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.

In a Displaced and Dilated Address Space (DDAS):

- Code pointers are decoupled from their true code location in the virtual address space.
- 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.

The Displaced and Diluted 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.

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.

We implement Displaced and Diluted 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)

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, with similarly low power and silicon area overheads.