Run-Time Protection

May 2021

Jan-Erik Ekberg

Adj. prof, Aalto

Head of Security, Huawei Finland

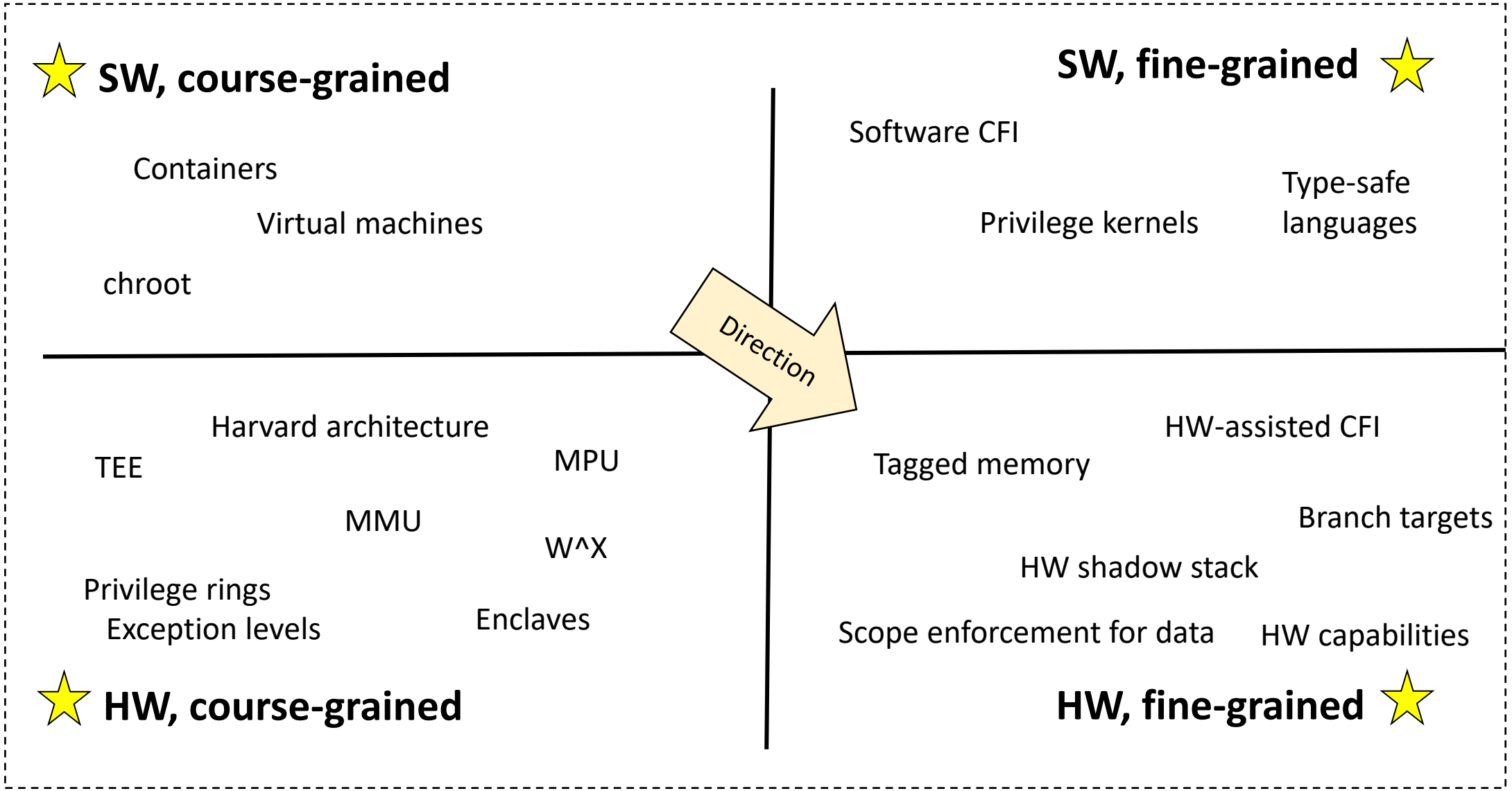
Contents

Run-time attacks / protection. One name for attacks / protection that occur when the computer is running , and primarily these leverage some software shortcoming and circumvent microarchitecture protections (where they exist). HW isolation is the first course grained feature that is instrumental to many of the protection mechanisms.

- Motivation, history (very brief)
- Course-grained HW isolation deplyed since the 80:s
- MMU-based HW protection mechanisms
- Fine-grained HW-based mechanisms
  - Point solutions
  - General primitives
- How are fine-grained solutions used. One achitecture example
- Conclusions: Where does architecture go from here

Note1: We will not consider HW isolation for trusted execution and enclaves. Other lecture
Note 2: When given the choice of architecture on which to present a feature, ARM is illustrated (most primitives exist in some form on Intel/ARM/RISC-V architectures)

Run-time protection / Memory Isolation Mechanisms

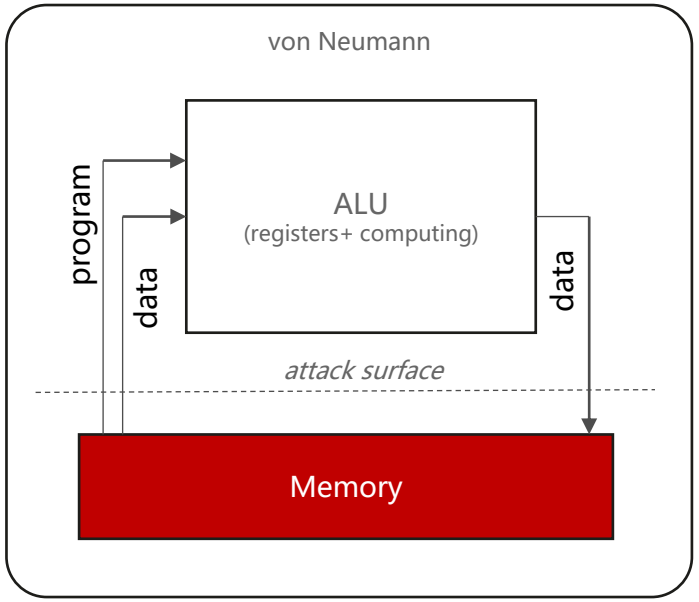


Just one categorization .. Does not include e.g. barriers use against Side-Channel attacks

Motivation

von Neumann architecture
a.k.a. the stored program computer

- When general-purpose computing was conceived in 1960s platform security was not relevant, machines were uniquely used for computing.
- Today, computers mainly perform data aggregation, storage, presentation & communication (value is in the data) AND the computing resource is multi-tasking for many stakeholders.
- Security is protection of (user) data and the protection of the computing system against attackers (that want to steal the data)



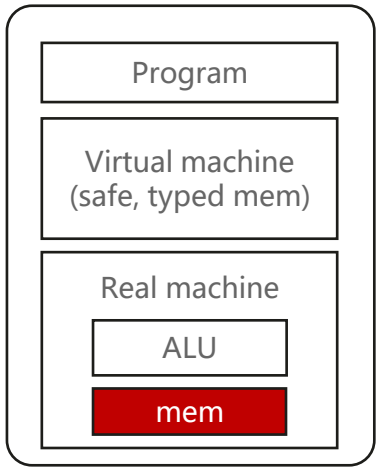
- From the processor perspective, the fundamental security flaw in von Neumann is that there is a type control mismatch between memory and ALU. At large, memory controllers have no support for typing. Example: Code and data can be confused and *code overwritten as data* by the attacker.
- Security issues in this category are typically called memory vulnerabilities, 70% of the successful attacks (viruses, root-kits, data theft) leverage at least one such vulnerability. The attacks are either spatial or temporal in nature.
- Since 1970s these problems have been kept in check by mechanisms like virtual memory, privilege rings, memory layout randomization and access control in the SW/HW interface, but today, these mechanisms are too course-grained for proper protection.

Solve the Memory Issue in Software

– what was tried and where we failed

- When hardware is not enough, software to rescue: Construct a safe virtual machine. Has been done: a) performance is not enough, b) only as memory safe as its implementation.
  - Also, what starts out as software virtualization traditionally breaks out of its box because of performance (e.g. Java JIT). a) → b)
- Use type-safe programming languages. For high-level, interpreted context could be a solution (but in browsers, JavaScript is more or less untyped ...). For systems programming, I/O, performance and regression breaks type safety (e.g. Rust)
- " Do not make errors in software – use good security tools" . Easier said than done. Programming is one of the most logically complex human endeavors, and maybe most telling is the Linux OS kernel. In this singular piece of software, with the best programmers worldwide working on it, some 50-100 memory vulnerabilities are uncovered per year. There are some small (simple) pieces of software with formal proofs for security, but the cost of such endeavors is prohibitively high.

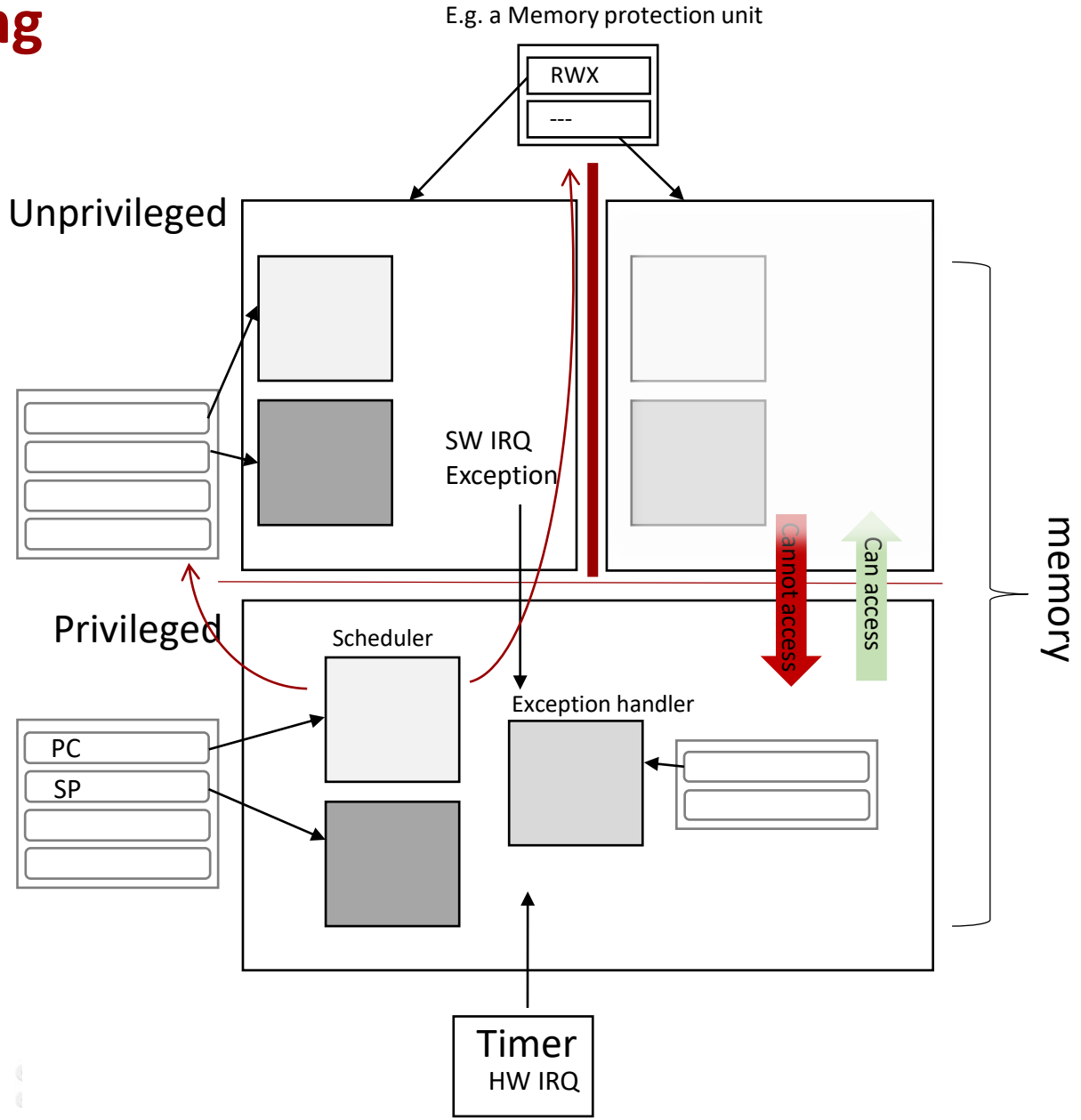
The power of indirection

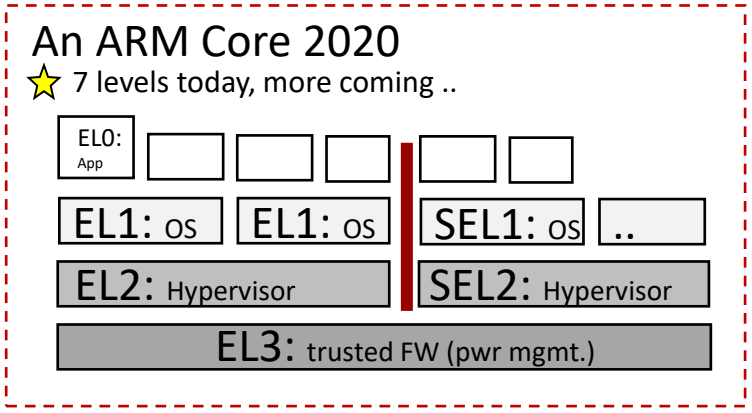


Whichever software protection mechanism we deploy, the processing overhead ranges from 30% to 500% or more for full memory protection. Not an an acceptable ratio.

Isolation: Privilege levels and Scheduling

- The privileged context is in charge of **scheduling**
- **Banked registers** allow for transitioning
- **Memory protection** hardware can shield memory (what memory is visible at what time)
- The **context switch** can happen **pre-emptively** based on a timer, or "cooperatively" using a SW IRQ (syscall)
- Security is ultimately based on the fact that higher privilege levels can configure HW, where lower ones cannot





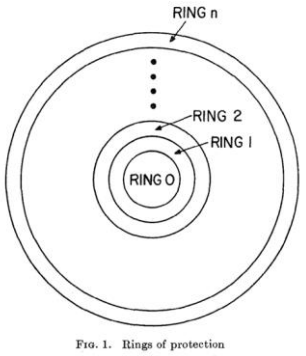
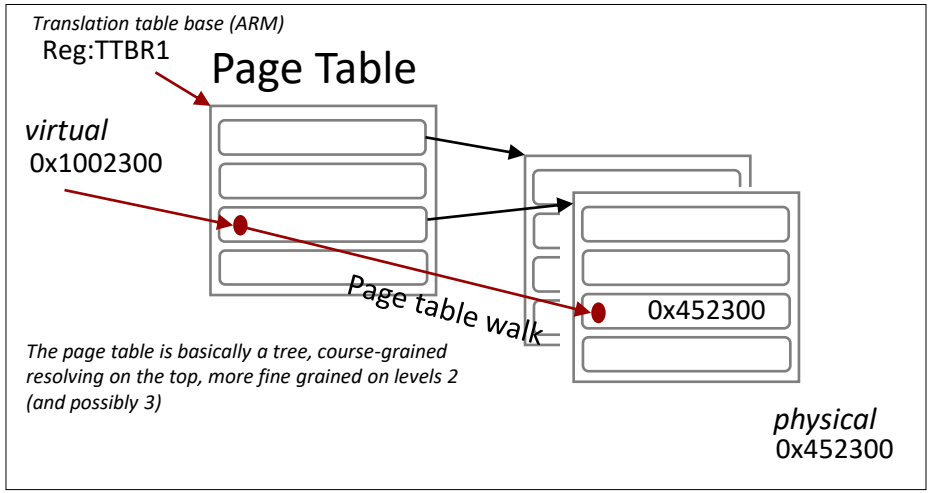


FIG. 1. Rings of protection

Isolation: Virtual Memory

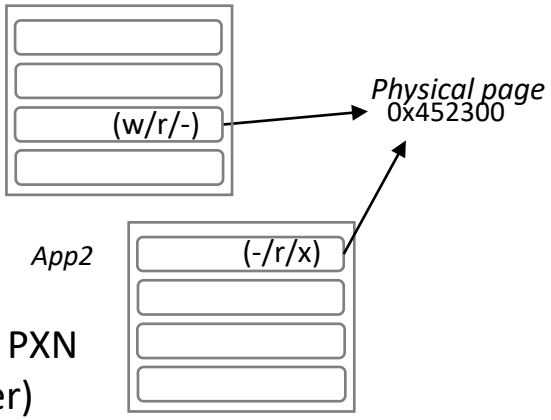
- Privileged context (kernel?) manages MMU and page tables. **Page tables** are stored in memory. System harmonized with caches (not shown here)
- Application sees only its own virtual memory / translation. If page table not configured (for a 4kB **page**), a trap happens. This is effectively a **sandbox**

Memory Management Unit (MMU)

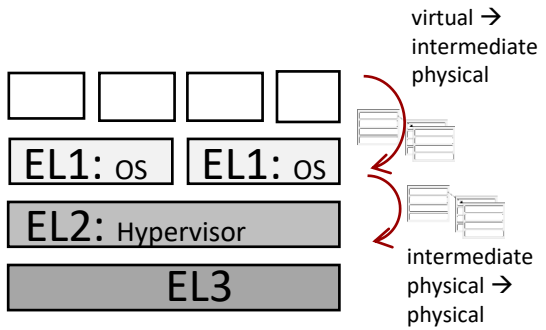


Page permission management (together with caches)

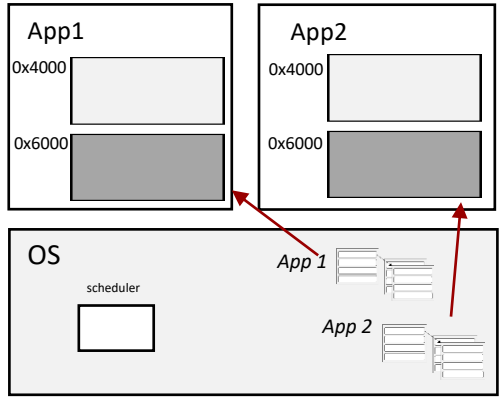
- The page translation can also apply access control for memory access
- These can be the "standard" R/W/X but also be constrained to privilege level and apply incrementally during page walk
- E.g. Execute Never (XN), PXN (Privileged Execute Never)



Translation domains



Same addresses / different mem.



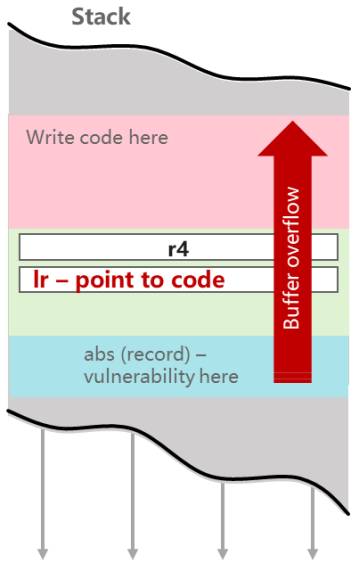
Traditional attacks and MMU fixes for them

- Since run-time attacks have been happening for a _long_ time, hardware has been adapting over the years, especially via memory management hardware

Execute from Stack

Mitigation: **W^X**

Use memory management rules to guarantee that writable memory pages cannot be executed. So stack can still be overflowed (it has to), but it cannot be executed.

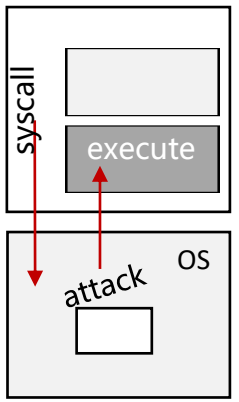


Return-to-User

Mitigation: **PXN (SMEP – intel)**

Since user-space is mapped to kernel space during system calls, kernel can be tricked to execute code in user space.

PXN (privileged execute Never) makes a memory page non-executable from kernel



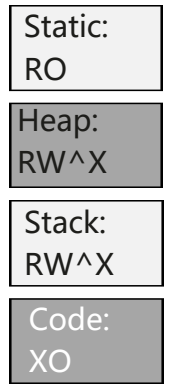
Execute-Only

Mitigation **XO**

Traditionally, code is readable, and also data can be interleaved with code in code segment (e.g. immediates for long jumps / branches)

Some attacks (ROP, see later) makes use of this opportunity to read code.

XOM is memory labeled to be non-readable while executable. Works well with ASLR (see later)



https://keenlab.tencent.com/en/2016/06/01/Emerging-Defense-in-Android-Kernel/

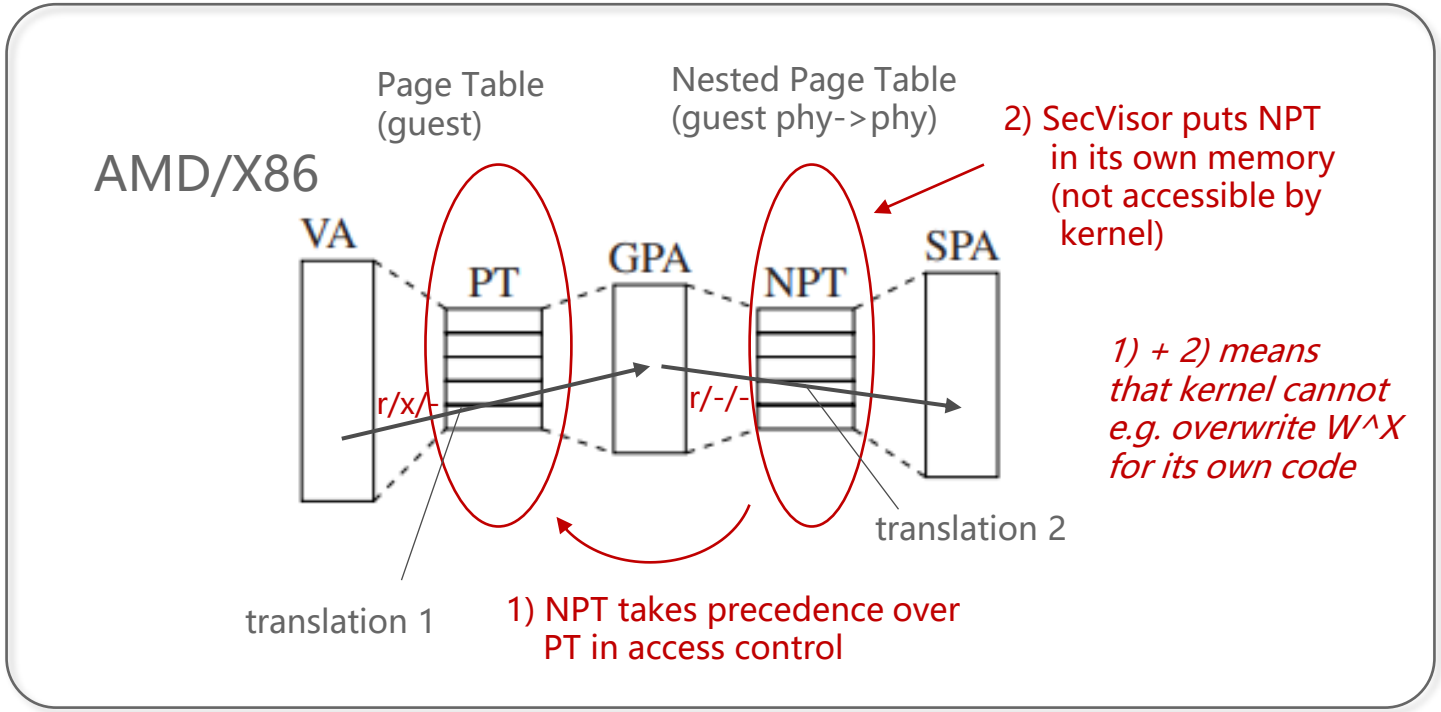
Kernel MMU protection (mobile)

- Most mobile phones use the HYP mode for run-time protection in ARM (Apple devices have hardware modification to them same effect)
  - SecVisor (2007) introduced this for the first time. (.. using AMD SVM tech. ..)

- The main insight is that **nested tables** are designed such that the **virtual physical → physical mapping access control takes precedence.** Even though the kernel needs to modify the first translation table, e.g. its own code (w^x) can be protected by the second translation, if the second translation table is "hidden away" from the kernel.

- By and large, **hypervisor control** for nested tables can provide memory protection for kernel (from itself) with

- **no significant extra overhead**, i.e. no need to check every page table update (expensive). There is a need for some extra checks (discussed later)



https://dl.acm.org/doi/pdf/10.1145/1294261.1294294

The Principle of a function call on the architecture level

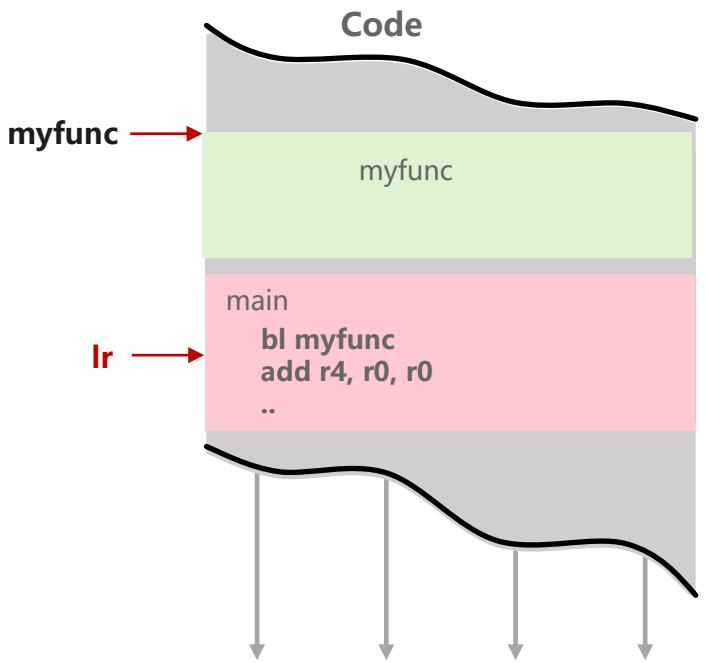
- The stack is a scratchpad of memory that holds a wide mix of **state data**, **program data**, temporary **register overflow** and other things

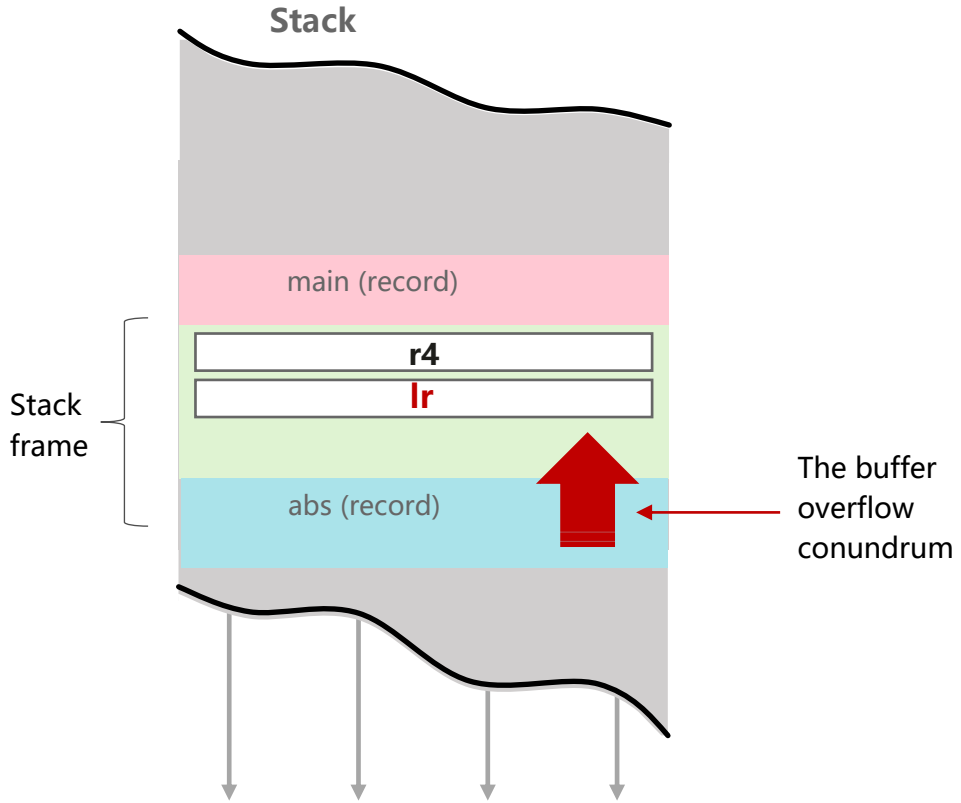
- Data in the stack is in most cases crucial to the correct / intended **call-flow** of the program

```
myfunc:
    push    {r4, lr}
    add     r4, r0, r1
    mov     r0, r2
    bl      abs
    add     r0, r4, r0
    pop     {r4, pc}

main:
    ...
    bl  myfunc
    ...
```

ARM assembler



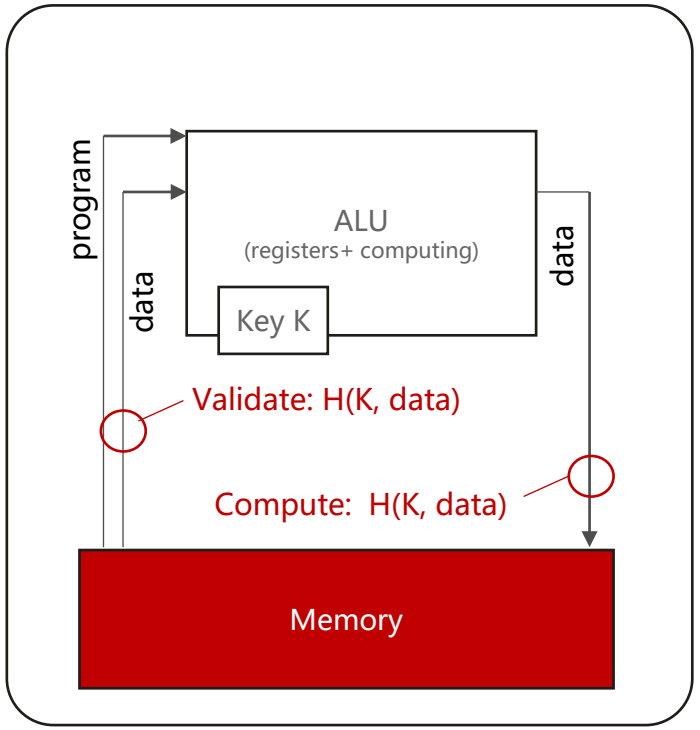


https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/how-to-call-a-function-from-arm-assembler

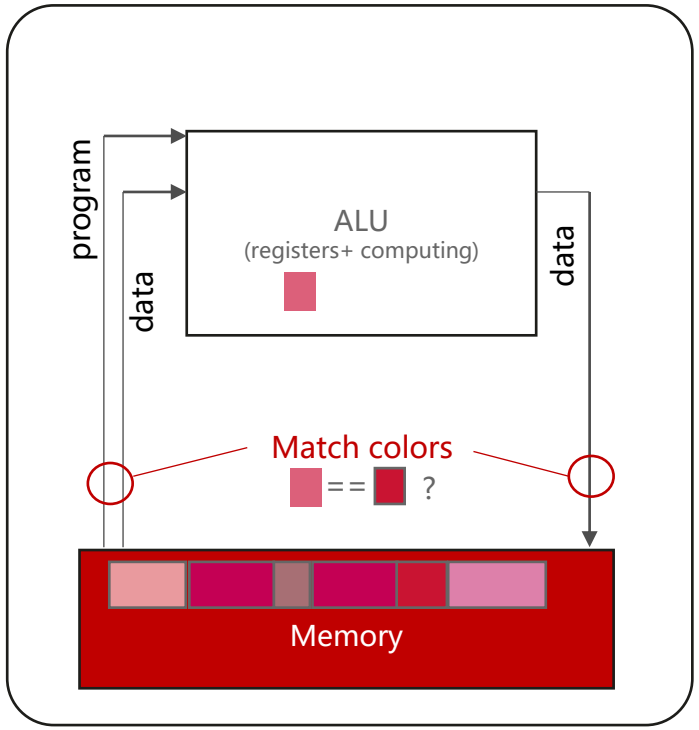
Processor Microarchitecture to the Rescue (1)

- The main APE designers (Intel, ARM, Oracle(?)) realized around 2010-2015 that memory protection is in dire straits and HW support is needed, and e.g. Intel is already in its 2nd generation of protection features. Later, the open RISC-V architecture has stimulated a lot of research (prototypes) in this area. The architecture additions range from general tools to specific point solutions:

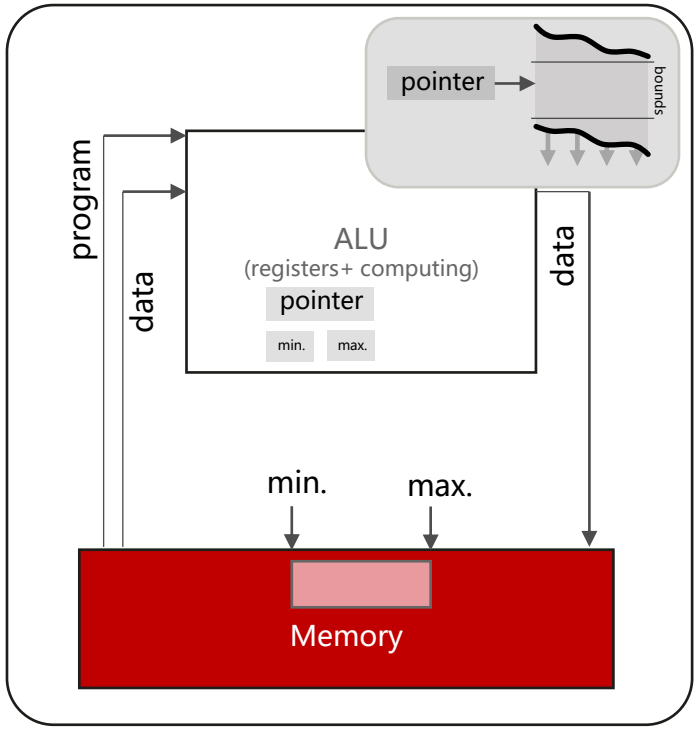
Cryptographic Pointer and Data Integrity
(ARMAv8.3 PA. RISC-V: pointers with residue codes)



Memory Tagging / Coloring
(ARMAv8.5 MTE. Oracle OpenSparc M7)



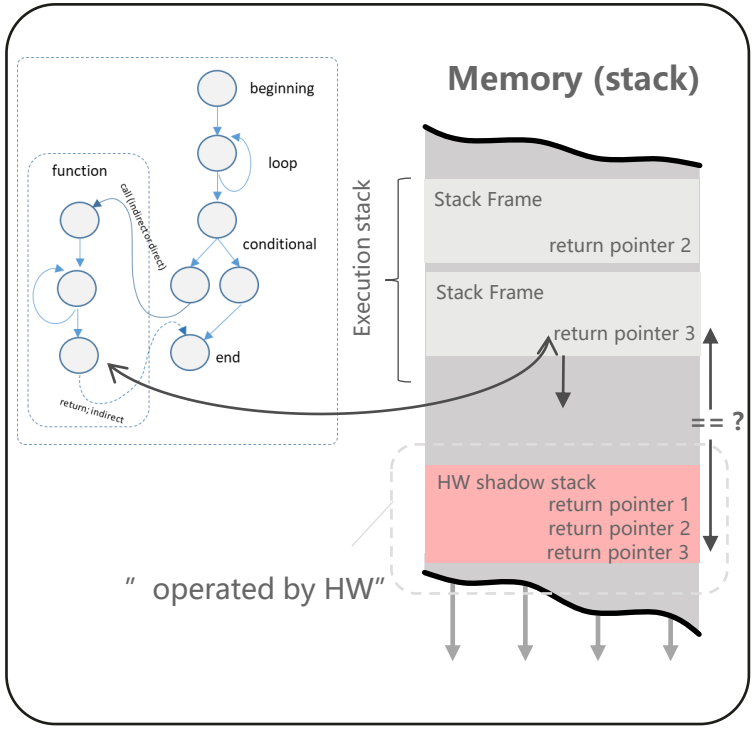
Pointer & reference bounds
(ARM Morello, Intel MPX (term.), RISC-V Shakti-T)



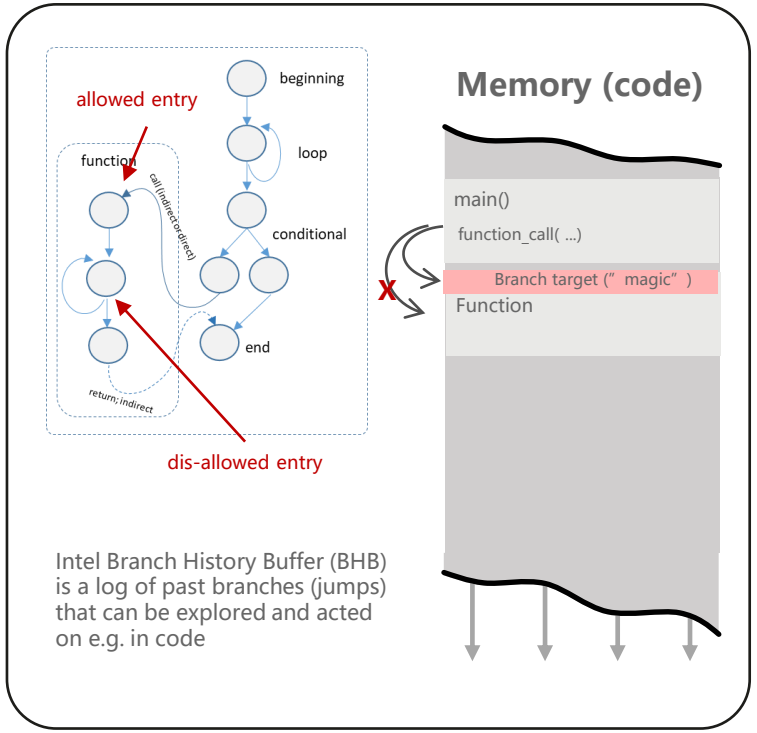
Processor Microarchitecture to the Rescue (2) – point solutions

- The main APE designers (Intel, ARM, Oracle(?)) realized around 2010-2015 that memory protection is in dire straits and HW support is needed, and e.g. Intel is already in its 2nd generation of protection features. Later, the open RISC-V architecture has stimulated a lot of research (prototypes) in this area. The architecture additions range from general tools to specific point solutions:

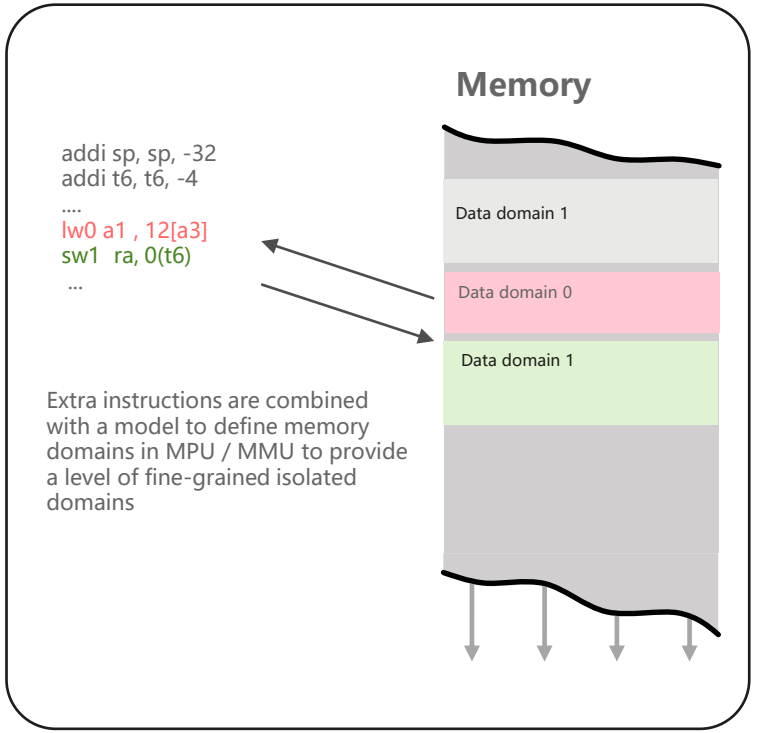
Hardware Shadow Stack
(Intel CET, RiSC-V Zipper)



Branch Targets
(ARMA8-8.5 BTI, [Intel BHB])

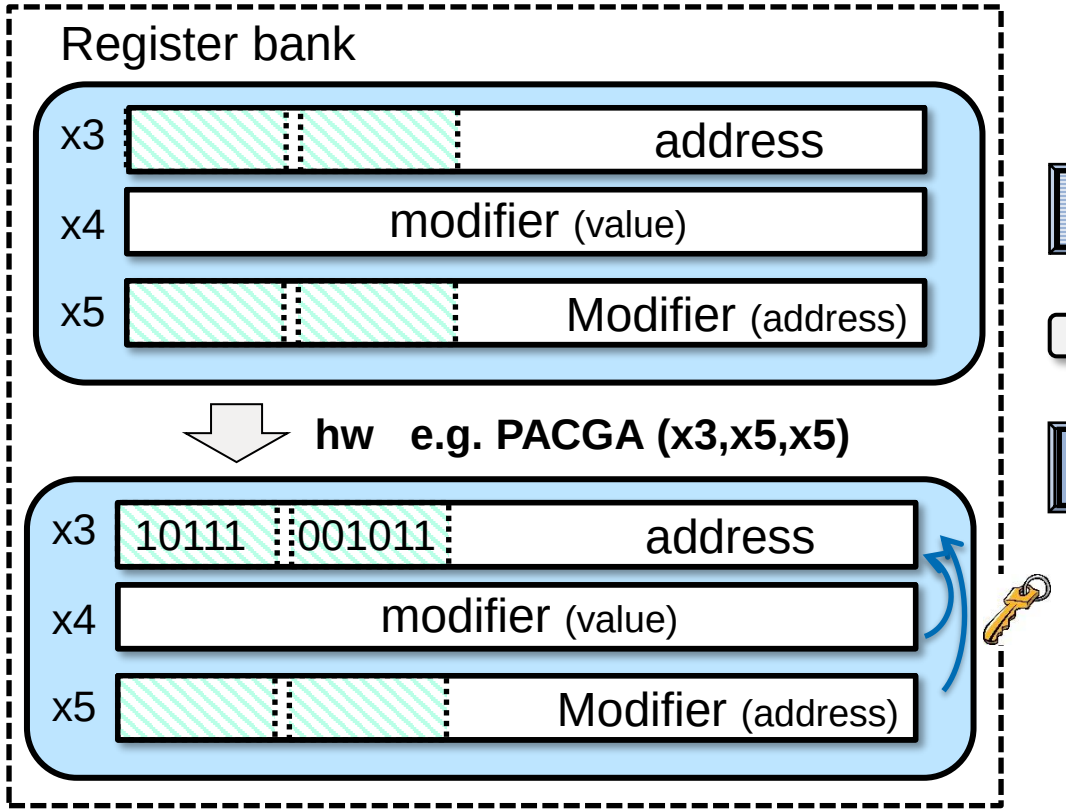


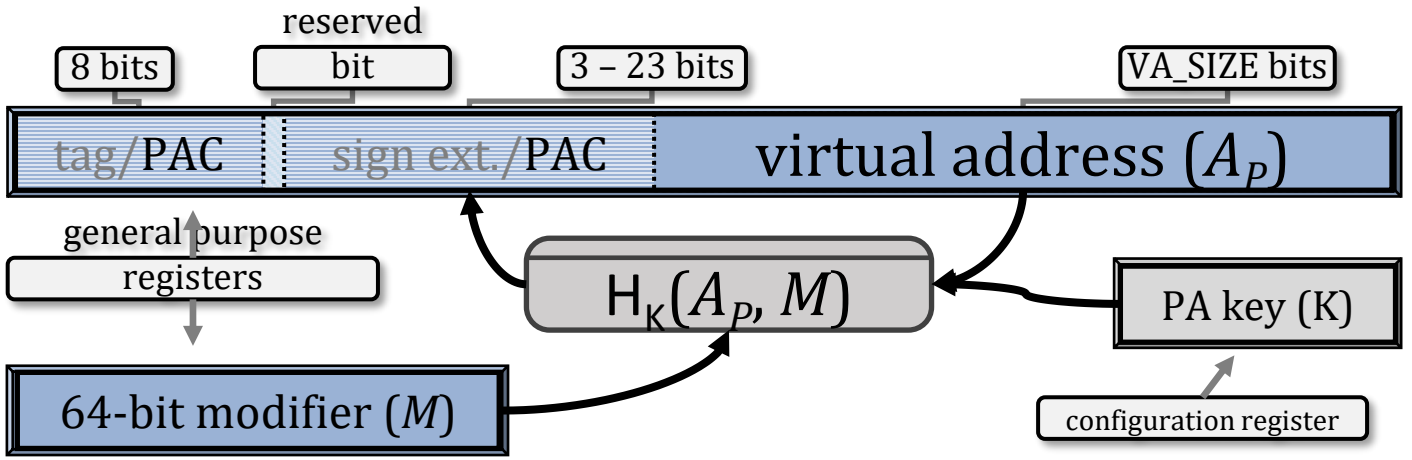
Fine-grained, instruction operated domains
(Intel Memory Keys, RISC-V RIMI)



Pointer Authentication in ARMv8.3-A

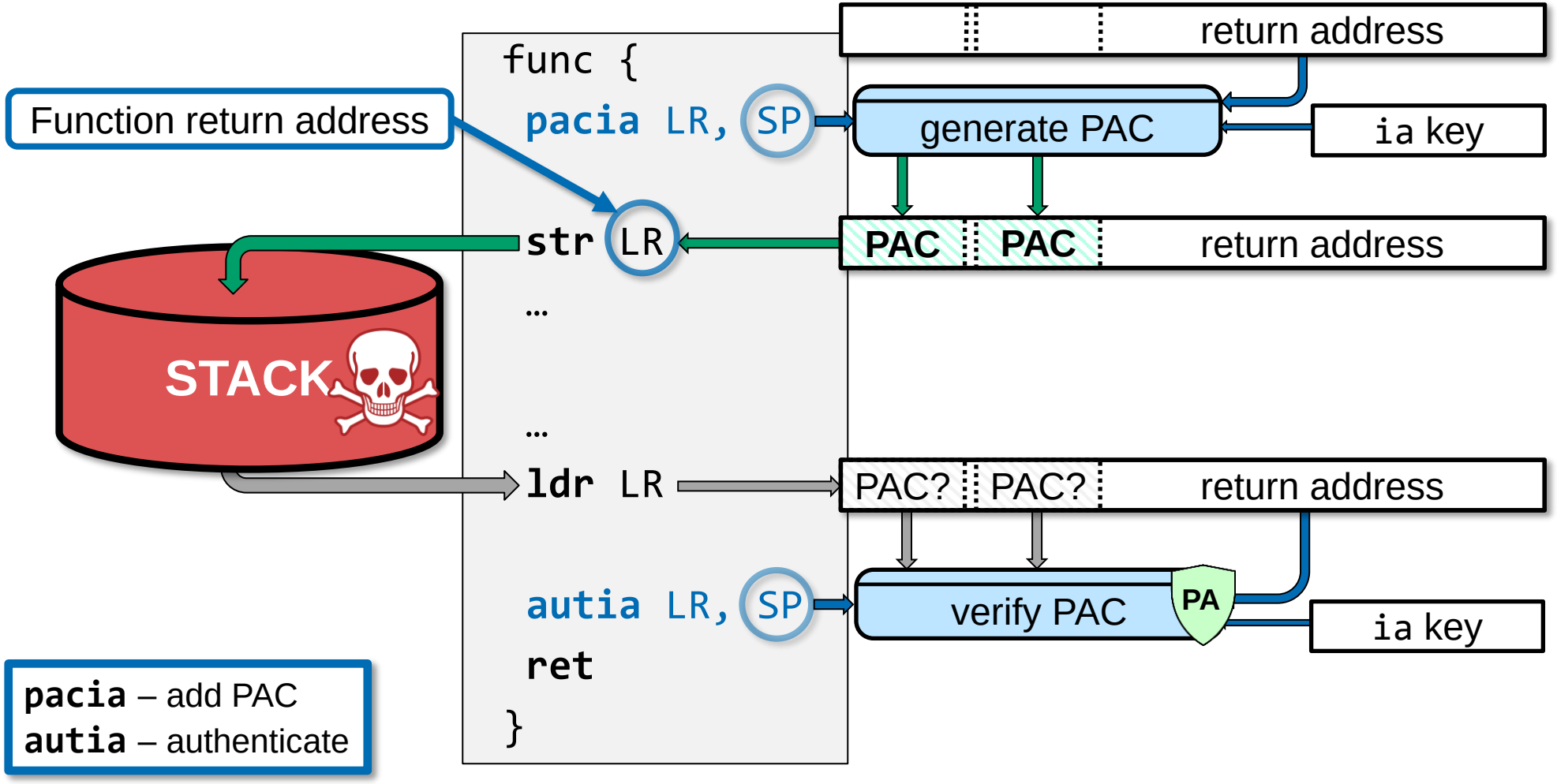
- **General purpose hardware primitive approximating pointer integrity**
- **Adds Pointer Authentication Code (PAC) into unused bits of pointer**
  - Keyed, tweakable MAC from pointer address and 64-bit modifier
  - PA keys protected by hardware, modifier decided where pointer created and used





Example: PA-based return address signing

Deployed as `-msign-return-address` in GCC and LLVM/Clang



Example: Tagging (in ARM MTE) -- principle

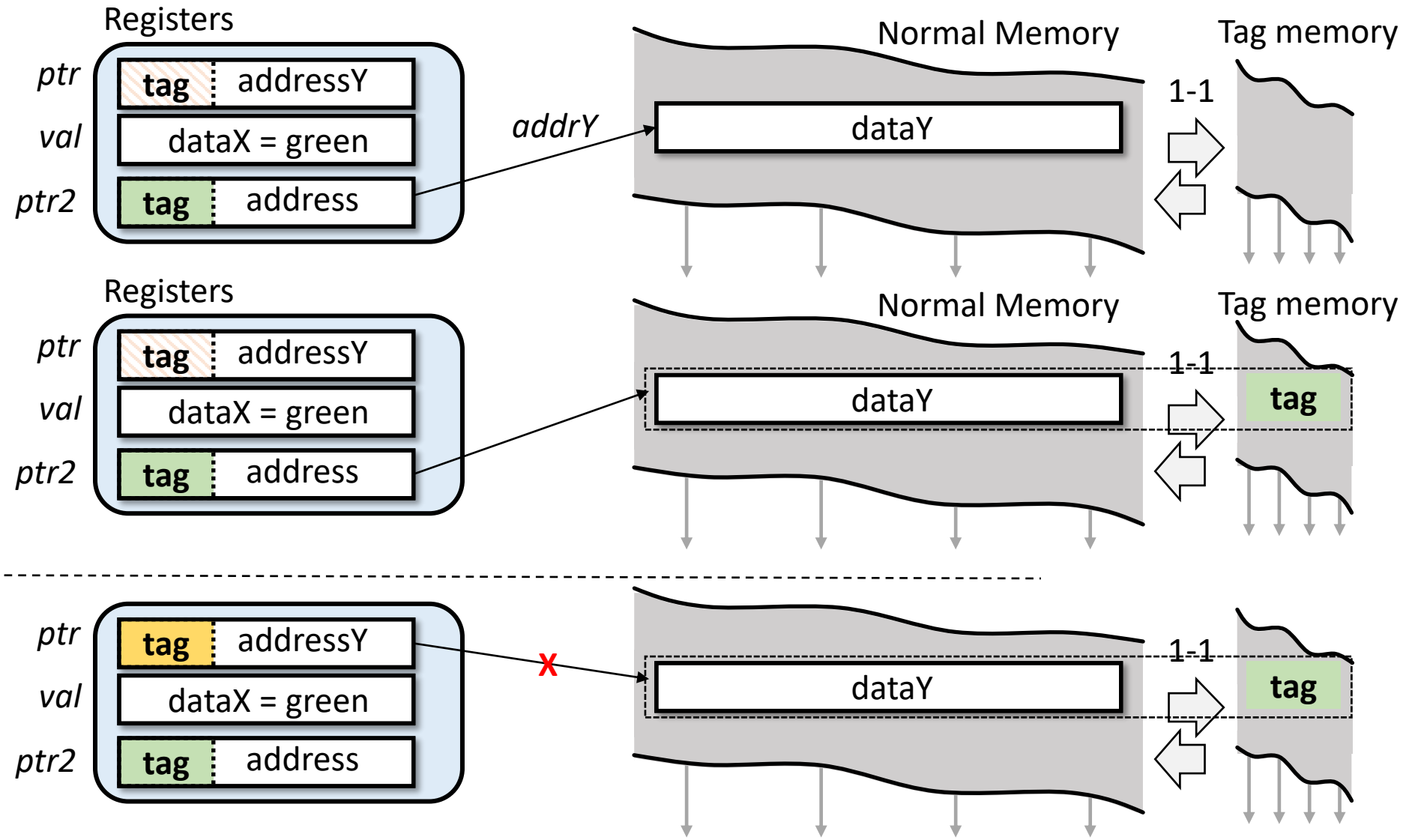
★ Writing colors to memory and assigning colors to pointers

1. **Assign tag color to pointer in register:**
   XRG(ptr2, ptr, val)
   .. or using arithmetic

2. **Color virtual memory based on indexed register (pointer) color:**
   STG(ptr2, 0)

3. **Most memory accesses will check tag color match:**
   LDR (val, ptr, 0) → fail

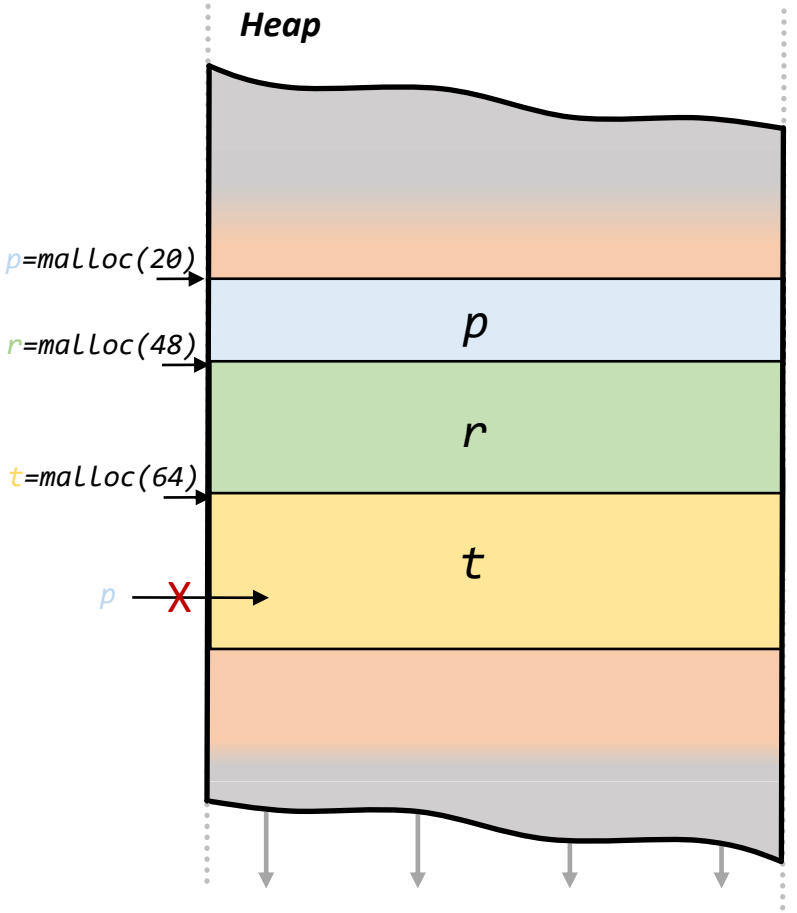
   tag <> **tag**



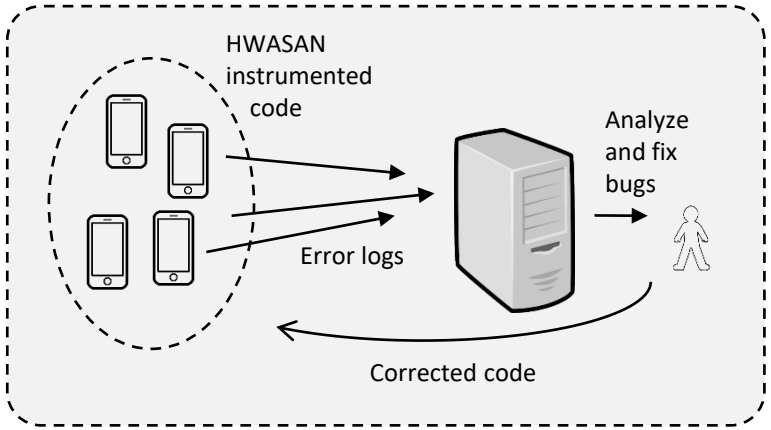
There are more details to consider. But the main principle is the comparison of tag colors

HWASAN (Google)

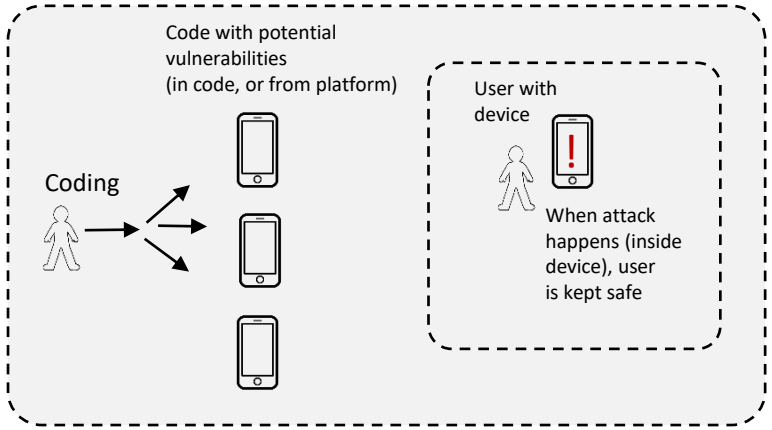
- **Statistical analysis** (mainly) from Google using memory tagging. Specifically a ***memory error detection tool***, not a memory protection tool
- Applies both to stack and heap allocations. Implemented in LLVM (not MTE yet?).
- Should allow for **massive testing** (testing on the field) without significant performance overhead (like with ASAN). Especially for codebases that are already connected (like a browser) this could be an ideal approach for maintaining code quality



Deployment principle (ASAN)



Deployment principle (memory protection)

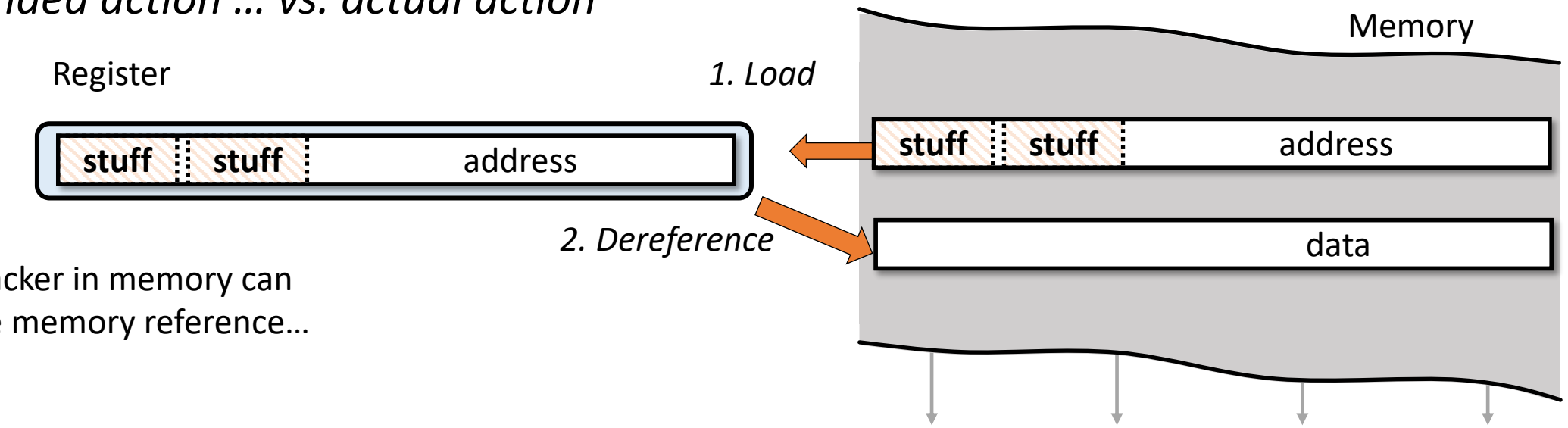


- If "blue" pointer p points (accidentally or intentionally) to the area of yellow t, MTE will trap. Chance of color mismatch is 1/16
- However, if colors are assigned randomly at each run, and program is run in a million devices, **trapping is virtually certain**

Even though ASAN and a deterministic solution cannot be used in parallel, we feel both are valuable

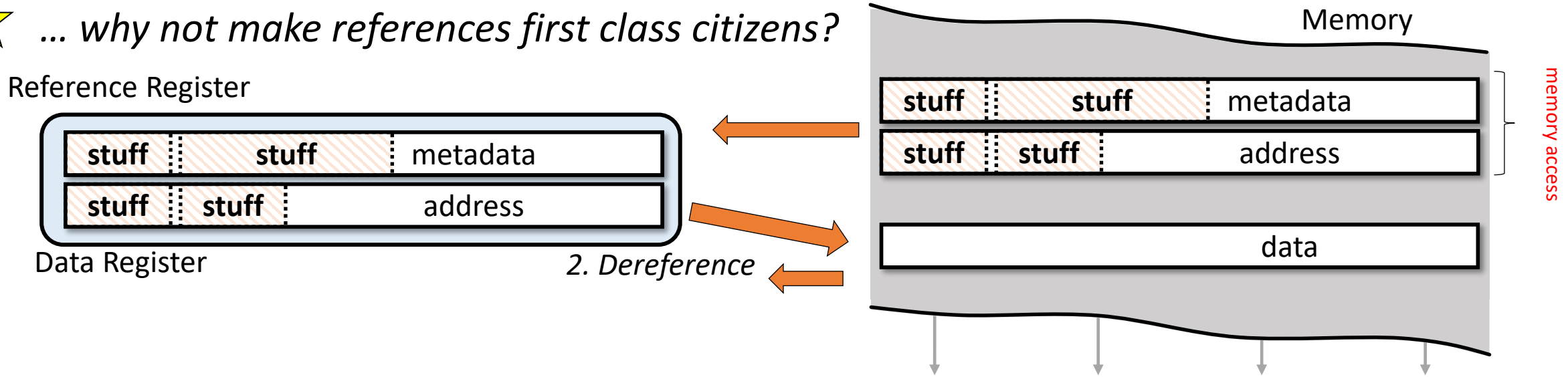
The problem we are facing today (even with PAC, MTE)

Intended action … vs. actual action



An attacker in memory can change memory reference…

… why not make references first class citizens?

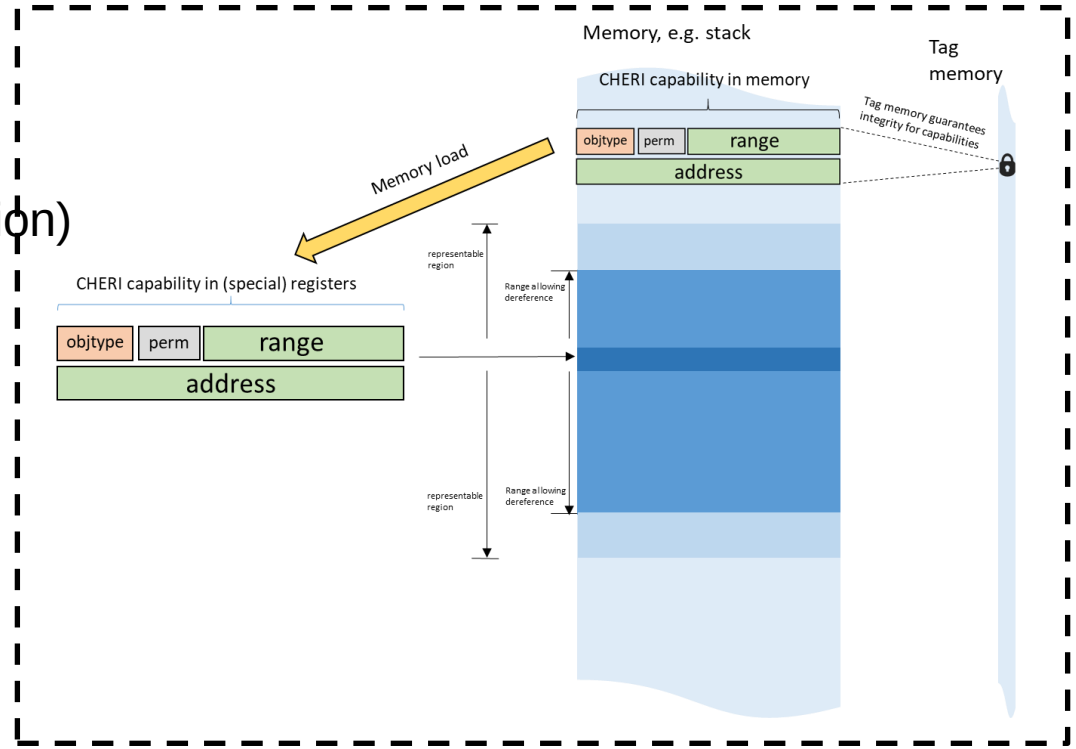


The idea of capabilities predates e.g. MMU as a mechanism for isolation

Run-time Protection, Capabilities (Cambridge Uni: ”CHERI”)

- **Capabilities for run-time protection** and **system protection** proceeds on many fronts currently
  - Capability-based ISA (machines): Recent ones include CHERI for 1) MIPS, 2) RISC-V
  - Capability-aware compilers (LLVM – not all back-ends compatible yet)
  - Full/partial-capability OSs (Google Zircon, CapROS. Google Hafnium for Hyp in Android)
- **Practical implementation** is still years away, but reasonable testing and architecture development can be done now (e.g. on RISC-V)
- **Capabilities in SW** and in **microcode** are two different things and can exist in indepence (cap-ISA can support e.g buffer-overflow protection, cap-SW can support access control). However, put together (full-capability implementation) e.g. significant speed benefits and consistent security architecture can be had.

https://github.com/CTSRD-CHERI
https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf
http://erights.org/elib/capability/index.html
https://fuchsia.googlesource.com/
https://fuchsia.miraheze.org/wiki/Main_Page
https://events.linuxfoundation.org/wp-content/uploads/2017/11/SecurityInZephyrAndFuchsia.pdf



Capabilites main benefits

CHERI (alone) is a HW design, and the benefit largely depends on how CHERI is used by software. Here are a few of the perceived benefits (all of them conditional to further research, innovation and architecture design), in order of complexity:

1. **Full-capability compilation** of software (kernel, applications, system services). This stops most (all) memory boundary violations, and therefore solves run-time attacks to a large degree

(however, full-capability runs into problems with IPC)

2. **Inter-process (inter-component)** communication is one place where the capability both causes problems and solves them. IPC authorization (where available) can be acheived with CHERI, and **big performance benefits** can be expected. However, the architecture design around IPC is complex, and may cause significant changes in ABI / APIs as well as compiler frameworks.

(and for small devices with no isolation, or in bigger devices with need for speed improvement)

3. **Tag / object type – based isolation.** On any level of abstraction (basic blocks, functions, libraries, applications, privilege levels) CHERI provides tools for workload / data isolation **at much higher performance levels** that software context switches or software access control checks. Significant software architecture design updates needed except in special cases (e.g. JNI / browser workload isolation)

The future ARM technologies are important, and come earlier, but their enforcement power is lower than CHERI

Implementation Pipeline: Security Microarchitecture

- Security Engineering is moving in the direction SW → HW

In RISC-V research. When leveraging features from ARM, Intel not done

Idea

Hardware design

HDL (e.g. In RISC-V)

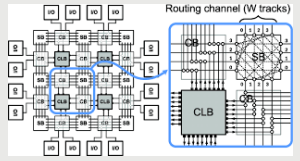
Prototype idea in simple processor

Update emulators

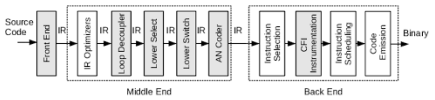
Adapt to FPGA board

Emulation / Synthethizing

QEMU, SPIKE, ..

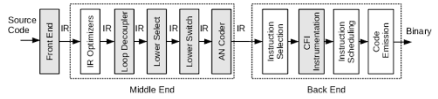


Compiler additions (LLVM)



Language-specific support (Rust ?)
Front-end mechanism adaptation
Back-end instruction generation

Architecture integration



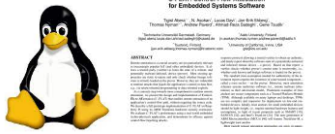
Add HW support in bootup / kernel
Solve hybrid SW conundrum
Protection: System services → Applications → Kernel

Performance measurement

- speed
- security
- reliability (resilience)

Product

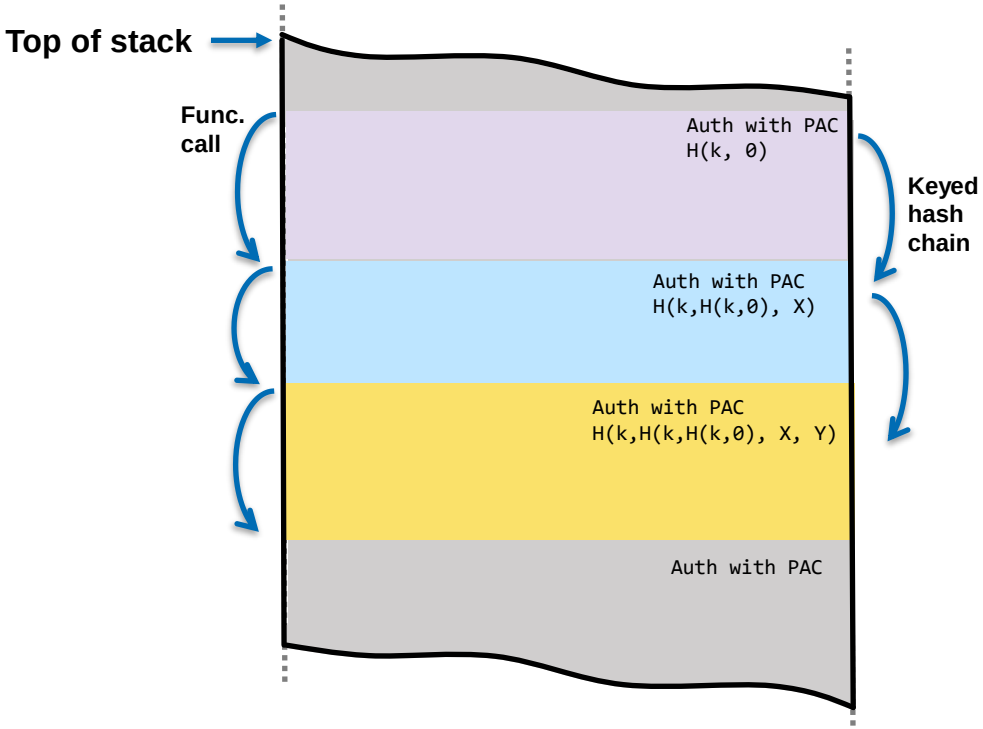
Dissemination

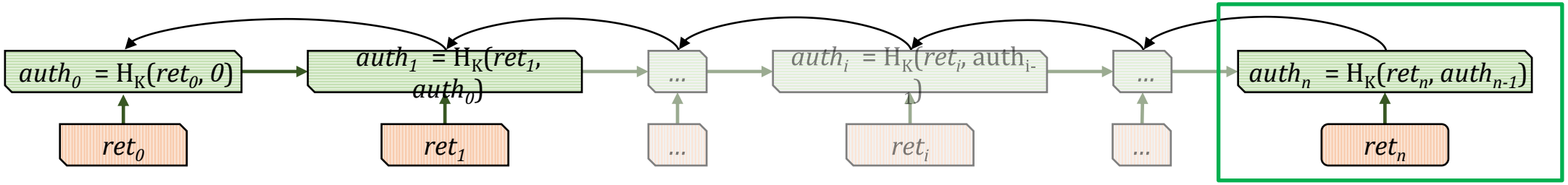


Authenticated Call Stack: high-level idea

Chained MAC of authentication tokens cryptographically bound to return addresses

- Provides modifier (*auth*) bound to all previous return addresses on the call stack
- Statistically unique to control-flow path
  - prevents reuse
  - allows precise verification of returns

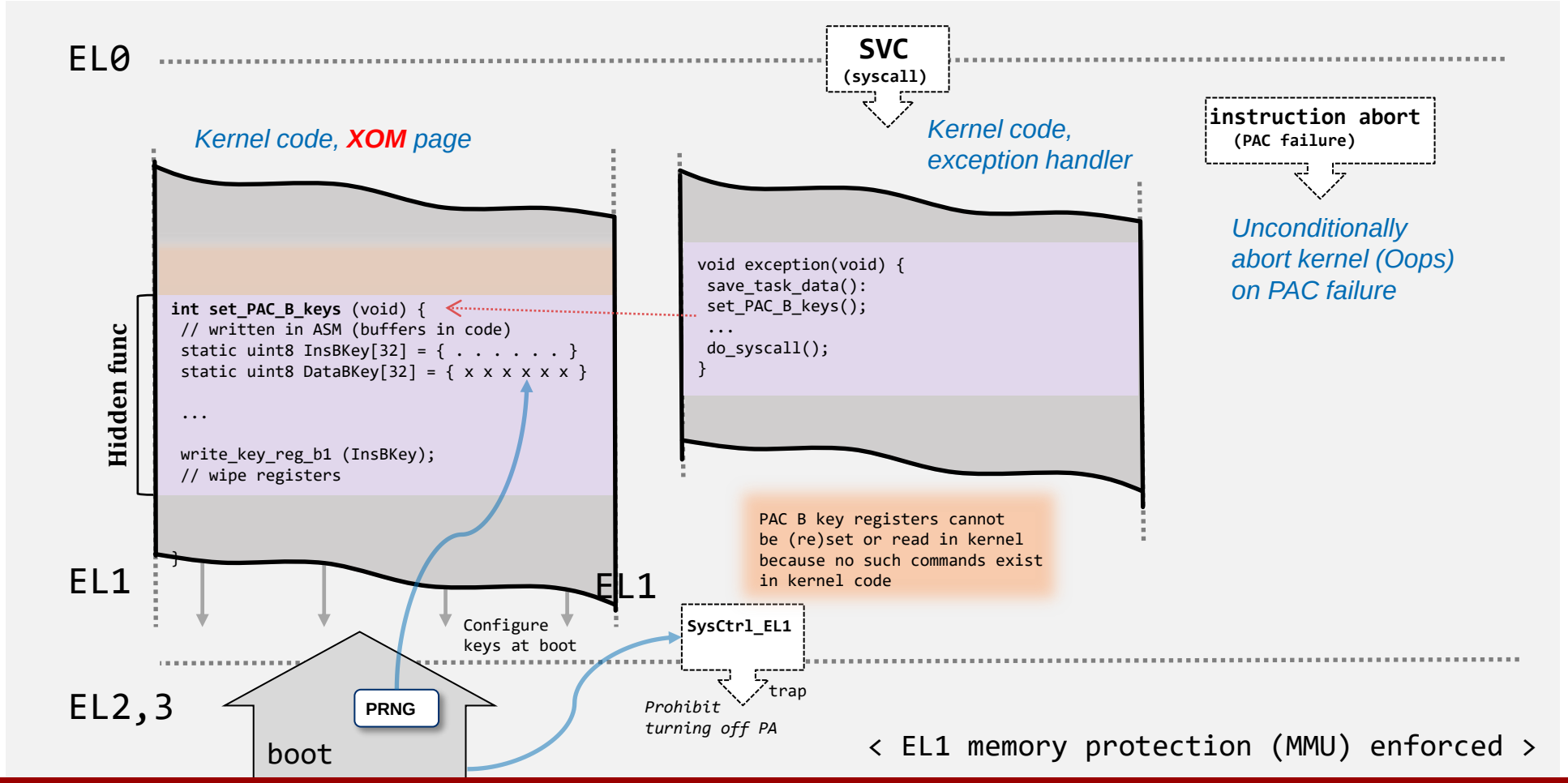




$auth_i$, $i \in [0, n-1]$ bound to corresponding return addresses, ret_i , $i \in [0, n]$, and $auth_n$

Kernel PAC Key Management Architecture

- The main challenge is to circumvent the kernel's own powers to configure the system and assign keys.
- XOM memory is used to maintain the confidentiality of kernel key assigment. EL2 protection applied for MMU safety
- User-space PAC keys are swapped in and out from task struct on context switch just like in upstream kernel / PARTS



DAC 2020: https://arxiv.org/pdf/1912.04145.pdf

Kernel PAC forward CFI

- For most indirect forward pointers in Linux kernel, access is via operation tables, not via function pointers. I.e. Forward CFI is implemented via an authenticated data pointer read and redirecting through a RO function pointer
- Applied using compiler attributes by semi-automated marking of function pointer types to be protected

Pointer Authentication

```
mov     w9, #0xfb45          type id
bfi     x9, x0, #16, #48     object
pacdb   x8, x9
str     x8, [x0, #40]
```

struct file address

Binds ”f_ops” in struct file

Pointer Use

```
 …
ldr     x8, [x0, #40]
mov     w9, #0xfb45          type id
bfi     x9, x0, #16, #48     object
autdb   x8, x9
ldr     x10, [x8, #112]
// ... blr x10               ”ext2_read”
```

pacdb – add PAC to data pointer
autdb – authenticate data pointer

Heap: allocated structs

```
struct file {
   …
40 const struct file_operations *f_ops;
   int f_mode;
…};
```

RO: vtable-like ops-structs

```
struct file_operations{
   …
112 ssize_t (*read) (..)
    ssize_t (*write) (..)
…};
```

RO: .text code

```
ssize_t ext4_read (..)
{
   …
}
```

Kernel PAC Return Address Signing

- Every task has its own kernel stack (at different places in kernel memory). Modifier is constructed based on stack address and executed kernel function (address), providing uniqueness between tasks and operations
- Applied using compiler automation. By using addresses, LTO is not needed (w.r.t PARTS)

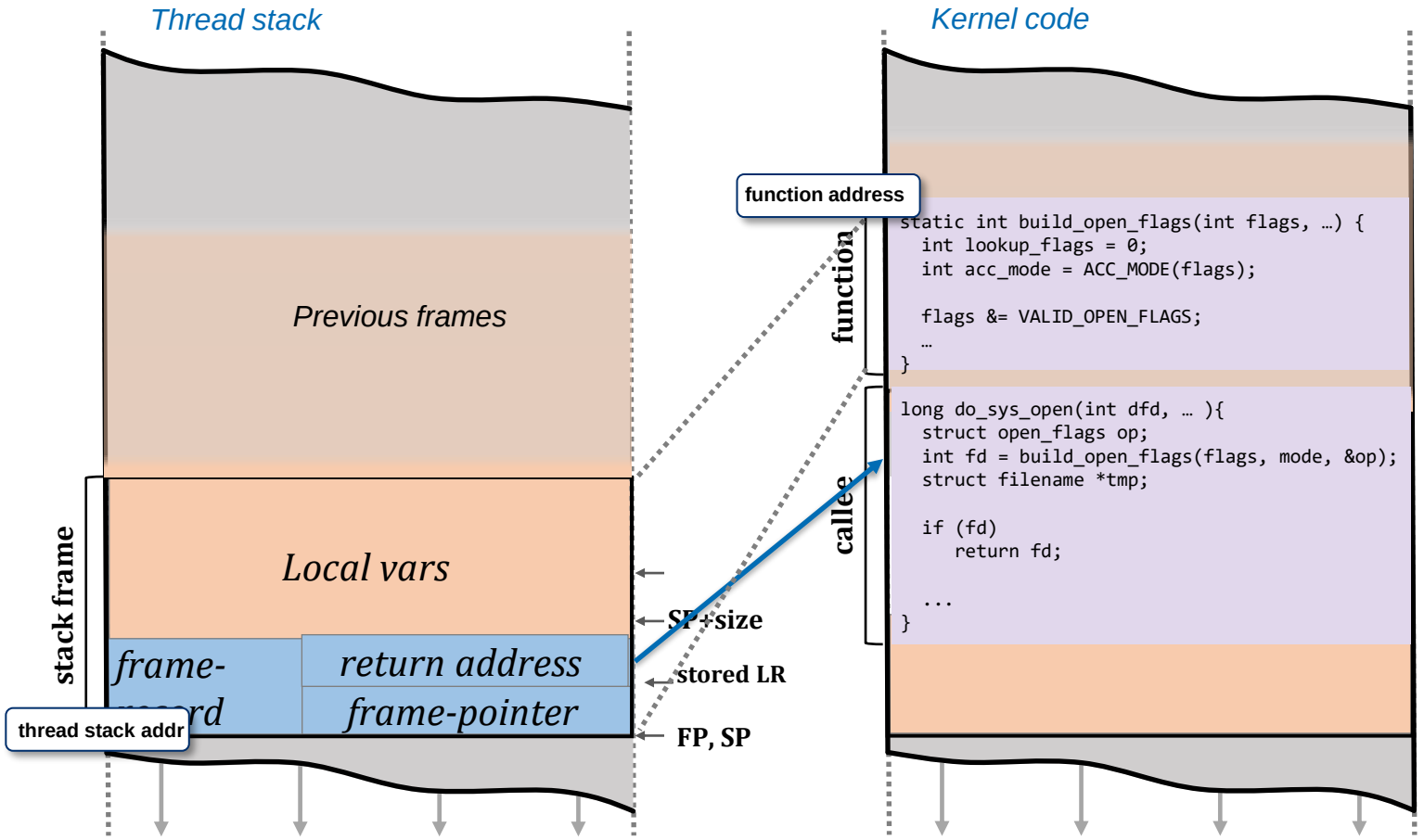
Function prologue

```
adr     ip0 , function      function address
mov     ip1 , sp            thread stack addr
bfi     ip0 , ip1 , #32, #32
pacib   lr , ip0
stp     fp , lr , [sp , #-64]!
mov     fp , sp
```

Function epilogue

```
…
ldp     fp , lr , [sp] , #64
autib   lr, ip0
ret
```

pacib – add PAC to code ptr
autib – authenticate code pointer



ReCap: Memory Protection in Hardware – History (1):

Solutions dedicated to solving contemporary memory protection issues (intel)

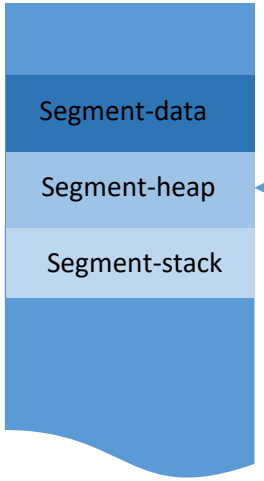
Hardware segments

- ★ First in MULTICS
- ★ Still visible in i286 / Win 3.1

Good memory separation, clumsy to operate, compile

Course granule

Ptr: SEG-register + segment offset

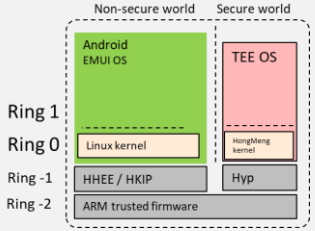


Hardware rings / hardware domains (TZ)

- ★ First in MULTICS
- ★ In significant security use especially on ARM

Protects domains from each other, hyp / kernel / user-space separation

Efficient specifically on macro-level



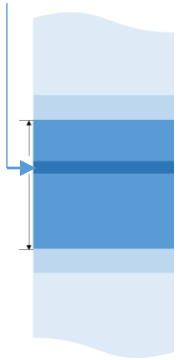
Hardware capabilities

- ★ First in Cambridge CAP computer
- ★ Not visible in contemporary devices

Fine-grained access control + boundary checks

Efficient, small granule

Ptr: base + limit



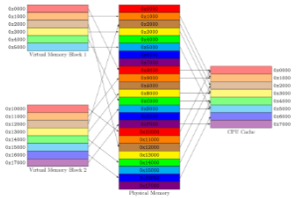
Hardware tags / coloring

- ★ First in Burroughs (1970s, later in Sparc -- 2010s)
- ★ 'Displaced' by virtual memory)

Temporal safety for bounds checking, memory reference type support

Efficient, restricted by tag resolution

*)

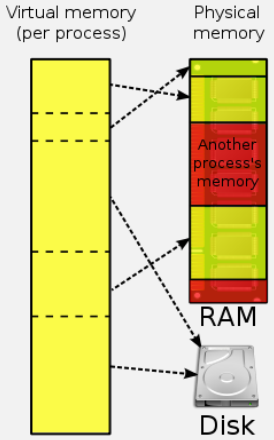


Hardware address translation

- ★ First in VAX/VMS (~1980)
- ★ Omnipresent in contemporary architectures

Page-based access control, cache support

Effective for e.g. process isolation



Privilege domains + VM

Main (only) HW Security solution 1985-2015

Granularity: 1 page (+ banked regs)

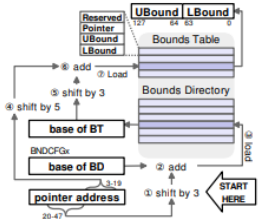
Code-controlled bounds checking

- ★ Intel MPX (2017-)
- ★ Very low overhead

User code-operated, stacked bounds enforcement

Bounds checking with non-fat pointers

**)



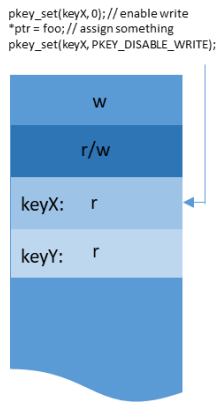
Memory Protection Keys for UserSpace

- ★ Intel MKU (Skylake ->)
- ★ First iteration IBM/360, in 60s but then for task isolation

User code-operated, page-based access control domains

Effective for e.g. rare-write protection

***)



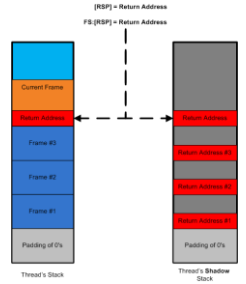
Hardware shadow stack

- ★ Intel CET (2019-)
- ★ To-be deployed

Hardware shadow stack, providing full return edge CFI,no overhead

Dedicated primitive for important CFI prot.

****)



1960 — 1970 — 1980 — 2015

*) http://www.feustel.us/Feustel%20&%20Associates/Advantages.pdf

**) https://intel-mpx.github.io/code/submission.pdf

***)https://www.kernel.org/doc/Documentation/x86/protection-keys.txt

****) https://eyalitkin.wordpress.com/2017/08/18/bypassing-return-flow-guard-rfg/

Traditional HW course-grained, and for page protection, higher resolution rare in commercial processors

ReCap Protection in Hardware – History & Future (2):

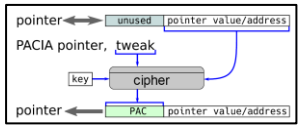
Pointer Authentication

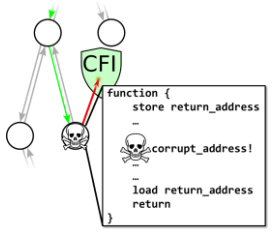
- ★ Design by QC
- ★ First in Apple Xs

HW-orchastrated cryptographic MAC for pointers

Tweaked cipher gives high speed (4 cycles / op)

Return CFI, other CFI variants

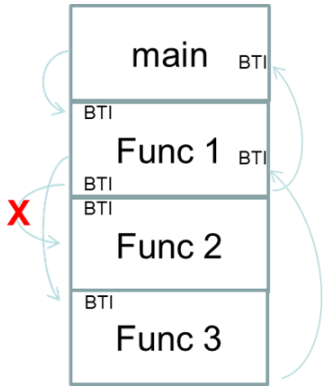




Branch Target Integrity

- ★ Function entry at beginning enforced
- ★ First in ARMv8-M (2015-). In A8.5

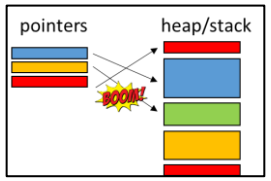
Low-overhead mechanism with lot of potential to support other primitives



Memory Tagging

- ★ To be lauched in ARM-Av8.5
- ★ Primarily intended as a statistical heap overflow protection

Like other tagging schemes, can be used for scope enforcement in limited scenarios



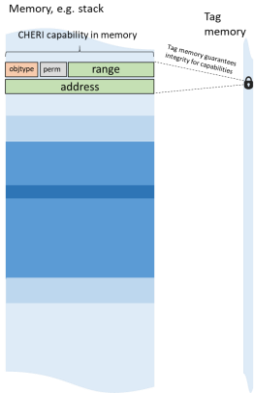
arm 2018 8.3 2020 8.5

Selected Public Research

CHERI (2014-)

- ★ Univ. Of Cambridge design for capabilites and permissions (computing domains)
- ★ Working prototypes for MIPS, ARM, RISC-V

Full-fledged capability system, powerful, but requires large SW arch. modifications



https://arxiv.org/pdf/1705.10295.pdf

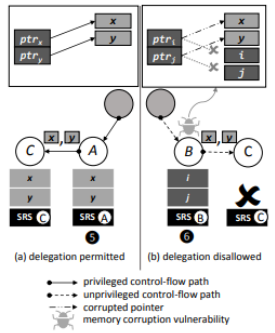
https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

HardScope (2017-)

- ★ Aalto / Darmstadt work on HW scope enforcement
- ★ Working PoC for RISC-V.

Architectural HW design with recursive refinement & delegation of memory scope.

Good performance overhead (~3%)

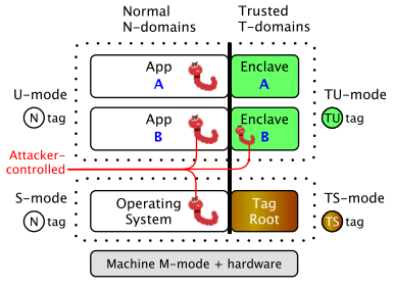


https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_10-3_Weiser_paper.pdf

Timber-V (2018)

- ★ Darmstadt / Graz work – Risc-V memory tagging
- ★ Domain construction (esp. for enclaves), sharing considered

Not necessarily fine-grained, more a MPU extension for emulation of SGX / TZ / ..



2014 2018

Fine-grained primitives on the horizon again, in new processors 2020→

Run-time protection in the future: Architecture changes

- There are indications that the linear computer architecture evolution has come to end, and we are facing a reboot of security design in the SW/HW interface
- It would be preferrable, that security support / functionality is considered from the start rather than being an add-on. Especially Integrity and Isolation for code and data are important features to consider at this level

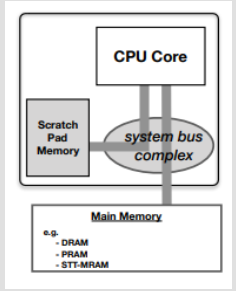
" Escaping" the von Neumann architecture

Non-Volatile RAM

Technologies like PRAM /MRAM etc. provide opportunities to make computing with no state loss possible. How is this beneficial / detrimental for security design?

Example: Security Aware Scratchpad Memory (SA-SPM). Encryption issues for NV_RAM

https://dl.acm.org/doi/pdf/10.1145/3316482.3326347



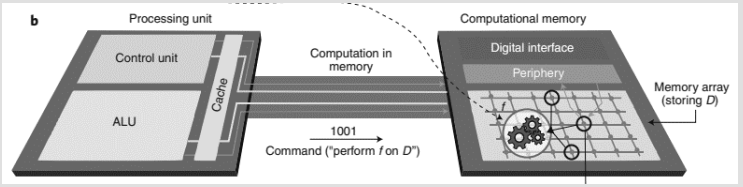
Domain-Specific Computing

E.g. Cluster, AI, bitcoin mining architectures or automotive digitalization shows the trend to adapt computing not via applying general-purpose computing to domain-specific problems, but by redesigning HW+SW+languages to the context at hand. What will definitions of security, safety or reliability mean in such architectures..

In-Memory Computing:

In-memory computing can be used to build security support, such as RNG, (PUF), trust roots directly in memory

However, flexible programmability also raises concerns about hardware attacks – how do we "program" e.g. isolation domains in in-memory computing arrays?



https://www.nature.com/articles/s41565-020-0655-z

Some interesting (?) references

Haeyoung Kim & al: **RIMI: Instruction-level Memory Isolation for Embedded Systems on RISC-V**
https://dl.acm.org/doi/pdf/10.1145/3400302.3415727

Menon A. & al: **Shakti-T: A RISC-V Processor with Light Weight Security Extensions**
https://dl.acm.org/doi/pdf/10.1145/3092627.3092629

De A. & al: FIXER: **Flow Integrity Extensions for Embedded RISC-V**
http://www.cse.psu.edu/~trj1/papers/date19.pdf

Liu L & al: **DRMaSV: Enhanced capability against hardware Trojans in coarse grained reconfigurable architectures (CGRA)**
https://ieeexplore.ieee.org/abstract/document/7984816/

Shilling R & al: **Pointing in the Right Direction – Securing Memory Accesses in a Faulty World**
https://arxiv.org/pdf/1809.08811.pdf

Li J & al: **Zipper Stack: Shadow Stacks Without Shadow?**
https://arxiv.org/pdf/1902.00888.pdf

Delshadtehrani L & al: PHMon: **A Programmable Hardware Monitor and Its Security Use Cases**
https://www.usenix.org/system/files/sec20spring_delshadtehrani_prepub.pdf

Thank you!

Questions ?