Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces

Lauren Biernacki, Mark Gallagher, Valeria Bertacco, Todd Austin

University of Michigan, Ann Arbor

Introduction

- Many prevalent security exploits are underpinned by precise knowledge of the *memory layout* of the victim machine. Specifically, to thwart control-flow attacks*, **code pointers must be protected.***
- A corrupted code pointer can effectively redirect program execution to attacker-chosen code gadgets, giving attackers the necessary foothold to circumvent system protections.
- Attackers corrupt code pointers by ***exploiting absolute and relative distances between code gadgets on the control plane.***
- In this work, we introduce ***Displaced and Dilated Address Spaces (DDAS)***, which obfuscates both absolute and relative distances of code objects to thwart control plane attacks.

DDAS Overview

In a Displaced and Dilated Address Space (DDAS):

- Code pointers are decoupled from their true code location in the virtual address space (VAS) and remapped to a location in the superimposed Displaced and Dilated Address Space (DDAS).
- DDAS exploits the vast unused portions of the virtual address space to ***displace code pointers*** by a 64-bit key, and ***dilate the code segment*** through the insertion of untouchable, invalid memory regions at an instruction-level granularity.
- To eliminate performance impacts on the memory system, code pointers are derandomized prior to use. Code pointers verified (*i.e.,* that they do not access dilated, invalid memory) and are translated from DDAS to the true VAS location prior to indirect jumps.

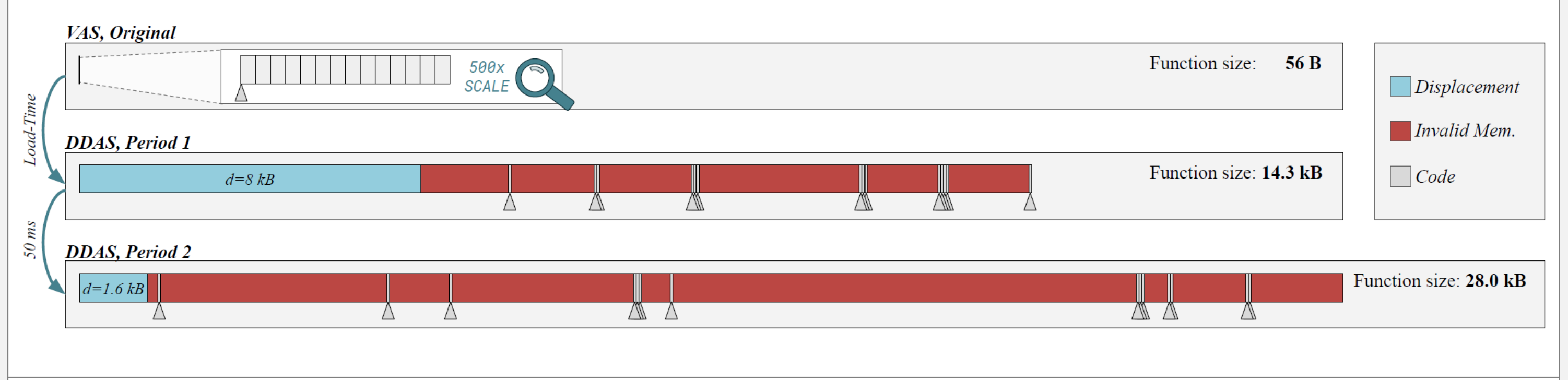

Figure 1. DDAS Overview. At load time, a 56 B function is displaced by 8 kB and dilated to 14.3 kB within the Displaced and Dilated Address Space. Invalid memory regions are inserted between instructions and trigger security exceptions when accessed. Every 50 ms, the function's layout re-randomizes, changing in size and location. In DDAS, this function can be maximally inflated to 14 MB, and the entire address space can be dilated by >250,000 times.

DDAS Memory Configuration

The Displaced and Dilated Address Space is configured ***programmatically*** by either a basic or table-based translation equation.

- The DDAS → VAS translation is determined by a function keyed by secret layout parameters (*e.g.,* size of displacement, size and location of dilation).
- The table-based translation serves to increase spatial diversity by varying the size and location of dilation across functions, however, the basic translation is presented below for simplicity.

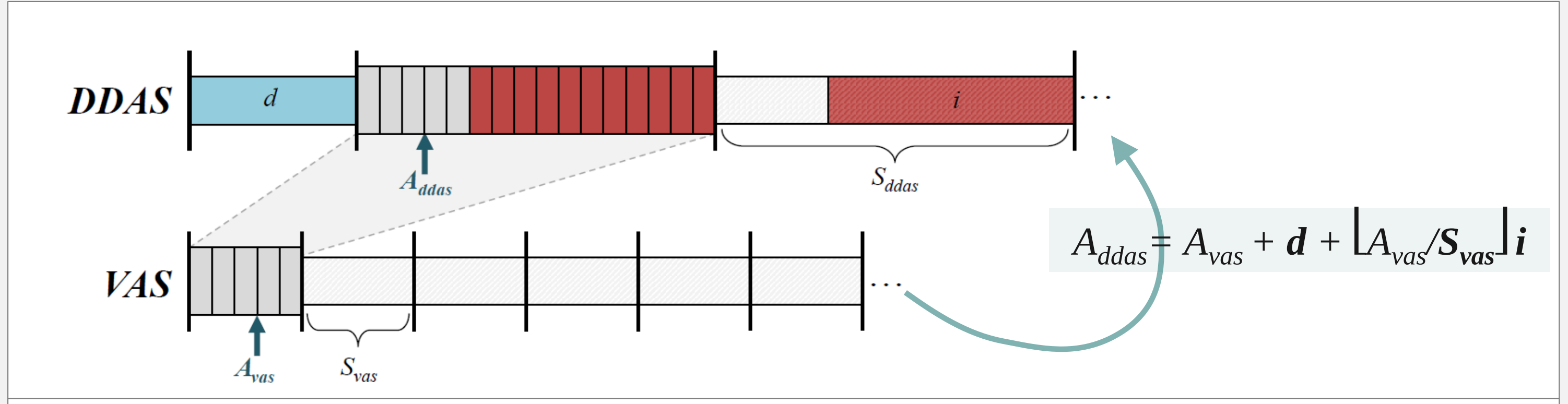
DDAS
d
i
A_{ddas}
S_{ddas}
VAS
A_{vas}
S_{vas}
$A_{ddas} = A_{vas} + d + \lfloor A_{vas}/S_{vas} \rfloor i$

Figure 2. Basic DDAS Configuration and Translation. 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 entire address space is shifted by ***d*** bytes.

Our defense leverages hardware support to achieve ***negligible performance overheads,*** at 1% with re-randomization every 50ms, while providing ***strong probabilistic guarantees against control-flow hijacking attacks***

DDAS Hardware Implementation

We implement Displaced and Dilated Address Spaces is a out-of-order RISC-V pipeline by instrumenting the execute stage to contain a ***functional unit for indirect jumps*** that performs the DDAS → VAS translation.

We also instrument the pipeline with support for ***runtime re-randomization*** of the DDAS memory configuration, including:

- Tagged memory to identify code pointer values
- Logic to generate new keys and update code pointers during runtime (termed *DDAS Remapper*)

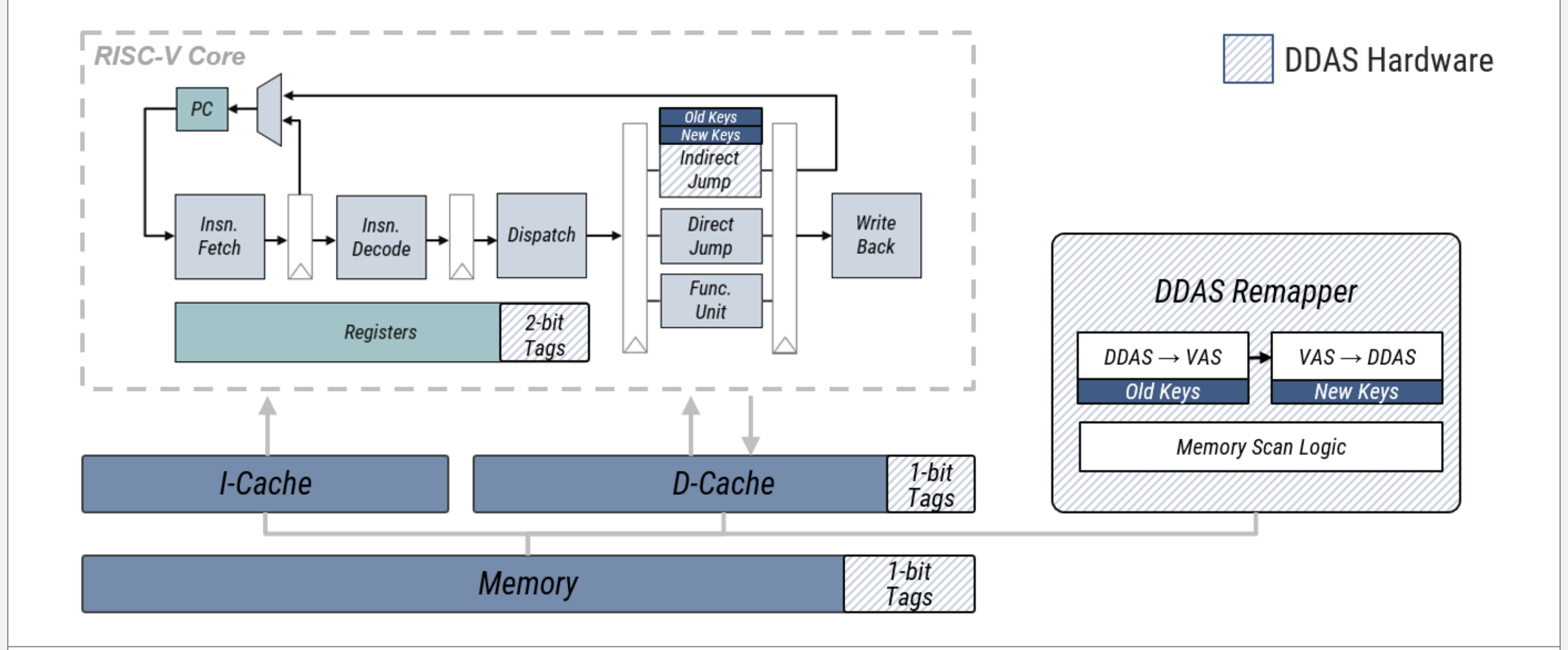

Figure 3. DDAS with Re-Randomization. The DDAS → VAS FU and register file are extended to accommodate the mixed state of pointers during re-randomization. The DDAS Remapper uses tagged memory to identify and update stale code pointers.

Evaluation

We prototyped DDAS control plane protections in gem5, a cycle accurate simulator, and ran experiments to measure the entropy and security of our defense for three different configurations with re-randomization.

- A jump to the next instruction was dilated by 100 kB on average
- Untouchable, invalid memory regions made up > 99.996% of memory on average, resulting in ***less than a 0.01% likelihood of forging a code pointer without detection***
- Performance overheads were well below 5% for all analyzed configurations, will similarly low power and silicon area overheads.

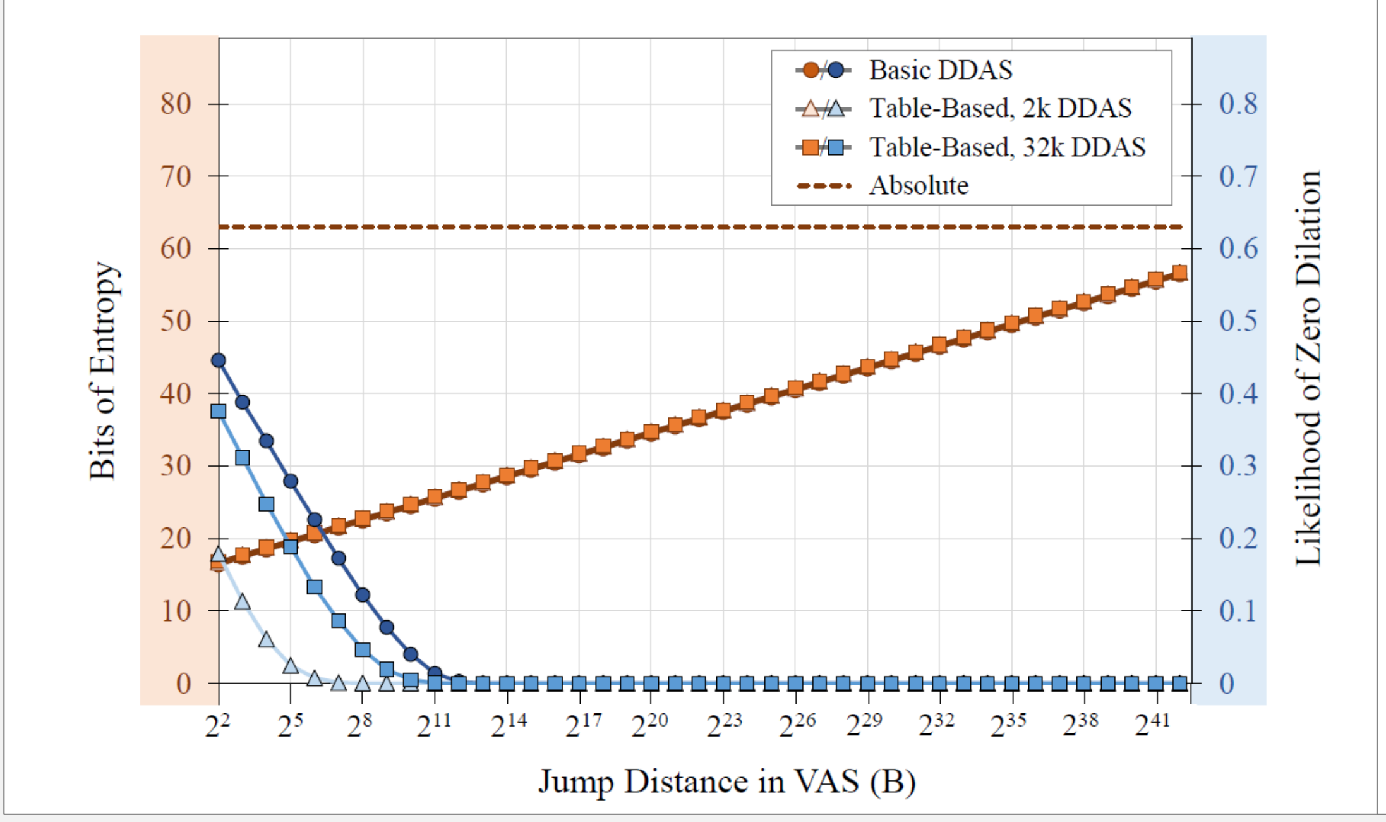

Figure 4. Entropy of Indirect Jumps. The primary axis, in orange, shows the entropy of indirect jumps, where entropy is modeled as log2 of the dilation in bytes. The secondary axis, in blue, shows the probability that a given configuration will not inflate an indirect jump.

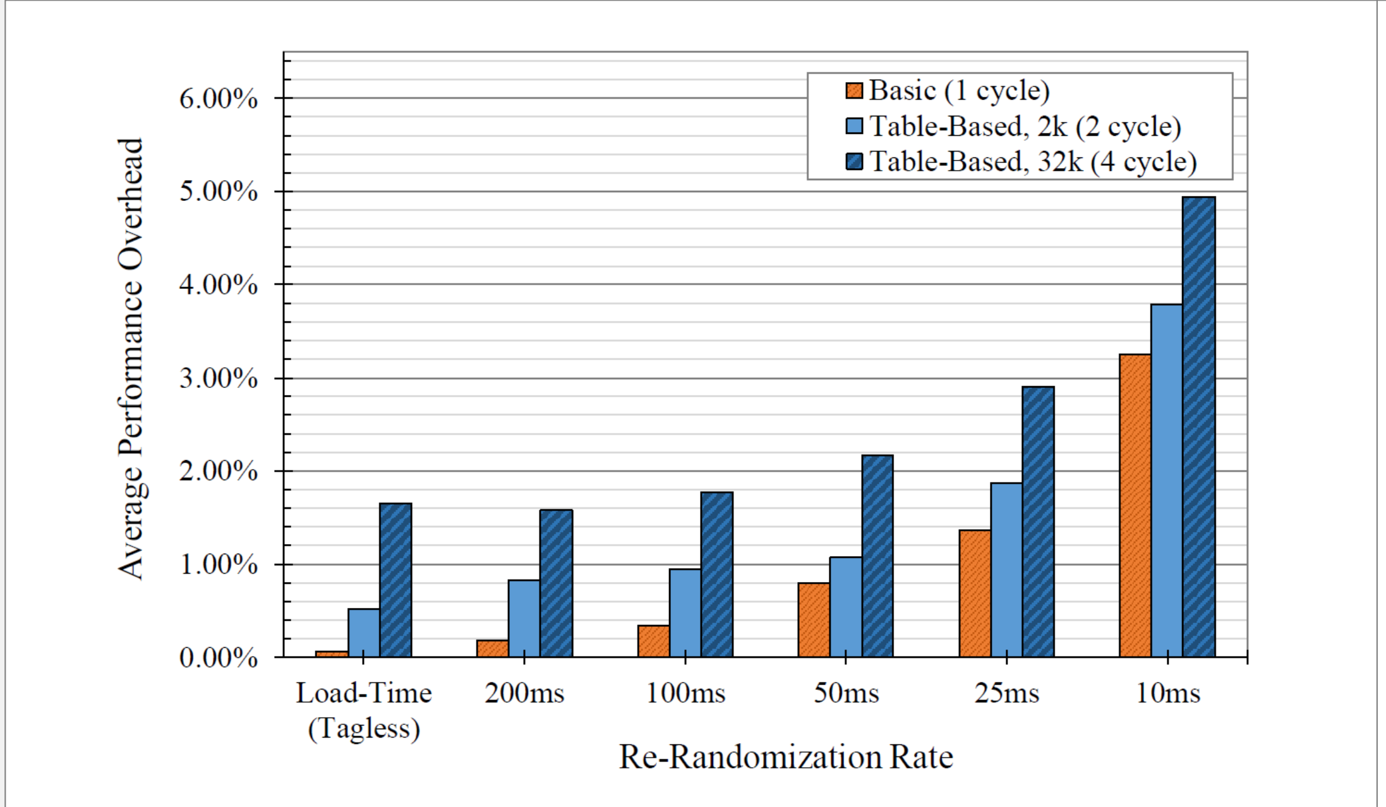

Figure 5. Average Runtime Overheads. The average overhead for the SPEC CPU2006 benchmark suite at varied re-randomization rates for the analyzed DDAS configurations. We analyzed three configurations: a basic translation and two differently sized table-based translations.