Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces

Lauren Biernacki
University of Michigan
lbiernac@umich.edu

Mark Gallagher
University of Michigan
markgall@umich.edu

Valeria Bertacco
University of Michigan
vale@umich.edu

Todd Austin
University of Michigan
austin@umich.edu

***Abstract*—To maintain the control-flow integrity of today's machines, code pointers must be protected. Exploits forge and manipulate code pointers to execute arbitrary, malicious code on a host machine. A corrupted code pointer can effectively redirect program execution to attacker-injected code or existing code gadgets, giving attackers the necessary foothold to circumvent system protections. To combat this class of exploits, we employ a Displaced and Dilated Address Space (DDAS), which uses a novel address space inflation mechanism to obfuscate code pointers, code locations, and the relative distance between code objects. By leveraging runtime re-randomization and custom hardware, we are able to achieve a high-entropy control-flow defense with performance overheads well below 5% and similarly low power and silicon area overheads. With DDAS in force, attackers come up against 63 bits of entropy when forging absolute addresses and 18 to 55 bits of entropy for relative addresses, depending on the distance to the desired code gadget. Moreover, an incorrectly forged code address will result in a security exception with a probability greater than 99.996%. Using hardware-based address obfuscation, we provide significantly higher entropy at lower performance overheads than previous software techniques, and our re-randomization mechanism offers additional protections against possible pointer disclosures.**

I. Introduction

Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine. Control flow exploits are some of the most dangerous, and most common, attacks of this nature [1]. In these exploits, attackers leverage knowledge of the location and representation of code pointers to abuse a memory corruption vulnerability and overwrite control data with malicious pointers, enabling the execution of arbitrary, malignant code on the victim machine [2], [3], [4]. Attackers without knowledge of the memory configuration employ techniques to readily discover this information at runtime [5], [6], [7]. This class of exploits occurs on the *control plane*, as attackers directly manipulate control data and require knowledge of the code segment and the representation of code pointers to be successful.

To deprive attackers of the information needed to synthesize an exploit, protections have employed *address obfuscation*. These defenses randomize the memory layout of the victim machine, effectively prohibiting attackers from accessing memory and forging pointer values at whim. There exists a significant body of work on address obfuscation defenses, yet the techniques used to randomize memory vary broadly. Address obfuscation techniques include address space displacement [8], [9], [10], [11], permutation of code at the function [12], [13], [14], [15], [16], basic block [17], [18], [19], [20], or instruction level [21], [22], insertion of padding between code objects [23], duplication of the code segment [24], [7], and insertion of traps within the code space [25].

While obfuscation-based protections can be both effective and efficient, they offer probabilistic security. Thus, achieving both high entropy and fine-grained, frequent randomization is critical for resilience [26]. Memory disclosures, or the leakage of memory addresses, can be readily used to derandomize the address space. For example, with techniques that only employ address space displacement, such as Address Space Layout Randomization (ASLR) [8], the entire code layout can be inferred from a single pointer disclosure. Even permutation-based defenses that obfuscate the relative distance between code objects can be derandomized by multiple disclosures. Fortunately, the use of runtime re-randomization to nullify leaked pointers [9], [10], [11], [13], [17], [18] and in-memory traps to detect probing attempts and trigger security exceptions [25] have been instrumental at fortifying address obfuscation techniques. However, systems with these added protections suffer from far greater performance overheads (14%-225% for function permutation [13], [18]), reducing the appeal of randomization-based defenses.

The deficiencies of low entropy, memory disclosures, and high overheads have hindered the adoption of software-based address obfuscation techniques. Secure hardware integrated into the processor pipeline can bridge this gap. Yet, existing hardware-based defenses are far behind their software counterparts, as they do not obfuscate relative distances [11]. In this work, we leverage custom RISC-V hardware to implement a defense with *all* of these necessities: high entropy, fine granularity, re-randomization, in-memory traps, and low overheads. To protect code pointers from external influence, we completely decouple pointers stored in the data segment from the true location of their targets by expressing all code pointers in the superimposed ***Displaced and Dilated Address Space (DDAS)***. Within DDAS, we introduce address space ***displacement***, a low cost, one-time shift of the base of the code segment. To obfuscate the relative distance between code objects, we introduce address space ***dilation***, the insertion of undefined memory regions at an instruction-level granularity.

The design of DDAS permits high entropy, fine-grained protections. Figure 1 illustrates the obfuscation of a function in DDAS. Since displacement occurs in the superimposed address space rather than in virtual memory, the code segment can be relocated anywhere. Thus, we are able to shift the

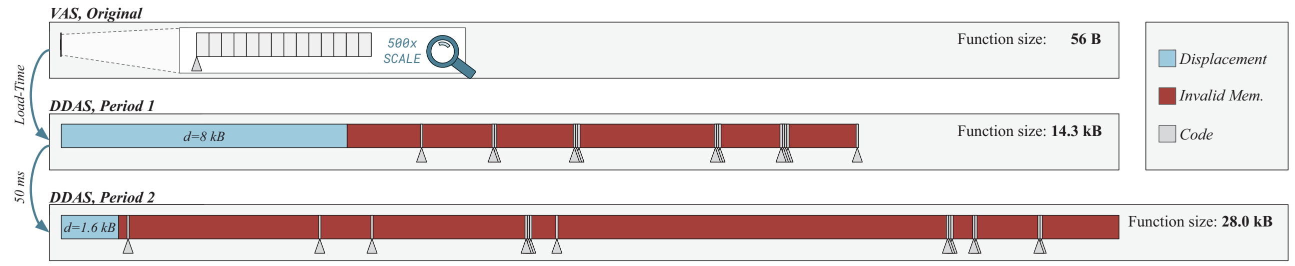


Fig. 1: **DDAS Overview.** At load time, a 56 B function is displaced by 8 kB and dilated to 14.3 kB within the Displaced and Dilated Address Space (DDAS). Invalid memory regions are inserted between instructions and trigger security exceptions when accessed. Every 50 ms, the function layout re-randomizes, changing in both size and location. In DDAS, this function can be maximally inflated to 14 MB, and the entire address space can be dilated by over 250,000 times. DDAS addresses are derandomized before entering the virtual memory system, thereby eliminating performance impacts on the memory system.

address space by a large, 64-bit key. To implement dilation, we leverage the vast unused portions of the address space. In RISC-V, the upper sixteen and lower two bits of code pointers are unused, implying that we can dilate memory by 2^{18} times with no loss of usable space. Similar spans of unused addresses are available in the ARM and x86-64 instruction sets.

Lastly, we defend against memory disclosures by employing both in-memory traps and runtime re-randomization. Since we dilate memory programmatically, we can detect accesses to the undefined memory regions that interleave instructions, effectively creating traps that detect attackers' attempts to leak code pointers. When these regions are accessed, the system triggers an ***immediate security exception***. Our analysis shows that, on average, 99.996% of DDAS consists of untouchable holes. Therefore, attackers have less than a 0.01% likelihood of randomly guessing a valid code pointer (*i.e.*, a pointer to addressable memory) without detection. To protect against potential memory disclosures, we re-randomize the addresses of code objects at runtime, effectively hardening this technique against advanced probing and side-channel attacks that have the potential to derive obfuscated memory locations.

By implementing DDAS in hardware, our defense is transparent to programmers while providing high entropy protections with low overheads. DDAS is integrated in the processor pipeline via custom functional units for pointer translations and secure storage for keys that parameterize the memory layout. Additional modifications, including a second lightweight core, are used to implement re-randomization. We maintain the program counter and indexing of microarchitectural structures as hardware-derandomized, virtual address space (VAS) pointers, allowing us to completely eliminate potential performance impacts within the memory system. With hardware support, our defense has negligible performance overheads, at $\sim$1% with re-randomization every 50 ms, and it makes control-flow hijacking attacks impracticable to execute.

A. Contributions of This Work

With Displaced and Dilated Address Spaces, we boost uncertainty in code addresses to a point where it is incredibly difficult to *i)* forge code pointers and *ii)* manipulate code pointers using relative distance information, effectively thwarting all known forms of control-flow attacks in the control plane. In this work, we make the following contributions:

- We introduce displacement to obfuscate absolute code locations by shifting the address space by a 64-bit key. We introduce dilation to create a super-inflated code plane and detect code pointer corruption, dilating consecutive instructions by a 100 kB untouchable hole, on average.
- We present a Displaced and Dilated Address Space (DDAS) implementation on RISC-V that can be maximally inflated to over 250,000 times the size of the existing virtual address space. We also propose DDAS-R (with re-randomization), where the memory layout is reconfigured under running programs to thwart memory disclosure vulnerabilities.
- We show that our proposed DDAS-R implementation (Table-Based, 2k-entries) introduces 55 bits of entropy for relative locations and 64 bits of entropy for absolute locations, and induces minimal overheads, with a 1.07% average slowdown at 50 ms re-randomization for the SPEC CPU2006 benchmarks.

This paper is organized as follows. Section II presents our address dilation technique. Section III details the hardware implementation, while Section IV provides support for runtime re-randomization. Section V presents our experimental setup, with Sections VI and VII analyzing the security and overheads (*i.e.*, performance, power, and silicon area) of our defense. This analysis is accompanied by a comparison to related work in Section VIII. Section IX concludes this paper.

II. DDAS: Displaced and Dilated Address Space

To defend against code reuse attacks in the control plane, we express all code pointers in a superimposed address space, termed the ***Displaced and Dilated Address Space (DDAS)***, layered atop the existing virtual address space (VAS). DDAS is addressable by 2^{64} distinct pointers, whereas the underlying VAS is only addressable by 2^{46} distinct pointers in RISC-V. We leverage this fact to ***dilate*** the address space by over 250,000 times through the insertion of ***undefined memory locations*** in the DDAS space. These regions do not exist as accessible memory in the virtual address space, therefore any access to these holes results in an immediate security

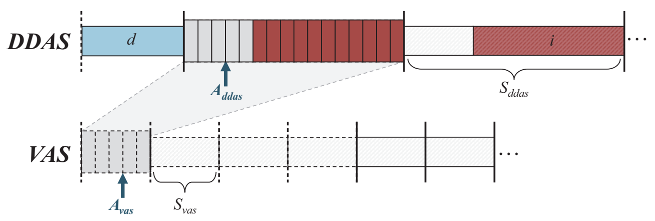


Fig. 2: **Basic DDAS Design.** The VAS is divided into segments of size S_{vas}. Each segment is mapped into DDAS and dilated by an i byte hole to create a DDAS segment of size S_{ddas}. The address space is shifted by d bytes.

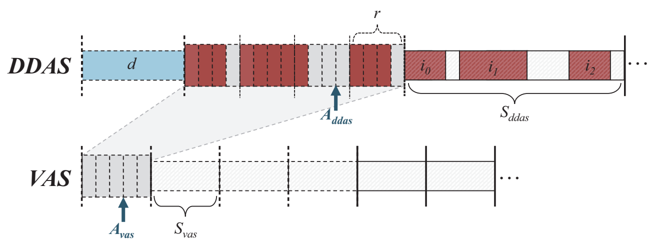


Fig. 3: **Table-Based DDAS Design.** Varying sized holes (i_0-i_n) that sum to i bytes are inserted between the valid addresses in each VAS segment. Each segment is divided into smaller ranges of size r to efficiently track the locations of holes in the Range-Map table.

exception. Thus, to disclose or manipulate a code pointer, one must guess the value of the code pointer through high-entropy defenses without accidentally touching an undefined memory location. This approach makes it incredibly unlikely for an attacker to successfully disclose or manipulate a code pointer. Combined with runtime re-randomization (Section IV), this challenge is coupled with a countdown clock of typically 50 ms to exfiltrate a pointer and synthesize an attack.

In this section, we first overview our threat model for single-gadget code reuse attacks on the control plane. Next, we present a simple, function-based DDAS implementation that dilates the address space uniformly. Then we examine a table-based DDAS implementation that sacrifices simplicity to gain increased spatial diversity. We then present constraints on these implementations that allow for more efficient translations.

A. Threat Model

The Displaced and Dilated Address Space (DDAS) is designed to protect against code reuse attacks on the control plane. In this effort, we focus only on control-plane attacks–although adapting DDAS to data pointers has the promise to also thwart data-plane attacks like DOP [27]. Code injection attacks are out of scope, as we assume existing protections like W⊕X [28] are in place. In our attack model, a powerful adversary is attempting to execute code gadgets, where the execution of a single gadget is indicative of a successful attack. The attacker can access a memory disclosure vulnerability to leak arbitrary information from the system and is able to analyze the program binary to derive the code layout. In addition, they have access to a memory corruption vulnerability that can be used to bootstrap gadget chains. While the attacker is knowledgeable about DDAS and the current re-randomization rate of the system, the keyed configuration of DDAS is secret at the start of the attack. Lastly, the system's random number source is assumed to produce true, non-repeating random numbers that cannot be predicted by the attacker.

B. Basic DDAS Translation

The DDAS memory configuration is determined programmatically and all code pointers in memory are expressed as DDAS pointers. Consider a simplified DDAS memory layout that lends itself to efficient dilation and compression, as shown in Figure 2. Here, the address space is shifted by a single displacement and divided into constant sized segments that are dilated uniformly by an undefined memory region (*i.e.*, a hole), shown in red. The layout of DDAS is parameterized by three keys: the displacement size, d, the segment size, S_{vas}, and the dilation amount per segment, i.

In DDAS, all code pointers are represented as 64-bit DDAS values. Thus, whenever a code pointer is created, it must reflect the current DDAS configuration. Whenever a code pointer is used, it must be translated from a DDAS address (A_{ddas}) to a VAS address (A_{vas}) to maintain correctness since memory remains unmodified (*i.e.*, indexed by VAS). Additionally, a check must be performed to assert that the pointer does not access an undefined memory region. These operations are performed with the following functions:

VAS→DDAS Translation:

$$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) \tag{1}$$

DDAS→VAS Translation:

$$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas} - d}{S_{ddas}} \right\rfloor * i) \tag{2}$$

Valid Memory Location Assertion:

$$(A_{ddas} - d) \bmod S_{ddas} \ < \ S_{vas} \tag{3}$$

Using these equations, we can translate between the VAS and DDAS segments and verify code pointers. If the above assertion ever fails, this indicates that a code pointer has been manipulated and a security exception is thrown.

C. Table-Based DDAS Translation

Although basic DDAS is effective at inflating the address space, there is little spatial diversity as i is a constant, single hole. To increase the entropy of our defense, we present a table-based memory configuration, shown in Figure 3, where each segment is recursively divided into a *range* of size r. The i byte inflation is now distributed throughout these ranges as multiple, distinct holes of varying sizes. This increases diversity by varying both the size and location of holes across the code space. The configuration of the holes within a segment is recorded by the *Range-Map table*. To translate from DDAS→VAS, the address's DDAS segment offset is used to index the table, T, returning the size of the holes within the segment that precede the queried address. This lookup also returns the size of the hole in the address's range, denoted

T_r, which is used to validate the DDAS code pointer. In our implementation, we index T and T_r with the address's range to minimize the total number of table entries required. To translate from VAS→DDAS, the address's VAS segment offset is used to index a separate, analogous table, T'.

Under this implementation, the layout of DDAS is parameterized by the Range-Map table, as well as the three keys used previously, d, i, and S_{vas}, and a new key, r. The operations for translating code pointers between DDAS addresses (A_{ddas}) and VAS addresses (A_{vas}) and asserting that code pointers are valid are given by the following functions:

VAS→DDAS Translation:

$$A_{ddas} = A_{vas} + d + (\left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor * i) + T'[A_{vas} \bmod S_{vas}] \quad (4)$$

DDAS→VAS Translation:

$$A_{vas} = A_{ddas} - d - (\left\lfloor \frac{A_{ddas} - d}{S_{ddas}} \right\rfloor * i) - T[R_{ddas}] \quad (5)$$

Where the DDAS range number (R_{ddas}) is calculated via:

$$R_{ddas} = \lfloor ((A_{ddas} - d) \bmod S_{ddas})/r \rfloor \quad (6)$$

Valid Memory Location Assertion:

$$(A_{ddas} - d) \bmod S_{ddas} \bmod r \ \geq \ T_r[R_{ddas}] \quad (7)$$

With this implementation, we can support dilation between every instruction in the code segment. Each table entry must correspond to a single 4 B instruction in the virtual address space to dilate at an instruction-level granularity. Since the table is implemented in hardware and its size is fixed, this constraint implies that S_{vas} can be no more than four times the number of table entries. We analyze the performance-security tradeoffs of different table sizes in Section VI and Section VII.

D. Architecting an Efficient DDAS Implementation

The keys that parameterize DDAS can be constrained to match the desired performance-security trade-offs of the application. In this work, we optimize the DDAS→VAS translation and valid memory location assertion as these functions are computed on every indirect jump. By constraining S_{ddas} to be a power of two, the division operations in Equations 2 and 5 are replaced by a right shift of $\log_2(S_{ddas})$ bits, and the modulo operators in Equations 3, 6, and 7 are replaced by a bitmask that selects the upper $64 - \log_2(S_{ddas})$ bits. These values are stored locally for efficient computation.

Constraining S_{ddas} removes lengthy operations on the critical path. Furthermore, by constraining other key parameters, we can further simplify outstanding multiply, divide, and modulo operations. For example, r must be a power of two since it is a divisor of S_{ddas}. This constraint further simplifies Equations 6 and 7. We can similarly constrain the hole size, i, to be a power of two to eliminate the multiplication in Equations 1, 2, 4, and 5. We employ this optimization when a multiply instruction is on the critical path, which occurs only for the basic DDAS configuration.

III. Hardware Implementation

We implement DDAS on a modified RISC-V architecture [29]. With hardware support, we achieve a strong layer of isolation between the software and the secret keys that parameterize DDAS. The parameters of the keyed configuration and Range-Map table are stored in dedicated hardware registers guaranteed to remain secret. Additionally, hardware allots reduced performance overheads on running programs. As DDAS only impacts indirect jumps during program execution, the pipeline only needs to be instrumented with a small, dedicated functional unit to translate code pointers before use. Although indirect jumps make up a small percent of instructions (about 2.4% in our analyzed workloads), instrumenting binaries to perform pointer translations before indirect control transfers would be costly and could expose the DDAS memory configuration to user programs, further motivating our hardware-based implementation. In addition to presenting load-time protections below, we present extensions that support runtime re-randomization in Section IV.

A. Extending the RISC-V Architecture to Support DDAS

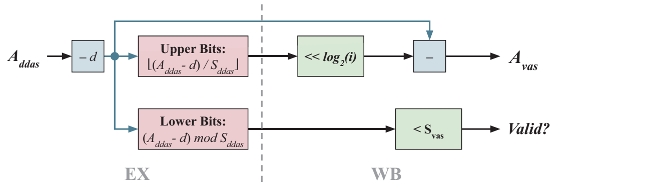
The use of unaligned, 64-bit DDAS code pointers introduces a layer of indirection that requires pipeline modifications to ensure correct control flow. To reduce the impact on direct jumps and instruction fetch, the program counter (PC) is maintained as a VAS value. Thus, no modifications to these operations are required. Further details on how the modified code pointer representation effects program execution are below.

1) Direct Jumps (JAL/BR): Direct jumps add a constant offset to the program counter, encoded in an immediate. Since we maintain the program counter in VAS, no modifications are made for direct jumps. To our knowledge, this does not have security implications as direct jump targets are encoded in the instruction and cannot be modified by malicious inputs.

2) Indirect Jumps (JALR): Indirect jumps handle function calls and returns, as well as other arbitrary control flow, by loading a code pointer from a register into the program counter. With DDAS, all code pointers stored in registers are DDAS values. Therefore, before loading the register value into the program counter, the pointer must be asserted as valid and translated to its VAS equivalent. This translation occurs implicitly in the microarchitecture through a ***dedicated functional unit***, detailed in Section III-B. Additionally, when the link register is written, the microarchitecture translates the $(pc + 4)$ link address from a VAS to a DDAS address.

3) Code Pointer Provenance (LUI/AUIPC): The targets of function calls are formed at runtime via a `lui` (or `auipc`) and `addi` instruction sequence. To generate a DDAS pointer rather than a VAS pointer, the compiler relocates code pointers to the global segment, which are then translated to DDAS values by the program loader. This implementation allows us to configure DDAS pointers without injecting backdoor instructions that perform VAS→DDAS translations at runtime.

4) Code Pointer Arithmetic (ADD/ADDI/etc.): DDAS is designed to thwart attacks that use relative distance information and arithmetic to forge code pointers. While DDAS



A_{ddas}
$-d$
Upper Bits: $\lfloor (A_{ddas} - d) / S_{ddas} \rfloor$
$<< log_2(i)$
$-$
A_{vas}
Lower Bits: $(A_{ddas} - d) \bmod S_{ddas}$
$< S_{vas}$
Valid?
EX
WB

Fig. 4: **Functional Unit for Basic DDAS.** This unit computes Equations 2 and 3 in parallel for a given code pointer, A_{ddas}.

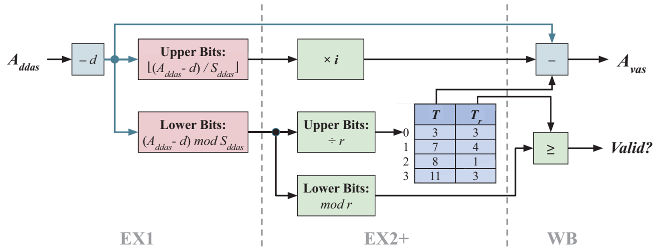


Fig. 5: **Functional Unit for Table-Based DDAS.** This hardware unit computes Equations 5 and 7. The Range-Map table, indexed by the range number, returns T for the A_{vas} calculation and T_r for the valid pointer assertion.

does not prohibit pointer arithmetic, it randomizes the result. Using a modified code pointer will result in an access to an unintended valid memory location or, with high probability, to an invalid hole. Thus, code pointer arithmetic is ill advised in DDAS. We argue that arithmetic on code pointers is not normal program behavior, as the C1x Standard [30] does not allow arithmetic operations on pointers to function types or incomplete types, like `void*` (§6.5.6-2 & §6.2.5.1). Although the GNU extension for C does permit operations on code pointers, DDAS supports this at the program's peril.

B. Integrating DDAS Support into the Processor Pipeline

To maintain correctness, all indirect control flow transfers are preceded by a DDAS$\rightarrow$VAS translation and valid pointer assertion. We instrument the execute stage of the pipeline with a dedicated functional unit to perform this computation for each indirect jump target (A_{ddas}). The latency of this translation is masked by the writeback stage, which is idle for indirect jump instructions. Figures 4 and 5 show the functional units for basic DDAS and table-based DDAS, respectively. The Range-Map table is implemented as a fixed-size memory structure with 8 B entries. We evaluate the exposed latency of these functional units with different table sizes in Section V.

1) Key Generation: The DDAS memory configuration is determined by secret parameters generated at load-time. For basic DDAS, a true random number generator (TRNG) is queried to determine the three keys: d, i, and S_{vas}. These values are stored in dedicated hardware registers. For table-based DDAS, r and a Range-Map key are generated in addition to those above. The Range-Map key is used to seed a pseudorandom number generator (PRNG) to deterministically populate the Range-Map table. This optimization allows for efficient context switching, as described below.

2) Context Switching Support: To be a viable defense in modern systems, the DDAS hardware must support context switching between individual processes that have unique DDAS configurations (*i.e.*, separate keys). During a context switch for basic DDAS, its keys are encrypted under a boot-time hardware key before being written out to memory. Context switching for the table-based implementation requires storing both the range size, r, and the Range-Map key that seeds the PRNG to generate the Range-Map table. When a process switches back in, the offloaded DDAS parameters are loaded and decrypted to be used again. For the table-based configuration, the Range-Map key seeds the PRNG to then generate the same table layout as before. With this optimization, the contents of the Range-Map table do not need to be encrypted and written to memory, saving a considerable amount of time given our large table sizes.

IV. DDAS-R: Runtime Re-Randomization

The research community has frequently cited re-randomization as a protection mechanism for both derandomization probes and memory disclosures [9], [10], [11], [13], [17], [18]. Additionally, re-randomization is effective at deterring multi-stage attacks that require the discovery and use of numerous code gadgets. DDAS with runtime re-randomization (DDAS-R) periodically re-keys the memory configuration and updates all code pointers accordingly with minimal stalling of running programs. This creates a hostile address space that inflates and contracts, with moving undefined memory locations that must be avoided.

When a re-randomization cycle begins, the pipeline is flushed to resolve in-flight instructions that contain stale code pointers. Then, custom hardware generates new keys to parametrize the re-randomized DDAS memory layout. These keys are loaded into dedicated hardware structures and the code pointers in registers are updated to reflect the new keys. At this point, program execution is resumed. All code pointers in the caches and DRAM are currently stale, *i.e.*, use the old key set. In parallel with program execution, these values are updated by the DDAS Remapper– a small, finite state machine that is responsible for pointer updates. The Remapper uses a DAS$\rightarrow$VAS and subsequent VAS$\rightarrow$DDAS functional unit to translate a pointer from the old to new key set.

A new re-randomization cycle can commence as soon as the memory scan in the prior period terminates. In our analysis, most benchmarks take no more than a few milliseconds to re-randomize, depending on their memory footprint. However, operating at high re-randomization rates can incur increased performance overheads (Section VII). In this work, we recommend a 50 ms re-randomization period because it is sufficiently faster than existing exploits without introducing much overhead. Namely, JIT-ROP exploits have been recorded to take 2.3 seconds or more [5], whereas Blind-ROP takes a minute on `nginx` [6]. Additionally, other interval-based re-randomization defenses (*e.g.*, Shuffler [13] and Morpheus [11]) operate at 50 ms with this justification.

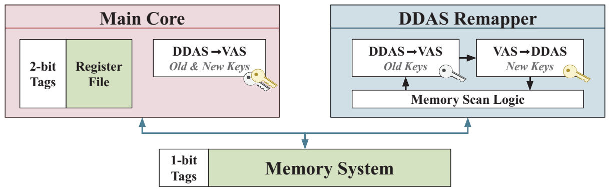


Fig. 6: **DDAS-R Hardware.** The DDAS→VAS FU and pipeline registers are extended to accommodate the mixed state of pointers during re-randomization. The DDAS Remapper uses tagged memory to identify and update stale code pointers.

A. *Hardware Support*

Hardware extensions to support re-randomization are shown in Figure 6. Similar to prior work [11], tagged memory is used to locate code pointers and identify the stale code pointers that require updating. The L1 and L2 caches are extended by a 1-bit tag for every pointer-width word (8 B). In physical DRAM, these tags are stored in a fixed block determined statically by the physical address. Tag information is obtained by an LLVM IR pass and loaded into the system via binary metadata. Lastly, the pipeline is extended to support the propagation of tag information throughout the system.

The processor is extended to include the DDAS Remapper, a small hardware unit responsible for key generation and code pointer updates. When a periodic re-randomization cycle begins, the DDAS Remapper generates a new set of DDAS keys and loads them into the associated structures before program execution resumes. The remapper then scans tag information sequentially to identify stale code pointer values, which are loaded into the remapper and updated using internal functional units. The new pointers are written back via a cache-coherent bus between the L1 and L2. The current VAS address being processed, termed the *threshold* value, is stored to indicate the progress of the remapper. All memory values above this address have been updated with the new key.

Since both updated and stale code pointers exist in memory during re-randomization, the pipeline must accommodate this mixed state. The functional unit is expanded to store the new and old key sets, and, for table-based DDAS, the block size of each Range-Map entry is doubled from 8 B to 16 B. During a load, code pointers are compared to the threshold value to determine if they are using the new or old key set. This result is reflected in a 1-bit register tag, which is fed into the functional unit to select the corresponding new or old key set and table entry, thereby maintaining correctness. Lastly, when a stale code pointer is stored to a region of memory that has since been updated, it must be updated with the new key set before being written back. Our simulations show that this case is rare and has negligible performance impacts.

B. *Support for Context Switching in DDAS-R*

When the DDAS Remapper is idle during a context switch (*i.e.*, all code pointers are updated), support for context switching is equivalent to DDAS without re-randomization (Section III-B). However, when a context switch occurs during a re-randomization cycle where the remapper is active, the complete DDAS context must be encrypted and stored to memory. This information includes both the new and old sets of DDAS keys and the threshold value, which indicates the progress of memory updates in the current re-randomization cycle. When a context is switched-in, the threshold value is loaded into the DDAS Remapper and re-randomization begins from this virtual address. Since a process may be switched-out for longer than the specified re-randomization period (*e.g.*, 50 ms), the remapper initiates another re-randomization cycle immediately upon completion to ensure that any code pointer's lifetime in an old key set is as short as possible.

V. Experimental Methodology

We implemented DDAS on a RISC-V out-of-order core in gem5 [31], [32] in system call emulation mode with the parameters listed in Table I. We modeled the memory contention for tagged memory and pointer updates using DRAMSim2 [33]. To determine the latency of the DDAS functional unit, we modeled the Range-Map table in CACTI 7 [34] at the 28 nm technology node. To analyze the security and performance trade-offs of our defense, we examined three distinct DDAS configurations with load-time and runtime randomization– *1)* Basic DDAS where i is restricted to a power of two (1 cycle of exposed latency), *2)* Table-based DDAS with a 2k-entry table (2 cycles of exposed latency), and *3)* Table-based DDAS with a 32k-entry table (4 cycles of exposed latency).

Core Type	2.5GHz, O3CPU (Out-of-Order)
Superscalar	4-wide
Cache Line Size	64B
L1 Cache Size	32KB with 2-cycle latency
L2 Unified Cache Size	256KB with 20-cycle latency

TABLE I: **Experimental Configuration.** The specifications of our baseline RISC-V out-of-order core in gem5 [31].

A. *Compiler Support and Benchmarks*

We implement our compiler passes in LLVM 5.0.0 configured for the RISC-V RV64IMA architecture [35]. All programs were compiled with optimization level -O2 and linked against the RISC-V Musl C library [36]. Function pointers that are normally formed by arithmetic code sequences during runtime are relocated to the global segment so they can easily be randomized by the gem5 loader or by the DDAS Remapper. We have found that this simple compiler change on average speeds up programs by $\sim$1.85%. We also analyzed the size increase of the binaries and found that they only increase by 1.8% on average. Since one could migrate code pointers to the static global section, we lend this optimization to the baseline in order to reveal the performance overheads incurred as a result of the DDAS hardware. We evaluate DDAS on the first 3 billion instructions of the SPEC CPU2006 C-code benchmarks [37] with the reference input: *perlbench*, *bzip2*, *gcc*, *mcf*, *milc*, *gobmk*, *hmmer*, *sjeng*, *libquantum*, *h264ref*, *lbm*, and *sphinx3*.

VI. Security Analysis

DDAS prevents code reuse attacks on the control plane by obfuscating the absolute and relative addresses of code objects. Unlike prior approaches, which permute code objects or encrypt pointers, DDAS does more than obfuscate the code segment. DDAS also catches when a pointer is overwritten by detecting any access to the undefined regions of memory that make up a majority of the address space. Notably, in table-based, 2k-entry DDAS, more than 99.996% of the address space is undefined memory on average. Of the remaining 0.004%, only a fraction is mapped and in the code segment. Thus, there is a high likelihood that any attempt to forge a code pointer will result in an exception from either accessing an invalid memory region, an unmapped region of memory, or the data segment. Additionally, frequent re-randomization of the memory layout during runtime prevents the reuse of leaked code pointers, effectively thwarting control-plane attacks.

Below we qualitatively reason about how DDAS prevents sophisticated code reuse attacks, including return-oriented programming (ROP) [2] and its variants, as well as Spectre Variant 2. Since DDAS is a probabilistic defense, the subsequent sections quantify the entropy of DDAS to demonstrate its effectiveness at stopping these control-plane exploits.

A. Qualitative Security Analysis for Classic Attacks

ROP: To dispatch ROP gadgets, a memory corruption vulnerability is exploited to overwrite a return address with a code pointer. In DDAS, this pointer is unknown and randomized every 50 ms on average. Additionally, if DDAS is maximally inflated, the attacker has less than an 0.01% chance of generating a valid code pointer, *i.e.* one to addressable memory. Variants of ROP that exploit other control flow statements, like jump-oriented programming [3], also fall under this category.

JIT-ROP: JIT-ROP [5] uses a leaked code pointer to read code pages and construct gadgets at runtime. Using a leaked DDAS pointer as the address of a load will result in an unintended memory access because the pointer will not be translated to VAS. Furthermore, a single leaked pointer cannot be used to entirely derandomize the address space. Even if an attacker can leak enough information to create a gadget chain, they must complete their attack before the next re-randomization cycle as all pointers will become stale.

Blind-ROP: Blind-ROP [6] repeatedly overwrites a single byte of the return address in a child process until the process does not crash, signaling that the guess was correct. This process is repeated for each subsequent byte to recover the entire pointer. With DDAS, these guesses will trigger exceptions with high probability. Additionally, this attack requires on average 1,500 requests to discover the ASLR offset and was recorded to have taken 1 minute on *nginx*. We randomize 1,200 times faster than the length of this attack at a period of 50 ms.

Spectre Variant 2: In Spectre Variant 2 [38], an attacker process mimics the control flow of a victim process to train the branch predictor to mis-speculate on a chosen indirect jump. This causes the victim process to execute an attacker-specified code gadget. In this exploit, an attacker must know the address of the target indirect jump instruction in the victim program. In DDAS, the attacker and victim process has different shifted and dilated code segments, preventing attackers from being able to understand and mimic the victim's jump patterns.

	Average Inflation per Insn	Average Inflation per Page	Average Percent of Holes	Relative Entropy
Basic	78.9 kB	80,650 kB	99.994%	54.89 bits
Table-Based, 2k	107.8 kB	111,341 kB	99.996%	55.30 bits
Table-Based, 32k	111.3 kB	111,784 kB	99.996%	55.31 bits

TABLE II: **Evaluation of DDAS.** The average inflation per instruction or page is the mean dilation incurred by a jump of that length in VAS. The table also includes the average percentage of memory occupied by untouchable holes and the calculated relative entropy (Section VI-B2) per configuration.

B. Quantitative Security Analysis of DDAS Entropy

Since DDAS provides probabilistic security, we quantitatively analyze the strength of our defense for our three studied configurations. Specifically, we analyze the likelihood an attacker can forge a valid code pointer at runtime. In our analysis, we run a 100,000 trial Monte Carlo simulation, where each trial has a randomly generated DDAS key set. For each trial, we record the amount of inflation incurred by an indirect jump in the VAS segment, starting from a jump to the next instruction. Additionally, for each indirect jump distance, we measure the inflation from 100 randomized starting locations per trial. We then use these measurements to derive statistics about the overall inflation of the control plane, including the likelihood that a specific jump will encounter no dilation.

1) Inflation of the Control Plane: Our results, summarized in Table II, show that both table-based configurations inflate the address space more than basic DDAS, dilating a 4 B jump by 107.8 kB or 111.3 kB on average, compared to 78.9 kB. The standard deviation of these metrics is also telling. For table-based DDAS with 2k entries, a 4 B jump has a standard deviation of 2086 kB, implying that there is a large variance in the inflation of the code segment, even for very short jumps. Page-length jumps were inflated by over 111,000 kB on average with a standard deviation of 200 MB. These results suggest that a malicious code gadget cannot be more than a few bytes away. Otherwise, the high entropy of dilation forces attackers to resort to brute-force guessing. Furthermore, guessing a pointer value subjects an attacker to accessing an untouchable hole and triggering an immediate security exception. Our analysis shows that a large portion of the address space is filled with these untouchable holes. Namely, in basic DDAS, 99.994% of memory on average contains untouchable holes, compared to 99.996% for both table-based configurations.

2) Entropy of Indirect Jumps: To measure the entropy of our defense, we consider the obfuscation of both absolute and relative addressing in the control plane. We define ***absolute entropy*** as log_2 of the average bytes of displacement. The uncertainty of the absolute location of code objects is attributed to the displacement key, d, which is 64 bits in length, giving

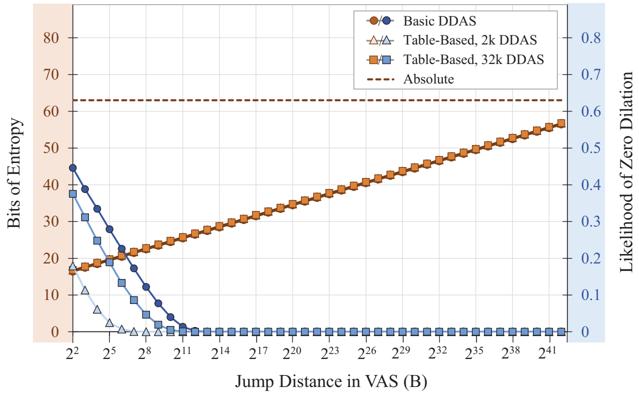


Fig. 7: **Entropy of Indirect Jumps.** The primary axis, in orange, shows the entropy of indirect jumps, where entropy is modeled as log_2 of the dilation in bytes. The secondary axis, in blue, shows the probability that a given configuration will *not* inflate an indirect jump.

a mean displacement of 2^{63} bytes. Thus, the absolute entropy is 63 bits. Similarly, we define ***relative entropy*** as log_2 of the mean bytes of dilation, averaged across varying jump distances. Previously, we quantified the entropy of dilation for specific jump distances, as shown on the primary axis of Figure 7. To measure the relative entropy of our whole defense, we repeat this calculation across *all* jump distances and find that our defense has an average relative entropy of 55 bits. This metric captures the overall difficulty of guessing the dilation for a jump of *any* distance. We find that the relative entropy of DDAS is well above existing defenses, like Shuffler [13], which typically has no more than 27 bits of entropy.

3) Likelihood of Zero Dilation: One limitation of DDAS is that, when a region of memory experiences zero inflation, the manipulation of jump targets within that range will succeed. This prompts us to evaluate how often zero inflation occurs for randomly chosen key configurations. We performed this analysis using the above simulation, now recording the frequency of zero dilation. The result of this analysis is shown on the secondary axis of Figure 7. Basic DDAS always inflates jumps greater than 8 kB. Table-based DDAS performs better, inflating all jumps greater than 4 kB (a page). For jumps within a page, there is a small likelihood that no dilation occurs. Table-based DDAS with 2k entries achieves a 95% likelihood of dilation for jumps greater than 32 B. Table-based DDAS with 32k entries and basic DDAS only achieves this likelihood for jumps greater than 256 B and 1024 B, respectively. For comparison, a function in our benchmarks is 890 B on average. Thus, limited dilation for short relative distances can be tolerated and would restrict attackers to using code gadgets within the current function. Additionally, with re-randomization, these uninflated jumps do not persist for long, at only 50 ms on average.

Overall, the evaluated DDAS configurations have comparable entropy but varying amounts of inflation for short jumps. Notably, although table-based, 2k-entry DDAS has a smaller keyspace (S_{vas} is constrained by the number of table entries), it is more effective than 32k-entry DDAS at inflating short jump distances. This is because, as the VAS segment becomes larger, the possibility for consecutive valid memory locations increases, causing the probability of dilation for short jumps to decrease. Given these results, ***we recommend table-based DDAS with 2k entries***. We find the performance overheads for this configuration to be reasonable, as presented in Section VII.

C. Attacking DDAS

With the DDAS secure hardware, existing control-plane exploits will be unsuccessful due to the obfuscation of code pointers in a high-entropy space. Attackers may try to circumvent DDAS by either brute-force guessing pointer values or derandomizing the memory layout. These techniques have been used to bypass randomization-based defenses like ASLR [5], [6], [39], [40], [41]. In this section, we analyze the effectiveness of these methods to circumvent DDAS.

1) Brute-Force Attacks: Given a leaked pointer and knowledge of uninflated relative distance information from the program binary, an attacker can attempt to brute-force guess a DDAS pointer for their desired target. We use a hypergeometric distribution to model how long it takes to accurately guess the target pointer when the system is re-randomized every 50 ms (*i.e.*, the attacker has a set number of guesses until the system re-randomizes and all prior attempts are invalidated). To be conservative in our analysis, we assume an attacker's guess results in a delay only when it accesses an undefined memory location, triggering an immediate security exception and stalling execution for 68 μs for exception handling [42]. We assume these security exceptions are taken and *ignored*, allowing the attack to continue uninterrupted.

For basic DDAS-R, 61 hours of continuous probing is required to correctly locate a code gadget that is 4 kB away in the original binary. For table-based DDAS-R, 86 and 87 hours is required for a 2k and 32k-entry configuration, respectively. To realistically capture relative distances, we profiled indirect jumps in the first 3 billion instructions of the SPEC benchmarks [43]. More than 68 days of probing is required to guess a jump that is the average length in the *first quartile* (75.7 kB) for both table-based configurations. Probing time increases for larger target relative distances, with jumps in the fourth quartile taking thousands of years for all configurations.

2) Derandomization: Timing side-channels [40], [41] and memory disclosures [5], [7], [24] have been leveraged to derandomize address obfuscation protections. DDAS is resilient to timing side-channel attacks because it does not affect cache mappings, nor are DDAS addresses used to index into microarchitectural structures (*i.e.*, caches, TLB, BTB), which are the traditional venues for side-channel exploits. Furthermore, the latency of the DDAS functional unit is not key or data dependent and is implemented in constant time. Thus, timing an address translation similarly does not reveal information about the existing DDAS configuration.

As side-channels are not viable, the only way to derandomize DDAS is to tactfully leak pointer values. For basic DDAS, an attacker can derive the dilation amount, i, if they obtain two DDAS/VAS pointer pairs in consecutive segments, allowing them to guess pointers with an increased likelihood of success.

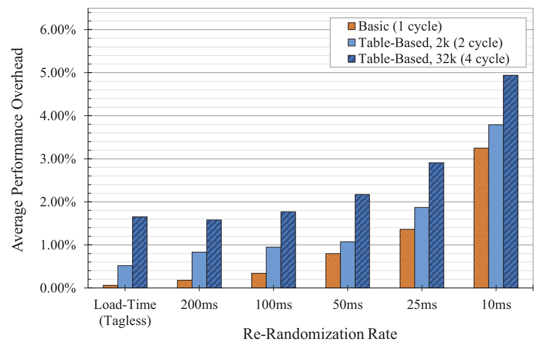


Fig. 8: **Average Performance Overheads.** The average overhead for the SPEC CPU2006 benchmark suite at varied re-randomization rates for the analyzed DDAS configurations.

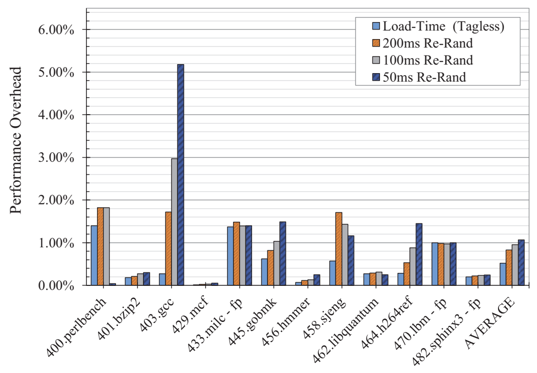


Fig. 9: **Overheads for Table-Based, 2k DDAS.** The performance overheads per benchmark for a table-based, 2k-entry DDAS configuration, with 2 cycles of exposed latency.

Although finding pointers in neighboring segments is difficult, an attacker can leverage the fact that the DDAS segment size is a power of two to narrow their search space. Table-based DDAS has more spatial diversity and therefore is more resilient to memory disclosures. With n ranges per DDAS segment, this previous technique exposes only $1/n^{th}$ of valid pointers in the address space. Furthermore, uncovering $1/n^{th}$ of the address space does not provide much of an advantage for derandomizing the rest of memory as an attacker does not know the range size or distance to the next range. Despite this, we regard memory disclosures as dangerous and employ runtime re-randomization to defend against derandomization.

By re-randomizing the DDAS configuration every 50 ms, attackers have a tight window to leak pointer pairs, perform additional probes, and synthesize an exploit. Prior work similarly uses re-randomization to shield from memory disclosures [9], [10], [13], [17], [18]. Additionally, we configure DDAS to re-randomize *immediately* when a security violation is detected. Thus, attempts to derandomize DDAS by leaking multiple pointers will fail, as leaked pointers will be invalidated by probing attempts that trigger security violations. To further harden our defense against memory leaks, the DDAS architecture could be expanded to include pointer encryption [44], [45], [46], [47]. Morpheus [11] and Shuffler [13] leverage encryption on top of address obfuscation to make leaked pointer values completely unusable. Obfuscating the virtual address space [8] or the program binary [45] is also viable, as derandomization attempts require knowledge of VAS pointers.

VII. Results and Evaluation

To measure the performance impacts of DDAS, we ran experiments against the SPEC CPU2006 C workloads [37], described in Section V-A, for the chosen configurations– basic (1 cycle latency), table-based, 2k-entry (2 cycles latency), and table-based, 32k-entry (4 cycles latency). Figure 8 shows the average performance overhead of these configurations for the analyzed workloads. The DDAS functional unit introduces latency in the pipeline that can delay branch prediction and cause bottlenecks. The overhead for a load-time (tagless) DDAS system illustrates the consequences of this added latency alone. To support re-randomization, tags for every pointer-width word must be stored and propagated in the memory hierarchy. Additionally, the re-randomization process accesses tag information and rewrites all code pointer values in the data segment. These updates invalidate code pointers in the data cache, increasing miss rates. Thus, compared to the load-time system, re-randomization incurs higher performance penalties and affects the memory system.

Our recommended configuration, DDAS-R with a 2k-entry Range-Map table, has reasonable slowdowns, at 1.07% during 50 ms re-randomization. A detailed breakdown of the overhead incurred by individual benchmarks for this configuration is shown in Figure 9. Performance overheads generally increase alongside re-randomization rates, especially for benchmarks with a large number of code pointers, *i.e. gcc* has 5.18% slowdowns at 50 ms. For these benchmarks, the DDAS Remapper must perform more pointer updates, invalidating values in the L1-cache and increasing both the number of read/write misses and the overall average miss latency for the main core. Some benchmarks with more `jalr` instructions experienced reduced slowdowns at faster re-randomization rates. This is likely a result of the DDAS remapper prefetching code pointers into the caches during re-randomization, effectively making these values more available to the core. However, as re-randomization rates increase, this benefit will likely diminish due to interference during pointer updates.

A. Area and Power Overheads

The hardware overhead of DDAS was estimated using McPAT 1.3 at the 28 nm technology node [48]. We modeled our system using an ARM Cortex-A9 out-of-order processor augmented with: *1)* the DDAS functional unit, *2)* memory system modifications (*i.e.*, tagged memory and the Range-Map table) modeled in CACTI 7 [34], and *3)* the DDAS remapper, estimated by the SiFive S51 standard core [49]. Table III shows the estimated area and power consumption for the analyzed DDAS configurations. The cost to implement table-based, 2k-entry DDAS is fairly low, at less than 3% area overhead for load-time randomization or 7% with runtime re-randomization Table-based, 32k-entry DDAS-R experiences much higher area overhead at nearly 40% because each table entry must double in size to accommodate both key sets.

	Area (mm^2)	Overhead	Power (W)	Overhead
Baseline	4.387	-	3.653	-
Basic DDAS	4.411	0.55%	3.674	0.57%
Basic DDAS-R	4.524	3.13%	3.682	0.80%
Table-Based DDAS, 2k	4.506	2.71%	3.704	1.40%
Table-Based DDAS-R, 2k	4.690	6.90%	3.804	4.13%
Table-Based DDAS, 32k	4.865	10.88%	3.850	5.39%
Table-Based DDAS-R, 32k	6.111	39.29%	4.370	19.61%

TABLE III: **Hardware Estimation.** The area and power estimates for the analyzed DDAS configurations.

B. Performance-Security Trade-offs

In this work, we constrain the parameters S_{ddas}, r, and i to be a power of two to make architecture-level optimizations in the functional unit. When removing these optimizations, the DDAS system has an increased keyspace, making it potentially more secure. Intuitively, it is harder to brute-force guess the value of keys like S_{ddas} in an unconstrained system. Derandomization becomes more challenging as attackers can no longer leverage the fact that keys are a power of two to constrain their search space. Yet, for a table-based, 2k-entry configuration, the unconstrained variant only adds 2.5 additional bits of relative entropy and exhibits a 99.5% likelihood of dilation for all jumps. Furthermore, this system incurs intolerable slowdowns, with 42 cycles of exposed latency in the pipeline (20 cycles per divide or modulo operation), compared to 2 cycles previously. At 50 ms re-randomization, this configuration incurs over 100% slowdowns over the baseline, which we regard as an intolerable performance degradation.

VIII. Related Work

The work related to address obfuscation protections is extensive. Here, we restrict our comparisons to the defenses that are most relevant, *i.e.* those that employ runtime randomization or in-memory traps. DDAS leverages hardware support to guarantee the necessary efficiencies while providing 63 bits of absolute entropy and 55 bits of relative entropy. With runtime overheads of 1.07% at a 50 ms re-randomization rate, DDAS outperforms many of its counterparts and offers increased security via finer grained, higher entropy protections.

Randomization of Absolute Addresses: Defenses have employed re-randomization to fortify Address Space Layout Randomization [8] against memory disclosures. These defenses re-randomize the displacement of the code segment in response to events that are indicative of an attack, like system calls [10] or process forks [9], or at regular intervals [11]. TASR [10] has high performance overheads of 30-40%. RuntimeASLR [9] performs better than DDAS, with 0.5% overheads, but does not obfuscate relative distances between code objects. Morpheus [11] leverages hardware to achieve fast address obfuscation, with 0.9% overheads at 50 ms randomization. However, Morpheus also does not randomize relative distances and thus can be derandomized by a single pointer disclosure.

Randomization of Relative Addresses: Permutation-based defenses are the leading approach to randomize relative distances between code objects. However, unlike DDAS, the entropy of this technique correlates with code size as only a fixed number of permutations exist, providing minimal protection for short functions. Remix [17] employs periodic re-randomization and is comparable to DDAS with a 2.8% overhead. However, Remix only randomizes relative distance within functions, not between them. DDAS and DDAS-R are able to achieve more entropy than these works and randomizes both intra-function and inter-function relative distance. We are only aware of two defenses, Shuffler [13] and Mixr [18], which obfuscate the relative distance *between* functions and re-randomize at runtime. However, these defenses suffer from higher overheads than DDAS, at 14.9% or 225% respectively, and have lower entropy (much less than 30 bits).

In-Memory Traps: In addition to code diversification, some defenses use traps to probabilistically detect attacks that probe the code segment. CodeArmor [25] is a software technique that instruments the program binary with “honey gadgets”, similar to undefined memory regions in DDAS. Attackers that mistakenly forge code pointers to these gadgets trigger three security exceptions on average before reaching their target. Using the same metric, an attacker would trigger over 100,000 exceptions in a DDAS system experiencing average dilation before reaching a code gadget. CodeArmor achieves low overheads, at 6.9% on SPEC CPU2006, but DDAS is able to provide overheads well below 5% with hardware support.

IX. Conclusions

To combat code reuse attacks on the control plane, we deter the manipulation of indirect jump targets by increasing the uncertainty of the code segment. We introduce the Displaced and Dilated Address Space, a superimposed address space where all code pointers in the program are expressed. We displace the address space by a 64-bit key to obfuscate *absolute* addresses. Additionally, we leverage the vast unused portions of the virtual address space to dilate memory by over 250,000 times, obfuscating *relative* addresses. We inject large, undefined memory regions between instructions that result in an immediate security exception when accessed. Finally, DDAS code pointers are derandomized in the pipeline before use, eliminating performance impacts on the memory system.

Compared to prior work, our defense achieves greater entropy in the control plane (63 bits for displacement, 55 bits for dilation), which attackers must overcome without touching the undefined memory locations that make up 99.996% of the address space on average. Additionally, with runtime re-randomization, this challenge is coupled with a time limit of 50 ms to synthesize an attack. By leveraging hardware support, we are able to implement this defense while keeping performance overheads well below 5%.

Acknowledgments

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] CVEs by Type. (2019) Cve details: Vulnerabilities by type. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012.

[3] 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*. ACM, 2011, pp. 30–40.

[4] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 385–399.

[5] 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 *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE, 2014, pp. 227–242.

[7] 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 *NDSS*, 2015.

[8] PaX Team. (2003) Pax address space layout randomization (ASLR). [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[9] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization." in *NDSS*, 2016.

[10] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 268–279.

[11] 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.

[12] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 339–348.

[13] 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.

[14] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *USENIX Security Symposium*, 2005, pp. 17–17.

[15] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 299–310.

[16] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense*. Springer, 2011, pp. 77–98.

[17] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-Demand Live Randomization," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16. New York, NY, USA: ACM, 2016, pp. 50–61.

[18] W. Hawkins, A. Nguyen-Tuong, J. D. Hiser, M. Co, and J. W. Davidson, "Mixr: Flexible runtime rerandomization for binaries," in *Proceedings of the 2017 Workshop on Moving Target Defense*. ACM, 2017, pp. 27–37.

[19] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 475–490.

[20] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[21] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.

[22] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 601–615.

[23] H. Xu and S. J. Chapin, "Improving address space randomization with a dynamic offset randomization technique," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 384–391.

[24] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing." in *USENIX Security Symposium*, 2014, pp. 433–447.

[25] X. Chen, H. Bos, and C. Giuffrida, "Codearmor: Virtualizing the code space to counter disclosure attacks," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 514–529.

[26] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[27] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[28] G. Shvets. (2018) Enhanced virus protection / execute disable bit.

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

[30] ISO/IEC 9899:201x, "Programming Languages – C," International Organization for Standardization, Geneva, CH, Standard, Apr. 2011.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[32] A. Roelke and M. R. Stan, "RISC5: Implementing the RISC-V ISA in gem5," in *Proceedings of Computer Architecture Research in RISC-V*, ser. CARRV '17, October 2017.

[33] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011.

[34] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, Jun. 2017.

[35] AndesTech. (2017, Sep.) Andes Technology GitHub - riscv-llvm. [Online]. Available: https://github.com/andestech/riscv-llvm

[36] rv8. (2018, Feb.) rv8.io github - musl-riscv. [Online]. Available: https://github.com/rv8-io/musl-riscv

[37] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[39] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 2004, pp. 298–307.

[40] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 40.

[41] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the mmu." in *NDSS*, vol. 17, 2017, p. 13.

[42] V. Bridgers, "Real Time Linux Scheduling Comparison," in *Embedded Linux Conference 2015 (ELC 15)*. CA: The Linux Foundation, 2015. [Online]. Available: http://events17.linuxfoundation.org/sites/events/files/slides/Real-Time-Linux-Comparison-Bridgers-ELC2015.pdf

[43] "Standard Performance Evaluation Corporation (SPEC). SPEC CINT2006 Benchmarks. http://www.spec.org/cpu2006/CINT2006."

[44] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard TM: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proceedings of the 12th conference on USENIX Security Symposium*, vol. 12. Berkeley, CA, USA: USENIX Association, 2003, pp. 91–104.

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

[46] 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: ACM, 2015, pp. 941–951.

[47] 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. New York, NY, USA: ACM, 2013, pp. 981–992.

[48] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480.

[49] SiFive, Inc. (2019) SiFive S51. [Online]. Available: https://www.sifive.com/cores/s51