Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces

Lauren Biernacki, Mark Gallagher, Valeria Bertacco, Todd Austin
Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine.
Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine.
Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine.
Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine.

Displaced and Dilated Address Space
Many prevalent security exploits are underpinned by precise knowledge of the memory layout of the victim machine.
Displaced and Dilated Address Space

- No impact on spatial locality;
- High entropy randomization;
- Attack Detection;
- Runtime Re-Randomization;
We **decouple code pointers** from true code location in the virtual address space (VAS) by representing them in a **superimposed address space** termed the Displaced and Dilated Address Space (DDAS).

---

**No impact on spatial locality;** High entropy randomization; Attack Detection; Runtime Re-Randomization;
We combine two techniques to obfuscate the code segment:

1.) Displacement by a 64-bit key
2.) Dilation by inserting holes at an instruction-level granularity
Displaced and Dilated Address Space

No impact on spatial locality; High entropy randomization; Attack Detection; Runtime Re-Randomization;

We *programmatically* translate pointers between the DDAS and VAS at runtime, allowing us to detect accesses to the dilated holes that interleave instructions.
To defend against memory disclosures, we leverage hardware to efficiently \textit{re-randomize} the DDAS layout under running programs.
Displaced and Dilated Address Space

No impact on spatial locality; High entropy randomization; Attack Detection; Runtime Re-Randomization;

With hardware support, our defense has *negligible performance overheads*, at 1% *with re-randomization every 50 milliseconds*, while providing strong probabilistic guarantees against control-flow hijacking attacks.
Displaced and Dilated Address Space

- DDAS → VAS Translation
- RISC-V Hardware Implementation
- Security & Performance Analysis
- Concluding Thoughts
The DDAS memory configuration is *determined programmatically* by a *translation function*, and all code pointers are expressed as 64-bit DDAS values.

**Code Pointer Creation**

<table>
<thead>
<tr>
<th>DDAS Value</th>
<th>VAS Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4ae01488</td>
<td>0xff658140</td>
</tr>
</tbody>
</table>

**Code Pointer Use (e.g., jalr)**

<table>
<thead>
<tr>
<th>DDAS Value</th>
<th>VAS Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4ae01488</td>
<td>0xff658140</td>
</tr>
</tbody>
</table>

**VAS to DDAS**

**DDAS to VAS + Security Check**
The DDAS memory configuration is determined programmatically by a translation function, and all code pointers are expressed as 64-bit DDAS values.

**Basic DDAS Translation**

\[
A_{ddas} = A_{vas} + d + \left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor i \\
A_{vas} = A_{ddas} - d - \left\lfloor \frac{(A_{ddas} - d)}{S_{ddas}} \right\rfloor i
\]

**Table-Based DDAS Translation**

\[
A_{ddas} = A_{vas} + d + \left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor i - T[A_{vas} \mod S_{vas}] \\
A_{vas} = A_{ddas} - d - \left\lfloor \frac{(A_{ddas} - d)}{S_{ddas}} \right\rfloor i - T[((A_{ddas} - d) \mod S_{ddas})/r]
\]
The DDAS memory configuration is determined programmatically by a translation function, and all code pointers are expressed as 64-bit DDAS values.

**Basic DDAS Translation**

\[
A_{ddas} = A_{vas} + d + \left\lfloor A_{vas}/S_{vas}\right\rfloor i \\
A_{vas} = A_{ddas} - d - \left\lfloor (A_{ddas} - d) \gg S_{ddas}\right\rfloor i
\]

**Table-Based DDAS Translation**

\[
A_{ddas} = A_{vas} + d + \left\lfloor A_{vas}/S_{vas}\right\rfloor i - T[A_{vas} \mod S_{vas}] \\
A_{vas} = A_{ddas} - d - \left\lfloor (A_{ddas} - d) \gg S_{ddas}\right\rfloor i - T[(A_{ddas} - d) \mod S_{ddas})/r]
\]
The DDAS memory configuration is **determined programmatically** by a **translation function**, and all code pointers are expressed as 64-bit DDAS values.

**Basic DDAS Functional Unit**

\[
A_{ddas} = A_{vas} + d + \left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor i \\
A_{vas} = A_{ddas} - d - \left\lfloor \frac{(A_{ddas} - d) \gg S_{ddas}}{S_{ddas}} \right\rfloor i
\]

**Table-Based DDAS Functional Unit**

\[
A_{ddas} = A_{vas} + d + \left\lfloor \frac{A_{vas}}{S_{vas}} \right\rfloor i - T(A_{vas} \mod S_{vas}) \\
A_{vas} = A_{ddas} - d - \left\lfloor \frac{(A_{ddas} - d) \gg S_{ddas}}{S_{ddas}} \right\rfloor i - T((A_{ddas} - d) \mod S_{ddas})/r
\]
RISC-V Hardware Implementation

The use of 64-bit DDAS code pointers introduces a layer of indirection that requires **pipeline modifications** to ensure correct control flow.

**Simplified RISC-V Pipeline**

- PC
- Insn. Fetch
- Insn. Decode
- Dispatch
- Registers
- Indirect Jump
- Direct Jump
- Func. Unit
- Write Back

- Upper Bits: \( A_{\text{high}} = 42,500 \)
- Lower Bits: \( A_{\text{low}} = 0,000 \)

- Upper Bits: \( A_{\text{high}} = 4,096 \)
- Lower Bits: \( A_{\text{low}} = 0,000 \)
During runtime re-randomization, the DDAS layout is **periodically re-keyed** and **code pointers are updated** accordingly.

**Simplified RISC-V Pipeline w/ Runtime Re-Randomization**

- DDAS Hardware
  - Main Core
    - PC
    - Insns. Fetch
    - Insns. Decode
    - Dispatch
    - Registers
    - Execute
      - Indirect Jump
      - Direct Jump
      - Func. Unit
    - Write Back
  - I-Cache
  - D-Cache
  - Memory
  - Memory Scan Logic
  - DDAS Remapper
    - DDAS → VAS
    - VAS → DDAS
    - Old Keys
    - New Keys
During runtime re-randomization, the DDAS layout is **periodically re-keyed** and **code pointers are updated** accordingly.
Results & Analysis: Methodology

We implemented DDAS on a RISC-V out-of-order core in the gem5 simulator in system call emulation mode.

We analyze three distinct implementations of DDAS, both with load-time and runtime re-randomization:

<table>
<thead>
<tr>
<th>Method</th>
<th>Functional Unit Latency</th>
<th>Power of 2 Constraints</th>
<th>Maximum segment size before repition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic DDAS</td>
<td>1 cycle</td>
<td>$s_{ddas}$ and $i$</td>
<td>N/A</td>
</tr>
<tr>
<td>Table-Based DDAS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2k entries</td>
<td>2 cycles</td>
<td>$s_{ddas}$ and $r$</td>
<td>8 kB</td>
</tr>
<tr>
<td>32k entries</td>
<td>4 cycles</td>
<td>$s_{ddas}$ and $r$</td>
<td>128 kB</td>
</tr>
</tbody>
</table>
Results & Analysis: Security

With Displaced and Dilated Address Spaces we:

• **Obfuscate valid code pointers** in a \(2^{64}\) byte address space

• Prevent **relative distances** from being used to derive code gadgets from a leaked pointer

• **Detect attempts** to forge a code pointer

<table>
<thead>
<tr>
<th>Jump to Next Insn</th>
<th>100 kB dilation, on average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump to Next Page</td>
<td>&gt;100 MB dilation, on average</td>
</tr>
<tr>
<td>Percentage of In-Memory Traps</td>
<td>&gt; 99.996, on average</td>
</tr>
</tbody>
</table>
Results & Analysis: Security

Entropy of Indirect Jumps for Varying Configurations

- $\log_2$ (Average bytes of displacement)
- $\log_2$ (Average bytes of dilation)
Results & Analysis: Security

Entropy of Indirect Jumps for Varying Configurations

95% likelihood that jumps > 32B are dilated
Results & Analysis: Performance

Average Performance Overheads on SPEC CPU2006 for Varying Configurations

- < 2% Overhead without re-randomization
- ~1.1% Overhead at 50ms re-randomization for recommended config.
We introduce Displaced and Dilated Address Spaces, a superimposed address space where all code pointers are expressed:

- Randomize **absolute addresses** with displacement *(63-bits of entropy)*
- Randomize **relative addresses** with dilation *(55-bits of entropy)*
- **Detections attempts** to forge a code pointers
Conclusions

We introduce Displaced and Dilated Address Spaces, a superimposed address space where all code pointers are expressed

- Randomize *absolute addresses* with displacement (*63-bits of entropy*)
- Randomize *relative addresses* with dilation (*55-bits of entropy*)
- *Detects attempts* to forge a code pointers

By leveraging hardware support, we are able to implement this defense while keeping *performance overheads well below 5%*
Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces

Questions?  LBiernac@UMich.edu  @LMBiernacki