of HERQULES with respect to Figure 1.

(1) At compile-time, our **compiler pass** instruments the program to send messages when security-relevant events occur. These messages and events are policy dependent.
(2) At run-time, the instrumented program enables HERQULES *(1a)*, causing the kernel to register it with a separate **verifier** process *(1b)*. Subsequently, the instrumented program sends messages to the verifier via AppendWrite *(2a, 2b)*.
(3) At some point, the instrumented program sends a system-call message *(3a)* and performs the system call *(3b)*, where it is initially paused by our **kernel module**, until the verifier confirms no policy checks have failed *(4a, 4b)*.

To safely bound the verifier's asynchrony, HERQULES pauses execution at system calls. A naive approach would then require a round-trip from the kernel to the verifier, imposing additional latency on the critical path. Hence, we pipeline the SYSTEM-CALL message *(3a)* with the system call itself *(3b)*, by instrumenting the program and runtime

IPC Primitive	Append-Only	Asynchronous	Cost	Time (ns)
Message Queue	✓	×	System Call	146
Named Pipe	✓	×	System Call	316
Socket	✓	×	System Call	346
Shared Memory	×	✓	Memory Write	12
AppendWrite-FPGA	✓	✓	Memory Write	102
AppendWrite-μarch	✓	✓	Memory Write	2

TABLE 1. Comparison of IPC primitives, split between existing software mechanisms (top) and our proposed AppendWrite.

libraries to send the SYSTEM-CALL message before system calls. When the verifier receives a confirmation message, it proactively notifies the kernel to approve the proximate system call. Note that the kernel and verifier communicate over a separate privileged channel that is not accessible to the instrumented program, as shown in Figure 1. If no confirmation arrives within a configurable epoch, the kernel declares a policy violation and the instrumented program is terminated.

5.1.1. AppendWrite IPC Primitive. Since the instrumented program is untrusted and may become compromised, our IPC primitive must guarantee *authenticity* and *integrity*. Namely, it must ensure all messages were sent by the instrumented program, and that no messages have been modified or erased after being sent. We provide the former by configuring the kernel to arbitrate creation of messaging channels, and ensure the latter by designing the IPC primitive as a simple *append-only* channel from an instrumented program to the verifier.

Existing software- and hardware-based primitives do not satisfy these constraints; thus, we design two variants of our new hardware-based primitive. One uses a programmable accelerator (§5.1.1.1), and the other adds a new AppendWrite instruction to the microarchitecture (§5.1.1.2), which we model in software. We provide a comparison of various IPC primitives in Table 1, and show the average runtime of a micro-benchmark that sends multiple 32-byte messages. Note that authenticity can be retrofitted onto many primitives using kernel arbitration.

Existing software-based primitives either perform poorly or lack our *append-only* property. Mechanisms that require a system call (including POSIX queues, pipes, and sockets) cost hundreds of nanoseconds, require a privilege transition that may flush hardware caches, and execute synchronously in the calling thread. Traditional performance workarounds, such as vectored I/O or other forms of client-side buffering, as well as fast primitives like shared memory, are all insecure in that a compromised program can alter or erase in-flight messages.

Many hardware peripherals already contain first-in first-out (FIFO) queues, which we initially attempted to repurpose for our IPC primitive. For example, network interface cards (NICs) are widely deployed, and contain separate receive/transmit packet queues for each physical port. However, we eventually abandoned this approach, because these hardware resources are not directly accessible to user-space programs without kernel bypass, which usually requires exclusive access and pre-assignment of programs to

Design	Mechanism	Precision	Detects UAF	Compatibility	Performance
Clang/LLVM CFI [37]	Language-level Types	•	×	••	••••
CCFI [102]	Cryptographic MACs	•••	×	•	•
CPI [91]	Information Hiding	••	×	•	••••
CPI [92]	Software Fault Isolation	••	×	•	•••
HQ-CFI-SfeStk	AppendWrite	••	✓	•••	•••
HQ-CFI-RetPtr	AppendWrite	•••	✓	•••	••

Table 2. Coarse- (top) and fine-grained (mid, bottom) control-flow integrity designs. More • is better.

physically-connected NIC ports. Kernel bypass also requires certain hardware capabilities, such as an IOMMU, as well as PCI Express (PCIe) Access Control Services on the root complex for subdividing IOMMU groups, which has limited availability. Software frameworks for bypass, such as the Data Plane Development Kit (DPDK), implement an entire networking stack, which adds overhead and must be trusted.

5.1.1.1. *Accelerator.* We designed an initial variant of AppendWrite using a cache-coherent FPGA. However, after evaluating it and exploring various optimizations, we observed that performance was fundamentally limited by memory pipeline stalls and PCIe bus overhead.

5.1.1.2. *Microarchitecture.* Hence, we designed a second variant that extends the microarchitecture with a new AppendWrite(reg) instruction, which stores the message in reg into a pre-configured *appendable memory region* and increments the append address. Multiple encodings of AppendWrite can support messages of different lengths by accepting, e.g., 64/128/256/512-bit registers.

An appendable memory region must be accessible to the instrumented program, but cannot be directly writable. Thus, the AppendWrite instruction can store to a memory page that is not writable only if it is in the appendable memory region. We add two privileged 64-bit control registers, AppendAddr and MaxAppendAddr, to configure a per-core appendable memory region. AppendAddr identifies the address where the next message should be written, whereas MaxAppendAddr identifies the end of the appendable memory region, such that AppendAddr cannot be incremented past MaxAppendAddr. When this would occur, the AppendWrite instruction triggers a processor fault, which the kernel can use to allocate a new buffer, or simply reset the appendable memory region.

5.2. Control-Flow Integrity

We provide a case study on using HerQules for control-flow integrity. Table 2 compares our designs against existing coarse-grained (§2.3.1) and fine-grained (§2.3.2) designs. Coarse-grained offers good performance but can fail to catch exploits due to large equivalence classes, whereas fine-grained is more precise but historically imposes nontrivial overhead. HerQules is the only design that protects against use-after-free (UAF) vulnerabilities, and is compatible with all benchmark programs.

5.2.1. Forward-Edge Transitions. Our HQ-CFI design protects the following forward-edge *control-flow pointers*:

(1) Function pointers: Direct pointers to executable code.

(2) Virtual method table pointers: Indirect pointers in C++ objects that refer to a global class-specific virtual method table (vtable), which is composed of function pointers. Note that vtables are typically 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 class-specific vtable table (VTT) that contains the relative offsets of vtables within other vtables.

We send messages to notify the verifier of certain operations on control-flow pointers, listed below. For example, programs may manipulate blocks (contiguous chunks) of memory with certain library functions. It is difficult to statically determine whether control-flow pointers are present within memory blocks, so we notify the verifier of these events at runtime. We describe the semantics for our messages below.

- POINTER-DEFINE(P, V): Initialize a control-flow pointer at P with value V.
- POINTER-CHECK(P, V): Validate that a control-flow pointer at P with current value V matches its previous definition. If not, this pointer is corrupt or a use-after-free.
- POINTER-INVALIDATE(P): Remove any control-flow pointer at P.
- POINTER-BLOCK-COPY(SRC, DST, SZ): Copy all control-flow pointers from [SRC, SRC + SZ) to [DST, DST + SZ). These ranges may intersect, and pre-existing control-flow pointers in the destination will be invalidated. This matches the behavior of `memcpy` and `memmove`[1].
- POINTER-BLOCK-MOVE(SRC, DST, SZ): Move all control-flow pointers from [SRC, SRC + SZ) to [DST, DST + SZ). These ranges must not intersect, and all pre-existing control-flow pointers in both will be invalidated. This matches the behavior of `realloc`.
- POINTER-BLOCK-INVALIDATE(P, SZ): Invalidate all control-flow pointers in the address range [P, P + SZ). This matches the behavior of `free`.

We use compiler instrumentation, which can be decomposed into the following three components, to automatically insert messaging on control-flow pointers. We also modify the standard library implementation of `longjmp` and `setjmp` to send messages on the function pointer used to implement non-local gotos.

(1) Language-Specific Annotations (Clang): Insert checks before calling function pointers or object methods, using built-in CFI annotations.
(2) Initial Lowering (LLVM): Before optimization, convert CFI annotations into runtime messaging calls.
(3) Final Lowering (LLVM/gold [145]): After built-in optimizations, execute our optimizations, instrument calls to certain library functions, initialize globally-scoped control-flow pointers, and optionally link our messaging runtime inline.

We also enable three C++-specific optimization passes, which execute separately. These devirtualization passes convert indirect method calls into direct calls: Virtual Pointer Invariance [117, 118], Whole Program Devirtualization [38], and Dead Virtual Function Elimination [139].

Our messages only support two arguments, 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

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

Design	Errors	False Ps.	Invalid	Ok	Design	Errors	False Ps.	Invalid	Ok
Baseline	0	0	0	48	CPI	14	8	14	31
Clang/LLVM CFI	3	16	3	32	HQ-CFI-SfeStk	0	0	0	48
CCFI	2	29	5	19	HQ-CFI-RetPtr	0	0	0	48

Table 3. Correctness of various CFI designs.

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.

5.2.2. Backward-Edge Transitions. We develop a similar design, HQ-RetPtr, to protect return pointers. Although it cannot defend against fundamental architectural races (§2.3.3), it is not vulnerable to disclosure attacks that have affected past work.

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

If a function may write to memory, is known to return, is not always tail called, and contains stack allocations, we send a Pointer-Define(p, v) message on the return address in the prologue of each stack frame, and a Pointer-Check-Invalidate(p, v) in the epilogue before returning.

5.3. Evaluation

We evaluate HerQules on multiple benchmarks: RIPE [124, 159], SPEC CPU2006 [82] v1.2, SPEC CPU2017 [25] v1.0.5, and NGINX [141] v1.18.0, using the musl C runtime library. On SPEC, we execute the reference input and apply patches for compatibility and memory safety bugs, including wrong type casts. On NGINX, we measure arithmetic mean request throughput using wrk [66] for 60 seconds. Our results demonstrate the correctness (§5.3.1), effectiveness (§5.3.2), and performance (§5.3.3) of HerQules.

We compare HerQules against three previous designs: modern Clang/LLVM CFI [37], CCFI [102], and CPI [91, 92] with SFI, each representing different trade-offs. Clang/LLVM CFI has a fast but imprecise coarse-grained design (2.3.1), while CCFI and CPI are state-of-the-art pointer integrity designs, which are slower but maximally precise (§2.3.2.1). Since CCFI and CPI use older versions of the Clang/LLVM compiler, their runtimes are normalized to a baseline without C++ optimizations. Otherwise, these optimizations are enabled when building for HerQules and modern Clang/LLVM CFI. For CCFI, we recompile all libraries to reserve eleven XMM registers, which breaks calling conventions [3]. Due to the prevalence of false positives in past work (§5.3.1), we configured all designs not to terminate programs after a policy violation.

We distinguish our policies as follows: HQ-CFI-SfeStk combines HQ-CFI with the Clang/LLVM safe stack (§2.3.3), while HQ-CFI-RetPtr combines HQ-CFI with HQ-RetPtr (§5.2.2). To identify IPC primitives, we use postfix -MQ for POSIX message queues (§5.1.1), -FPGA for the accelerator, and -Model for the software model of our microarchitecture-based primitive.

Design	BSS	Data	Heap	Stack	Total
Baseline	214	234	234	272	954
HQ-CFI-SfeStk	10	10	10	0	30
HQ-CFI-RetPtr	0	0	0	0	0

Table 4. Successful RIPE exploits, grouped by overflow origin.

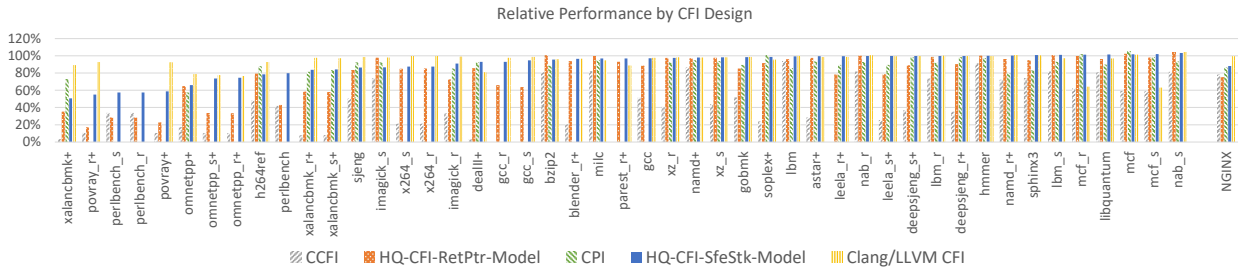

Figure 2. Relative performance of CFI designs, sorted on HQ-CFI-SfeStk-Model (left to right). Suffix '+' denotes C++.

5.3.1. Correctness. To evaluate each design, we built and executed all of our benchmarks with CFI enabled. We summarize our results in Table 3, distinguishing between errors (crashes/hangs), false positive violations, invalid results (output doesn't verify), and successful runs. Note that some categories are not mutually exclusive.

5.3.2. Effectiveness. To show the effectiveness of our policies, we execute the RIPE test suite, which contains hundreds of buffer overflow exploits. We use RIPE64 [124], a port for 64-bit systems that also adds 100 exploits, because the original [159] only supports 32-bit programs. On all configurations, we disable ASLR, non-executable stack, and stack canaries. Under HerQules, we also disable enforcement of system call confirmations after the `execve` system call, because shellcode system calls are used by RIPE to verify exploits.

On an initial run of the SPEC2006 and SPEC2017 benchmarks, HQ-CFI detected policy violations from use-after-free bugs in the the `omnetpp*` benchmarks. They both suffer from a subtle C++ *static initialization order* bug, which occurs because the language provides no guarantees about initialization/destruction order of `static` objects across compilation units. We note that this bug has persisted despite over 11 years of continuous development, as both benchmarks correspond to different versions of the OMNeT++ simulator [153].

5.3.3. Performance. We show a comparison of HQ-CFI-SfeStk-Model and HQ-CFI-RetPtr-Model against related work in Figure 2. Note that we omit measurements on benchmarks where related work encountered errors or produced invalid output, but not false positives.

CHAPTER 6

Developing effective protections for program integrity

In this section, we propose to explore designs for more effective program protection against memory corruption bugs. We focus on improving precision over past work by modeling intra-object overflows and avoiding pointer analysis, while reducing overhead by eliminating unnecessary instrumentation. Two different approaches for achieving this goal are data-flow integrity and memory safety; however, it may ultimately be faster to eagerly detect out-of-bounds memory stores, rather than instrumenting both memory loads and stores to lazily detect load-time corruption. In addition, given increasing availability of non-volatile (persistent) memory, greater memory safety is needed to ensure consistency [35, 49, 162, 167] of persistent memory regions. We do note, however, that neither proposed design protects directly against bad casts, as our approaches utilize the static type to help determine if stores are authorized.

6.1. Data-flow Integrity

Data-flow integrity [30] (§2.4) validates memory values at runtime against pre-computed static reaching definitions. It instruments both memory loads and stores, in order to insert checks and prevent corruption of the runtime definitions table. Unfortunately, their approach suffers from numerous drawbacks. The underlying pointer analysis is not field-sensitive, and relies on approximate pointer analysis, which the authors acknowledge can result in false negatives. Usage of software fault isolation to protect the definitions table also imposes high runtime overhead, because every memory store must be masked. Furthermore, past work [63] has exploited the inherent undecidability [123] of pointer aliasing to defeat similar control-flow integrity protections.

One proposed design could track definitions at runtime rather than relying on static alias analysis, much like pointer integrity for control-flow integrity (§2.3.2). Such a design could utilize our past work for program integrity via message queues (§5), which eliminates the need for intra-process isolation mechanisms like software fault isolation. Not only would this improve analysis precision, but it would also reduce scope by only instrumenting relevant memory loads and stores.

A key design decision that must be addressed is to determine what program data should be protected. Past work on pointer integrity (§2.3.2) already includes certain types of data that may affect program control flow, so any new design must at least offer similar protections as well. At the same time, it is not feasible to simply protect all program data, for multiple reasons. There must be a mechanism to distinguish between "authorized" and "unauthorized" stores, otherwise memory safety bugs will be implicitly become authorized. For example, pointer integrity only permits explicit stores to sensitive pointers, whereas data-flow integrity assumes that "correct programs do not

Design	Type	Pointer Checks	Invariant	Metadata	Complete
AddressSanitizer	Location-based	Loads, Stores	Memory Validity	Shadow State	×
LowFat	Pointer-based	Loads, Stores, Arithmetic	Pointer Validity	Object Sizes	✓
Write Integrity Testing	Pointer-based	Stores, Indirect Calls	Memory Coloring	Points-To Sets	×

TABLE 1. Comparison of various memory safety designs.

use pointer arithmetic to navigate between independent objects in memory" [30], or relies on static dependency analysis [136] to identify explicit accesses to sensitive data used by access control checks. We propose to extend our existing pointer integrity protections to:

(1) Protect the heap memory allocator, which is commonly targeted for developing read/write exploit primitives.
(2) Protect values that may affect system call arguments.

6.1.1. Performance. As suggested previously, perhaps the most important consideration in determining the feasibility of a proposed approach is performance. Indeed, past work has limited the scope of their protection to only direct control-flow pointers [91] for pointer integrity, or only control-data (return addresses) and local variables for data-flow integrity [30]. Below, we identify a number of optimizations that we plan to explore, in descending order of estimated impact:

(1) Identify "pure data" that cannot affect program behavior either directly or indirectly, and elide instrumentation for memory stores to it. For example, this could include data that is only read from and written to disk/network/etc.
(2) Enable safe stacks (§2.3.3), and elide instrumentation for variables or register spills placed on the safe stack.
(3) Identify sibling and/or tail-recursive function calls, and elide instrumentation on return addresses for such calls.
(4) Identify read-only system calls, and elide synchronization with the verifier at such calls.

6.2. Spatial Memory Safety

In comparison to data-flow integrity, past work on memory safety (§2.1) focuses directly on identifying out-of-bounds accesses. These include LowFat [55, 56], AddressSanitizer [132] (ASan), and Write Integrity Testing [10], which we discuss below and compare in Figure 1.

Write integrity testing shares a similar design with that of data-flow integrity [30], as both use pointer analysis to identify the set of authorized writes for each memory location. However, whereas data-flow integrity relies on reaching definitions, write integrity testing directly partitions and assigned identifiers (colors) based on points-to sets, by merging intersecting sets until a fixed point is reached. Because this can permit false negatives, the authors attempt to remediate imprecision by inserting guards between objects to detect overflow, and by checking separate equivalence classes of indirect callsites and call targets–a form of coarse-grained control flow integrity (§2.3.1). Nevertheless, this latter protection still relies on the same imprecise pointer analysis, and

neither protection can detect intra-object overflows. Also, unlike other memory safety designs discussed below, write integrity testing does not detect out-of-bounds loads.

AddressSanitizer, which has been integrated into both the Clang/LLVM and GCC compilers, uses shadow state to track memory validity at 8-byte granularity. However, their approach also permits false negatives, because out-of-bounds accesses can coincide with other valid objects, despite the presence of adjacent guard regions. In contrast, LowFat is sound and guaranteed to detect out-of-bounds pointers, but also incurs additional false positives, because some programs may deliberately create pointers to valid objects in this manner (§6.3.1). Their approach checks that certain pointer arithmetic and all pointer dereferences are always within bounds. Each object allocation is rounded up to the nearest predefined power-of-2 size, and a custom allocator assigns addresses such that allocation sizes can be computed from a table lookup of bitwise arithmetic on the pointer address, which avoids storing size metadata inside pointers or objects themselves.

However, both approaches suffer from common drawbacks of memory safety designs, which include additional memory and performance overhead, as well as lack of support for intra-object overflows. In fact, very few designs [109, 116] can detect intra-object overflows, and those that do either lack support for C++ [109] or rely on the deprecated [77] MPX [23] extension. Our experiments with the SPEC CPU2006 [82] and CPU2017 [25] benchmarks demonstrate that both designs have significant overhead, and measure a geometric mean of 48% relative performance with respect to an uninstrumented baseline, as shown in Figure 1. They also show that LowFat is incompatible with both the `502.gcc_r` and `602.gcc_s` benchmarks, which crash during execution, and that Clang/LLVM AddressSanitizer can miscompile programs by incorrectly removing virtual function calls, which we have now fixed[1].

One proposed design could focus exclusively on the safety of memory stores, which are needed by exploit primitives. In comparison with past work, this could improve precision by supporting intra-object subfields, while decreasing overhead by only instrumenting memory stores that may go out-of-bounds. Such a design could track the type and offset/provenance of every memory store at runtime, and detect potential out-of-bounds accesses when either there is a type mismatch, or the offset/provenance is not within the parent object, as outlined by the following invariants. Additional instrumentation may be necessary to improve precision of heap or stack allocations, to ensure that the runtime is aware of the type and size of each memory object.

(1) Memory stores should only modify variables of the same type from a compatible base pointer.
(2) Memory stores should never overwrite variables that are not directly address-taken.

6.2.1. Performance. As with data-flow integrity, performance of the final design is critical; however, a memory safety approach may ultimately be faster than a data-flow integrity approach, for the following reasons. By eagerly detecting out-of-bounds accesses, instrumentation only needs to be inserted at memory allocations and stores, instead of

[1] `https://reviews.llvm.org/D88368`

Design	Errors	False Ps.	Invalid	Ok
Baseline	0	0	0	47
Clang/LLVM ASan	0	0	0	47
LowFat	2	9	0	36

TABLE 2. Correctness of various memory safety designs.

memory loads and stores, as with data-flow integrity. Below, we identify a number of further optimizations that we plan to explore, in descending order of estimated impact:

(1) Maintain memory store metadata in-process, which allows for removal of the messaging primitive, external verifier, and system call verification. No additional protection is needed for the metadata region, because unsafe program writes will never have a correct offset/provenance.
(2) Check only for mismatched invariants on stores, without instrumenting allocations.
(3) Statically analyze allocation boundaries, and elide instrumentation for memory writes with constant or loop-indexed offsets that are known to be safe (in bounds).
(4) Identify loop indices, and only emit a single range check for memory writes within a loop that perform a homogeneous contiguous linear traversal.
(5) Enable safe stacks (§2.3.3), and elide instrumentation for variables or register spills placed on the safe stack.

6.3. Evaluation

6.3.1. Correctness. We manually reviewed all warnings issued by each design on the SPEC benchmark suites. Most were for known [55, 116, 132] memory safety bugs in the CPU2006 benchmarks, so we patched the benchmarks to fix these, and reran our experiments. We summarize the final results in Table 2, distinguishing between errors (crashes/hangs), false positive violations, invalid results (output doesn't verify), and successful runs.

All warnings issued by LowFat were determined to be false positives, as out-of-bounds accesses do not occur until invalid pointers are actually dereferenced. Many SPEC benchmarks intentionally create pointers to valid objects from out-of-bounds pointers to other objects [55]. Other warnings occur on actual out-of-bounds pointers to invalid objects, but are never dereferenced. For example, in `400.perlbench`, out-of-bounds pointers are used as a sentinel value, whereas in `526.blender_r`, out-of-bounds pointers created during initialization of a doubly-linked list are immediately overwritten afterwards. Finally, in `525.x264_r` and `625.x264_s`, an out-of-bounds pointer is passed as an argument to a prefetching function, but because prefetch is defined not to fault on invalid addresses, this does not affect program correctness.

6.3.2. Performance. Since past work [10, 30, 136] on data-flow integrity is not open-source or directly-comparable, we plan to evaluate our ultimate design by comparing its effectiveness and performance against related work [55, 56, 132] on spatial memory safety. In addition to the SPEC benchmark suite, we also plan to evaluate against other real-world applications, such as the NGINX webserver, as well as multi-threaded and persistent memory applications.

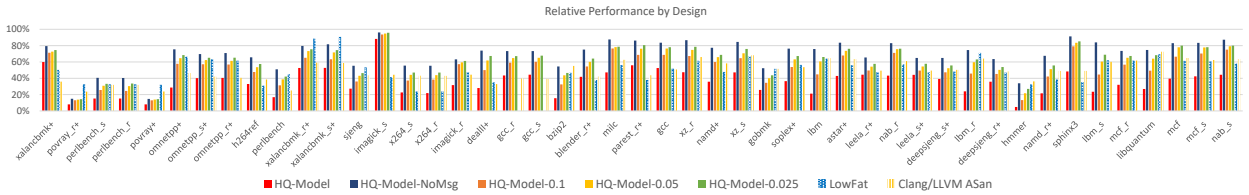

FIGURE 1. Relative performance of memory safety designs and our naive prototype, with failed runs not shown. Suffix '+' denotes C++.

In Figure 1, we compare past work on memory safety against a naive prototype, only for the purpose of evaluating performance overhead. Our prototype does not actually implement any memory integrity policy; rather, it simply inserts instrumentation at every memory store, with the exception of those involving objects on the safe stack (§2.3.3), and executes with both inlining and system call checking disabled. Measuring this baseline performance is important, because both proposed designs add additional instrumentation; either memory loads (for data-flow integrity), or memory allocations (for memory safety). Using our software model of our microarchitecture-based IPC primitive (§5.3), we show four different variants of this prototype: HQ-MODEL, which sends a message on every memory store, HQ-MODEL-0.1, which sends a message on every 1 in 10 memory stores, HQ-MODEL-0.05, which sends a message on every 1 in 20 memory stores, and HQ-MODEL-0.025, which sends a message on every 1 in 40 memory stores. We also show a fifth variant, HQ-MODEL-NOMSG, which calls an empty instrumentation function that does not send any messages.

These results demonstrate that our final approach will need a geometric mean relative performance of approx. 48% to achieve parity with AddressSanitizer and LowFat, whereas currently HQ-MODEL, HQ-MODEL-0.1, HQ-MODEL-0.05, HQ-MODEL-0.025, and HQ-MODEL-NOMSG are at 31.2%, 48.3%, 55.3%, 58.4%, and 65.4% respectively. Since our goal is to improve precision by modeling intra-object overflows and avoiding imprecise pointer analysis, the exact performance target is somewhat negotiable, but higher performance is always better. Enabling inlining and other optimizations will also provide an additional performance boost. These results suggest that the performance of a memory safety approach is likely to be higher than that of a data-flow integrity approach, because memory loads are likely more frequent than memory allocations, and because our proposed optimizations for memory safety (§6.2.1) can reduce overhead without affecting protection scope, which is not true for that of data-flow integrity (§6.1.1).

6.4. Thesis Timeline

- October 2020: Give thesis proposal, develop proposed work.
- November 2020 - March 2021: Continue development.
- March - May 2021: Thesis writing, continue development, consider paper submission (Oakland: March/June, CCS/NDSS: May, USENIX Security: June).
- May 2021: Defend.

Bibliography

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

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

[3] 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.

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

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

[6] WebAssembly Specification, Sept. 2020. URL https://webassembly.github.io/spec/core/.

[7] M. Abadi, M. Budiu, Ú. 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.

[8] D. Adrian, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, P. Zimmermann, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, and E. Thomé. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 5–17. ACM Press, 2015. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813707.

[9] 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.

[10] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (Sp 2008)*, pages 263–277, May 2008. doi: 10.1109/SP.2008.30.

[11] 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.

[12] 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.

[13] 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.

[14] B. Baumann. Kompromat, Mar. 2015. URL https://github.com/BenBE/kompromat.

[15] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, ATEC '05, pages 41–46. USENIX Association, 2005. doi: 10.5555/1247360.1247401.

[16] 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.

[17] 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.

[18] 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`.

[19] 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.

[20] 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.

[21] 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.

[22] A. J. Bonkoski, R. Bielawski, and J. A. Halderman. Illuminating the Security Issues Surrounding Lights-Out Server Management. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, pages 10–10. USENIX Association, Aug. 2013. doi: 10.5555/2534748.2534761.

[23] 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`.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, pages 21–24, 2016. ISBN 1-891562-41-X. doi: 10.14722/ndss.2016.23415.

[33] 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.

[34] 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.

[35] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. doi: 10.1145/1950365.1950380.

[36] 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.

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

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

[39] 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.

[40] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium - SEC '14*, SEC'14, pages 95–110. USENIX Association, Aug. 2014. ISBN 978-1-931971-15-7. doi: 10.5555/2671225.2671232.

[41] 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.

[42] 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.

[43] A. Cui and S. J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-Area Scan. In *Proceedings of the 26th Annual Computer Security Applications Conference on - ACSAC '10*, pages 97–97. ACM Press, 2010. ISBN 978-1-4503-0133-6. doi: 10.1145/1920261.1920276.

[44] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *Proceedings 2013 Network and Distributed System Security Symposium*. The Internet Society, 2013. URL `http://www.internetsociety.org/doc/when-firmware-modifications-attack-case-study-embedded-exploitation`.

[45] 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.

[46] 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.

[47] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*, SEC'13, pages 463–478. USENIX Association, Aug. 2013. ISBN 978-1-931971-03-4. doi: 10.5555/2534766.2534806.

[48] 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.

[49] J. E. Denny, S. Lee, and J. S. Vetter. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 125–136, New York, NY, USA, May 2016. Association for Computing Machinery. ISBN 978-1-4503-4314-5. doi: 10.1145/2907294.2907303.

[50] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS XIII*, volume 36, pages 103–103. ACM Press, 2008. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346295.

[51] 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.

[52] 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.

[53] 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.

[54] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, pages 4:1–4:8, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7226-8. doi: 10.1145/3337167.3337171.

[55] 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.

[56] 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.

[57] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 605–620. USENIX Association, 2013. ISBN 978-1-931971-03-4. doi: 10.5555/2534766.2534818.

[58] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 475–488, New York, NY, USA, Nov. 2014. Association for Computing Machinery. ISBN 978-1-4503-3213-2. doi: 10.1145/2663716.2663755.

[59] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A Search Engine Backed by Internet-Wide Scanning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 542–553, New York, NY, USA, Oct. 2015. Association for Computing Machinery. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813703.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] D. Gohman. WASI: WebAssembly System Interface, Apr. 2019. URL `https://github.com/WebAssembly/WASI`.

[68] 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.

[69] 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.

[70] 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.

[71] R. D. Graham. Masscan, Oct. 2020. URL `https://github.com/robertdavidgraham/masscan`.

[72] 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. URL `https://www.ndss-symposium.org/wp-content/uploads/2017/09/ndss2017_09-1_Gras_paper.pdf`.

[73] 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`.

[74] 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.

[75] 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.

[76] 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. pages 185–200. ACM Press, 2017. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363.

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

[78] 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.

[79] 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.

[80] C. Heffner. Littleblackbox, Feb. 2014. URL `https://github.com/devttys0/littleblackbox`.

[81] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 205–220. USENIX Association, 2012. doi: 10.5555/2362793.2362828.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 107–120. USENIX Association, 2019. ISBN 978-1-939133-03-8. doi: 10.5555/3358807.3358817.

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

[88] 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.

[89] 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.

[90] D. Kumar, Z. Wang, M. Hyder, J. Dickinson, G. Beck, D. Adrian, J. Mason, Z. Durumeric, J. A. Halderman, and M. Bailey. Tracking Certificate Misissuance in the Wild. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 785–798, May 2018. doi: 10.1109/SP.2018.00015.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] H. Li, D. Tong, K. Huang, and X. Cheng. FEMU: A firmware-based emulation framework for SoC verification. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 257–266, Oct. 2010.

[97] 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.

[98] 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 Security Symposium - SEC '19*, page 18, 2019.

[99] 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.

[100] 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.

[101] G. Lyon. Nmap security scanner. URL `https://nmap.org/`.

[102] 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.

[103] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham. Mouse Trap: Exploiting Firmware Updates in USB Peripherals. In *Proceedings of the 8th USENIX Conference on Offensive Technologies*, WOOT'14, pages 1–10. USENIX Association, Aug. 2014. URL `https://www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz`.

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

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

[106] H. D. Moore. Metasploit. Rapid7. URL `http://www.metasploit.com/`.

[107] H. D. Moore. Ssh-badkeys, Jan. 2015. URL `https://github.com/rapid7/ssh-badkeys`.

[108] A. Murdock, F. Li, P. Bramsen, Z. Durumeric, and V. Paxson. Target generation for internet-wide IPv6 scanning. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 242–253, New York, NY, USA, Nov. 2017. Association for Computing Machinery. ISBN 978-1-4503-5118-8. doi: 10.1145/3131365.3131405.

[109] 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.

[110] 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.

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] K. Nohl, S. Krißler, and J. Lell. BadUSB—On accessories that turn evil, Aug. 2014. URL `https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf`.

[116] 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.

[117] 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.

[118] 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`.

[119] 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.

[120] 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.

[121] 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`.

[122] 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.

[123] 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.

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

[125] A. Rossberg. Reference Types for WebAssembly, Mar. 2018. URL `https://github.com/WebAssembly/reference-types`.

[126] A. Rossberg. Multiple Memories for Wasm, Oct. 2019. URL `https://github.com/WebAssembly/multi-memory`.

[127] 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.

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

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

[140] 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.

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

[142] 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.

[143] 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.

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

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

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

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

[148] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. 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. URL `https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf`.

[149] 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`.

[150] 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.

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

[152] 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.

[153] 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.

[154] 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.

[155] 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.

[156] C. Watt. Mechanising and verifying the WebAssembly specification. pages 53–65. ACM Press, 2018. ISBN 978-1-4503-5586-5. doi: 10.1145/3167082.

[157] R.-P. Weinmann. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, WOOT'12, pages 1–10. USENIX Association, 2012. doi: 10.5555/2372399.2372402.

[158] 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.

[159] 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.

[160] 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.

[161] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468. IEEE, June 2014. ISBN 978-1-4799-4394-4. doi: 10.1109/ISCA.2014.6853201.

[162] 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.

[163] 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.

[164] 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.

[165] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings 2014 Network and Distributed System Security Symposium*, pages 23–26. Internet Society, 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23229.

[166] 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.

[167] L. Zhang and S. Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 897–911, USA, July 2019. USENIX Association. ISBN 978-1-939133-03-8.

[168] 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.

[169] 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.