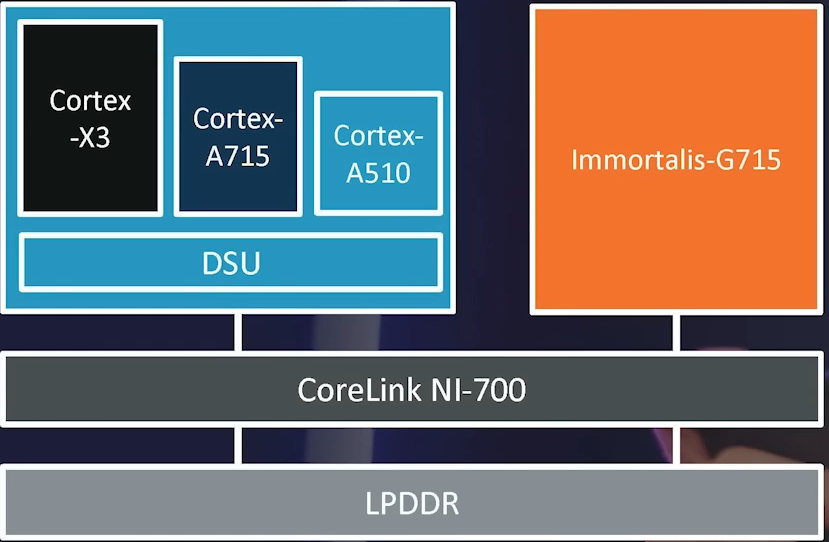
* Doing a timing analysis in ASYNC mode to determine if we can derive any info from security violations. I.e. can we deduce any useful information from the time difference in memory access
  + I.e. Can measure the time it takes to access protected memory regions unintentionally leak tag-related information.
* If a coprocessor (such as a GPU or DSP) can be compromised, it could act as a proxy to perform memory operations on behalf of malicious code?
* If we run 16 programs simultaneously, (i.e. apply memory pressure) can we force the system into reusing tags more frequently?
* In this website <https://llvm.org/docs/ScudoHardenedAllocator.html> they mention “

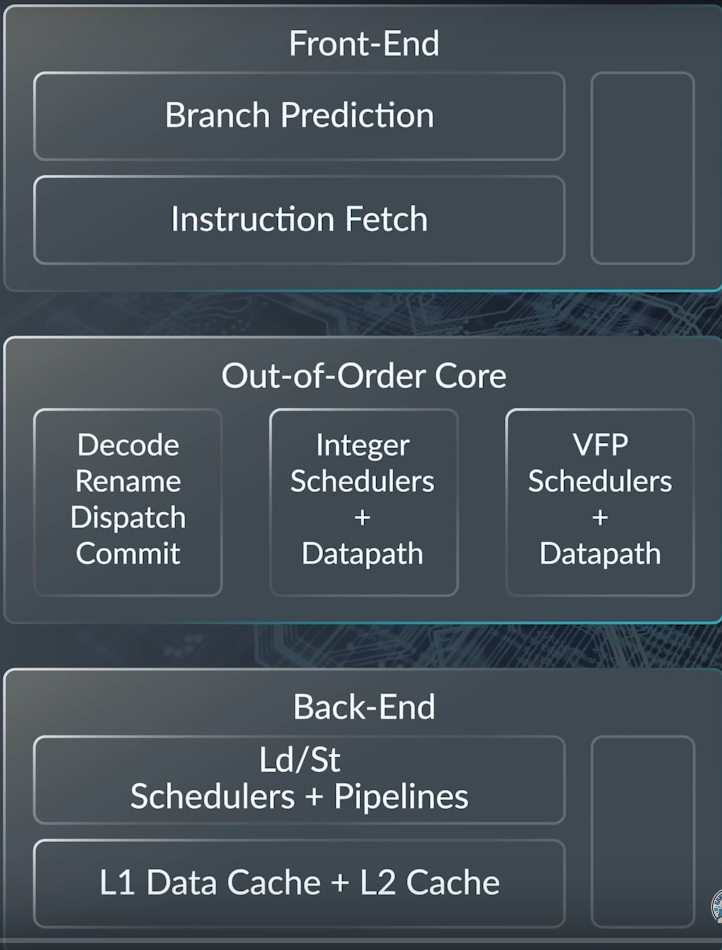
| allocation\_ring\_buffer\_size | 32768 | If stack trace collection is requested, how many previous allocations to keep in the allocation ring buffer.  This buffer is used to provide allocation and deallocation stack traces for MTE fault reports. The larger the buffer, the more unrelated allocations can happen between (de)allocation and the fault. If your sync-mode MTE faults do not have (de)allocation stack traces, try increasing the buffer size.  Stack trace collection can be requested using the scudo\_malloc\_set\_track\_allocation\_stacks function. |
| --- | --- | --- |

* ” how would the ring buffer work?
* <https://developer.arm.com/documentation/ka005620/1-0/?lang=en>

Random Notes:

According to a paper (Memory Tagging: A Memory Efficient Design), “Tags are prefetched when cache lines are loaded for tagged memory. There is also a tag cache used for caching tags”





Tensor G3:

* Nine-cores, 1+4+4 configuration
  + 1 Arm Cortex X3 “Big” core -> heavy games etc.
    - 40-bit address width
    - L1 Cache - 128KiB (64 KiB I-cache with parity, 64 KiB D-cache)
    - L2 Cache - 512–1024 KiB
    - L3 Cache - 512 KiB – 16 MiB (optional)
  + 4 ARM Cortex-A715 “Middle” core -> launching and running apps etc
    - Decode width 5 bits
    - L1 Cache - 64/128 KiB (32/64 KiB I-cache with parity, 32/64 KiB D-cache)
    - L2 Cache - 128–512 KiB
    - L3 Cache - 256 KiB – 16 MiB (optional)
  + 4 Arm Cortex-A510 “Little” core ->
    - 3-wide in-order design
    - 3-wide fetch and decode front-end as well as 3-wide issue and execute on the back-end

Power analysis

* Sync mode definitely *uses the Big Core*, same with Async.
* MTE uses the little core *less*
* Uses the GPU less (much less in SYNC mode)

## Theoretical Calculations:

### **Hardware Specifications:**

* **Cortex-X3 (Big core):** 1 core, with a large cache size that varies, but let's consider the maximum possible cache size for the calculations.
* **Cortex-A715 (Middle core):** 4 cores, each with its own L1 and L2 cache sizes.
* **Cortex-A510 (Little core):** 4 cores, with smaller cache sizes compared to the middle cores.

### Total Tags

* **1,048,576 tags** in its L3 cache (assuming a 16 MiB maximum size),
* **1,073,741,824 tags** in its RAM (assuming a 16 GB size).

Bluehat 2022 Presentation Notes - “Security Analysis of MTE Through Examples”

* What bugs/errors this prevents:
  + memcpy-style bugs
  + strictly linear overflows/underflows
  + Heap overrun/ovrerread (adjacent)
  + UAF
  + Heap OOB R/W (non-adjacent)
  + SMBGhost is deterministically mitigaged
* Bypassing
  + if we know the tag, or fake a pointer to tagged memory, we can corrupt absolute 64-bit pointers
  + if we dont move a pointer out of bounds, or trigger an OOB to memory that has the same tag) corrupt the LSB of a pointer/,move it backward/forward in memory
  + Intra-object corruption
  + if you have an OOB in a JS engine, you can trigger a side channel via speculative execution to leak tags
  + Straightforward type confusion bugs are not mitgated by MTE
    - 1st primitive is a type confusion
    - creation of type confusion scenarios rooted/built on other bugs (OOB/UAF)are mitigated
* Virtual Address Tagging - Translation Control Register (TCR\_ELn) has additional field TopByteIgnore (TBI) that provides tagged addressing support
* In Synchronous mode, loads/stores cannot retire until tag is read from memory and checked
* For every allocation, malloc needs to
  + align the allocations
  + choose random rag
  + tag underlying memory for newly allocated chunk (O(n))
  + return tagged pointer to newly allocated chunk
* For every free, retag the allocation
  + could catch UAF before before reallocation
* I dont think the stack/global are tagged(?)

Websites to check out:

* <https://msrndcdn360.blob.core.windows.net/bluehat/bluehatil/2022/assets/doc/Security%20Analysis%20of%20MTE%20Through%20Examples__Saar%20Amar.pdf>
* <https://news.ycombinator.com/item?id=39668053>
* <https://vulners.com/googleprojectzero/GOOGLEPROJECTZERO:1F994A0EE7506346DF0F2E86C2628C13>
* <https://www.youtube.com/watch?v=JL_D9mm5wGo>

**3am haze notes:**

**ndk-gdb does not support mte**

**Lldb also does not**

**Need to to implement it myself**

**Iterate pids  
For each pid, open /proc/<pid>/smaps  
For each memory range listed with the ‘mt’ flag set, run ptrace on the segment using the MTE\_PEEK https://dri.freedesktop.org/docs/drm/arm64/memory-tagging-extension.html**