Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn

Mark Gallagher
University of Michigan

Lauren Biernacki
University of Michigan

Shibo Chen
University of Michigan

Zelalem Birhanu Aweke
University of Michigan

Salessawi Ferede Yitbarek
University of Michigan

Misiker Tadesse Aga
University of Michigan

Austin Harris
University of Texas at Austin

Zhixing Xu
Princeton University

Baris Kasikci
University of Michigan

Valeria Bertacco
University of Michigan

Sharad Malik
Princeton University

Mohit Tiwari
University of Texas at Austin

Todd Austin
University of Michigan

Abstract

Attacks often succeed by abusing the gap between program and machine-level semantics– for example, by locating a sensitive pointer, exploiting a bug to overwrite this sensitive data, and hijacking the victim program's execution. In this work, we take secure system design on the offensive by continuously obfuscating information that attackers need but normal programs do not use, such as representation of code and pointers or the exact location of code and data. Our secure hardware architecture, Morpheus, combines two powerful protections: ensembles of moving target defenses and churn. Ensembles of moving target defenses randomize key program values (*e.g.*, relocating pointers and encrypting code and pointers) which forces attackers to extensively probe the system prior to an attack. To ensure attack probes fail, the architecture incorporates churn to transparently re-randomize program values underneath the running system. With frequent churn, systems quickly become impractically difficult to penetrate.

We demonstrate Morpheus through a RISC-V-based prototype designed to stop control-flow attacks. Each moving target defense in Morpheus uses hardware support to individually offer more randomness at a lower cost than previous techniques. When ensembled with churn, Morpheus defenses offer strong protection against control-flow attacks, with our security testing and performance studies revealing: *i)* high-coverage protection for a broad array of control-flow attacks, including protections for advanced attacks and an attack disclosed after the design of Morpheus, and *ii)* negligible performance impacts (1%) with churn periods up to 50 ms, which our study estimates to be at least 5000x faster than the time necessary to possibly penetrate Morpheus.

CCS Concepts • **Security and privacy → Systems security**; *Hardware-based security protocols*; Tamper-proof and tamper-resistant designs; • **Computer systems organization** → *Architectures.*

Keywords moving target defense; runtime randomization

ACM Reference Format:
Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3297858.3304037

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304037

1 Introduction

Building vulnerability-free, complex systems is challenging and further complicated by legacy code and changing threat models. Control-flow exploits are still prevalent [21] despite several layers of defenses based on safe pointers [26], control-

and data-flow integrity [1, 16, 43], limiting the use of data as code (NX-bit) [67], and address space layout randomization (ASLR) [54]. A similar arms race is being played out in the microarchitecture, where side-channels are abusing speculation [42] and memory characteristics [56] to circumvent architectural isolation guarantees. Current approaches primarily address security as a memory safety or speculative execution problem. In contrast, we propose to attack the attacker and systematically thwart their efforts by making vulnerable systems exceedingly hard to exploit.

We introduce a secure system design approach called ***ensembles of moving target defenses (EMTDs) with churn***. We assume that the system contains vulnerabilities, and attackers are attempting to penetrate system security via those vulnerabilities. A large number of these attacks exploit execution-level semantics (*e.g.*, code location and pointer values) that are undefined at the language level and to which normal programs remain agnostic. Furthermore, over time, sophisticated attacks require progressively more execution-level information as they attempt to evade successive layers of defenses such as NX-bit, ASLR, and control-flow integrity. Hence, we propose to systematically randomize all undefined semantics that are critical to craft successful attacks–using hardware to enable an *ensemble* of such moving target defenses to be deployed concurrently. Additionally, we introduce *churn*, a mechanism that transparently re-randomizes the moving target program values while the system is in operation. By limiting our defenses to values associated with undefined program semantics, we construct a secure system that is transparent to normal programs but intentionally hostile to malicious programs. Crucially, even though concrete exploits and vulnerabilities grow rapidly, *security-relevant undefined semantics do not*. Thus, EMTDs with churn provide a more durable and proactive layer of defense than patching specific vulnerabilities.

To demonstrate our security claim, we present ***Morpheus***, a RISC-V-based system that incorporates EMTDs with churn to thwart control-flow attacks. Control-flow protection represents a critical step in the design of secure systems as control-flow attacks have consistently been the most common vulnerability tracked by the CVE database since its inception in 1999 [21, 49]. Our secure architecture protects critical information by using two novel moving target defenses: *i)* ***pointer displacement***, which randomly and independently places code and data in the address space, and *ii)* ***domain encryption***, which randomizes the representation of code, code pointers, and data pointers using strong encryption. These new defenses provide a significantly larger randomization space at lower overheads than previous techniques. Specifically, Morpheus defenses have a randomization space of 504 bits (referred to as "bits of entropy" for the remainder of this paper), leading to 2^{504} possible key configurations.

Once deployed, EMTDs force attackers to extensively probe the system to discover the randomized values needed for an attack. Morpheus sabotages the attacker's ability to probe the system and gain critical values by using churn to re-randomize these values at runtime. To further strengthen the value of churn, Morpheus incorporates an ***attack detector*** to sense when an attack is ongoing and immediately trigger a churn cycle to halt the attack.

To get a sense of how Morpheus works, consider the code reuse attack shown in Figure 1. This attack exploits a buffer overflow vulnerability to invoke `target()` when `vulnerable()` returns. The function `main()` is malicious because the string passed to `strcpy()` is too long for the array `buf[]`, allowing the string to overwrite the return address with the entry-point of `target()`. With Morpheus defenses, two complications arise for the attacker: *i)* 60-bits of the address of `target()` have been randomized by the pointer displacement defense, and *ii)* the representation of code pointers (expressed as "`\xf0\x03\x02\x01`") is now encrypted with a 128-bit key, making the pointer in the string incorrectly encoded as plaintext. To overcome these challenges, the attacker must either brute-force guess information or probe Morpheus to discover *i)* the location of `target()` and *ii)* the representation of code pointers. We show in Section 5.2 that circumventing Morpheus protections could take several minutes or more with the most advanced probes. Unfortunately for the attacker, the churn mechanism will re-randomize the address of `target()` and its pointer representation within 50 ms. Even worse, if the attacker's probes trigger the attack detector, a new churn cycle is immediately invoked.

1.1 Contributions of This Work

We build on the successes moving target defenses have shown for network and software security [38]. Specifically:

- We introduce *ensembles* of moving target defenses (EMTDs) with *churn*. EMTDs randomize undefined values crucial to perpetrating security attacks (*e.g.*, code and pointers). Churn re-randomizes the protected values to defeat probing while being transparent to normal programs. ***This paper advances the security of state-of-the-art moving target defenses (e.g., ASLR and instruction-set randomization), and introduces hardware-based ensembling with churn to create a new class of vulnerability-tolerant secure systems.***
- We present the ***Morpheus*** architecture – a RISC-V-based processor that is designed to stop control-flow attacks using EMTDs with churn. Using compiler, architecture, and runtime support, we implement EMTDs with support for attack-detector-driven churn periods. Our moving target defenses for code, code pointers, and data pointers, termed pointer displacement and domain encryption, create significant barriers to control-flow attacks.

- We show that Morpheus, with 504 bits of entropy and 50 ms churn period, has a 1% average slowdown (7% worst case) for SPEC'06 and MiBench benchmarks, despite re-randomizing key program values 5000x faster than estimated attack times. In addition, we show that Morpheus provides strong security against control-flow attacks, stopping a broad array of advanced attacks including return-oriented programming [62] and the Back-Call-Site attack [75], which was disclosed after the design of Morpheus. ***These analyses demonstrate that the Morpheus architecture is an effective defense against known and potentially future control-flow attacks.***

The remainder of this paper is organized as follows: Section 2 motivates our vulnerability-tolerant design approach, Section 3 presents our threat model and Section 4 details the Morpheus architecture. In Section 5, we present security and performance analyses of Morpheus, examining its ability to stop control-flow attacks and at what cost. Finally, Section 6 examines related work before concluding.

2 Vulnerability-Tolerant Secure Architectures

Our goal of building a vulnerability-agnostic secure system is achieved by designing a computing platform that is transparent to normal programs but hostile to malicious ones. To illustrate our approach, consider the malicious program shown in Figure 1, which exploits a buffer overflow vulnerability in `vulnerable()` to perform a code reuse attack.

2.1 Malicious Programs are from Mars; Normal Programs are from Venus

In Figure 1, one might limit the difference between a malicious and non-malicious use of `vulnerable()` to the observation that a malicious program overflows `buf[]`, and a non-malicious program (when the input string is length 4 or less) does not. However, if one takes a slightly more nuanced view, more distinctions exist. ***We observe that normal programs utilize defined program-level semantics, while malicious programs lean heavily on undefined semantics.***

Undefined semantics are execution-level semantics that are not explicitly documented, often because they are properties of the underlying implementation. Examples of undefined semantics include out-of-bounds array accesses, uninitialized variable values, execution timing, microarchitectural sharing characteristics, *etc.* While savvy programmers know these values exist and often are static, they should not build a program that relies on undefined semantics, since these values can easily change from one build of the program to the next or from one CPU architecture to another.

In contrast, malicious programs routinely utilize undefined semantics to subvert security measures. For example,

```
void target() {
  printf("You overflowed successfully, gg");
  exit(0);
}
void vulnerable(char* str1) {
  char buf[5];
  strcpy(buf, str1);
}
int main() {
  vulnerable("fffffffffffffff\xf0\x03\x02\x01");
  printf("This only prints in normal control flow");
}
```

Figure 1. Example Attack Code. This code exploits a buffer overflow vulnerability in `vulnerable()` to overwrite a return address. We note that the attack code utilizes undefined semantics associated with array overflows, stack frame organization, and code addresses. Morpheus randomizes values associated with undefined semantics to thwart malicious attacks while remaining benign to normal programs.

the code in Figure 1 copies memory past the end of `buf[]` (*undefined*: out-of-bounds array access), which overwrites the return address (*undefined*: return address location) with the value of `0x010203f0` (*undefined*: address of a code object). We are not the first to make the observation that attacks rely heavily on undefined semantics; debugging tools have long focused on finding undefined semantics (*e.g.*, UBSan [4] or STACK [80, 81]).

Indeed, programs can utilize undefined semantics without malice, as we have seen in some isolated cases in systems code and buggy code. For the former case, we have accommodated these operations (*e.g.*, pointer alignment) in the design of Morpheus defenses. In the latter case, the program needs to be debugged. After building nearly 1 million lines of code to run on Morpheus with no source-level changes required (including all of SPEC's C-based benchmarks), we have found that we can readily accommodate these isolated cases without making demands on developers and without taking the heat off malicious programs.

2.2 Boosting Uncertainty with Moving Target Defenses

Given that malicious programs use undefined semantics and normal programs avoid them, an attractive approach to vulnerability-tolerant secure design is to keep well-defined semantics as such and randomize undefined semantics each time a program is executed. The use of randomization on undefined semantics as a defensive measure is an example of a ***moving target defense*** [38]. Moving target defenses deliberately introduce change into a system, which in turn increases the attacker's uncertainty of the system's state. By randomizing the undefined semantics of a program each time it runs, the attacks built for a system with static undefined semantics will no longer work as key attack values have become randomized. As shown in Figure 2, ***moving target***

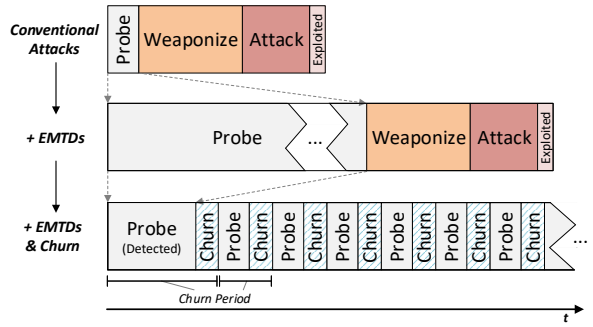

Figure 2. Thwarting Attacks via EMTDs with Churn. Exploits progress with the following phases: *probe*, *weaponize*, and *attack*. Ensembles of moving target defenses (EMTDs) increase uncertainty in key values needed to weaponize attacks, thus, probe times increase significantly. To ensure the patient attacker does not complete their probes, churn re-randomizes key program values at regular intervals. An attack detector senses when probe-like activity is occurring and reduces the churn period to strengthen defenses.

defenses force the attacker to probe the system for the unknown values needed to implement an attack. Probing involves conducting an experiment where the outcome yields some information about the uncertain value.

For example, if an attacker must contend with a moving target defense applied to the location of code in Figure 1, they must first probe the system to determine the location of the function `target()`. The attacker could (naively) introduce a memory scanner that searches the 48-bit address space for the code of function `target()`. With a mountain of patience, the scanner will eventually locate `target()` and then the corrected string can be injected into the call to `strcpy()`.

Fortunately, we have a prime example of how the attacker community responds to moving target defenses with address space layout randomization (ALSR) [54]. ASLR randomizes the location of the code, heap, and stack each time a program runs, which has the effect of applying a moving target defense to code/data locations and pointer values. Today, ASLR is widely deployed in Windows, Linux, iOS, and Android. Table 1 lists ASLR defenses and the subsequent probes attackers developed to overcome the uncertainty introduced by ASLR defenses. The table also shows the entropy of defenses (or number of bits recovered) and estimated time for an attack. While one-time moving target defenses like ASLR are very powerful stumbling blocks for attackers, today they form only temporary barriers since, with enough time and ingenuity, attackers have overcome all variants of ASLR. For example, the AnC attack is able to recover a full 48-bit x86_64 virtual address in about 150 seconds [30].

Table 1 also shows that attacking moving target defenses with high entropy requires the use of either *i)* more advanced probing techniques or *ii)* significant probe time. We would similarly expect increased probe times for a defense introduced in this work, shown in the last line of the table, which has significantly higher entropy than current ASLR implementations. Given this trend, our proposed design technique leverages high entropy to guarantee long probe times, and re-randomization (detailed in Section 2.3) to assure that attack probes fail.

Table 1. ASLR Defenses and Successive Attacks. This table (ordered by release date) lists ASLR defenses and attack probes used to recover randomized addresses for that defense. It is evident that strong moving target defenses (*e.g.*, 64-bit ASLR) result in long probe times.

Type	Name	Entropy / Bits Recovered	Time (s)
Defense	32-bit PaX ASLR [54]	16 bits	n/a
- Attack	Blind calls (Pfaff) [66]	16 bits	216
Defense	64-bit ASLR	30 bits	n/a
- Attack	BROP [11]	30 bits	1,200
- Attack	CROP [29]	30 bits	14,580
- Attack	Dedup Est Machina [13]	30 bits	1,800
- Attack	JUMP over ASLR [28]	9 bits	0.06
- Attack	AnC [30]	30 bits	150
Defense	Morpheus Pointer Displacement	60 bits	n/a

2.3 Ensembles of Moving Target Defenses with Churn

In this work, we seek to elevate moving target defenses by combining them with a runtime re-randomization technology called churn. EMTDs alone randomize program values attackers need, boosting uncertainty. It is conceivable that, with enough time, the attacker may eventually succeed in probing the system for information needed to launch an attack. Thus, we introduce churn as a mechanism to thwart even the most patient attackers. As shown in Figure 2, ***churn re-randomizes the values that attackers need to craft successful attacks.*** With fast churn periods, attack probes fail to discern the unknown values for which they search.

With moving targets deployed under the code in Figure 1, an attacker would need to search for the function `target()`. With churn, the function `target()` repeatedly moves in the address space as the search progresses. Thus, the probe will only succeed on the infinitesimal chance that `target()` moves to the immediate vicinity of the probe's search.

To create an even more hostile environment for attackers, churn incorporates a reactionary component. Not only can churn cycles be initiated periodically, but they can also be triggered in response to a potential attack. As illustrated by Figure 2, when an attack probe attempt is detected an additional churn cycle is initiated, severely limiting the time available for a successful probe. Furthermore, repeated probe attempts result in continuous churn, making it incredibly more difficult for attackers. More details on this attack detector are found in Section 4.1.

3 Threat Model

Morpheus is designed to mitigate control-flow attacks, many of which utilize memory exploits [73]. Specifically, we seek to thwart attacks where a trusted but vulnerable victim program processes untrusted inputs that exploit memory errors to hijack control-flow of the victim. These attacks may make use of buffer overflows, format-string, heap-spray, double-free, return/jump-oriented programming, and other execution-level semantics to override language-level protections and hijack control flow. Denial-of-service (DoS) and side-channel attacks, while good future targets for Morpheus, are outside the scope of this work.

We assume the following about the attacker: *i)* they cannot physically threaten the system via power analysis, fault injection, *etc.*; *ii)* they cannot manipulate the system's boot sequence or anticipate the output of the random number generator; *iii)* they interact with the system via an interface such as the network or keyboard; *iv)* they cannot modify the original binary; and *v)* they are able to locate a memory corruption or disclosure vulnerability in the target program to exploit. The trusted computing base (TCB) includes the specialized Morpheus hardware, as well as some software support from the loader and OS scheduler.

4 The Morpheus Secure Architecture

As shown in Figure 3, Morpheus is a 64-bit RISC-V-based [60] secure architecture that uses EMTDs with churn to thwart control-flow attacks. Morpheus deploys moving target defenses to randomize key values needed for these attacks: *i)* code, *ii)* code pointers, and *iii)* data pointers. These domains can be aggressively churned without breaking normal programs in the architecture.

Morpheus' moving target defenses rely on the ***domain tagging*** mechanism (Section 4.1) to precisely track the domain of all memory objects at runtime. Morpheus leverages these tags to implement two moving target defenses – ***pointer displacement*** (Section 4.2), which obscures pointer values by adding a random displacement to them by domain, and ***domain encryption*** (Section 4.3), which encrypts all domains in the program under their own keys. Both defenses can be re-randomized at runtime by the ***churn unit*** (Section 4.4). To do this efficiently, the churn unit updates the necessary values while program execution continues. Additionally, Morpheus includes an ***attack detector*** (Section 4.1) to sense when a potential attack is in progress and ramp up the churn rate to strongly repel the attack.

4.1 Precise Runtime Domain Tagging

The domain tagging infrastructure is at the center of the Morpheus architecture and serves to precisely track the domain of each memory object during execution. Domain tags are used by the churn unit to correctly re-randomize values at runtime and by the attack detector to identify operations

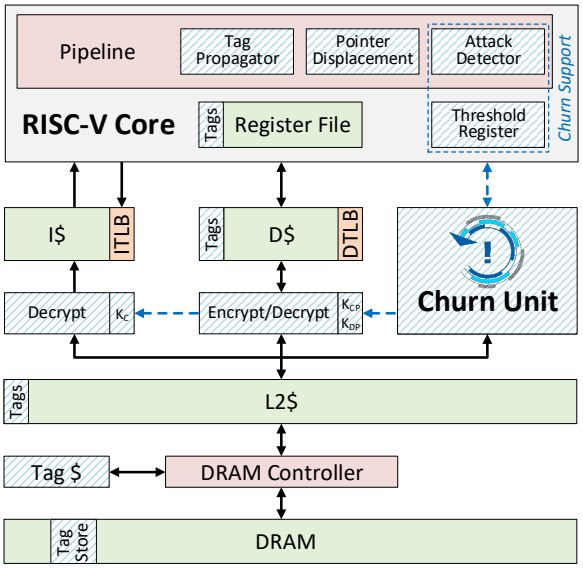

Figure 3. The Morpheus Secure Architecture. Morpheus implements EMTD protections with churn to stop control-flow attacks. Components hashed with diagonal lines augment the baseline RISC-V system to support Morpheus defenses. The dotted line is a bus used for churn control signals and transmitting keys. Churn Support includes the Attack Detector and Threshold Register, while logic for propagating tags and translating pointers is added to the pipeline.

indicative of undefined semantics. Morpheus tracks four distinct domains using 2-bit domain tags: code (C), code pointers (CP), data pointers (DP), and other data (D). The pipeline is responsible for propagating tags; specifically, domain tags are fetched into the pipeline and used to compute the tag of the output value for every instruction. Initial tag values come from the compiler, while the microarchitecture is augmented to support tag storage.

The Morpheus LLVM-based [44] compiler extensions provide the locations of code and pointers in the executable to enable precise tracking of domains at runtime. The input to the compiler is unmodified C source files, which are converted into the LLVM intermediate representation by the Clang front-end. Then, a global variable domain analysis labels each memory object in statically initialized data sections as data, a code pointer, or a data pointer. Subsequently, an instruction labeling pass in the Clang back-end identifies and labels instructions that initialize dynamically created memory objects (*i.e.*, values on the stack, heap, and `.bss` segment). This produces a labeled binary and a domain tag file that contains the initial tags for memory objects.

Storing tags requires modifications to the microarchitecture. All registers are extended to include a 2-bit tag. To reduce the overhead of memory tags, Morpheus only attaches one tag to each 64-bit aligned word, as pointers in the RISC-V RV64 ISA are 64 bits wide. Since RISC-V instructions are 32

Table 2. Attack Detector Logic. ABORT rules monitor particularly grievous operations and trigger an exception that terminates the program. CHURN rules detect undefined behavior that may be indicative of an ongoing attack. In response to these violations, the attack detector initiates a churn cycle. Tags are defined as follows: C = Code, CP = Code Ptr, D = Data, DP = Data Ptr.

	<OP>	Check Condition	Rule
ABORT	Execute	Insn.tag != C	Only execute C
	ANY	R1/R2.tag == C	No C in the pipeline
	JAL(R)	R1.tag != CP	Jump target must be CP
	LD/ST	R1.tag != DP	Address must be a DP
CHURN	COMPARE	R1.tag != R2.tag	No inter-domain compares
	ANY (not JAL(R))	R1.tag == CP	CP arithmetic suspicious
	ANY (not LD/ST)	R2.tag == DP	DP arithmetic suspicious, except add/sub D
	ANY	Overflow Occurs	Overflows are undefined
	SHIFT	Shift > RegWidth	Invalid shift is undefined

bits wide, we augment the linker to minimally NOP pad each object file so that every 64-bit code location contains two 32-bit instructions. The tag information resides in physical DRAM at an offset of $tagstart + phyaddr/32$, where $tagstart$ is the address of the tags in DRAM. Also, by concentrating tags to a fixed location in DRAM, we simplify the churning process as domain types can be efficiently located within the tag store. To improve tag access latency, tags are cached throughout the memory hierarchy. The 64-byte tag groups read from DRAM are cached in a *tag cache*, and all data cache blocks are extended with 2-bits per 64-bit word, to store the additional domain tag bits with each cache block.

Security Implications: The domain tagging's propagation rules enforce closure for pointers under all computation; *i.e.*, all computation with a pointer produces a pointer. Consequently, pointers are always under Morpheus defenses, which protects the system against pointer disclosures.

As shown in prior tagged architectures [24, 45, 71, 72, 84], memory tagging offers security benefits through policy enforcement. Morpheus similarly makes use of tag checking policies to provide additional security guarantees via its ***attack detector***. This detector watches for suspicious operations that are indicative of ongoing attacks and responds in one of two ways: raising an exception or triggering a churn cycle. Morpheus raises security exceptions on a few grievous operations, listed under the "ABORT" rules in Table 2. Four policies are enforced: *i)* only code can be executed, *ii)* code cannot be loaded into a register, thus it can never be read or written, *iii)* only data pointers can be load/store addresses, and *iv)* only code pointers are valid jump targets.

Other runtime-typed architectures often enforce much finer-grained rules, such as not permitting code pointer arithmetic. To ensure minimal impact to programmers, we allow any of these legitimate (albeit suspicious) operations. Rather than raising an exception, the attack detector initiates a churn cycle to destroy any information gained by the attacker. If churn is already ongoing, the system schedules a subsequent churn cycle. Table 2 lists these triggers under the "CHURN" rules. For example, a program is permitted to test individual bits of a pointer (*e.g.*, $p\&(1 << n)$), making it possible to re-encode the pointer as a non-pointer data value. However, these operations will immediately initiate a churn cycle, which has the effect of destroying the utility of any leaked pointer. It is important to note that churn is always occurring at a baseline rate in Morpheus, and the attack detector only serves to *increase* the churn rate when a potential attack is detected.

Non-malicious programs rarely exhibit these behaviors, therefore occasional churn cycles initiated by non-malicious code will result in negligible slowdowns. Since we can deliver increased security without making additional demands on programmers, we have found that our approach is preferred over systems that force programmers to re-write or annotate potentially dangerous code (*e.g.*, CHERI [84]). In addition, if the detector trigger rate rises above a concerning level, the attack detector raises a signal to the OS to indicate that an ongoing attack is likely.

4.2 Pointer Displacement Defense

Virtually all control-flow attacks require knowledge of where memory objects reside. To boost uncertainty in the location of these values, Morpheus utilizes ***pointer displacement*** to create two randomly displaced address spaces (DAS_C and DAS_D) above the virtual address space (VAS). As shown in Figure 4, code is displaced by d_{CODE} bytes in DAS_C, and data is displaced by d_{DATA} bytes in DAS_D. In the programmer's view, DAS_C contains a displaced image of the code objects and DAS_D contains this for data objects, with both supporting wraparound of addresses. This is implemented by incrementing all code pointers by d_{CODE}, and all data pointers by d_{DATA}.

A program sees all pointers as displaced for its lifetime, including pointers in the registers, caches, and memory. Pointer computations (*e.g.*, $p + const$) proceed on displaced pointers as normal without being impacted by the displacement. Morpheus supports a 60-bit random displacement, with the lower 4 bits of the displacement always zero. Keeping these bits zero accommodates codes, such as `memcpy()`, that want to enforce physical memory alignment on pointers. Whenever a translation from DAS→VAS is performed at fetches, loads, and stores, the hardware reads the pointer's tag and subtracts the appropriate displacement (d_{CODE} or d_{DATA}).

To ensure that no exposed latency occurs from displacing the address space, we pipeline the DAS→VAS translation. In the decode stage, the displacement key is subtracted from the load/store offset: $offset - d_{DATA}$. Then in the execute stage, this *delta* is added to the base register to produce the effective address. A similar approach is used for JAL(R)/RET

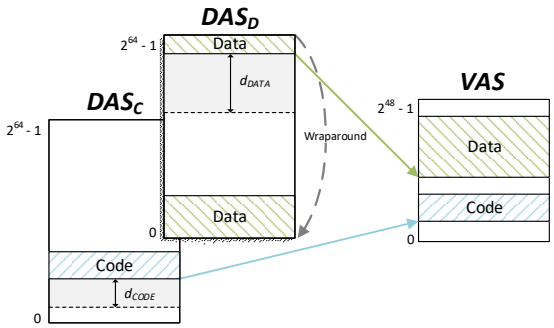

Figure 4. Pointer Displacement. Pointer displacement creates two displaced address spaces: DAS_D and DAS_C, each shifted by a random displacement of d_{DATA} and d_{CODE}, respectively. By having two displaced address spaces, it is possible for each address space to utilize any displacement without colliding with other objects in the other domain, resulting in higher entropy for the code and data segments.

targets, except using the code pointer displacement (d_{CODE}). Using this approach, the addresses presented to the memory system are reformed into their native virtual address counterparts, thus retaining localities and incurring no performance penalties on the memory system.

The operating system is responsible for securely loading and unloading key sets at context switches (as detailed in Section 4.4). If a program utilizes shared memory, then processes/threads sharing memory need only share displacement keys if they also share code or pointers. Typically, this will be required only for tightly threaded programs and shared libraries, since other popular uses of shared memory paradigms such as inter-process communication (IPC) do not share code or pointers. Similarly, forked processes can simply churn at creation time and continue execution with their own private keys. Finally, the kernel possesses the sets of keys for both privileged and user memory, allowing it to access both address spaces without switching key sets.

Security Implications: Displacing pointers delivers a number of powerful security properties. First, the defense permits a full 60-bits of entropy during DAS displacement, which is significantly more entropy than can be produced by an ASLR-based defense [54]. ASLR typically has at most 30-bits of entropy, due to architectures only partially implementing virtual addresses (*e.g.*, RISC-V and x86_64 have 48-bit virtual addresses) and because code and data have to avoid collisions when placed. It is also interesting to note that ASLR and pointer displacement may be complementary: if combined, a displaced pointer disclosure would only reveal the fixed bits in the ASLR randomization algorithm, leaving the ASLR-randomized address bits still unknown.

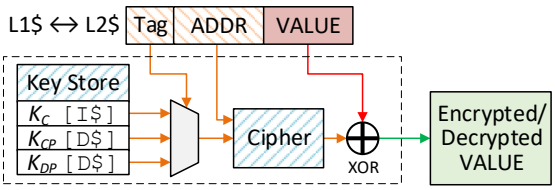

Figure 5. Domain Encryption Defense. Morpheus' encryption makes use of per-domain keys, denoted as K_C, K_{CP}, and K_{DP} for *code, code pointer,* and *data pointer*, respectively. These keys are selected by a value's domain tag and used to protect it in memory. Non-pointer data is unencrypted.

4.3 Domain Encryption Defense

To prevent attackers from inspecting and forging vital program values, the domain encryption defense randomizes the representation of code, code pointers, and data pointers in memory using a strong cipher. These assets are encrypted in memory under their own distinct domain keys. As shown in Figure 5, protected domains are decrypted when memory is read (load or instruction fetch) and encrypted when memory is written (store) between the L1-L2 boundary, keeping the L2 cache and DRAM encrypted. The tag of the accessed value is used to select the appropriate cipher key, either the code key K_C, code pointer key K_{CP}, or data pointer key K_{DP}. The key and the physical address are combined via the cipher to decorrelate memory locations that contain the same value. Encrypting the address allows us to use the cipher in counter-mode, where a *keystream* is generated and XOR'ed with the protected value to encrypt or decrypt it. Note that the address encryption can happen in parallel with the L2 cache access, reducing the performance penalty. A similar technique was used to speed up access to the encrypted instruction cache in the Polyglot architecture [68]. To further reduce the complexity of domain encryption and performance overheads, variable-sized non-pointer data values are not encrypted. Our Morpheus implementation utilizes a strong block cipher introduced for Arm's Pointer Authentication technology [58] called QARMA [5], specifically, $\text{QARMA}_7\text{-64-}\sigma_1$, which encrypts 64-bit blocks with a 128-bit key in 16 rounds.

The three keys used for domain encryption (K_C, K_{CP}, and K_{DP}) are stored in microarchitectural registers that are not accessible by software, so they are private to a process's address space. Multi-processing execution contexts need only share encryption keys if they are sharing code or pointers, otherwise, each process will have its own unique keys. Finally, the kernel will have two sets of keys: one for privileged memory and one for user space.

Security Implications: Attacks trying to forge or leak information will face several obstacles. First, using multiple encrypted domains significantly increases the computational burden of acquiring a broad set of attack information (*e.g.*, code and pointers). Second, code pointer forgery, which is

required by nearly all control-flow attacks, and data pointer forgery, which is used for advanced control-flow mimicry attacks (like DOP [34]), is difficult to accomplish in Morpheus. New pointers can only be derived from knowing an existing, protected pointer value. Since code and pointers are always encrypted outside of the pipeline and L1 caches, any attempt to exfiltrate them by writing them to an I/O location, or via DMA, RDMA, or cold-boot attack (all of which access DRAM or the L2 cache) will result in the capture of a useless encrypted instruction or pointer. Attackers are now limited in finding base pointer values to launch their attacks. While our non-pointer data is stored in plaintext, this does not provide many benefits to CFG attacks since they primarily focus on code and pointers.

Our approach surpasses prior pointer encryption techniques, such as PointGuard [18] or ASIST [53], where assets are encrypted by XOR'ing the value with an in-memory key. Instead, Morpheus uses a strong cipher to encrypt multiple domains within the same program address space. This provides the physical penetration protections of traditional enclaves (e.g. Intel SGX [48], XOM [74]), while adding protections that prevent programs from attempting to disclose or forge data within their own address space (*e.g.*, buffer overflow or JavaScript-based attack).

4.4 Churning Moving Target Defenses

In Morpheus, the domain encryption and pointer displacement defenses are re-randomized at runtime beneath live execution. The churn unit implements re-randomization of code and pointers in coordination with the main core. In addition, the churn unit supports secure context switching for address spaces that have not completed their churn cycle.

Churning Under Live Execution: To accommodate live churn, the churn unit updates system state concurrently with the running processor. This serves to minimize the performance impact on running programs. As shown in Figure 6, the churn unit maintains a ***threshold register*** to indicate how far churn has progressed in the running process's address space. State that has been processed to use the new keys is "clean", while state using old keys and awaiting update is "stale". To make state clean, code and pointers must be re-encrypted under the new domain encryption key (unless in a register), and the difference between the new and old displacements ($d_{NEW} - d_{OLD}$) must be added to all pointers.

The steps of the churn cycle are illustrated in Figure 6. First, the pipeline is flushed and fetch is halted to allow churn to update registers in the next stage. Next, a new set of keys is generated for all defenses, and then the register values are updated. To simplify register updates, a process context switch is initiated, which stores the stale registers to memory before the churn cycle starts. The context is reloaded once the churn cycle begins, which has the effect of updating the registers as they are loaded back into the register file.

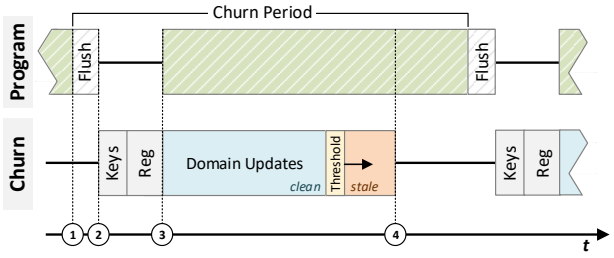

Figure 6. Churning During Execution. The churn cycle starts with ① a pipeline flush and ② new key generation and register updates. Next, ③ a threshold register is used to coordinate updating values in memory under the new displacements and encryption keys. When ④ churn completes, all domains have been updated, effectively disposing of any information the attacker may have acquired previously.

Next, fetch is resumed and the churn unit walks the virtual address space from low addresses to high addresses, page by page, updating all code and pointers indicated in the tag store. As memory locations are updated, the threshold register address is updated to indicate the progress of churn. The churn unit updates memory by accessing storage on the coherent L1-to-L2 bus (as shown in Figure 3), after indicating the address to be accessed in the threshold register. If the memory location accessed is in the data cache and marked "dirty", it is first written back and invalidated from the L1 before being accessed by the churn core. To prevent race conditions between the churn unit and kernel, virtual memory pages are locked down while being updated by the churn unit. Once updates are completed, the system can initiate churn again immediately, or it can delay the next churn cycle until it meets the desired security-performance trade-offs.

To simplify coordination with the main core, the churn unit maintains the following invariants: *i)* all pipeline state (*e.g.*, instructions and pointers in registers and latches) is clean, *ii)* all memory values *below* the threshold address are clean, iii) all memory values *above* the threshold are stale, and *iv)* the memory value *at* the threshold address is currently being processed by the churn unit. Given these invariants, the domain encryption unit selects the appropriate keys depending on if the value is below or above the threshold register (K_{NEW} or K_{OLD} respectively). By contrast, only new keys are necessary for the pointer displacement defense, since the pointer displacement unit only generates effective addresses with clean registers. Additionally, when a load accesses a pointer *above* the threshold address, the stale pointer loaded from memory is updated before entering the register file by adding ($d_{NEW} - d_{OLD}$) to the value. Similarly, when a store writes a pointer to memory *above* the threshold register, the register pointer value is clean, but it must be stored as a stale pointer under the old displacement. This

is done by subtracting ($d_{NEW} - d_{OLD}$) from the pointer before writing it to memory. Finally, if the main core makes a memory access *at* the threshold address, the core must stall the access until the churn unit updates the threshold register. This last requirement prevents potential write-after-read and write-after-write hazards that could occur when the core and churn unit are accessing the same memory address. The churn unit gets priority over accesses to the conflicting memory location, to ensure that a rogue core could not perform a denial-of-service attack on the churn unit.

Support for Context Switching: To prevent adverse impacts to OS scheduling, the churn unit supports context switching during an active churn cycle. The OS can request the current context from the churn unit, which is encrypted with a boot-time private churn key. The encrypted context is passed to the kernel, which stores it in the kernel's process control block. The churn context contains the threshold register, the old keys for all defenses, the new keys for all defenses, and the time that the last churn cycle was initiated. To prevent attacks on mostly idle programs, the churn unit initiates a churn cycle on a program after it has been idle longer than its normal churn cycle (as indicated by the timestamp in the churn context).

5 Morpheus Architecture Study

In this section, we first detail our experimental setup of the Morpheus secure architecture. Next, we assess the security of the architecture by examining its ability to stop a wide range of control-flow attacks, and by exploring how long it could take an attacker to penetrate Morpheus by crafting novel attacks directed specifically at our system. Finally, we assess the performance costs of providing EMTDs with fast churn for the SPEC'06 and MiBench benchmarks.

5.1 Experimental Framework

To assess security and performance, we implemented our Morpheus prototype on the RISC-V port of the gem5 simulator [10, 61]. The Morpheus core is built on top of the gem5 MinorCPU 4-stage in-order core, using the configuration parameters listed in Table 3. We chose this core model because it would demonstrate the impact of EMTD protections for a core with little capability to tolerate any introduced latencies (as opposed to an out-of-order core). Morpheus' encryption layer uses QARMA$_7$-64σ_1, which, when synthesized in [5], had a minimum delay of 3.25 ns (9-cycle latency at 2.5 GHz). This is masked by the L2 access. We used DRAMSim2 [63] to model the memory system and assess the performance of tag scanning and churn operations. Specifically, the churn unit communicates between gem5 and DRAMSim2 to scan tags and update pointers and code. The churn unit is implemented as a simple FSM with access to the cache-coherent bus between the main core's L1-cache and the L2-cache. In addition, the churn unit has a back-channel connection to the main core to force it to drain the pipeline before a churn cycle begins. The simulations model a single process context, using the system call emulation capabilities of gem5.

Table 3. Morpheus Microarchitecture Configuration. Beyond the baseline components, a 4KB tag cache is added to the EMTD-enabled microarchitecture.

Core Type	MinorCPU (In-Order)
CPU Frequency	2.5GHz
Cache Line Size	64B
L1 Instruction Cache Size	32KB with 2-cycle latency
L1 Data Cache Size	32KB with 2-cycle latency
L2 Unified Cache Size	256KB with 20-cycle latency
Tag Cache Size	4KB

5.2 Security Analysis

To gauge the security benefits of Morpheus, we perform a simulation-based study of its ability to stop a wide range of control-flow attacks, ranging from long-standing attacks (*e.g.*, buffer overflow) to advanced attacks (*e.g.*, return-oriented programming). We then attempt to build Morpheus-specific attacks, to assess the value of ensembles and churn.

Penetration Testing Results: To build confidence that the Morpheus architecture is capable of stopping control-flow attacks, we performed penetration testing with real-world control-flow attacks running on the simulated Morpheus architecture. We ran tests from an ongoing port of the RIPE control-flow attack suite [82]. The port, being developed by Draper Labs [25], does not yet implement the entire RIPE attack suite, but it did give us access to stack overflow [2], heap overflow [22, 36], and ROP [62] attacks. In addition to RIPE, we included hand-crafted implementations of additional attacks, including heap spray [57], format string [52, 59], integer overflow [12], and back-call-site [75] attacks. All attacks aimed to overwrite an existing return address or function pointer as a means to manipulate control flow.

The Morpheus architecture stopped all of the attack classes from our penetration testing suite. Table 4 lists the attacks tested and which Morpheus defenses aided in stopping the attack. Stopping a comprehensive control-flow attack suite shows that the Morpheus architecture is capable of stopping state-of-the-art control-flow attacks. Sophisticated software protections that come close (e.g., CPI [43]) are significantly more expensive and advanced hardware protections such as the soon to be released Intel CET [37] extensions have notably lower coverage for advanced attacks.

Of the control-flow attacks in Table 4, one of particular note is the *Back-Call-Site* attack, which is a recently published control-flow attack designed to circumvent advanced CFI-based control-flow protections [75]. Morpheus stops this attack when it attempts to forge a code pointer to redirect a function return. It is interesting to note that ***we learned of***

Table 4. Penetration Testing Results. The Morpheus secure architecture was shown in simulations to stop all of the attacks listed. Additionally, all attacks trigger our attack detector. For each attack, the table shows how Morpheus stops the attack, with Morpheus domain encryption (E) and pointer displacement (P) applied to code (C), code pointers (CP), and data pointers (DP).

Attack	Defenses Engaged					Bits of Entropy
	E-C	E-CP	E-DP	P-CP	P-DP	
Stack Buffer Overflow [2]	X	✓	X	✓	X	188
Heap Overflow [22, 36]	X	✓	X	✓	X	188
Heap Spray [57]	X	✓	X	✓	X	188
Format String [52, 59]	X	✓	✓	✓	✓	376
Integer Overflow [12]	X	✓	✓	✓	✓	376
ROP [62]	✓	✓	✓	✓	✓	504
Back-Call-Site Attack [75]	✓	✓	✓	✓	✓	504

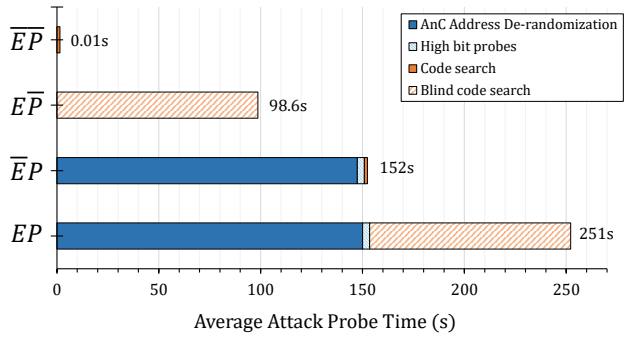

Figure 7. Estimated Probe Times. This figure shows the average estimated attack time to penetrate a de-featured Morpheus, which has fewer defense keys and no churn. Probe times are shown for Morpheus with (E) and without ($\overline{E}$) domain encryption, and with (P) and without ($\overline{P}$) pointer displacement. As more defenses are engaged, probe times grow, with fully-activated defenses requiring a probe time 5020x longer than the normal churn period (50 ms).

this last attack after our Morpheus architecture was designed, suggesting that Morpheus provides some level of future-proofing against unknown future CFG attacks.

Table 4 shows that the number of defenses stopping the attack grows as attacks become more advanced. This growth in the effective strength of Morpheus occurs because new attacks nearly always need to acquire additional information to circumvent defenses introduced to stop earlier attacks. For example, buffer overflow attacks [2] do not need knowledge of program code. However, the introduction of non-executable memory led to code reuse attacks like ROP [62], that do need knowledge of the program code. ***Our analysis suggests that, since Morpheus defends critical attack assets rather than patching vulnerabilities, Morpheus is more effective even as attacks become more advanced.***

Attacking Morpheus: We can gain a deeper understanding of Morpheus' security benefits by crafting new attacks designed to penetrate Morpheus defenses. Our attack scenario involves a local program attacking a victim program (*i.e.*, SPEC's *gobmk* in our study) via an IPC interface, where the program under attack has exception reporting and crash recovery comparable to vanilla Linux. The goal of the attack is to call the C-library `system()` function, thereby giving an attacker a shell in the victim program's process context. We assume that the organization of the code has been randomized at build time (similar to [68]), and the victim program is crash-resilient.

Crafting successful attacks to penetrate Morpheus required degrading its protections in three ways: *i)* churn and the attack detector are disabled to give the attacker unlimited time to probe the system, *ii)* a single displacement key is used across all address spaces, since microarchitectural sharing attacks are not possible with different displacements and *iii)* a single key for domain encryption is used to allow code pointer forgery.

Brute force attacks in this context are infeasible due to the large randomization space of Morpheus defenses, leading attackers to use side-channels to leak information. When attacking Morpheus, we take this approach by combining techniques from the AnC de-randomization attack [30] and blind code search [11]. We simulated all aspects of each attack described below, except for IPC calls, exception handling, and AnC de-randomization. Instead, reported times were used [14, 30, 78]. Being conservative, we assume AnC can recover all 48-bits of the virtual address in 150 s, whereas the original work recovers fewer bits due to limited entropy in the experimental setup.

As shown in Figure 7, we evaluate the time it takes to attack the de-featured Morpheus for each of the four combinations of moving target defenses: $\overline{EP}$, with no defenses, $E\overline{P}$, with only encryption, $\overline{E}P$, with only pointer displacement, and EP, with all defenses engaged.

$\overline{EP}$— To execute a call to `system()` when no defenses are engaged, an attacker needs to simply search the victim's code to locate this function. This can be done through exploiting a stack buffer overflow to call `memchr()`, which returns the address of `system()` in 10 ms.

$E\overline{P}$— When encryption is engaged, this technique is not applicable because functions like `memchr()` would incorrectly compare to encrypted code. Rather, an attacker can simply make blind calls [11] into the code segment, until successfully calling `system()`. We can make these calls using mimicry [79] to form the code pointer, and use a stack buffer overflow to test the pointer. This process takes about 98.6 sec to identify `system()` and effectively defeats the encryption defense. It should be noted that these operations would trigger the attack detector on fully featured Morpheus.

$\overline{EP}$— When pointer displacement is engaged, an attacker needs to determine the displacement key to locate the code segment before searching for the `system()` function. To leak a pointer and uncover the key, an attacker can use an AnC de-randomization probe to reveal a virtual address, followed by high-bit probes to determine where the code resides among the remaining 2^{16} possible locations (*i.e.*, the DAS address). Breaking displacement takes 150 sec for AnC and 2.3 sec for high bit probes. After the pointer displacement defense is defeated, an attacker can perform a normal code search as described above. Overall, this exploit takes about 152 sec.

EP— Finally, with all defenses engaged, the above techniques are combined to first de-randomize pointer displacement, and then blindly search the code segment for a call to `system()`. This approach uses a combination of an AnC de-randomization probe to recover the 48-bit virtual address, high-bit probes with blind calls to recover the upper DAS address, and a full blind-call search for `system()`. Combined, this attack takes a total of 251 sec.

It is clear that ensembling defenses carry much value, since attack times progressively increase as more moving target defenses are engaged. Moreover, the time needed to attack the fully engaged defenses is 251 sec. This provides a useful metric for assessing churn periods, which should be much faster than 251 sec. With churn on, attackers resort to brute-force guessing, which will take substantially longer.

When examining the individual attack details, it is possible to see the direct benefits of ensembles. Clearly, with all defenses engaged, it is the worst of both worlds for the attacker, since they do not know where the code is located in the displaced address space, and once found, they cannot inspect it to easily find the location of `system()`. It is also interesting to note that the increase in attack time is purely additive, since, for this attack, defeating one defense provides no value in penetrating the other defense.

5.3 Performance Impact of EMTDs with Churn

To assess the performance overheads of EMTD defenses, we ran experiments on our augmented gem5 simulator with all Morpheus protections engaged, *i.e.*, domain encryption and pointer displacement. We evaluated Morpheus using benchmarks from MiBench [32] and SPEC'06 [33]. By comparing the performance impacts on these two suites, we can get an understanding of the relative costs for small embedded programs (MiBench) and larger desktop/server workloads (SPEC'06). Simulations were run with a 3 billion instruction cap. From MiBench, we analyzed the performance of 13 programs: *basicmath*, *bitcount*, *qsort*, *susan*, *dijkstra*, *patricia*, *stringsearch*, *blowfish*, *rijndael*, *sha*, *crc32*, *fft*, and *adpcm*; and from SPEC'06, we analyzed the performance of all C-code benchmarks, 12 in total, using the reference input: *perlbench*, *bzip2*, *gcc*, *mcf*, *milc*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *lbm*, and *sphinx3*. All programs were built using LLVM 5.0.0 for the RISC-V RV64IMA architecture [3] with optimization level -O2. The benchmarks were linked against a Morpheus-built RISC-V Musl C library [64].

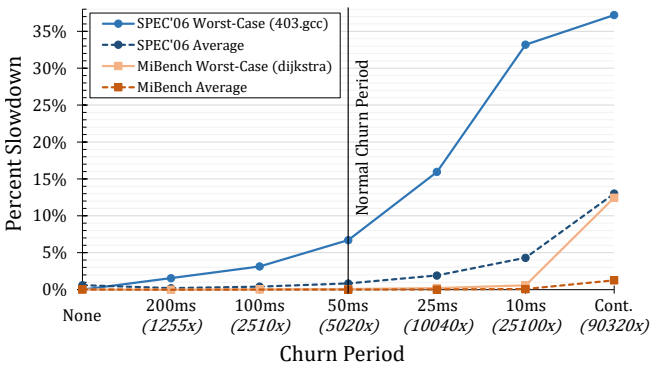

Figure 8. Performance Overhead. This graph shows the average and worst-case performance overheads of the chosen MiBench and C-based SPEC'06 benchmarks, with churn periods varying from no churn (left) to continuous churn (right). The expected number of churn periods that would complete in the time estimated to penetrate Morpheus defenses, based on the attack study in Section 5.2, is shown below each churn period. Given these results, we target 50 ms as a "normal" churn period.

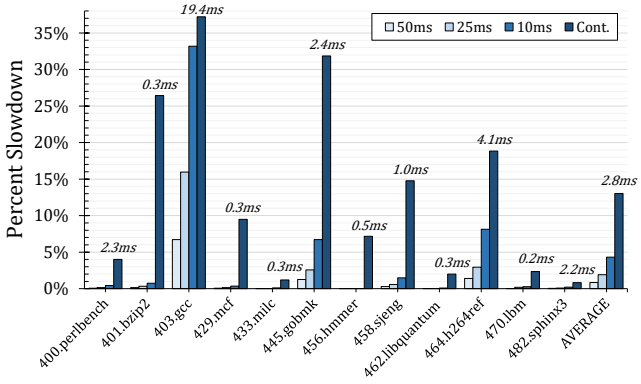

Figure 9. SPEC'06 Performance Overheads. The graph shows SPEC'06 performance overheads for varied churn periods. The continuous churn period is listed above the bar.

Figure 8 shows the slowdown for the MiBench and SPEC'06 benchmark suites with full Morpheus protections at varied churn periods. The slowdowns measured are with respect to the program running on the baseline architecture without Morpheus protections. For both MiBench and SPEC'06, the graph shows the average slowdown at each measured churn period, plus the benchmark with the highest slowdown (*i.e.*, *dijkstra* for MiBench and *gcc* for SPEC'06). The points on the far right of the graph represent a continuous churn period, where a new churn cycle starts immediately after the previous one ends. For these points, the churn period varies by program; the average continuous churn period for SPEC'06

was 2.8 ms, with *gcc* having the slowest churn time of 19.4 ms. For MiBench, the average churn time was much smaller, at 0.4 ms, with the worst case being *patricia* at 2.1 ms.

As shown in Figure 9, slowdowns for short churn periods are quite low, with an average overhead of 0.84% (and a worst-case slowdown of 6.71%) at 50 ms. Overheads even seem tolerable up to 10 ms, but quickly grow as the system approaches continuous churn. These low overheads are the result of efficient hardware-based churn, and the fact that code and pointers only make up a small fraction of overall memory. Non-pointer data is not accessed during the churn phase. Surprisingly, running without churn results in a slightly higher overhead than running with 200 ms churn (0.6% vs. 0.2%), because our churn unit is essentially acting as a cache prefetcher. To better understand the churn unit's workload, we analyzed *gcc* since it had the largest slowdown. We found that *gcc* has *i)* the largest data segment, with an average footprint of 47.8 MB, *ii)* the most pointers, with over 300,000, and *iii)* the largest codebase, at 5.3 MB. The combination of these factors resulted in more work for the churn unit, and therefore higher overheads.

It is interesting to consider how fast the churn period should be. In existing network security protocols, churn periods are very large. The re-key period for the TLS secure web protocol defaults to 1 hour (3.6 million ms) [85]. In the previous section, we found that it took 251 sec to penetrate the de-featured Morpheus architecture. The number of churn cycles that would complete in that time is shown in parenthesis on the x-axis of Figure 8. Here, we see that ***a churn period of 50 ms is 5020x faster than the expected attack time, thus we see 50 ms as a reasonable "normal" churn period.*** In all likelihood, an attack would frequently trigger the attack detector, leading to continuous churn, which is on average 90,320x faster than the expected attack time.

6 Related Work

Networking and software security has adopted many forms of moving target defenses (MTDs), including popular examples like randomizing network-service ports and ASLR. However, prior research typically focuses on protecting a single asset due to the high overhead introduced by applying multiple MTDs. Morpheus supports multiple MTDs in hardware to provide protection for more assets with lower overhead. A comparison of Morpheus against prior works is shown in Table 5. In this table, we show the information assets protected by each work, their associated entropies, average overhead, and if they support runtime churn.

Displacement: ASLR [54] randomizes the base addresses of code and data segments each time a program is loaded, but is susceptible to insufficient randomness [11, 66] – worse, a single address leakage through a memory disclosure vulnerability can reveal the location of all code and data [28, 30, 65, 69]. Kil *et al.* [41] proposed a finer-grained permutation of procedures and data objects for ASLR, such that the leakage of a single address does not immediately reveal the location of all program information. Remix [17], TASR [9], and RuntimeASLR [46] augment ASLR with a run-time component either by rearranging direct calls, direct jumps, and indirect jumps (leaving indirect calls intact and hence being vulnerable to return-to-libc attacks [70]) or by re-randomizing the memory layout after sensitive system calls.

Encryption: PointGuard [18] and related techniques [77] have employed encryption to obfuscate pointers as a means to defend against control-flow attacks. Besides having higher overheads, these approaches have consistently been compromised due to either weak encryption (typically XOR-based), or read attacks that extracted the key. In contrast, Morpheus uses strong encryption and encrypts disparate domains (an idea in [20]) with distinct keys held in protected registers.

Encryption has also been used to protect code, as in binary-translation based Instruction Set Randomization (ISR) [6, 35, 40] – with ∼10% overhead common for network-facing services and up to 75% for databases. This overhead decreases to ∼1% with hardware support in ASIST [53]. Randomization schemes have also been employed for data [8, 15], in which static analysis is used to partition memory into classes that each have their own distinct, random mask. Its average slowdown for SPEC is 14%. While these techniques are appealing as they obscure code and data efficiently, they use weak encryption, both from a brute-force protection and cryptanalysis standpoint. As such, Morpheus forgoes these approaches and adopts a strong cipher.

Another approach to program randomization is N-version systems [7, 19, 51], which require an attacker to break defenses in multiple domains simultaneously. Overheads range from 28%-129% for latency and ∼50% for server throughput. While these techniques are powerful, Morpheus defends against the same attacks with lower overheads.

Comparison with Shuffler: The value of hardware-based EMTDs comes into focus when we contrast Morpheus to a software-based moving target defense. The fastest previous continuous address space reorganization technology we are aware of is Shuffler [83], which moves code periodically in the address space of a running program. Shuffler also encrypts return addresses with XOR.

There are some advantages Morpheus has over Shuffler: *i)* hardware enables faster churn periods, *ii)* more randomization with a total of 504 key bits and strong encryption, *iii)* more domain coverage, including code and code & data pointers, and *iv)* lower overhead at a 50 ms churn period – <1% compared to 15% (with a worst-case of 45%). An advantage of Shuffler is that it destroys relative distance between code objects, whereas Morpheus' displacement does not. However, Morpheus' hardware-based defenses have lower overhead while delivering more randomization.

Table 5. Comparison Against Prior Works. The assets protected by each work is listed. A ✓* indicates protection for only return addresses (for CP), or frame pointers (for DP). Displacing the code segment is listed as protection for code pointers (CP), rather than for code (C). For Fixed Interval systems, the average overhead is quoted at 50 ms. †TASR's overhead is originally 2.1%, however, as noted in Shuffler [83], TASR uses -Og, which can slow down SPEC benchmarks by ~30% compared to -O2.

	Name		Assets C	Assets CP	Assets DP	Entropy/Key Size	Runtime Churn	Avg. Overhead
Displacement	64-bit PaX ASLR [54, 55]		-	✓	✓	29-30 bits (48-bit vaddr)	No	3.6%
	TASR [9]		-	✓	✓	29-30 bits (48-bit vaddr)	At I/O Only	30-40%†
	Remix [17]		-	✓	X	ASLR + $\log_2$(basic blocks per func.) Apache on x86 (32-bit): 16+4 = 20 bits max	Random Interval	2.8% (one-time)
	RuntimeASLR [46]		-	✓	✓	28-48 bits (48-bit vaddr)	At `fork()` Only	0.5%
Encryption	PointGuard [18]		X	✓	✓	64 bits (weak XOR cipher, on 64-bit ISA)	No	10.0%
	CCFI [47]		X	✓	✓*	128 bits (strong cipher)	No	23.0%
	ASIST [53]		✓	✓*	X	32-128 bits (weak XOR cipher) or 32 bits (weak transposition)	No	1.0%
	Polyglot [68]		✓	X	X	163 bits (strong cipher)	No	4.6%
Enc. + Disp.	Shuffler [83]	Displacement	-	✓	X	27 bits (48-bit vaddr)	Fixed Interval	14.9%
		Encryption	X	✓*	X	64 bits (weak XOR cipher)		
	Morpheus	Displacement	-	✓	✓	60 bits per segment	Fixed Interval	0.9%
		Encryption	✓	✓	✓	128 bits per asset (strong cipher)		

Tagged Architectures: Secure architectures often utilize tags to provide defenses with more information at runtime. For example, Dynamic Information Flow Tracking (DIFT) [23, 27, 31, 39, 50, 71, 76] enables analyses to assign labels to sensitive data, code, or inputs. These labels are tracked as the program executes and raise an exception if a 'tainted' label is used for a sensitive operation (*i.e.*, as an information flow 'sink'). This approach is quite powerful at ferreting out vulnerabilities; however, finding all of the vulnerabilities would require *i)* full path coverage of the program, and *ii)* a very detailed vulnerability model. Both of these challenges ultimately keep detection of all vulnerabilities beyond reach. Our EMTDs, on the other hand, are vulnerability-agnostic, since they hide the information needed to exploit vulnerabilities.

Other architectures, such as CHERI [84], lowRISC [45], the Dover processor [72], and PUMP [24], have implemented tags in hardware to enforce security policies. Like Morpheus, these systems rely on precise tagging of memory objects. However, these systems often face high false-positive rates, leading to the failure of benign programs. Morpheus operates differently, as false-positive security violations from the attack detector only trigger a churn cycle.

7 Conclusions and Future Directions

While traditional security protections work to find and fix every last vulnerability, EMTDs with churn take the approach of protecting a system by randomizing the information assets that attackers need to craft successful attacks. In this paper, we presented the Morpheus secure architecture, which brings together multiple moving target defenses to protect a system from control-flow attacks. Pointer displacement and domain encryption work in tandem to protect the code and pointers needed to launch control-flow attacks. A hardware-based churning mechanism is able to re-randomize these values at runtime, without impacting normal programs. Together, these protections demonstrate a high level of protection against control-flow attacks with very low overheads.

Looking ahead, we see great potential for EMTD technologies. Beyond control-flow attacks, we envision that a similar approach could be adopted to protect against side-channel attacks, timing attacks, Rowhammer attacks, and even cache attacks. To address each of these additional challenges, we will explore what assets the attacker needs and then develop efficient mechanisms to boost uncertainty and stifle attacks.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by DARPA under Contract HR0011-18-C-0019. 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 and Valeria Bertacco.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Aleph One. 1996. Smashing the stack for fun and profit. Phrack Magazine. Retrieved April 6, 2018 from http://www.phrack.org/archives/49/P49-14

[3] AndesTech. 2017. Andes Technology GitHub - riscv-llvm. Retrieved August 2, 2018 from https://github.com/andestech/riscv-llvm

[4] Apple Corporation. 2018. Undefined Behavior Sanitizer. Retrieved August 6, 2018 from https://developer.apple.com/documentation/code_diagnostics/undefined_behavior_sanitizer

[5] Roberto Avanzi. 2017. The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Transactions on Symmetric Cryptology* 2017, 1 (Mar. 2017), 4–44. https://doi.org/10.13154/tosc.v2017.i1.4-44

[6] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 281–289. https://doi.org/10.1145/948109.948147

[7] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

[8] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*. Springer-Verlag, Berlin, Heidelberg, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[9] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/2810103.2813691

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[11] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 227–242. https://doi.org/10.1109/SP.2014.22

[12] blexim. 2002. Basic integer overflows. Phrack Magazine. Retrieved April 6, 2018 from http://phrack.org/issues/60/10.html

[13] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Washington, DC, USA, 987–1004. https://doi.org/10.1109/SP.2016.63

[14] Vince Bridgers. 2015. Real Time Linux Scheduling Comparison. In *Embedded Linux Conference 2015 (ELC 15)*. The Linux Foundation, CA. http://events17.linuxfoundation.org/sites/events/files/slides/Real-Time-Linux-Comparison-Bridgers-ELC2015.pdf

[15] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. 2008. *Data Randomization*. Technical Report. Microsoft Research, Redmond, WA, USA. 14 pages. https://www.microsoft.com/en-us/research/publication/data-randomization/ MSR-TR-2008-120.

[16] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 147–160. http://dl.acm.org/citation.cfm?id=1298455.1298470

[17] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-Demand Live Randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/2857705.2857726

[18] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, Vol. 12. USENIX Association, Berkeley, CA, USA, 91–104.

[19] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 9. http://dl.acm.org/citation.cfm?id=1267336.1267344

[20] John Criswell and Vikram Adve. 2010. Chaos for a Fast, Secure, and Predictable Future. In *Fun and Interesting Thoughts at the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (FIT PLDI '10)*. 2. http://pldi10fit.blogspot.com/2010/05/chaos-for-fast-secure-and-predictable.html

[21] CVEs by Type. 2018. CVE Details: Vulnerabilities By Type. Retrieved April 6, 2018 from https://www.cvedetails.com/vulnerabilities-by-types.php

[22] CWE-122. 2018. CWE-122: Heap-based Buffer Overflow. Retrieved April 6, 2018 from https://cwe.mitre.org/data/definitions/122.html

[23] Michael Dalton, Hari Kannan, and Christos Kozyrakis. 2007. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 482–493. https://doi.org/10.1145/1250662.1250722

[24] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. 2015. 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)*. ACM, New York, NY, USA, 487–502. https://doi.org/10.1145/2694344.2694383

[25] Draper Laboratory. 2018. Draper Laboratory GitHub - hope-RIPE. Retrieved April 6, 2018 from https://github.com/draperlaboratory/hope-RIPE

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

[27] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. https://doi.org/10.1145/2619091

[28] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[29] Robert Gawlik, Benjamin Kollenda, Philipp Koppe, Behrad Garmany, and Thorsten Holz. 2016. Enabling Client-Side Crash-Resistance to

Overcome Diversification and Information Hiding. In *Proceedings of the Network and Distributed System Security Symposium 2016 (NDSS '16)*. Internet Society, Reston, VA, USA. https://doi.org/10.14722/ndss.2016.23262

[30] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings of the Network and Distributed System Security Symposium 2017 (NDSS '17)*. Internet Society, Reston, VA, USA. https://doi.org/10.14722/ndss.2017.23271

[31] Joseph L. Greathouse, Ilya Wagner, David A. Ramos, Gautam Bhatnagar, Todd Austin, Valeria Bertacco, and Seth Pettie. 2008. Testudo: Heavyweight Security Analysis via Statistical Sampling. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 117–128. https://doi.org/10.1109/MICRO.2008.4771784

[32] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, Washington, DC, USA, 3–14. https://doi.org/10.1109/WWC.2001.15

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

[34] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Washington, DC, USA, 969–986. https://doi.org/10.1109/SP.2016.62

[35] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments (VEE '06)*. ACM, New York, NY, USA, 2–12. https://doi.org/10.1145/1134760.1134764

[36] Yan Huang. 2016. Heap Overflows and Double-Free Attacks. Retrieved April 6, 2018 from http://homes.soic.indiana.edu/yh33/Teaching/I433-2016/lec13-HeapAttacks.pdf

[37] Intel Corporation. 2017. Control-flow Enforcement Technology Preview. Retrieved August 6, 2018 from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[38] Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang. 2011. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats* (1st ed.). Springer-Verlag, New York, NY, USA. https://doi.org/10.1007/978-1-4614-0977-9

[39] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium 2011 (NDSS '11)*. Internet Society, Reston, VA, USA.

[40] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering Code-injection Attacks with Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 272–280. https://doi.org/10.1145/948109.948146

[41] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Washington, DC, USA, 339–348. https://doi.org/10.1109/ACSAC.2006.9

[42] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[44] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[45] lowRISC. 2018. lowRISC Project. Retrieved April 5, 2018 from http://www.lowrisc.org/

[46] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the Network and Distributed System Security Symposium 2016 (NDSS '16)*. Internet Society, Reston, VA, USA. https://doi.org/10.14722/ndss.2016.23173

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

[48] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. ACM, New York, NY, USA, Article 10, 8 pages. https://doi.org/10.1145/2487726.2488368

[49] National Institute of Standards and Technology. 2018. National Vulnerability Database. Retrieved April 6, 2018 from https://nvd.nist.gov

[50] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium 2005 (NDSS '05)*. Internet Society, Reston, VA, USA.

[51] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. 2008. Security through redundant data diversity. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE Press, Piscataway, NJ, USA, 187–196. https://doi.org/10.1109/DSN.2008.4630087

[52] OWASP. 2015. Format string attack. Retrieved April 6, 2018 from https://www.owasp.org/index.php/Format_string_attack

[53] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. 2013. ASIST: Architectural Support for Instruction Set Randomization. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/2508859.2516670

[54] PaX Team. 2003. PaX address space layout randomization (ASLR). Retrieved April 6, 2018 from http://pax.grsecurity.net/docs/aslr.txt

[55] Mathias Payer. 2012. *Too much PIE is bad for performance.* Technical Report. ETH Zurich, Department of Computer Science, Zürich. https://doi.org/10.3929/ethz-a-007316742 Technical Reports D-INFK.

[56] Colin Percival. 2005. Cache missing for fun and profit. In *Proceedings of the Technical BSD Conference 2005 (BSDCan '05)*.

[57] Jonathan Pincus and Brandon Baker. 2004. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security & Privacy* 2, 4 (July 2004), 20–27. https://doi.org/10.1109/MSP.2004.36

[58] Qualcomm Product Security. 2017. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions.* Technical Report. Qualcomm Technologies, Inc., San Diego, CA, USA.

[59] riq and gera. 2001. Advances in Format String Exploiting. Phrack Magazine. Retrieved April 6, 2018 from http://phrack.org/issues/59/7.html

[60] RISC-V Foundation 2017. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2.* RISC-V Foundation, Berkeley, CA, USA. Editors Andrew Waterman and Krste Asanović. May 2017.

[61] Alec Roelke and Mircea R. Stan. 2017. RISC5: Implementing the RISC-V ISA in gem5. In *Proceedings of Computer Architecture Research in RISC-V* (14) *(CARRV '17)*. 7.

[62] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[63] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Archit. Lett.* 10, 1 (Jan. 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[64] rv8. 2018. rv8.io GitHub - musl-riscv. Retrieved August 2, 2018 from https://github.com/rv8-io/musl-riscv

[65] Fermin J. Serna. 2012. The info leak era on software exploitation. In *Black Hat USA*.

[66] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[67] Gennadiy Shvets. 2018. Enhanced Virus Protection / Execute Disable Bit. Retrieved January 24, 2019 from http://www.cpu-world.com/Glossary/E/EVP_XD.html

[68] Kanad Sinha, Vasileios P. Kemerlis, and Simha Sethumadhavan. 2017. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE Press, Piscataway, NJ, USA, 21–28. https://doi.org/10.1109/HST.2017.7951732

[69] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 574–588. https://doi.org/10.1109/SP.2013.45

[70] Solar Designer. 1997. lpr LIBC RETURN exploit. Retrieved April 6, 2018 from http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[71] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1024393.1024404

[72] Gregory T. Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. 2017. The Dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE Press, Piscataway, NJ, USA, 1–5. https://doi.org/10.1109/THS.2017.7943502

[73] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 48–62. https://doi.org/10.1109/SP.2013.13

[74] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 168–177. https://doi.org/10.1145/378993.379237

[75] Michael Theodorides and David Wagner. 2017. Breaking Active-Set Backward-Edge CFI. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE Press, Piscataway, NJ, USA, 85–89. https://doi.org/10.1109/HST.2017.7951803

[76] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. 2009. Quantifying the Potential of Program Analysis Peripherals. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*. IEEE Computer Society, Washington, DC, USA, 53–63. https://doi.org/10.1109/PACT.2009.38

[77] Nathan Tuck, Brad Calder, and George Varghese. 2004. Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 37)*. IEEE Computer Society, Washington, DC, USA, 209–220. https://doi.org/10.1109/MICRO.2004.20

[78] Aditya Venkataraman and Kishore Kumar Jagadeesha. 2015. *Evaluation of Inter-Process Communication Mechanisms.* Technical Report. University of Wisconsin-Madison, Madison, WI, USA.

[79] David Wagner and Paolo Soto. 2002. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*. ACM, New York, NY, USA, 255–264. https://doi.org/10.1145/586110.586145

[80] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[81] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

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

[83] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 367–382. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king

[84] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[85] James Yonan. 2010. OpenVPN 2.0.x man pages. Retrieved April 6, 2018 from https://openvpn.net/index.php/open-source/documentation/manuals/65-openvpn-20x-manpage.html