Enabling Hardware Randomization Across the Cache Hierarchy in Linux-Class Processors

Max Doblas¹, Ioannis-Vatistas Kostalabros¹, Miquel Moretó¹ and Carles Hernández²

¹Computer Sciences - Runtime Aware Architecture, Barcelona Supercomputing Center
{ max.doblas, vatistas.kostalabros, miquel.moreto } @bsc.es

²Department of Computing Engineering, Universitat Politècnica de València
carherlu@upv.es
Introduction

- Cache-based side channel attacks are a serious concern in many computing domains
- Existing randomizing proposals can not deal with virtual memory
  - The majority of the state-of-the-art is focussing at the LLCs
- Our proposal enables randomizing the whole cache hierarchy of a Linux-capable RISC-V processor
Cache Side Channel Attacks

1. Calibration
2. Signalling (Precondition, Detection)
3. Signal
4. Decoding

Attacker Addresses
Cache Side Channel Attacks

1. Calibration

Prime+Probe Example

4 sets, 2 way associative cache

Attacker’s Blocks
Victim’s Blocks
Cache Side Channel Attacks

Prime+Probe Example
1. Calibration
2. Prime (precondition)
Cache Side Channel Attacks

Prime+Probe Example
1. Calibration
2. Prime (precondition)
3. Wait (execution of the victim)
Cache Side Channel Attacks

Prime+Probe Example
1. Calibration
2. Prime (precondition)
3. Wait (execution of the victim)
4. Probe (detection)
State of the art

Cache-layout randomization schemes

● Parametric functions that randomize the mapping of a block inside the cache
  ○ Use a key-value to change the hashing applied to the address
  ○ At every key change a new calibration has to be performed
  ○ Protection is provided by modifying the key frequently

● It can be used in single or multiple security domains
State of the art

- (a) Some solutions use an Encryption-Decryption scheme
  - Introduces latency -> Potential high impact in cache latency
  - Improves design simplicity by not altering the cache structure
(b) Randomization function produces the cache-set’s index

- Latency can be partially hidden -> feasible for first level caches
- Needs to increase the Tags to recover block address
- Extra mechanism is needed to enable the virtual memory
Randomization Functions Quality

- Randomization functions need to balance security performance trade-off
- CEASER’s LLBC
  - Inherent linearity deems it useless for SCA thwarting [1]
- Balance time randomized functions examples [2]:
  a) Hash Function
  b) Random modulo


Skewed Caches

- Enhances the security of the cache
  - It is more difficult to calibrate an attack
  - Increases the resources used by multiplying the number of randomization functions.
Virtual memory Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Virtual memory Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

**Process A:** `sb X -> 0x0001`

**Process B:** `ld 0x1001 -> r1`

**Page Table A**

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Page Table B**

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Virtual memory Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Proc A: \( sd \, X \rightarrow 0x0001 \)
Virtual memory Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Proc A: `sd X -> 0x0001`
Proc B: `ld 0x1001 -> r1`
Virtual memory Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Proc A:  sd X -> 0x0001  
Proc B:  ld 0x1001 -> r1

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Coherency protocol access to addr 0x3001
● Adds supports the coherence protocol in finding any valid block.
  ○ Even after a key or a page-table’s translation modification.
● Every cache, keeps track of the valid blocks in the lower level cache.
  ○ This tracking is done by storing the last random index used by the lower level cache for every valid block.
  ○ Using this information, the cache probes any block of the lower level cache.
Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

**Proc A:** `sd X -> 0x0001`

**Proc B:** `ld 0x1001 -> r1`
Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Proc A: \( sd \ X \rightarrow 0x0001 \)  
Proc B: \( ld \ 0x1001 \rightarrow r1 \)

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Coherency protocol access to addr 0x3001
Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Coherency protocol
invalidating addr 0x3001

Coherency protocol
provides X

Page Table A

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Page Table B

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example: Shared data

- Two processes A and B
  - Two different Page Tables
  - Shares data on 0x3000
  - First level caches are VIPT

Coherency protocol invalidating addr 0x3004

Coherency protocol provides X

Proc B: ld 0x1001 -> r1

**Page Table A**

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Page Table B**

<table>
<thead>
<tr>
<th>Virtual Addr</th>
<th>Physical Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

CPU Virtual Address

**L2 Physical Address**

L2 Physical Address

f(addr)
Example of a Three Level Cache Hierarchy

L1 Data Cache
- Meta & Data
  - rnd_idx
generator & coherency
- rnd_idx1
vaddr
req.core

TLB

L2 Private
- Meta & Data
  - coh
tag
  - rnd_idx1
- rnd_idx2
coh_msg
paddr

L3 Shared LLC
- Meta & Data
  - coh
tag
  - rnd_idx2
- rnd_idx3
coh_msg
Implementation on a RISC-V Core

We have implemented this mechanism in the lowRISC SoC.

- There are two different randomizers on the first level cache.
  - Hash function and Random modulo.
- L2 incorporates the directory which track the L1 Blocks.
- Both caches have been augmented with tag array extensions to handle collisions produced by the randomizers.
- The Coherency protocol has been modified.
  - Able to issue probe requests using the random index stored.
Performance Evaluation

- We used the non-floating point benchmarks from the EEMBC suite.
  - 1000 iterations with 1000 different randomized keys.
- The hash function version has a very small impact on performance.
  - Other configurations increase the performance in this benchmarks.
Security Evaluation

- NIST STS testing proves uniform set distribution.
- Non-linear randomization function.
  - Thwarts linear cryptanalysis attacks.
- Security vulnerability analysis based on the cost of attack calibration

<table>
<thead>
<tr>
<th>Processor</th>
<th>L1</th>
<th>L1 (skewed)</th>
<th>L2</th>
<th>L2 (skewed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rocket</td>
<td>4288</td>
<td>68608</td>
<td>35456</td>
<td>2269184</td>
</tr>
<tr>
<td>Neoverse</td>
<td>17152</td>
<td>274432</td>
<td>2269184</td>
<td>18153472</td>
</tr>
<tr>
<td>Skylake</td>
<td>17728</td>
<td>1134592</td>
<td>67584</td>
<td>1081344</td>
</tr>
</tbody>
</table>

Number of attacker accesses to build eviction set
## Resources Evaluation

FPGA resources utilization for different configurations of the caches

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>FF</th>
<th>CLAs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>3249</td>
<td>2514</td>
<td>82</td>
</tr>
<tr>
<td>HF</td>
<td>4587 (+41.2%)</td>
<td>2598 (+3.3%)</td>
<td>87 (+6.1%)</td>
</tr>
<tr>
<td>RM</td>
<td>3553 (+6.0%)</td>
<td>2598 (+3.3%)</td>
<td>87 (+6.1%)</td>
</tr>
<tr>
<td>HF Skewed</td>
<td>7862 (+142.0%)</td>
<td>2676 (+6.4%)</td>
<td>87 (+6.1%)</td>
</tr>
<tr>
<td>RM Skewed</td>
<td>3718 (+14.4%)</td>
<td>2676 (+6.4%)</td>
<td>87 (+6.1%)</td>
</tr>
<tr>
<td><strong>L2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>11047</td>
<td>3778</td>
<td>85</td>
</tr>
<tr>
<td>Others</td>
<td>13607 (23.2%)</td>
<td>3999 (+5.8%)</td>
<td>93 (+9.4%)</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline</td>
<td>15301</td>
<td>7636</td>
<td>199</td>
</tr>
<tr>
<td>HF</td>
<td>19199 (+25.5%)</td>
<td>7941 (+4.0%)</td>
<td>212 (+6.5%)</td>
</tr>
<tr>
<td>RM</td>
<td>18055 (+18.0%)</td>
<td>7941 (+4.0%)</td>
<td>212 (+6.5%)</td>
</tr>
<tr>
<td>HF Skewed</td>
<td>22474 (+46.9%)</td>
<td>8019 (+5.0%)</td>
<td>212 (+6.5%)</td>
</tr>
<tr>
<td>RM Skewed</td>
<td>18330 (+25.5%)</td>
<td>8019 (+5.0%)</td>
<td>212 (+6.5%)</td>
</tr>
</tbody>
</table>

- The HF has a higher cost.
- In the RM case, randomization module consumes very few resources.
Conclusions

● Novel randomization mechanism for the whole cache hierarchy.
● Enables the use of virtual and physical addresses.
● Maintains cache coherency.
● Has a small impact on performance and consumed resources.
● We achieved integration into a RISC-V processor capable to boot Linux.
● Achieved increased security against cache-based side-channel attacks.
Future work

● Analyze implications and implementation of more complex coherence protocols.
● Implement our proposal in a complex processor design.
● Enable the utilization of multiple security domains.
Thank you

max.doblas@bsc.es