Arm A-Profile Architecture
Arm A-Profile Architecture

- Crypto
- AArch32
- AArch64
- SIMD
- VHE
- Atomics

Armv8.0-A
Armv8.1-A
Arm A-Profile Architecture

Crypto
AArch32
SIMD
AArch64

SVE
RAS
Stat Profiling
VHE
Atomics

Armv8.0-A

Armv8.1-A
Armv8.2-A
Arm A-Profile Architecture

- Crypto
- AArch32
- AArch64
- SIMD
- VHE
- Stat Profiling
- Atomics
- FP
- Nested Virt.
- SVE
- RAS
- Pointer Auth
- RCpc

Armv8.0-A
Armv8.1-A
Armv8.2-A
Armv8.3-A
Arm A-Profile Architecture

- Crypto
- AArch32
- AArch64
- SIMD
- VHE
- Atomics
- FP
- Nested Virt.
- RAS
- SVE
- Stat Profiling
- RCpc
- Pointer Auth
- Secure EL2
- Mem Model
- Armv8.0-A
- Armv8.1-A
- Armv8.2-A
- Armv8.3-A
- Armv8.4-A
Overview of Armv8.5-A features

• Memory Tagging Extensions
• Branch Target Identifiers
• Random Number Generating instructions
• Architecture support for protecting KASLR
• Spectre and Meltdown related enhancements
• FP support for porting code from other architectures – Conversion to Integers
• Small scale enhancements to virtualization (traps on Cache/TLB maintenance)
• Support for Exception/Entry without context synchronization
• Cache Clean to point of Deep Persistence
Vulnerabilities and exploits

The **vulnerability** is the ‘open door’ allowing the attacker in...
- There are many and they cannot all be found in lab conditions

An **exploit** occurs when the attacker takes advantage of the vulnerability...
- For profit, nuisance or warfare
Buffer Overflows and Use-After-Free

Many vulnerabilities are caused by Buffer Overflows or Use-After-Free errors:

- Buffer overflow: Running past the end of an allocation
- Use-after-free: Accessing freed memory through a stale pointer
Memory Tagging Extension (MTE)

- Memory tagging probabilistically detects memory safety vulnerabilities
  - Update software *before* defenses tested
  - Detects issues such as use-after-free and buffer overflows

- When memory is allocated or freed it is given a Tag
  - All accesses to that memory must be made by an address with the same Tag
  - Lock and Key scheme
Memory Tagging Extension (MTE)

• Memory tagging probabilistically detects memory safety vulnerabilities
  • Update software *before* defenses tested
  • Detects issues such as use-after-free and buffer overflows

• When memory is allocated or freed it is given a Tag
  • All accesses to that memory must be made by an address with the same Tag
  • Lock and Key scheme
Memory Tagging Extension (MTE)

- Memory tagging probabilistically detects memory safety vulnerabilities
  - Update software before defenses tested
  - Detects issues such as use-after-free and buffer overflows

- When memory is allocated or freed it is given a Tag
  - All accesses to that memory must be made by an address with the same Tag
  - Lock and Key scheme
Memory Tagging Extension (MTE)

- Memory tagging probabilistically detects memory safety vulnerabilities
  - Update software *before* defenses tested
  - Detects issues such as use-after-free and buffer overflows

- When memory is allocated or freed it is given a Tag
  - All accesses to that memory must be made by an address with the same Tag
  - Lock and Key scheme
Memory Tagging Extension (MTE)
Use & Deployment Model

Tag checking can be introduced incrementally
- Deploying MTE aware memory allocator will guard heap allocations
- Guarding stack allocations requires a recompile with an MTE aware toolchain

Priority on checking memory allocations in unsafe languages such as C/C++

Can view MTE as on-device hardware acceleration of tools such as Address Sanitizer
- MTE has a substantially lower performance overhead compared to Address Sanitizer
- Can be used in production environments.
Pointer Authentication
Armv8.3-A

Strong defense against Return Oriented Programming (ROP)

- Raises the bar against attackers
- ROP attacks trick functions to return to the wrong place
**Pointer Authentication**

**Armv8.3-A**

**Strong defense against Return Oriented Programming (ROP)**

- Raises the bar against attackers
- ROP attacks trick functions to return to the wrong place
- With pointer authentication, hardware ensures return to the correct place
- Uses upper-bits of a 64-bit pointer to hold a pointer authentication code (PAC)
- Added and checked by dedicated instructions
- The PAC is cryptographically strong to resist forging
Branch Target Identifiers (BTI)

Analysis suggests BTI enables a massive reduction in viable Jump-Oriented Programming (JOP) gadgets

code:
blr x0
ret

good:
bti c
...
ret

bad:
mov x0, #16
...
ret
Branch Target Identifiers (BTI)

Analysis suggests BTI enables a massive reduction in viable Jump-Oriented Programming (JOP) gadgets

Pages can be marked as containing BTI

Indirect branches can only branch to identified locations with BTI instructions

code:
  blr x0
  ret

  good:
  bti c
  ...
  ret

  bad:
  mov x0, #16
  ...
  ret
Branch Target Identifiers (BTI)

Analysis suggests BTI enables a massive reduction in viable Jump-Oriented Programming (JOP) gadgets

Pages can be marked as containing BTI
Indirect branches can only branch to identified locations with BTI instructions
Branching to other locations causes an exception

code:
  blr x0
  ret

good:
  bti c
  ...
  ret

bad:
  mov x0, #16
  ...
  ret
Pointer Authentication & Branch Target Identifiers together

PAC & BTI produce a significant reduction in available gadgets for ROP & JOP.

- PAC & BTI increase the length of gadgets
- Many remaining gadgets are whole functions

<table>
<thead>
<tr>
<th>Type</th>
<th>No PAC/BTI</th>
<th>With PAC/BTI</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP</td>
<td>16,585</td>
<td>264</td>
<td>59x</td>
</tr>
<tr>
<td>JOP</td>
<td>5,845</td>
<td>11</td>
<td>531x</td>
</tr>
<tr>
<td>Total</td>
<td>22,430</td>
<td>275</td>
<td>82x</td>
</tr>
</tbody>
</table>

glibc (Ubuntu 14.04) gadgets <= 10 instrs
Random number generation instructions

TRNG

True Random Number Generator providing entropy
Must conform to:
• NIST SP800-90B
• NIST SP800-22
• FIPS 140-2
• BSI AIS-31

DRNG

Deterministic Random Number Generator providing random numbers from a cryptographically secure algorithm
Seeded from the TRNG
Reseeded after an Implementation
Defined count of numbers have been generated
Random number generation must conform to NIST SP800-90A Rev1

Whole scheme must conform to NIST SP800-90C
Protecting Kernel Address-Space Layout Randomization

EL0 can probe EL1 address space to look for VA space used by EL1
  • Typically looking at the timing distinction between Translation and Permission faults

KPTI (Kernel Page Table Isolation) has been suggested to address this (and Meltdown)
  • But also makes the buffers used by SPE (or self-hosted trace) inaccessible during user application execution

Armv8.5-A adds TCR_EL1.E0PDx fields to the Translation Control System Register
  • Forces all EL0 accesses to the TTBRx space to give a Translation Fault in all cases
  • For OSes where the TTBR1 space is only used by EL1 code, this protects KASLR from this sort of probing
  • Doing this avoids using KPTI, and also avoids having a separate TTBR for the SPE system

© 2018 Arm Limited
Concurrent modification & execution with fetch speculation

JITs rely on certain behavior when optimizing running code

- This has *technically* been undefined behavior
- Although **all** Arm-based CPUs behaved this way
- The architecture now defines that this will work as expected
Concurrent modification & execution with fetch speculation

JITs rely on certain behavior when optimizing running code
- This has *technically* been undefined behavior
- Although all Arm-based CPUs behaved this way
- The architecture now defines that this will work as expected

Key: Executing Processor, Modifying Processor

0 Executing processor runs code path (repeatedly)
Concurrent modification & execution with fetch speculation

JITs rely on certain behavior when optimizing running code

- This has *technically* been undefined behavior
- Although all Arm-based CPUs behaved this way
- The architecture now defines that this will work as expected

Key: Executing Processor, Modifying Processor

0 Executing processor runs code path (repeatedly)

1 Modifying processor writes new optimized routine in memory, flushes cache
Concurrent modification & execution with fetch speculation

JITs rely on certain behavior when optimizing running code
- This has *technically* been undefined behavior
- Although all Arm-based CPUs behaved this way
- The architecture now defines that this will work as expected

Key: Executing Processor, Modifying Processor

0 Executing processor runs code path (repeatedly)

1 Modifying processor writes new optimized routine in memory, flushes cache

2 Modifying processor writes branch to new optimized routine in memory, flushes cache
Concurrent modification & execution with fetch speculation

JITs rely on certain behavior when optimizing running code
- This has *technically* been undefined behavior
- Although **all** Arm-based CPUs behaved this way
- The architecture now defines that this will work as expected

Key: Executing Processor, Modifying Processor

0 Executing processor runs code path (repeatedly)

1 Modifying processor writes new optimized routine in memory, flushes cache

2 Modifying processor writes branch to new optimized routine in memory, flushes cache

3 Executing processor now runs new code path **AND MUST GET NEW CODE**
Floating-Point rounding

Different architectures handle FP to Integer conversions different for out of range & NaN

This difference is very rarely important as most conversions are in range, but it can produce a difference in behavior of programs

New RINT[32,64][Z,X] instructions provide the alternate behavior on Arm

<table>
<thead>
<tr>
<th>Number</th>
<th>'Classic' Arm</th>
<th>New Alternate</th>
</tr>
</thead>
<tbody>
<tr>
<td>+OutOfRangeIntRange</td>
<td>MostPosInt</td>
<td>MostNegInt</td>
</tr>
<tr>
<td>-OutOfRangeIntRange</td>
<td>MostNegInt</td>
<td>MostNegInt</td>
</tr>
<tr>
<td>NaN</td>
<td>0 (+Invalid Exception)</td>
<td>MostNegInt</td>
</tr>
</tbody>
</table>
Virtualization efficiency improvements

• Various coarse grained EL2 traps have been split out into multiple trap groups
  • Reducing the number of Hypervisor entries needed

• Separate EL2 traps for:
  • Access to CLIDR_EL1/CSSELR_EL1/CCSIDR_EL1 vs CTR_EL0
  • IC IALLUIS vs DC CVAU/IC IALLU/IC IVAU
  • TLBI …IS instructions for the non-broadcast cases

• Separate ID for which translational granules support for Guest OS vs Hypervisor
  • Allows the Nested Hypervisor to distinguish Stage 2 rules from Stage 1 rules as part of the nesting
Context Synchronization on Exception Entry/Exit

Current Arm architecture requires that Exception Entry and Exit are Context Synchronization Events.

For some supervisory code this is an unnecessary requirement.

Armv8.5-A adds SCTLR_ELx.{EIS,EOS} to control context synchronization on ELx Entry/Exit.
Cache Clean to Point of Deep Persistence

Systems involving Persistent Memory have the concept of “Point of Persistence”

Armv8.2-A introduced DC CVAP to cache clean to the Point of Persistence

However, the industry has moved forward to identify an additional point of “deep” persistence

Armv8.5-A introduces DC CVADP to cache clean to this point of deeper persistence
Arm A-Profile Architecture

- Crypto
- AArch32
- AArch64
- SIMD
- SVE
- RAS
- Pointer Auth
- VHE
- Stat Profiling
- RCpc
- Nested Virt.
- Mem Model
- Secure EL2
- Crypto ++

Armv8.0-A
Armv8.1-A
Armv8.2-A
Armv8.3-A
Armv8.4-A
Arm A-Profile Architecture
Find out more

https://developer.arm.com/products/architecture/cpu-architecture/a-profile/exploration-tools

Exploration Tools

System Register XML for Armv8.4 (00bet7).
This package provides descriptions in XML and HTML format for the System registers and memory-mapped registers defined in the Armv8-A Architecture, for Armv8.4.

Download XML

A64 ISA XML for Armv8.4 (00bet7).
This package provides descriptions in XML and HTML format for the A64 Instruction Set Architecture in the Armv8-A Architecture, for Armv8.4.

Download XML

View HTML

AArch32 ISA XML for Armv8.4 (00bet7).
This package provides descriptions in XML and HTML format for the A32 and T32 Instruction Set Architecture in the Armv8-A Architecture, for Armv8.4.

Download XML
Thank You
Danke
Merci
谢谢
ありがとうございます
Gracias
Kiitos
감사합니다
धन्यवाद
תודה
Matthew Gretton-Dann
Director, Product Management
Architecture and Technology Group
matthew.gretton-dann@arm.com

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks