


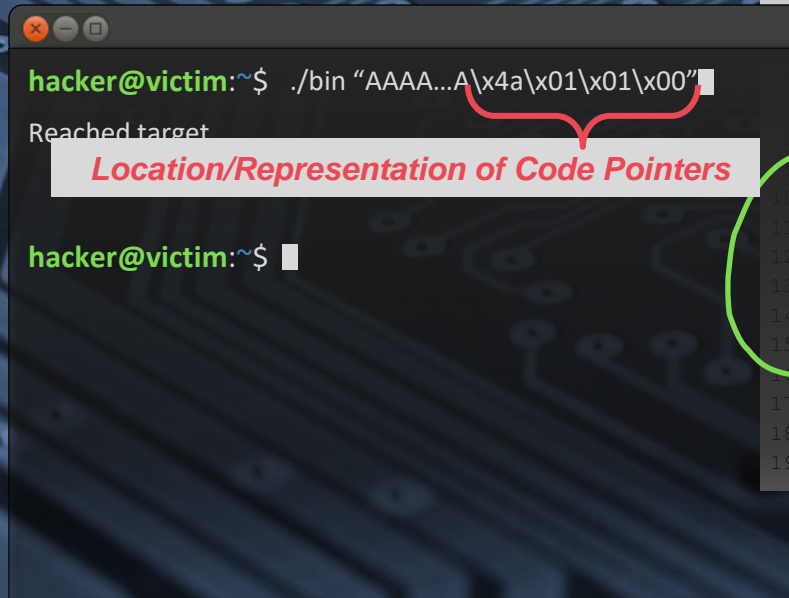
# *Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces*

**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.

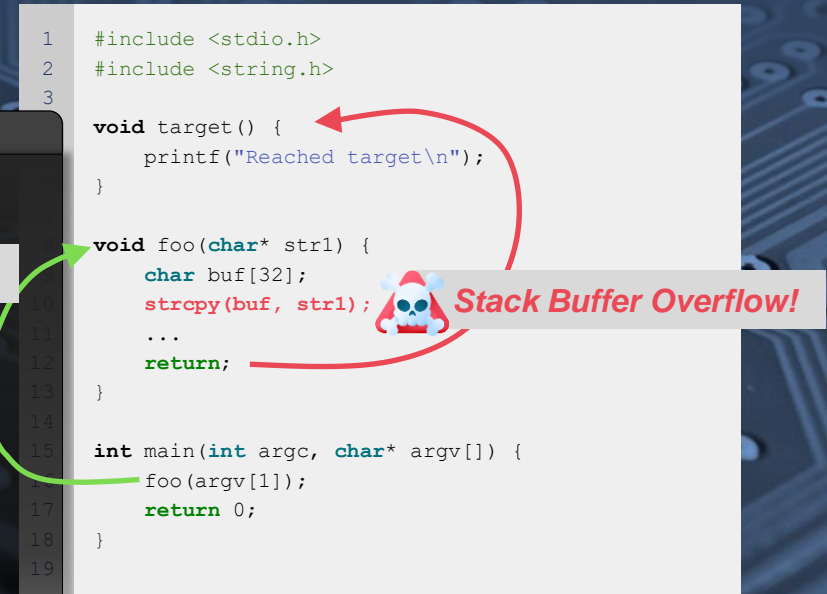
Many prevalent security exploits are underpinned by **precise knowledge of the memory layout of the victim machine.**



A terminal window with a dark background. The prompt is `hacker@victim:~$`. The command entered is `./bin "AAAA...A\x4a\x01\x01\x00"`. The output is `Reached target`. A red bracket underlines the hex sequence `\x4a\x01\x01\x00` in the command. A callout box with a red border and the text **Location/Representation of Code Pointers** points to this bracket.

```
hacker@victim:~$ ./bin "AAAA...A\x4a\x01\x01\x00"
Reached target
```

**Location/Representation of Code Pointers**



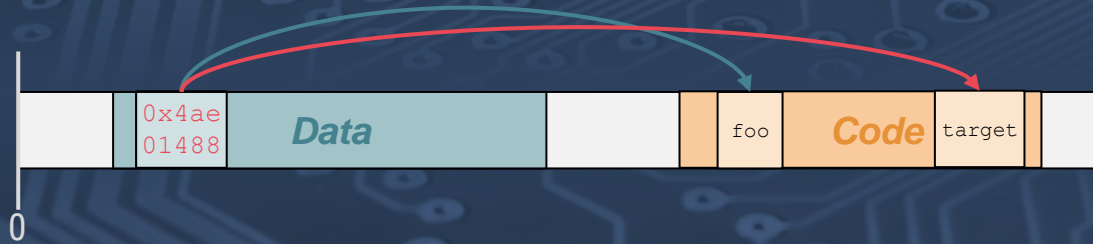
A C code snippet with line numbers 1 to 19. It includes `<stdio.h>` and `<string.h>`. It defines a `target()` function that prints "Reached target\n". It defines a `foo(char* str1)` function that declares a `buf[32]` array, copies `str1` into it, and returns. It defines a `main` function that calls `foo` with `argv[1]`. A red arrow points from the `target` function to the `printf` statement. A green arrow points from the `foo` function to the `main` function. A red skull icon and the text **Stack Buffer Overflow!** are placed next to the `strcpy` call in the `foo` function.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void target() {
5      printf("Reached target\n");
6  }
7
8  void foo(char* str1) {
9      char buf[32];
10     strcpy(buf, str1);
11     ...
12     return;
13 }
14
15 int main(int argc, char* argv[]) {
16     foo(argv[1]);
17     return 0;
18 }
19
```

**Stack Buffer Overflow!**



Many prevalent security exploits are underpinned by **precise knowledge of the memory layout** of the victim machine.

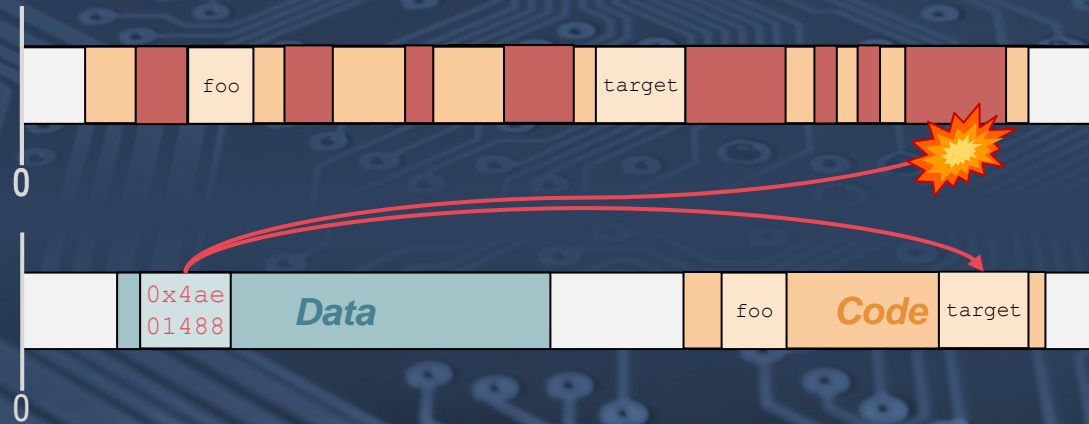


Many prevalent security exploits are underpinned by **precise knowledge of the memory layout** of the victim machine.

## *Displaced and Dilated Address Space*



Many prevalent security exploits are underpinned by **precise knowledge of the memory layout** of the victim machine.



*No impact on spatial locality;*

*High entropy randomization;*

*Attack Detection;*

*Runtime Re-Randomization;*

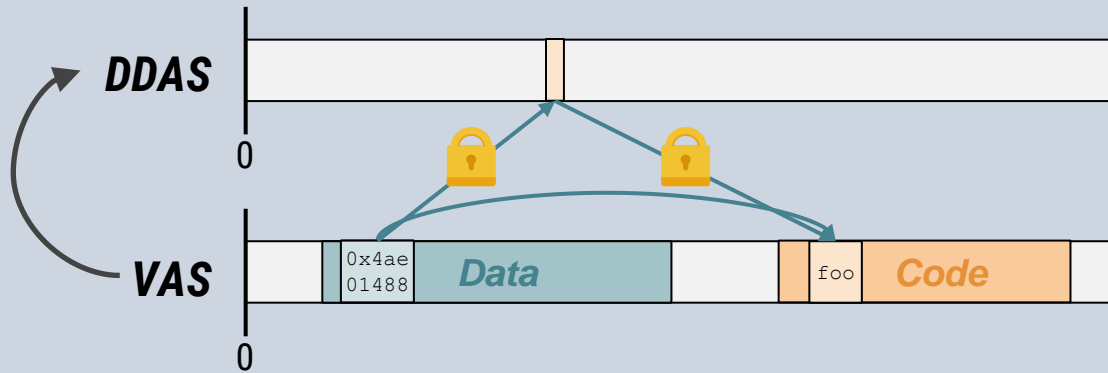
# *Displaced and Dilated Address Space*

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

# 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)



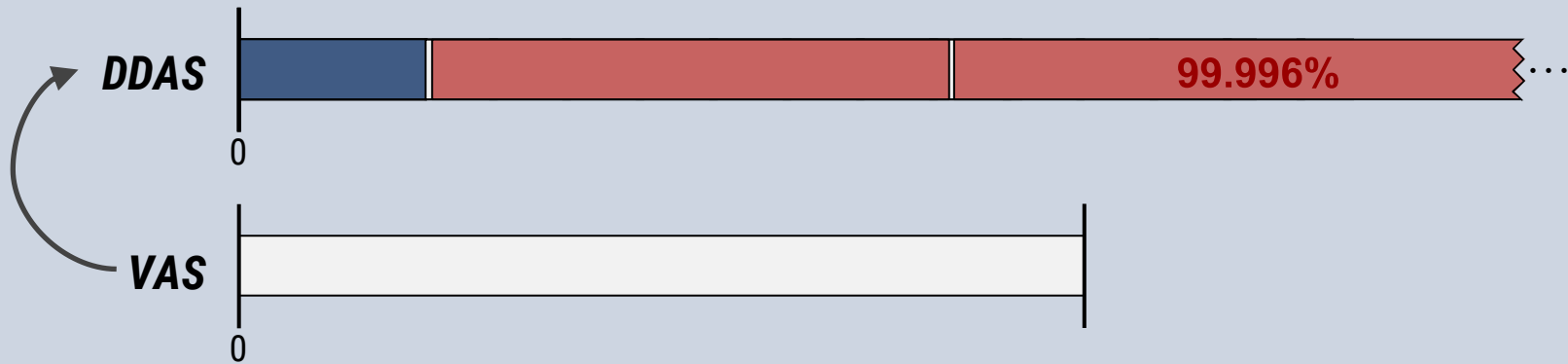


# Displaced and Dilated Address Space

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

We combine two techniques to obfuscate the code segment:

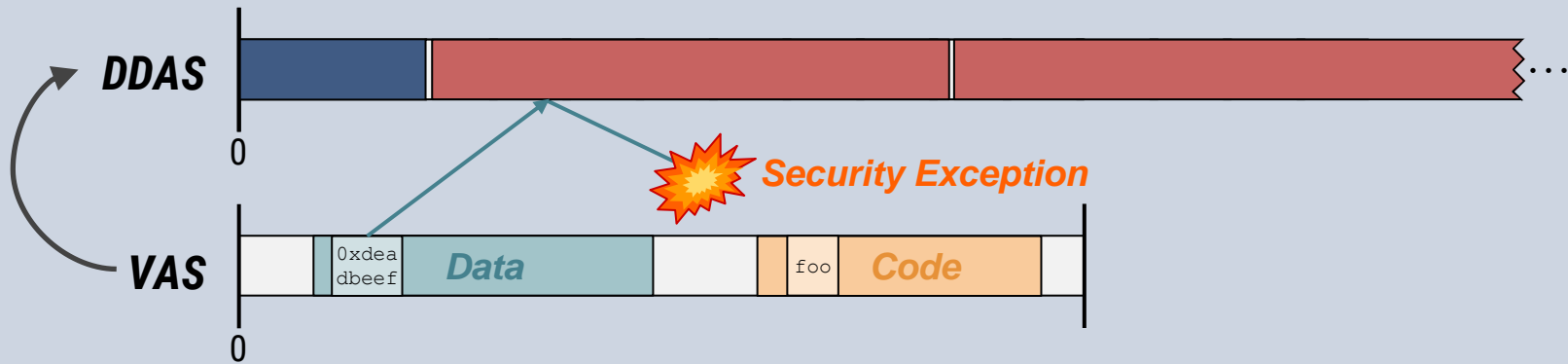
- 1.) Displacement by a 64-bit key
- 2.) Dilation by inserting holes at an instruction-level granularity



# Displaced and Dilated Address Space

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

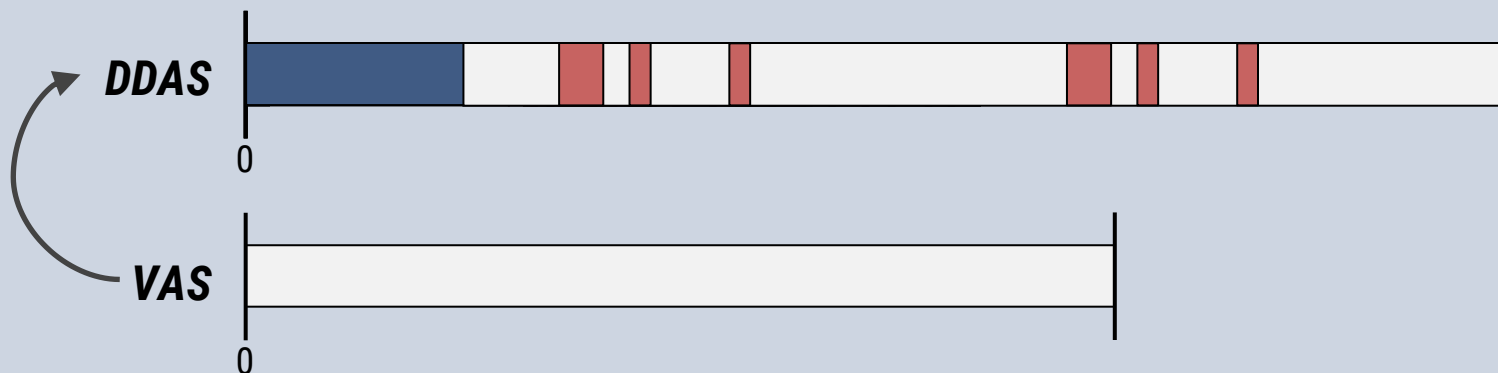
We **programmatically** translate pointers between the DDAS and VAS at runtime, allowing us to **detect accesses** to the dilated holes that interleave instructions



# Displaced and Dilated Address Space

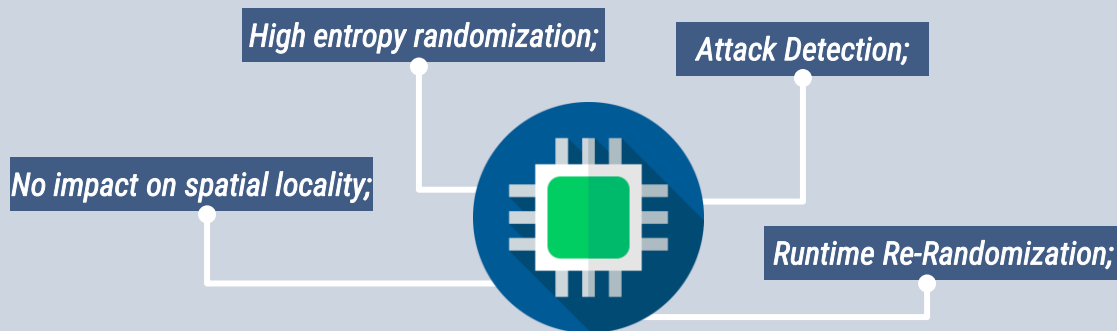
→ No impact on spatial locality; High entropy randomization; Attack Detection; Runtime Re-Randomization;

To defend against memory disclosures, we leverage hardware to efficiently **re-randomize** the DDAS layout under running programs



# Displaced and Dilated Address Space

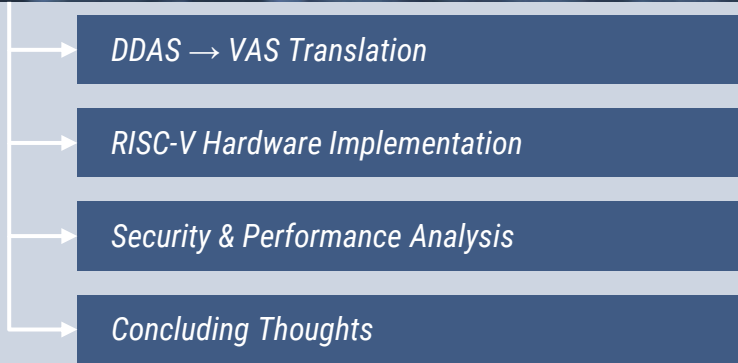
No impact on spatial locality; High entropy randomization; Attack Detection; Runtime Re-Randomization;



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

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**

0x4ae01488



0xff658140

**VAS to DDAS**

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

0x4ae01488



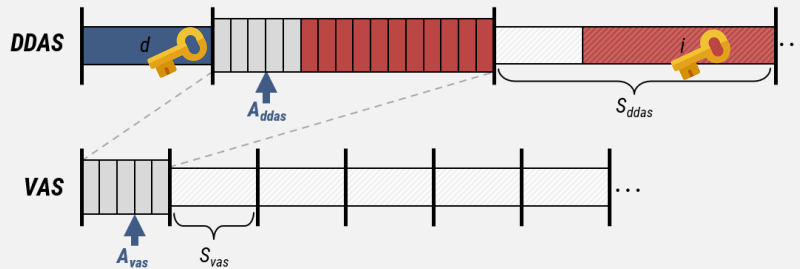
0xff658140

**DDAS to VAS +  
Security Check**

# DDAS → VAS Translation

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

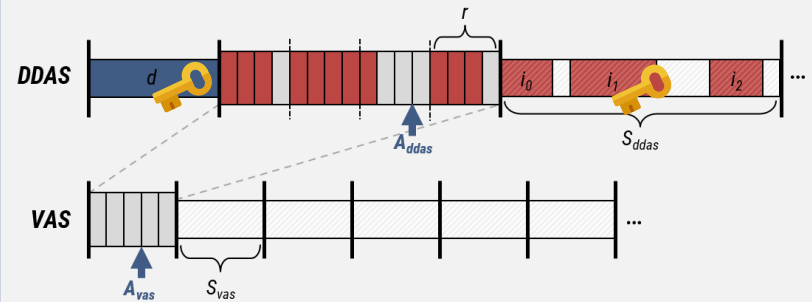
Basic DDAS Translation



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

$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) / S_{ddas} \rfloor i$$

Table-Based DDAS Translation



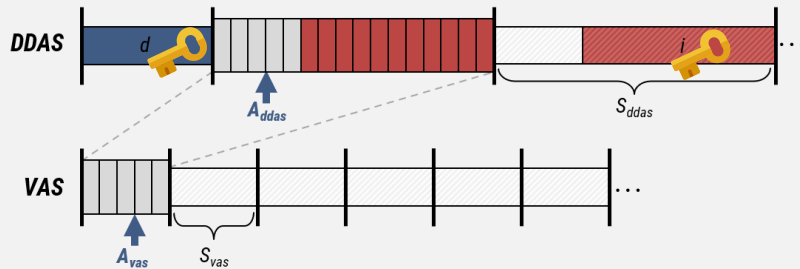
$$A_{ddas} = A_{vas} + d + \lfloor A_{vas} / S_{vas} \rfloor i - T[A_{vas} \bmod S_{vas}]$$

$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) / S_{ddas} \rfloor i - T[(A_{ddas} - d) \bmod S_{ddas}] / r$$

# DDAS → VAS Translation

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

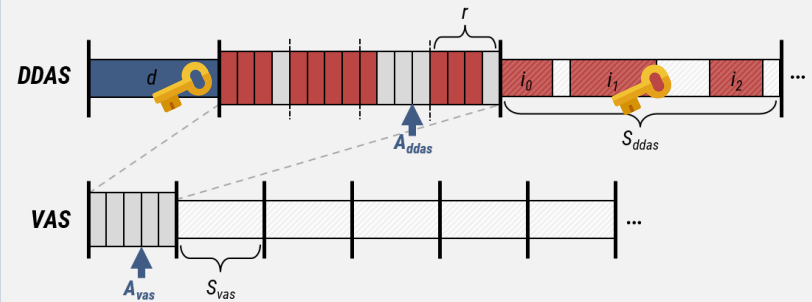
Basic DDAS Translation



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

$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) \gg S_{ddas} \rfloor i$$

Table-Based DDAS Translation



$$A_{ddas} = A_{vas} + d + \lfloor A_{vas} / S_{vas} \rfloor i - T[A_{vas} \bmod S_{vas}]$$

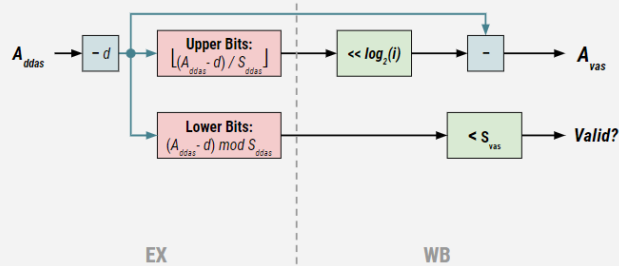
$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) \gg S_{ddas} \rfloor i - T[(A_{ddas} - d) \bmod S_{ddas}] / r$$



# DDAS $\rightarrow$ VAS Translation

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

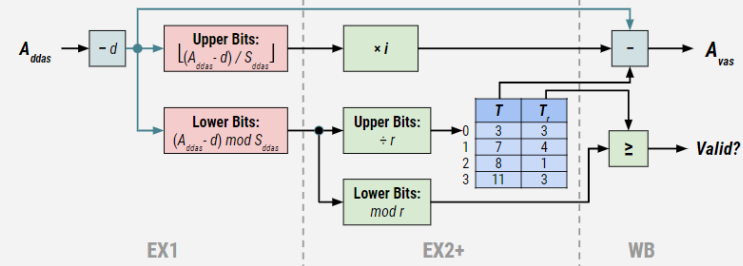
Basic DDAS Functional Unit



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

$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) \gg S_{ddas} \rfloor i$$

Table-Based DDAS Functional Unit



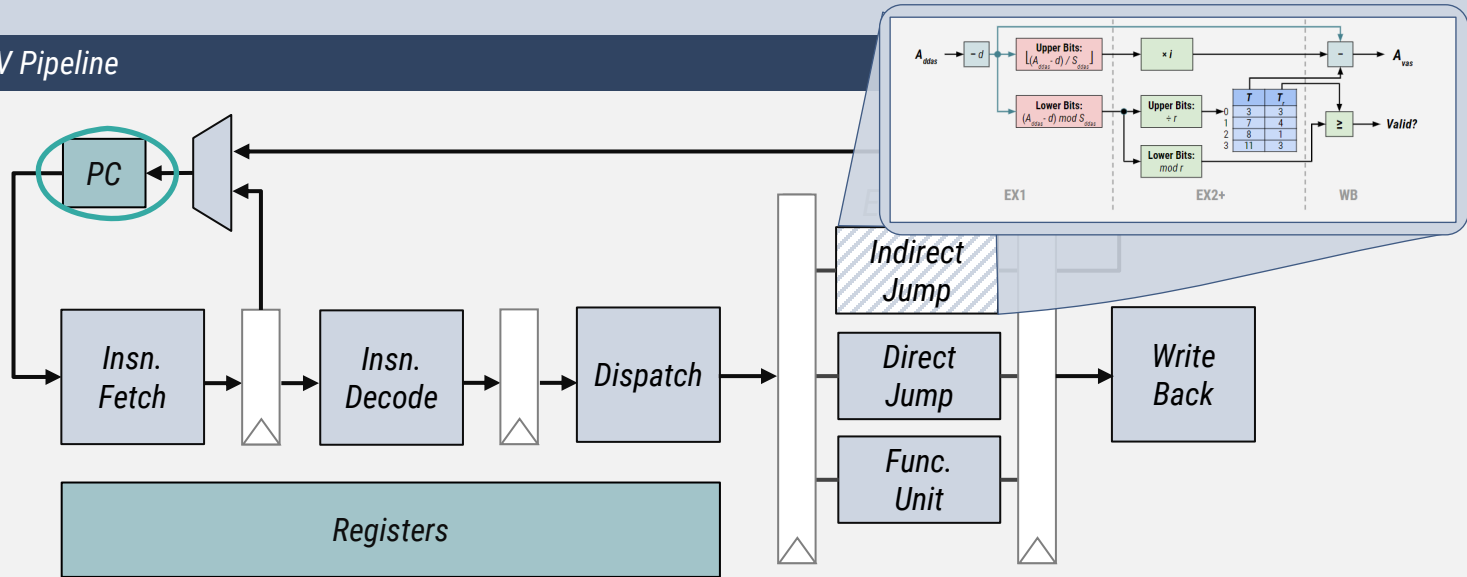
$$A_{ddas} = A_{vas} + d + \lfloor A_{vas} / S_{vas} \rfloor i - T[A_{vas} \bmod S_{vas}]$$

$$A_{vas} = A_{ddas} - d - \lfloor (A_{ddas} - d) \gg S_{ddas} \rfloor i - T[(A_{ddas} - d) \bmod S_{ddas}] / r$$

# 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

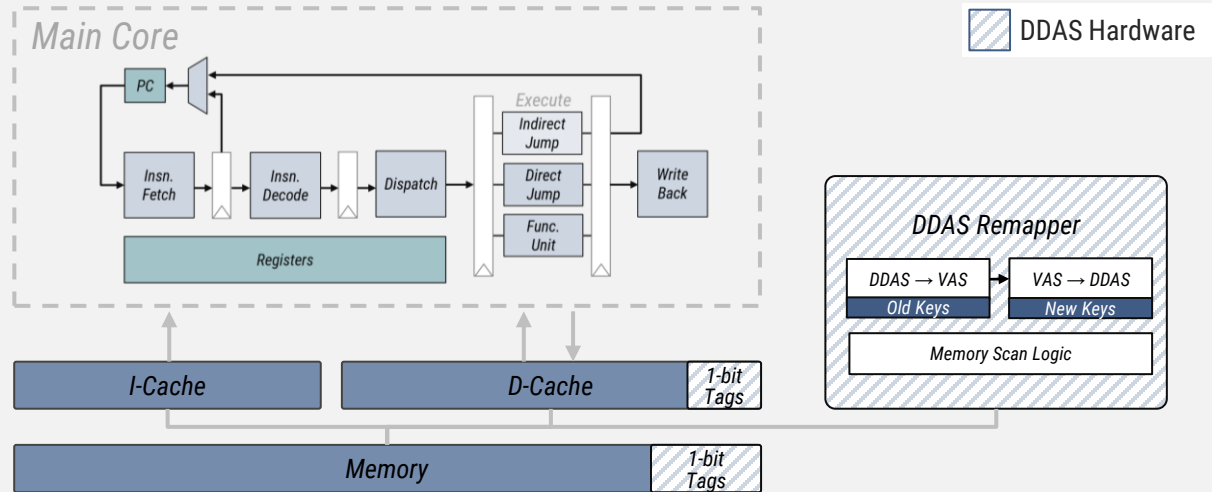
Simplified RISC-V Pipeline



# RISC-V Hardware Implementation

During runtime re-randomization, the DDAS layout is **periodically re-keyed** and **code pointers are updated** accordingly

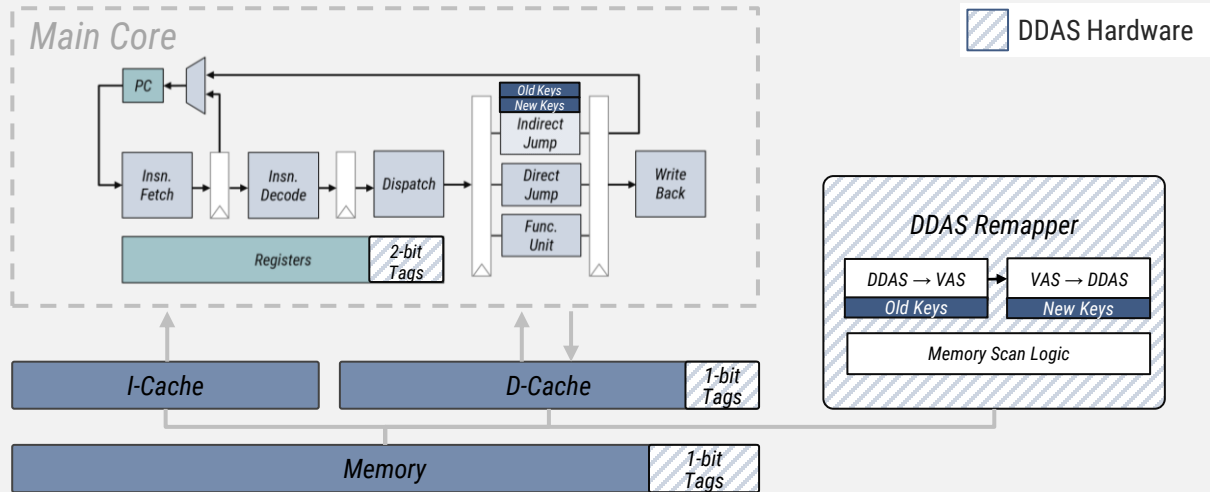
Simplified RISC-V Pipeline w/ Runtime Re-Randomization



# RISC-V Hardware Implementation

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





# 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

	Functional Unit Latency	Power of 2 Constraints	Maximum segment size before repition
<b><i>Basic DDAS</i></b>	1 cycle	$S_{ddas}$ and $i$	N/A
<b><i>Table-Based DDAS 2k entries</i></b>	2 cycles	$S_{ddas}$ and $r$	8 kB
<b><i>Table-Based DDAS 32k entries</i></b>	4 cycles	$S_{ddas}$ and $r$	128 kB

# Results & Analysis: Security

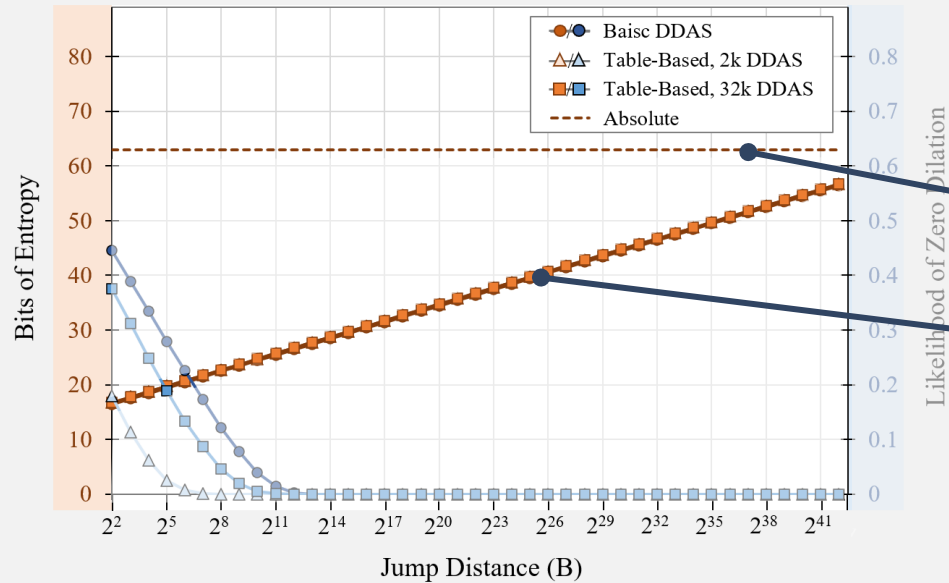
With Displaced and Dilated Address Spaces we:

- **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

<i>Jump to Next Insn</i>	<b>100 kB dilation</b> , on average
<i>Jump to Next Page</i>	<b>&gt;100 MB dilation</b> , on average
<i>Percentage of In-Memory Traps</i>	<b>&gt; 99.996</b> , on average

# Results & Analysis: Security

Entropy of Indirect Jumps for Varying Configurations

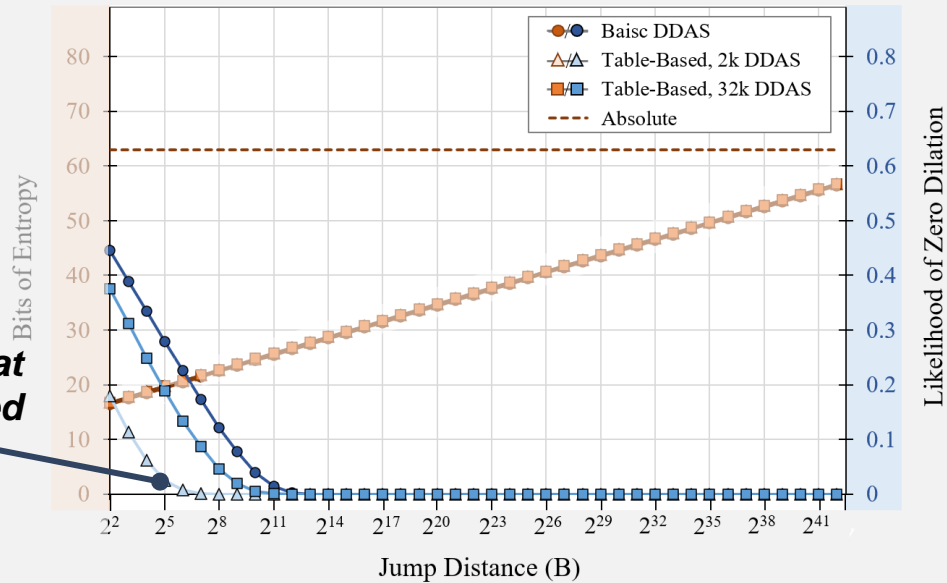


$\log_2(\text{Average bytes of displacement})$

$\log_2(\text{Average bytes of dilation})$

# Results & Analysis: Security

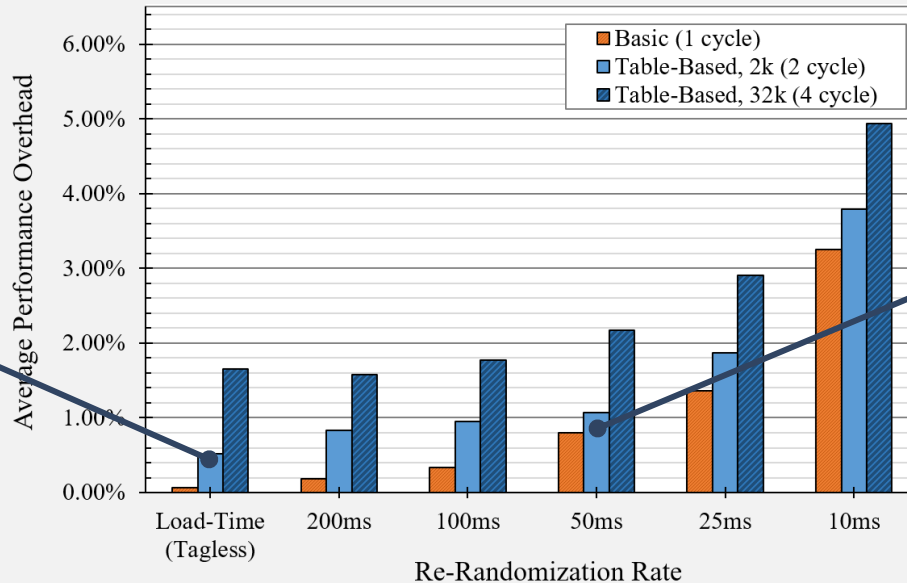
Entropy of Indirect Jumps for Varying Configurations





# Results & Analysis: Performance

Average Performance Overheads on SPEC CPU2006 for Varying Configurations



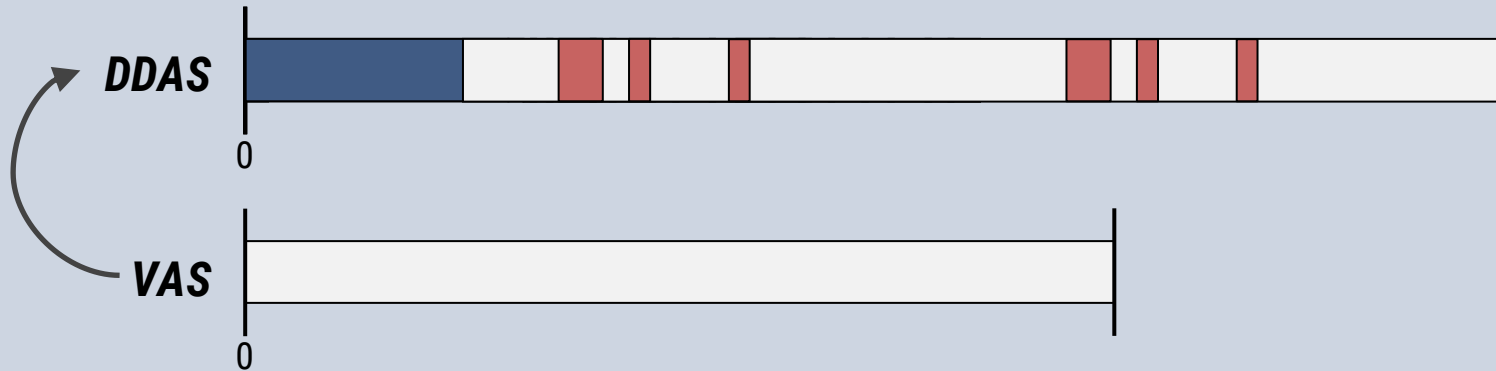
**~1.1% Overhead at 50ms  
re-randomization for  
recommended config.**

**< 2% Overhead without  
re-randomization**

# Conclusions

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

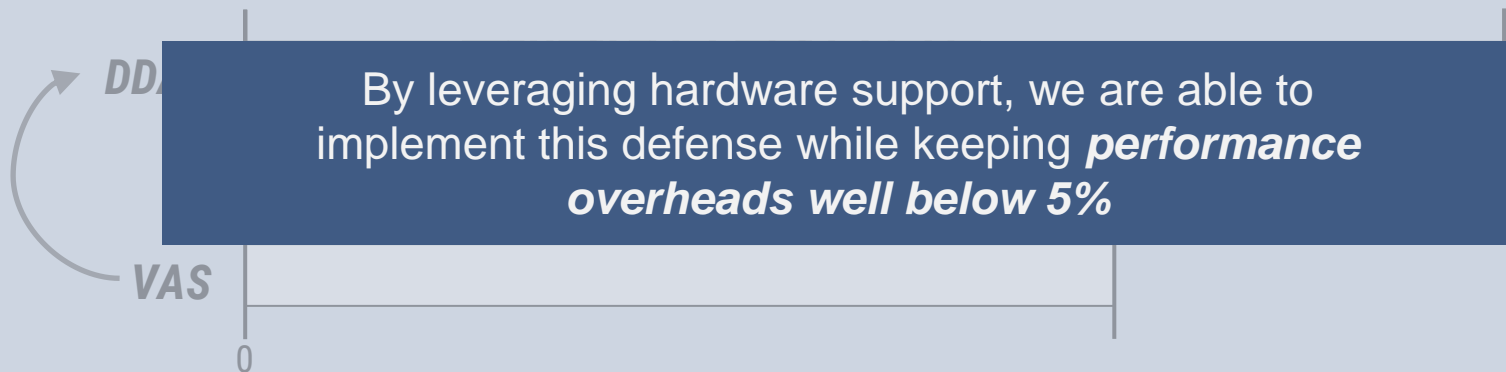
- Randomize **absolute addresses** with displacement (*63-bits of entropy*)
- Randomize **relative addresses** with dilation (*55-bits of entropy*)
- **Detects attempts** to forge a code pointers



# 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



# *Thwarting Control Plane Attacks with Displaced and Dilated Address Spaces*

Questions?



LBiernac@UMich.edu



@LMBiernacki