Morpheus II: A RISC-V Security Extension for Protecting Vulnerable Software and Hardware

Austin Harris[1], Tarunesh Verma[2], Shijia Wei[1], Lauren Biernacki[2], Alex Kisil[3], Misiker Tadesse Aga[2], Valeria Bertacco[2], Baris Kasikci[2], Mohit Tiwari[1], and Todd Austin[2,3]

[1]University of Texas, [2]University of Michigan, [3]Agita Labs

[1]{austin.harris, shijiawei, tiwari}@austin.utexas.edu

[2]{tarunesh, lbiernac, misiker, valeria, barisk, austin}@umich.edu

[3]{alex, austin}@agitalabs.com

***Abstract*—Morpheus II is a secure processor designed to prevent control flow attacks. Morpheus II strengthens the defenses of the Morpheus [1] processor, by deploying always-on encryption to obfuscate code and pointers along with runtime churn to thwart side-channel attacks. Focusing on Remote Code Execution attacks, we modified the RISC-V Rocket core to support always-encrypted code and code pointers with negligible performance impact and less than 2% area overhead. Morpheus II was deployed running a web server interface to a mock medical database on AWS F1 instances, where it was red-teamed for three months by over 500 security researchers. No vulnerabilities were discovered in Morpheus II. In addition, we evaluated Morpheus II against a range of CWE attack classes including a Blind ROP attack on the web server. We show that Morpheus II defenses increase Blind ROP probe time for gadgets from weeks to likely thousands of years.**

I. Introduction

With the growth of cloud computing and IoT, data security has never been more important. With cloud computing, we hand over our personal and private information to cloud providers and their customers, and we can only hope that they steward our data well. For IoT devices, we install them everywhere in our homes, cars, and workplaces, and then we trust these devices to not spy on us. We extend trust to the manufacturers and vendors of computing systems today, and, in many cases, they are letting us down. The world of computing is replete with examples of data breaches and poor stewardship of sensitive data, suggesting that stronger security measures are surely needed.

Today's Defenses Lack Durability: In modern computer security, there are two primary means by which systems are protected. The first is a ***patch-based security*** approach where software and hardware vulnerabilities are addressed by patching the system's software. The key challenge with this approach is that attacks will not stop until the system is free of vulnerabilities.

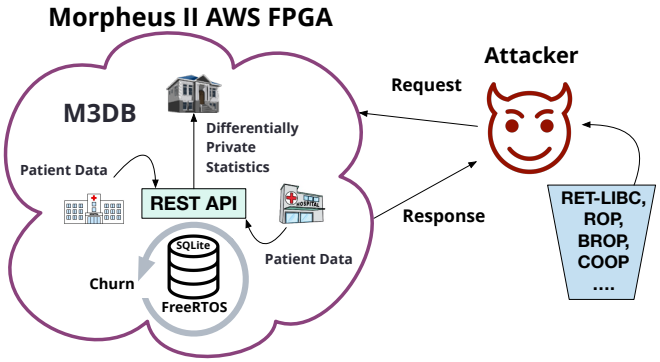

Figure 1: Deployed Morpheus II system for the DARPA Fett Challenge [2]. Attackers attempted to exploit a mock COVID-19 SQLite database (M3DB) running on a FreeRTOS-based web server by sending requests to our AWS FPGA prototype. Morpheus II always encrypts code and code pointers in M3DB, making control-flow hijacking attacks infeasible.

Unfortunately, the complexity of modern software and hardware, combined with the rate at which new software is created, ensures that patched systems always have plenty of additional vulnerabilities for motivated attackers to exploit.

A more powerful approach is to outfit the system's software or hardware with a ***targeted defenses against well-known attacks***. Examples of these defenses include no-execute stacks [3], which prevents code injection on the stack, or CATalyst [4] which uses Intel's Cache Allocation Technology (CAT) [5] to silence covert channels in the last-level cache. These defenses are superior to patch-based defenses because they can typically defend against an entire class of attacks. However, targeted attacks often have limited scope, thus, attackers will devise ways to step around these defenses to continue exploiting vulnerable systems. For example, when no-execute stacks were introduced, the attack community quickly perfected techniques to inject code into the heap with heap-spray attacks [6].

Unfortunately, the sum total of today's security defense only throws moderately-strong barriers in the face of oncoming attackers for existing attacks. If new "zero-day" vulnerabilities are discovered, systems are completely unprotected from attacks. As such, there is a great need for new thinking in the arena of computer security, in particular, for defenses that are more durable against a fast-growing slate of security attacks.

Morpheus Defenses Work Despite Vulnerabilities: Moving target defenses like the Morpheus [1] security technology works in a vulnerability-agnostic fashion, allowing it to stop attacks on vulnerable software and hardware. Where traditional security defenses focus on specific vulnerabilities, Morpheus defenses instead obfuscate the information assets required by attacks. This approach denies attackers timely access to the critical information assets needed to attack systems. The critical information assets that Morpheus can protect include the following:

- Code representation
- Code and data pointer representation
- Code and data layout (both absolute and relative)
- Function and return pointer representation

Critical information assets are protected using ***encryption and churn***. By encrypting code and data pointers, attacks lose the ability to find code gadgets, inject pointers into the stack, perform relative address attacks, and so on. Yet, savvy attackers can adapt to even high-entropy encryption by utilizing memory disclosures and side-channel attacks to eventually acquire the information assets they need to attack a system. Consequently, Morpheus uses churn to re-key the cipher used to protect information assets on a regular basis, thereby destroying any information assets that were disclosed or inferred by attackers.

A key aspect of Morpheus is that it only protects information assets within the program and micro-architecture implementation. These assets possess undefined semantics because they are the internal workings of compilers and micro-architectures. Consequently, encrypting and churning these assets, while breaking attacks, has little to no effect on normal software. This property allows Morpheus to imbue defenses into vulnerable software and hardware without putting undue burdens on users. It is quite challenging to find any attack that doesn't utilize some subset of these critical information assets. Thus, protecting these information assets broadly stops security attacks [7], ranging from control-flow attacks, to privilege escalation, to disclosure attacks and beyond. Moreover, it is likely that attacks discovered in the future will also utilize some subset of these information assets, thus, it is possible that Morpheus systems could have some measure of immunity from attacks in the near future.

Morpheus Design Challenges: A drawback of the original Morpheus design is that it has significant overheads, invasive changes to the micro-architecture, and requires complex software support. Simulation estimates in gem5 [8, 9] for Morpheus show worst-case overheads of 6.71% at 50ms churn with overheads over 30% as the churn rate increases, without protecting system calls or kernel code. Morpheus requires memory tagging to implement runtime churn that updates the represention of all pointers in the system. These tags require adding 2 bits to the register file and caches, as well as a dedicated portion of DRAM. A dedicated tag cache is required for acceptable performance. While most Morpheus protections can be implemented with minimal software burden and compiler support, data pointer encryption requires a complex LLVM pass and manual porting of code that acccesses raw addesses (*e.g.*, device drivers).

Morpheus II Secure Processor: In this paper, we introduce Morpheus II, which implements a subset of the Morpheus protections with negligible overhead, less than 2% area, and simple hardware modifications. Morpheus II is a modified RISC-V Rocket [10] core prototyped on Xilinx FPGAs and deployed on Amazon AWS F1 instances. It provides always-on code and code pointer encryption, protecting against a large class of high-value control-flow hijacking attacks. Morpheus II consists of simple LLVM support to identify function pointers and transform instructions, and only requires software modifications in rare handwritten assembly. Instead of configurable run-time churn, Morpheus II only changes encryption keys when a program starts or an exception is encountered, greatly simplifying the implementation of churn. Fig. 1 shows our deployment of Morpheus II on AWS, consisting of a SQLite mock COVID-19 medical database (M3DB) running on a FreeRTOS web server. As part of the DARPA FETT bug bounty [2], Morpheus II was attacked by over 500 researchers for 3 months. Attackers were given knowledge of the software stack and potential vulnerabilites, and attempted to exploit them by sending malicious requests to the M3DB web server. None of the attackers were successful in infiltrating the database running on our protected Morpheus II system.

II. Background & Related Work

In this section, we briefly recall classic control flow attacks and their proposed defenses. While we describe

potential defenses, we leave the direct comparison with related work to Section VIII.

A. Control Flow Attacks.

Control flow attacks refer to attacks that result in the attacker being able deviate the program execution from normal behaviors or execute arbitrary code in the victim program. A vast majority of control flow attacks stem from memory errors. These software vulnerabilities are categorized into spatial memory safety issues like buffer overflows [11–13], format string [14], and temporal memory safety issues like use-after-free [15] and double-free [16]. Memory errors are usually introduced by memory unsafe languages, which are pervasive as they are often used to implement run-time libraries and system software for even memory-safe languages. These vulnerabilities, once exploited, can grant the attacker arbitrary read and/or write access within the execution context of the victim.

To achieve code execution, code injection or code reuse techniques are typically employed. In code injection attacks, adversaries input malicious code into victim memory (*e.g.,* the stack) using the write access gained from above vulnerabilities; and overwrite the control flow targets (*e.g.*, return addresses) to redirect the victim program to execute the provided code (*e.g.*, code to launch a shell).

Code reuse attacks were invented to bypass defenses against code injection attacks. Instead of supplying an execution payload to the victim program, attackers reuse instructions that already exist in the address space. For example, return-to-libc attacks [17] divert control flow to `libc` functions for process and memory management with malicious function arguments. Another example of powerful code reuse attacks is return oriented programming (ROP) [18]. Instead of `libc` functions, ROP leverages existing instruction sequences that end with a return instruction. The sequence of instructions is called a *gadget*. Attackers overwrite return addresses on the stack to chain *gadgets* together in order to execute arbitrary code. Variants that target jump (indirect branch) instructions (JOP) [19], C++ virtual functions (counterfeit object-oriented programming, also known as COOP) [20], and just-in-time (JIT) compilation [21, 22] have also been exploited. The ability to identify these *gadgets* in the victim program's address space is the key to successful execution of code reuse attacks.

B. Control Flow Defenses.

Enforcing memory safety. Completely enforcing both spatial and temporal memory safety can stop all memory error exploits, thus preventing all control flow attacks. CHERI [23] uses capabilities to enforce memory safety, requiring tags in hardware and extra capability registers to define access regions and permissions. CHERI requires recompilation and significant porting effort for system-level software, in addition to overheads to store and access capabilities. Systems such as lowRISC [24], the Dover processor [25], and PUMP [26] also use hardware tag support to enforce various policies, including memory safety. Intel MPX [27] has registers and instructions to manage base and bounds associated with data structures. REST [28] provides coarse-grained memory safety by placing random tripwires around stack and heap allocations, utilizing hardware support for minimal overhead. Califorms [29] strives for fine-grained memory safety within objects (*e.g.*, members of a `struct`) by providing byte granularity blacklisting instead of cache line granularity provided by REST.

Restricting control flow behavior. Restricting the control flow behavior of programs limits attackers' ability to execute arbitrary code. The NX-bit [30] is a widely-deployed example that restricts data pages from being executed, but is not effective against code reuse attacks. Traditional control flow integrity (CFI) solutions restrict the source and destination pairs of control flow transfers. A wide range of CFI techniques have been proposed. For example, classification-based solutions [31–33] classify control flow pointers based on pointer categories (*e.g.,* function pointer vs. return address vs. vtable pointer) and their static properties (*e.g.,* number of parameters). Prohibiting control flow transfer between two distinct classes provides *coarse-grained CFI*. Labeling approaches [34–36] aim for *fine-grained CFI* by labeling control flow sources and destinations along a statically computed control flow graph (CFG) and dynamically restricting control flow to follow the labels. However, static analysis used to compute the CFGs may not guarantee precision and can lead to high run-time check overheads. Shadow call stack solutions [31, 35–37] aim to enforce backward-edge CFI by storing an extra copy of the return address onto a shadow stack on each function call and verifying the integrity of return address upon each return instruction. Two industrial techniques have been productized: Intel CET [37], ARM BTI [38].

Software and hardware code pointer integrity solutions [39–41] propose to verify the integrity of pointers

at run-time using cryptographic MACs. At run-time, each time a control flow pointer is stored into memory, a MAC of the pointer address, its label, and certain run-time properties are computed. The integrity is verified whenever the said pointer is loaded and used. By enforcing the pointer integrity dynamically, the control flow behavior is limited to verified pointers.

Obfuscating and hiding attack assets. PointGuard [42] and similar systems [43] obfuscate pointers with randomization, utilizing weak XOR-based encryption with the same key across all types of pointers. Instruction Set Randomization (ISR) [44–47] schemes use encryption to protect code against injection attacks. Recent proposals of ISR [48, 49] incorporate the idea of code randomization and strong encryption to hide *gadgets* in order to thwart code reuse attacks. A recent version of the ASIST [49] ISR system re-encrypts code with a new key when the process crashes. N-version systems [50–52] execute programs with multiple versions, each with different randomized layouts and then compare the outputs. An attacker must infiltrate every version for a successful attack. Shuffler [53] is a software-based moving target defense (MTD) that periodically randomizes the code layout during run-time, along with encrypting return addresses. Phantom Name System (PNS) [54] creates multiple random addresses for every instruction (or coarser, such as every basic-block), forcing an attacker to guess which is actually executed at run-time. PNS also encrypts pointers when they are created and decrypts them at the call site to prevent attackers' tampering with the return. Morpheus [1] is a hardware-based MTD with strong encryption and fast run-time randomization of code and pointers. We next describe Morpheus in detail as its design is closest to Morpheus II.

C. Morpheus Summary

The Morpheus [1] architecture randomizes the semantics of program execution below the language-level such as code and data pointer addresses (relative and absolute), code representation, and return pointers. Morpheus uses encryption to implement this randomization, enabling these undefined program semantics to be periodically *churned* at configurable intervals. Periodically changing these representations makes it infeasible for attackers to utilize these semantics to perform attacks like control-flow hijacking, even for unknown variations. Two bit tags in the micro-architecture identify domains: code, code pointers, data pointers, and other data. These tags are stored alongside the register file, caches, and a dedicated portion of DRAM. Relative distances of pointers are protected using *pointer displacement*, which offsets the code and data address spaces in the program with randomly generated offsets. All of the Morpheus defenses combined provide 504 bits of true random entropy to thwart attackers. Since very fast churn rates have significant overheads, an attack detector can be used to only increase churn times when it observes suspicious activity such as invalid code execution or jumps through data pointers. The gem5 [8, 9] prototype of Morpheus had overheads from 0.84% on average to 6.71% in the worst-case (gcc) with a 50ms churn time. The gem5 evaluation was done with system-call emulation mode, so overheads associated with changes in the system call implementation are not accounted for and the kernel is not protected. The QARMA cipher was used for encrypting and decrypting assets in Morpheus, with an estimated delay of 3.25ns. These overheads are due to extra memory requests by the churn unit and the pipeline potentially stalling during churn with very fast rates.

III. Threat Model

Memory errors are still among the most prevalent software errors in the Mitre CWE top 25 [55]. Morpheus II primarily protects against control-flow hijacking attacks that arise from these memory errors such as return/jump-oriented programming, double-free, counterfeit object-oriented programming, return-to-libc, and more. In addition, Morpheus II's always-on encryption throughout the caches, RAM, and disk can prevent disclosure of encrypted code and pointers through vectors like cold-boot attacks. We do not consider an attacker performing Denial-of-service (DoS) attacks, fault injection, modifying the boot sequence, or tampering with the random number generator. Unlike the Morpheus [1] design, Morpheus II does not protect against attacks that modify data pointers to leak information or escalate privilege. The trusted computing base (TCB) consists of the Morpheus II processor core, compiler passes that generate appropriate encrypted instructions, and a small amount of loader code.

IV. The Morpheus II Architecture

Morpheus II is a refinement of the original Morpheus design [1] built in real hardware. Morpheus II was developed in the DARPA SSITH [56] program, which had a requirement that designs be placed into the DARPA FETT [2] program, where it was to be built on an FPGA and red-teamed for potential vulnerabilities. Thus, while the original Morpheus prototype in simulation provided full code, code pointer, and data pointer encryption, the

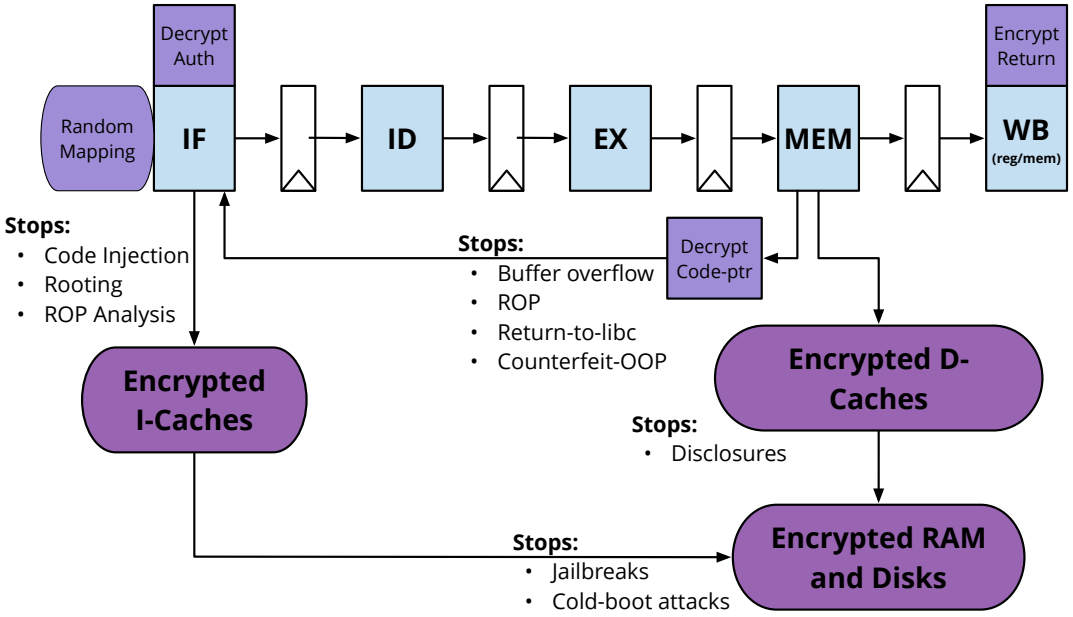

Figure 2: Processor Architecture with Morpheus II. Encrypted code from the I-Cache is decrypted during the fetch stage after an I-Cache response. Instructions that source code pointers decrypt them during execute, while instructions that write code pointers encrypt them during writeback.

goal of Morpheus II was to capture the main security strengths of Morpheus defenses while being buildable and performant. The Morpheus II architecture is implemented as a RISC-V extension applied to the Rocket Core in-order scalar pipeline [10].

Additionally, Morpheus II was tuned to primarily stop remote code execution (RCE) attacks, which are a high-value class of attacks that allow remote attackers to inject code into vulnerable machines. For example, Morpheus II RCE defenses would handily stop the Microsoft Exchange Server RCE attacks that were in the news at the time of this writing [57]. As another example, one of the largest security breaches in US history, the 2017 Equifax breach, was also initiated via an RCE attack on a vulnerability in the Apache Struts library that would have been easily stopped with Morpheus II defenses.

A. Morpheus II Architectural Features

Morpheus II implements ***always-encrypted code pointers***. A twelve-round Simon cipher [58] uses a randomly generated key to strongly encrypt all code pointers (*i.e.*, function pointers and return pointers). From the programmer or attacker's perspective, all code pointers are encrypted all the time. As such, code pointers in DRAM, caches, or registers are encrypted. Code pointers are only decrypted during the execution of Morpheus II jump instructions immediately before a function pointer or return address is placed into the PC register and within the pipeline during encrypted arithmetic instructions.

When code pointers are always encrypted, it complicates the attacker's ability to forge code pointers. Additionally, relative address attacks become very difficult to synthesize because the computation of a relative address from an encrypted pointer is cryptographically hard. Since virtually all RCE attacks involve some form of code pointer injection or manipulation, these attacks become significantly more challenging.

In addition, Morpheus II implements always-encrypted code. Like code pointers, a twelve-round Simon cipher [58] uses a randomly generated key to strongly encrypt all instructions. Code remains always encrypted in the binary, DRAM, and all caches. Thus, instructions are only decrypted in the pipeline for execution. By always encrypting code, it becomes very challenging for attackers to identify code gadgets or synthesize new code for injection.

Fig. 2 depicts the 5-stage Rocket pipeline with Morpheus additions. For always-encrypted code, a Simon cipher is placed in the Fetch stage between the I-Cache and the fetch queue to decrypt instructions before they are sent through the pipeline. Always-encrypted code pointers are implemented with decryption in the Execute stage after reading the source registers, and encryption in the Writeback stage when writing a pointer. The final Morpheus II design was optimized to place the twelve-round Simon ciphers within the existing Rocket pipeline stages, without impacting the baseline frequency.

Instruction Class	Example	Semantics
Arithmetic	enc_add r1, r2, 4	r1 = enc(dec(r2) + 4)
Relational	enc_sleq r1, r2, r3	r1 = dec(r2) $\leq$ dec(r3)
Indirect Jump	enc_jalr r2, LR	LR = enc(PC), PC = dec(r2)
Jump	enc_jal r2, TGT	LR = enc(PC), PC = TGT

Table I: Morpheus II RISC-V ISA Extensions. Morpheus II adds three classes of instructions: *i)* always-encrypted pointer ALU operations, *ii)* decrypting pointer relational tests, and *iii)* decrypting indirect jumps and returns. Note that the *enc()* and *dec()* interfaces encrypt and decrypt always-encrypted pointers within the pipeline.

Figure 3: Morpheus II Software support. The frontend LLVM pass analyzes the Intermediate Representation to identify and mark pointer manipulation instructions. The backend emits instructions to initialize function pointers and replaces pointer manipulation instructions and returns with the encrypted version. Finally, a small program encrypts the instructions in the resulting binary.

To thwart disclosures and side-channel attack, Morpheus II churns encryption keys whenever the system boots, reboots, or when a security violation has been detected (*e.g.*, segmentation fault or misaligned instruction fetch). On each of these events, the system is very quickly warm-booted, which reloads the code under a new encryption key, and reconstitutes all code pointers from the original binary, again under a new encryption key. The churn process ensures that any valuable information gathered by attackers since the last churn cycle will be lost due to the re-keying of the Simon ciphers. In addition, by limiting churn to system warm boots, Morpheus II did not require tagged memory, which significantly reduced the complexity of the changes needed for the Rocket core, which in turn led to a fast build time for a small academic design team.

B. Morpheus II Software Support

Fig. 3 describes the Morpheus II compilation process consisting of a LLVM frontend pass, backend pass, and ELF encryptor to encrypt instructions in the final binary. The frontend pass marks instructions that manipulate pointers so that the backend pass can replace them with the Morpheus II equivalent that performs encryption and decryption as required. For example, the code that generates the address for a function pointer will now generate an encrypted address and jump to the function using *enc_jalr* which performs decryption on the base register. Morpheus II's protections apply to both user and kernel code—a RISC-V port of FreeRTOS [59] and the newlib C library are fully supported.

C. Morpheus II Evolution

Morpheus [1] implemented moving target defenses using a **domain tagging** mechanism to identify all pointers in memory, **pointer displacement** to obscure relative pointer distances, **domain encryption** to encrypt all pointers, and a configurable runtime **churn unit** to re-randomize the program by changing encryption keys. The original Morpheus prototype was evaluated in simulation using gem5 [8, 9] with average overheads under 1% and worst-case under 7%, depending on the churn period. In addition, the domain tagging mechanism required microarchitectural support in the register file, caches, and memory. Software support for Morpheus required a complex LLVM pass to identify data pointers as well as manual identification of pointers that access raw memory (*e.g.*, device drivers).

As we began to adapt similar protections for Morpheus II in the RISC-V Rocket core, we quickly decided to pursue a design without explicit tags in the microarchitecture to simplify the modifications—instead domains can be identified by the types of instructions that manipulate them, expressed by the compiler. Table I details the instruction classes the were added to support always-encrypted pointers. Rocket provides a tightly-coupled accelerator interface (RoCC) to easily add accelerators running reserved RISC-V instructions. Our initial Morpheus II implementation added a RoCC accelerator to perform the encrypted variants of instructions that manipulate pointers. In addition, we added an extra pipeline stage to decrypt code as it is fetched from the instruction cache. Unfortunately, the overheads associated with the RoCC accelerator and extra pipeline stage resulted in significant overheads (of up to 68%). Thus, we refined the Morpheus II implementation by adding direct support for our instructions to the Rocket pipeline. We were also able to move the decryption of instructions to between the instruction cache and the fetch queue, eliminating the extra pipeline stage without impacting the critical path. To simplify the compiler support and software porting effort, we decided to focus on Remote Code Execution (RCE) attacks and eliminated data pointer protections. Note that Morpheus II hardware still supports data-pointer encryption and we are currently adding compiler support. Ultimately, the final Morpheus II design eliminated runtime performance impacts and simplified the microarchitecture while still protecting against control-flow attacks.

V. Evaluation Methodology

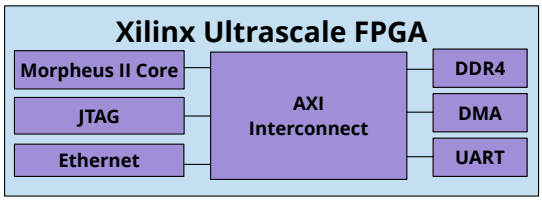

Figure 4: FPGA prototype of Morpheus II with a modified Rocket core and peripherals.

Arch	RV32IMAC
Frequency	50 MHz
Core	5-stage
Caches	4k I-cache / 4k D-cache
Mul/Div	Pipelined 8-cycle
Memory	2GB DDR4
Devices	Ethernet, UART
Morpheus II Crypto	12-Round Simon

Table II: Morpheus II and Baseline Hardware Configuration

We developed Morpheus II on a Xilinx UltraScale Plus VCU118 FPGA as part of the DARPA SSITH [56] program. Fig. 4 describes the VCU118 setup with the RISC-V Morpheus II core, ethernet, UART, and DRAM. Table II describes the specific parameters of the baseline Rocket core processor [10] provided by DARPA, which we modified to include support for Morpheus II with a 12-round Simon [58] cipher. We used the VCU118 FPGA for our internal performance and security evaluation. In addition, in the summer of 2020, the Morpheus II architecture was entered into the DARPA FETT Challenge [2], along with six additional secure CPU designs. The security firm Synack ran the challenge with their crowd-sourced security researchers providing red-team testing. The computer science research firm Galois adapted FireSim [60] for the test harness, which allowed the Morpheus II design to be deployed into an Amazon AWS F1 cloud instance. When a researcher wanted to attack one of our machines, they would spin up an AWS F1 instance with FireSim, with the core replaced with a DARPA SSITH secure core. All teams in the DARPA FETT Challenge were required to provide a challenge application with known software vulnerabilities. We built the Mock Medical Database (M3DB) which was a mock COVID-19 patient database for medical research, with a network-facing REST interface for querying the patient data in a differentially private manner. Attackers had to penetrate this database and modify or exfiltrate patient data to successfully attack the Morpheus II target.

All teams were requested to provide known software vulnerabilities in the code base. This requirement grew out of the DARPA SSITH program, which was focused on hardware techniques for protecting vulnerable software. In addition, each team had to make their build environment available to the researchers, so they could install and build their own code for testing and study purposes. Each team also had to build a mission list, which was a step-by-step guide detailing easy-to-hard attack scenarios. Bug bounty payouts were gauged to the mission difficulty. DARPA paid bug bounties for finding vulnerabilities, which were never disclosed, but DARPA did say publicly that the bounties went at least up to US $50k.

For our FETT deployment, attackers were given full knowledge of our software stack containing vulnerabilities that consisted of FreeRTOS [59], device drivers, and the SQLite database used for M3DB. While attackers were not given the deployed binary, they were provided with a full toolchain to produce Morpheus II-enabled binaries with a different boot-time key than the deployed system. Attackers had access to a web server running a REST interface to query M3DB, but they were not able to access the UART or the JTAG to run a debugger.

VI. Security Analysis

In this section, we give details of the DARPA FETT security evaluation and recount additional penetration testing we performed on Morpheus II. Finally, we discuss to what extent Morpheus II could stop other classes of attacks.

A. Morpheus II Bug Bounty

During the three months of the FETT challenge, the Morpheus II target had 535 researchers trying to penetrate it. Despite being the second most attacked target in the FETT program, the ***Morpheus II target was never penetrated and no exploitable vulnerabilities were discovered***. By the end of the program, DARPA disclosed that 10 vulnerabilities were discovered by the researchers (on targets other than Morpheus II). Given the deployment setup described in Section V, attackers were expected to analyze the FreeRTOS and SQLite code-base to find vulnerabilites, then exploit them by sending maliciously crafted requests over the M3DB REST interface. Unfortunately, the terms of the bug bounty program did not permit us to interact with our attackers to learn what barriers they were encountering. However we expect that attempts to inject code were thwarted by code encryption and ROP-style attacks were stopped with code pointer encryption.

B. BlindROP Analysis

For further security analysis, we performed our own penetration testing by exploring the utility of Blind

CWE Class	Attack Scenario	Result	Defense Details
Buffer Overflow	CWE-121: Stack-based buffer overflow	**Stopped**	Injected return address not encrypted
Permission, Privileges, & Access Control	CWE-257: Storing Passwords in a Recoverable Format	**Successful**	Requires password data to be encrypted
Resource Management	CWE-416: Use After Free	**Stopped**	Injected code pointer not encrypted
Code Injection	CWE-94: Improper Control of Generation of Code	**Stopped**	Injected return address and code not encrypted
Information Leakage	CWE-209: Information Exposure through an Error Message	**Successful**	Disclosed pointer is an unencrypted data pointer
Numeric Errors	CWE-190: Integer Overflow or Wraparound	**Stopped**	Injected return address and code not encrypted

Table III: Set of CWE attacks evaluated on Morpheus II.

Return Oriented Programming (BROP) [61] attacks on the M3DB application. The BROP approach is suitable for attacking closed-source proprietary services with non-public binaries. BROP requires a stack overflow vulnerability and a service which restarts after a crash (for example, a web server or a database application), both of which are represented in the Morpheus II-protected M3DB target application. The attack works by first breaking Address Space Layout Randomization (ASLR) and then finding remote gadgets to call a `write()` function, which leaks the binary from the memory to the attacker over the network. The attacker then performs a conventional ROP attack using knowledge of the leaked binary. [61] provides further details about the various gadgets required for a successful BROP attack.

For the baseline system without Morpheus II protections, scanning for a particular ROP gadget from known start of code range until known start of stack takes about 57,500 tries. The attacker runs `curl` to send address that can overflow on stack. Given how the application is designed, it takes the attacker 1.29 seconds to determine that the attempt was unsuccessful. Moreover, each unsuccessful attempt causes a soft reboot which has a latency of 13.6s for the 50 MHz frequency of the platform. It would thus take almost 10 days to probe just for a single ROP gadget. Note that the M3DB web server supports 20 requests in parallel, but unsuccessful requests must be tried sequentially while the application performs a soft reboot. After guessing the correct address, the attacker successfully executes the gadget.

Once Morpheus II protections are turned on, the attacker must guess the encrypted address of any and all gadgets required to break ASLR, execute `write`, and then execute ROP from the full 32-bit address range. Since the encryption key changes at warm boots (rerandomizing all key assets which BROP expects to be static across reboots [61]), each attempt at an BROP will be a pure random guess into the 32-bit address space. Given that the expected number of trials (of a Bernoulli process) for a domain of size N is N trials, a successful BROP attack will require approximately 4 billion attempts *for each gadget*. With each attempt taking 14.89 seconds, one would expect an expected latency for this attack of nearly 2000 years! In addition, any attempts which crash the system would need to restart the entire probe process, and wait for the M3DB application to restart, further delaying the attempted hack.

C. Bare-metal CWE Evaluation

In addition to attacking the M3DB application, we wrote a set of bare-metal tests described in Table III that implement attacks from several CWE classes. Morpheus II's code and code pointer encryption were able to sucessfully thwart all except for privilege escalation and information leakage, which require data and data pointer defenses.

D. Qualitative Security Analysis

Next, we describe additional modern attacks and how Morpheus II would prevent them. Counterfit Object-Oriented Programming (COOP) [62] is an advanced code-reuse attack that relies on chaining gadgets from the C++ virtual function table. Morpheus II stops COOP by preventing injection of counterfeit objects (code encryption) and requiring the attacker to forge encrypted pointers for the addresses of gadgets in existing objects (code pointer encryption).

Since brute-force attacks on Morpheus II cryptography are infeasible, attackers must resort to side-channels to leak information. Encryption keys are stored in the hardware, and only utilized during the fixed-latency encryption and decryption within the pipeline. In addition, micro-architectural structures like caches never hold plaintext pointer values as they never leave the pipeline unencrypted. Similarly, cold-boot attacks on DRAM or disk are unsuccessful as pointer values are fully encrypted throughout the system.

JIT ROP attacks [21, 22] utilize gadgets generated by the just-in-time compiler based on attacker controlled inputs. If Morpheus II was extended to generate encrypted JIT code, an attacker would not be able to disclose

Database Query	Average Execution Time (seconds)			95% Confidence Intervals	
	Baseline	Morpheus II	Δ	Baseline	Morpheus II
HELP Page	0.0244	0.0254	0.0010	0.1047	0.161
avg(Recovered)	1.5293	1.5648	0.0355	0.0248	0.0728
avg(TestPositive)	1.5310	1.5504	0.0194	0.0551	0.0371
avg(RecentTravel)	1.5382	1.5281	-0.0101	0.0265	0.0286
avg(RecentTravel) where (gender==M")	1.4663	1.5158	0.0495	0.0680	0.0449
avg(RecentTravel) where (zipcode==48105 && gender==\"M\"&& reqvent==1)	1.0349	1.0662	0.0313	0.0042	0.0305

Table IV: Morpheus II execution time (in seconds) for sample database queries compared to Baseline.

the plaintext code needed to mount the attack in [21]. While [22] doesn't rely on disclosing plaintext code, the encrypted code pointers in Morpheus II would prevent a successful attack.

VII. Power, Performance, and S/W Impacts

FPGA LUTs	1.29%
FPGA Regs	0.06%
Estimated Power	0.21%
Max Frequency	125MHz
LLVM Modifications	1K SLOCs
Rocket Chisel Modifications	369 SLOCs
Software Stack Modifications	3 SLOCs

Table V: Morpheus II Overhead Summary

Benchmark	Morpheus II Slowdown
Coremark (2000 Iterations)	0.045%
adpcm decode	-0.0228%
adpcm encode	-0.0239%
aes	-0.0018%
basicmath	-0.0009%
blowfish	-0.0003%
crc	-0.0089%
fft	-0.0098%
limits	0.0587%
qsort	-0.0046%
randmath	0.0016%
rc4	-0.0042%
MiBench Avg	-0.0022%

Table VI: Morpheus II Slowdown Compared to Baseline

The power increase due to the extra logic and ciphers was only 0.21% for the entire Rocket Core design, including the DRAM controller and the XDMA PCIE bus controller. Area overheads were uniformly low, at only a 1.29% increase in LUT (logic) resources and 0.06% increase in registers. The baseline Rocket system provided for the DARPA FETT Challenge ran at 50MHz. We successfully stress tested the frequency of Morpheus II and the baseline Rocket to 125MHz before timing began to fail at 150MHz in the debug module, confirming that our pipeline changes were not on the critical path.

We evaluated Morpheus II on a suite of benchmarks including Coremark [66], MiBench [67], and our deployed M3DB application. The baseline Rocket-based system was able to compile and run a subset of benchmarks from MiBench 2: adpcm decode, adpcm encode, aes, basicmath, blowfish, crc, fft, limits, qsort, randmath, and rc4. As described in Section IV, no additional pipeline stages were added to implement code and code pointer encryption. Since the code decryption occurs between the instruction cache and the fetch queue, there is no overhead associated with our code protections. One overhead for code pointer encryption is an additional instruction to encrypt each function pointer when it is initialized. Coremark and MiBench do not contain any function pointers, while the M3DB application has around 27K. Section VI-C shows the execution time for a set of example queries on the M3DB application. We see that any Morpheus overheads are within the run-to-run variation due to the network stack and I/O. Table VI breaks down the overhead of Coremark and MiBench, showing the overhead is essentially zero since there are no function pointers.

Software and design impacts were also low. The changes necessary to LLVM to support Morpheus II compilation were less than 1k lines of code. In addition, the changes to the Chisel code to accommodate the Morpheus II extensions on the Rocket Core totaled only 369 additional lines, including the cipher engine. Finally, few software changes were required in the software running on the Morpheus II system for the FETT Challenge. Our platform, described in Section V, was running the Mock Medical Database (M3DB) running on a FreeRTOS web server with SQLite database, totaling more than 200K lines of code. To accommodate Morpheus II defenses, only three lines of code in the FreeRTOS assembly files needed to be changed.

VIII. Discussion

Morpheus II denies attackers the ability to forge or analyze code and pointers using always-on encryption, forcing attackers to use stochastic methods. By leaning into strong cryptography and physical isolation, Mor-

System	Real Hardware	Overheads	Area	Power	Software Mods	Protections	Randomization rate
Shuffler[53]	Not required	14.9%	N/A	N/A	Recompile	Code pointer (relative and absolute)	50ms
Morpheus[1]	Simulation	0.9%	N/A	N/A	Recompile, manual porting	User Code, code/data pointer (relative and absolute)	50ms
PNS[54]	Yes	6%	2%	N/A	DBI/Recompile	Code pointer (absolute)	10ns
MVU[63]	No	0.034%	N/A	N/A	None	Code pointer (absolute)	N/A
ASIST[45, 49]	Yes	<3%	10.6% Reg, 8.3% LUT (AES)	N/A	None	User and kernel Code, return addresses	Creation, exception[49]
PolyGlot[48]	Yes	<5%	72% LUT	N/A	None	User and Kernel Code, return	N/A
Isomeron[64]	Not required	19%	N/A	N/A	DBI	User Code pointer (absolute)	1ms
PointGuard[42]	Not required	10%	N/A	N/A	Recompile	Code/data pointer (absolute)	N/A
ZeRØ[65]	Cache-only	Negligible	5.41%	3.37%	DBI/Recompile	Code/data pointer (absolute)	N/A
Morpheus II	Yes (deployed)	Negligible	0.06% Reg, 1.29% LUT	0.2%	Recompile	User and Kernel Code, code pointer (absolute)	Creation, exception

Table VII: Comparison of control-flow protection systems. Morpheus II has minimal area cost and the overhead for deployed benchmarks on FreeRTOS was not perceivable within run-to-run variation.

pheus II provides durable security mechanisms. Our FPGA deployment using 12 rounds of the Simon cipher demonstrates the effectiveness of Morpheus II with almost zero overhead. Further research into hardware efficient ciphers is particularly relevant to scale Morpheus II to a high-frequency ASIC design. While removing data pointer encryption opens up Morpheus II to attacks like data-oriented programming [68], information leaks, and privilege escalations, our prototype is robust enough to withstand a wide range of attacks with negligible overhead.

Table VII compares Morpheus II to other systems that aim to hide or obfuscate attack assets. Morpheus provides the strongest protections, obfuscating both relative and absolute addresses for code and all pointer types, but with significant overheads, non-trivial compiler support and software porting, no kernel protection, and a limited evaluation in simulation. In addition, Morpheus is vulnerable to side-channels that leak unencrypted pointer values residing in the caches, while Morpheus II only operates on unencrypted pointers within the pipeline. Instruction set randomization proposals like ASIST [45, 49] provide similar protections for code injection attacks as Morpheus II's always-on code encryption. The latest ASIST version also provides the same encryption of the link register upon a call and decryption upon return as does Morpheus II, but does not prevent tampering with function pointers. PNS [54] also implements strong-encryption of code pointers like Morpheus II, utilizing special instructions for encryption and decryption. Morpheus II instead implements encrypted variants of existing address generation and jump instructions, removing the extra overhead of instructions specifically for encryption and decryption. However, Morpheus II requires recompilation while PNS can implement partial protections using binary instrumentation. Concurrent work ZeRØ [65] uses special instructions to provide code and data pointer integrity with neglibile performance overhead, but does not protect code and has higher area and power overheads than Morpheus II due to cache modifications to track pointers.

Our current work includes extending Morpheus II to a FreeBSD system where userland code is protected by Morpheus II with minimal modifications to transfer pointers between the kernel and applications. In addition, we are exploring the use of always-on encryption for non-pointer values. This facility could further extend the security scope of Morpheus II, allow the defenses to provide protections for information leakage and privilege escalation attacks.

IX. Conclusion

This paper proposed Morpheus II, a RISC-V processor that prevents control-flow hijacking attacks by encrypting code and code pointers. We built and deployed a FPGA-based system on AWS running a real-time operating system, a full network stack with web server, and a sensitive medical database—all of these components are fully protected by Morpheus II's always-on encryption. Our internal security analysis shows that Morpheus II offers strong protections against a wide-range of attacks such as ROP, BROP, COOP, and JIT-ROP. In addition, over 500 hundred security researchers unsuccessfully attempted to infiltrate the medical database running on Morpheus II over the course of three months. Morpheus II accomplishes this without perceivable performance impact and less than 2% area, all built by a small academic team.

Acknowledgment

This work was supported by DARPA under Contract HR0011-18-C-0019, NSF 1453806, 1704778 and 1817020, and SRC UTA19-001350 and 2019-TS-2965. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA. Technology detailed in this paper has been licensed from the University of Michigan by Agita Labs Inc., an ongoing concern of Todd Austin, Valeria Bertacco, and Alex Kisil.

References

[1] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco,

S. Malik, M. Tiwari, and T. Austin, "Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 469–484.

[2] DARPA. (2020) DARPA FETT bug bounty program. [Online]. Available: https://fett.darpa.mil

[3] G. Shvets. (2018) Enhanced virus protection / execute disable bit. [Online]. Available: http://www.cpu-world.com/Glossary/E/EVP_XD.html

[4] F. Liu, Q. Ge, Y. Yarom, F. McKeen, C. V. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 406–418.

[5] K. T. Ngyuen, *Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family*, 2016.

[6] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1050

[7] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 48–62.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[9] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian, "The gem5 Simulator: Version 20.0+," *arXiv:2007.03152 [cs]*, Sep. 2020.

[10] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The Rocket Chip Generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[11] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 119–129.

[12] CWE - CWE-121: Stack-based buffer overflow (4.4). https://cwe.mitre.org/data/definitions/121.html. (Accessed on 05/13/2021).

[13] CWE - CWE-122: Heap-based buffer overflow (4.4). https://cwe.mitre.org/data/definitions/122.html. (Accessed on 05/13/2021).

[14] CWE - CWE-134: Use of externally-controlled format string (4.4). https://cwe.mitre.org/data/definitions/134.html. (Accessed on 05/13/2021).

[15] CWE - CWE-416: Use after free (4.4). https://cwe.mitre.org/data/definitions/416.html. (Accessed on 05/13/2021).

[16] CWE - CWE-415: Double free (4.4). https://cwe.mitre.org/data/definitions/415.html. (Accessed on 05/13/2021).

[17] A. Peslyak. (1997, Aug.) Bugtraq: Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63. (Accessed on 05/13/2021).

[18] 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*, 2007, pp. 552–561.

[19] 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*, ser. ASIACCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 30–40. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/1966913.1966919

[20] 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*. IEEE, 2015, pp. 745–762.

[21] 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*, May 2013, pp. 574–588.

[22] G. Maisuradze, M. Backes, and C. Rossow, "What Cannot Be Read, Cannot Be Leveraged? Revisiting Assumptions of JIT-ROP Defenses," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 139–156.

[23] J. Woodruff, R. N. 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 *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468.

[24] lowRISC Project. (2018) lowrisc. [Online]. Available: https://lowrisc.org

[25] G. T. Sullivan, A. DeHon, S. Milburn, E. Boling, M. Ciaffi, J. Rosenberg, and A. Sutherland, "The Dover inherently secure processor," in *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, Apr. 2017, pp. 1–5.

[26] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., 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*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 487–502.

[27] 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*, vol. 2, no. 2, pp. 28:1–28:30, Jun. 2018.

[28] K. Sinha and S. Sethumadhavan, "Practical memory safety with rest," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. IEEE Press, 2018, p. 600–611. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00056

[29] 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*,

ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 558–571. [Online]. Available: https://doi.org/10.1145/3352460.3358299

[30] Microsoft. Data execution prevention - win32 apps | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. (Accessed on 05/14/2021).

[31] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. USA: IEEE Computer Society, 2013, p. 559–573. [Online]. Available: https://doi.org/10.1109/SP.2013.44

[32] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 941–955.

[33] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha, "Hardware-assisted run-time monitoring for secure program execution on embedded processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 14, no. 12, p. 1295–1308, Dec. 2006. [Online]. Available: https://doi.org/10.1109/TVLSI.2006.887799

[34] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[35] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE Press, 2016, p. 1–6. [Online]. Available: https://doi.org/10.1145/2897937.2898098

[36] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 38–49. [Online]. Available: https://doi.org/10.1145/2857705.2857722

[37] 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*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/3337167.3337175

[38] ARM. (2018, Sept.) Arm architecture Armv8.5-A announcement - branch target indicators (BTI) - Arm community. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/arm-a-profile-architecture-2018-developments-armv85a. (Accessed on 05/14/2021).

[39] 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*, ser. OSDI'14. USA: USENIX Association, 2014, pp. 147–163.

[40] 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*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 941–951. [Online]. Available: https://doi-org.ezproxy.lib.utexas.edu/10.1145/2810103.2813676

[41] Q. ProductSecurity. (2017, Jan.) Pointer authentication on ARMv8.3: Design and analysis of the new software security instructions. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[42] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard™: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[43] N. Tuck, B. Calder, and G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, Dec. 2004, pp. 209–220.

[44] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03. Washington D.C., USA: Association for Computing Machinery, Oct. 2003, pp. 272–280.

[45] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis, "ASIST: Architectural support for instruction set randomization," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. Berlin, Germany: Association for Computing Machinery, Nov. 2013, pp. 981–992.

[46] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. Austin, Texas, USA: Association for Computing Machinery, Dec. 2010, pp. 41–48.

[47] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The effectiveness of instruction set randomization," in *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, P. D. McDaniel, Ed. USENIX Association, 2005.

[48] K. Sinha, V. P. Kemerlis, and S. Sethumadhavan, "Reviving instruction set randomization," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2017, pp. 21–28.

[49] G. Christou, G. Vasiliadis, V. Papaefstathiou, A. Papadogiannakis, and S. Ioannidis, "On Architectural Support for Instruction Set Randomization," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 4, pp. 1–26, Dec. 2020.

[50] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 158–168, Jun. 2006.

[51] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. USA: USENIX Association, 2006.

[52] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, "Security through redundant data diversity," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Jun. 2008, pp. 187–196.

[53] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 367–382.

[54] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, V. P. Kemerlis,

and S. Sethumadhavan, "Using Name Confusion to Enhance Security," *arXiv:1911.02038 [cs]*, Aug. 2020.
[55] MITRE, "2020 CWE top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25, 2020.
[56] DARPA. (2017) System security integration through hardware and firmware (SSITH). [Online]. Available: https://www.darpa.mil/program/ssith
[57] Hafnium targeting exchange servers with 0-day exploits. [Online]. Available: https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers/
[58] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "SIMON and SPECK: Block Ciphers for the Internet of Things," Tech. Rep. 585, 2015.
[59] R. Barry *et al.* (2008) FreeRTOS. [Online]. Available: https://freertos.org
[60] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanović, "Firesim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Los Angeles, California: IEEE Press, Jun. 2018, pp. 29–42.
[61] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 227–242.
[62] 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*, May 2015, pp. 745–762.
[63] Y. Lee and G. Lee, "Detecting Code Reuse Attacks with Branch Prediction," *Computer*, vol. 51, no. 4, pp. 40–47, Apr. 2018.
[64] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.
[65] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, and S. Sethumadhavan, "ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks," in *Proceedings of the 48th International Symposium on Computer Architecture*, ser. ISCA '21. Virtual Event: IEEE Press, Jun. 2021.
[66] P. K. Krause, "Stdcbench: A benchmark for small systems," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 43–46. [Online]. Available: https://doi.org/10.1145/3207719.3207726
[67] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. USA: IEEE Computer Society, 2001, p. 3–14.
[68] S. Chen, J. Xu, and E. C. Sezer, "Non-Control-Data Attacks Are Realistic Threats," in *14th USENIX Security Symposium (USENIX Security 05)*, 2005.