40 Years Since Dusk:

Will Hardware Capabilities Finally Make Our Systems More Capable?

Lluís Vilanova
(Technion)
vilanova@technion.ac.il

Why Hardware Capabilities Are a Good Idea for Security *and* Performance

Lluís Vilanova
(Technion)
vilanova@technion.ac.il

Outline

- Brief history of memory virtualization and protection
  - Memory paging
  - Memory segmentation
- Classic hardware capability systems
- Recent hardware capability systems
- Some ideas for the future

Origins of Memory Virtualization and Protection

- ***Virtualization*** → ***programmability & efficiency***
  - Automate swapping between memory & disk
    - Growing program sizes
    - Complex to do manually
  - Share code and data across processes
    - Fight memory scarcity

- ***Protection*** → ***security***
  - Multiprogramming
    - Run other programs during I/O wait times

Why Do We Want Capabilities?

- Coined by Dennis and Van Horn *[CACM'66]*
  - A ***communicable*** and ***unforgeable*** token that at the same time ***authorizes*** and ***identifies*** the destination of an operation
- Think of «fat pointers» with permission bits

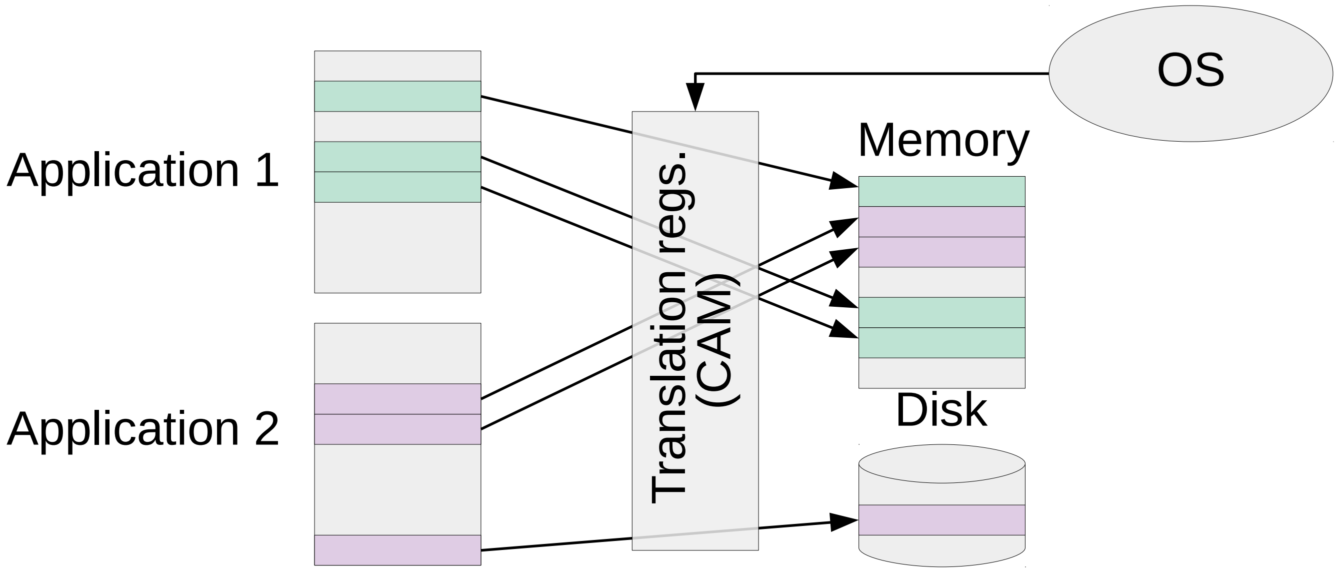
Base address | Size | {Read,Write,Call,...}

  - Fine-grained memory protection
  - Fine-grained, user-defined isolation domains
    - Safely managed by user: «diminish» rights
      - E.g., forbid writes, or make range smaller
    - Enforced by HW: tamper-proof structures

Virtual Memory Paging: The Atlas Computer

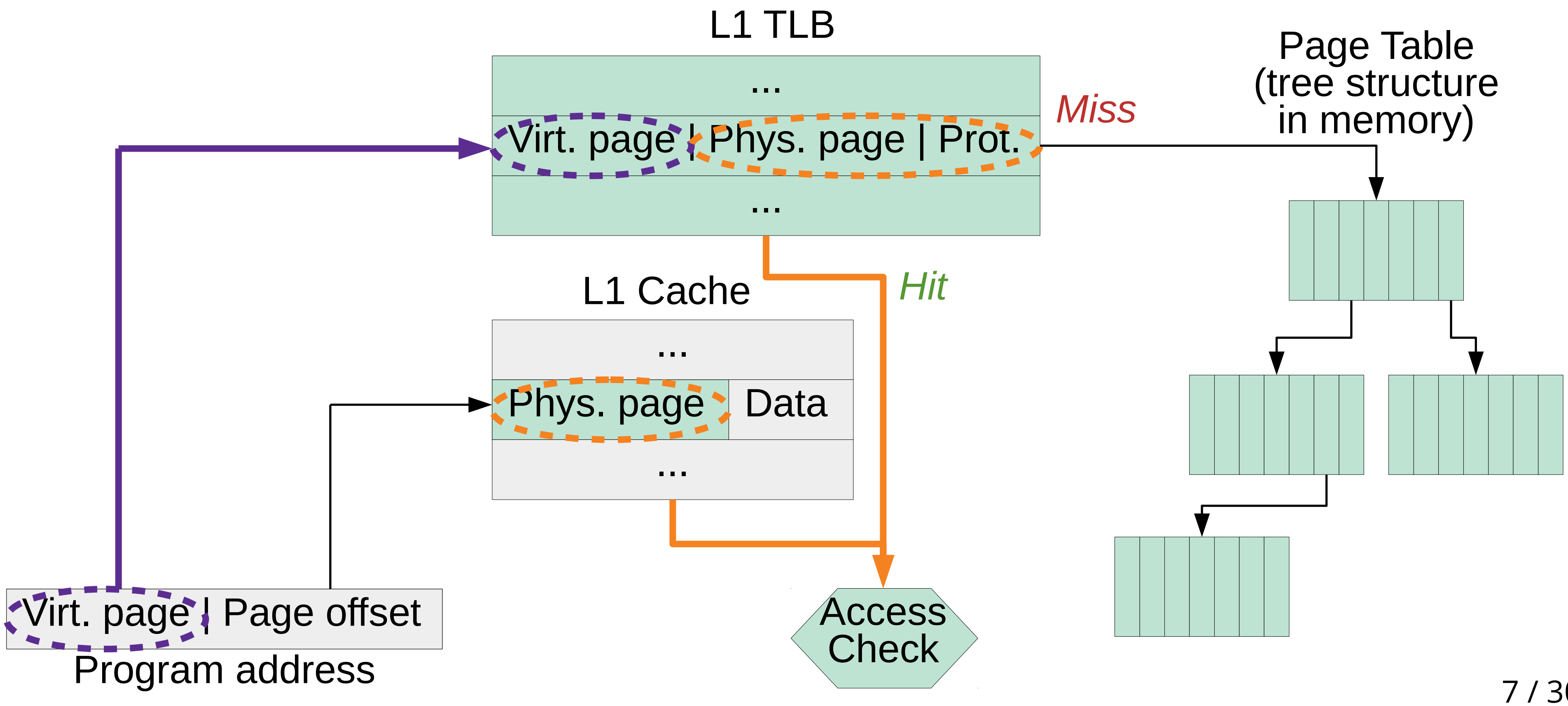
Source: University of Manchester

- University of Manchester (1956-1962)
- One-level storage system (demand paging)
  - Transparently move fix-sized data blocks (pages) between memory and disk on-demand



Virtual Memory Paging: Modern Implementation

- Most modern processors use paging
- Typically: *virtually-indexed*, *physically-tagged* L1$



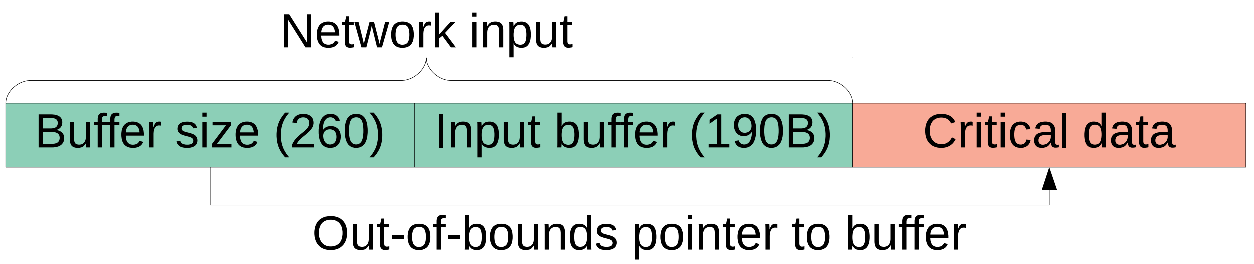
Virtual Memory Paging: Properties

- **The good**
  - Flat, private address space
  - Transparent paging & sharing
    - Memory oversubscription
- **The bad**
  - Fixed (page) granularity
  - TLB misses in big data apps
    - Up to 50% of the cycles

      [Basu et al. «Efficient Virtual Memory for Big Memory Servers», ISCA'13]
- **The ugly**
  - Complex (but well understood) HW design
  - Inter-process communication (IPC) is very costly and complex
    - Each process has its own page table
    - Communication goes through the OS
      - TLB and page tables managed by the OS
    - Data copies or explicit page sharing (cannot just pass a pointer)

Recent Security Problems: Protection Granularity

- **Heartbleed** *[went public in April 2014]*
  - Missing *buffer bounds check* in OpenSSL
    - Pages have fixed granularity

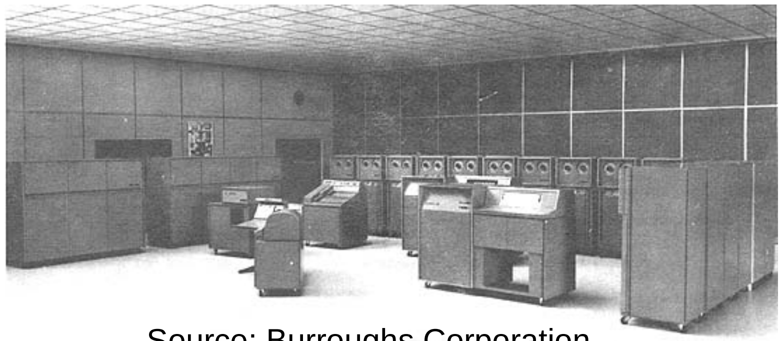


  - ***Servers:*** remote theft of private keys
    - ~17% of internet servers (~0.5 million)
  - ***Clients:*** remote theft of session cookies & passwords
  - Problem undetected for 2 years

Recent Security Problems: Function/Protection Separation

- **Meltdown** *[went public in January 2018]*
  - Allows reading **arbitrary** memory
    1) Force branch to mis-predict
    2) An ***invalid*** speculative read loads data into the cache before it is squashed
    3) Do a ***valid*** memory read that depends on previous value (different lines present depending on previous value)
    4) Time cache access to discover value
  - **Any CPU** with speculative execution is potentially vulnerable
    - Intel, IBM and ARM chips are affected

Virtual Memory Segmentation: The B5000 *[1/2]*


Source: Burroughs Corporation

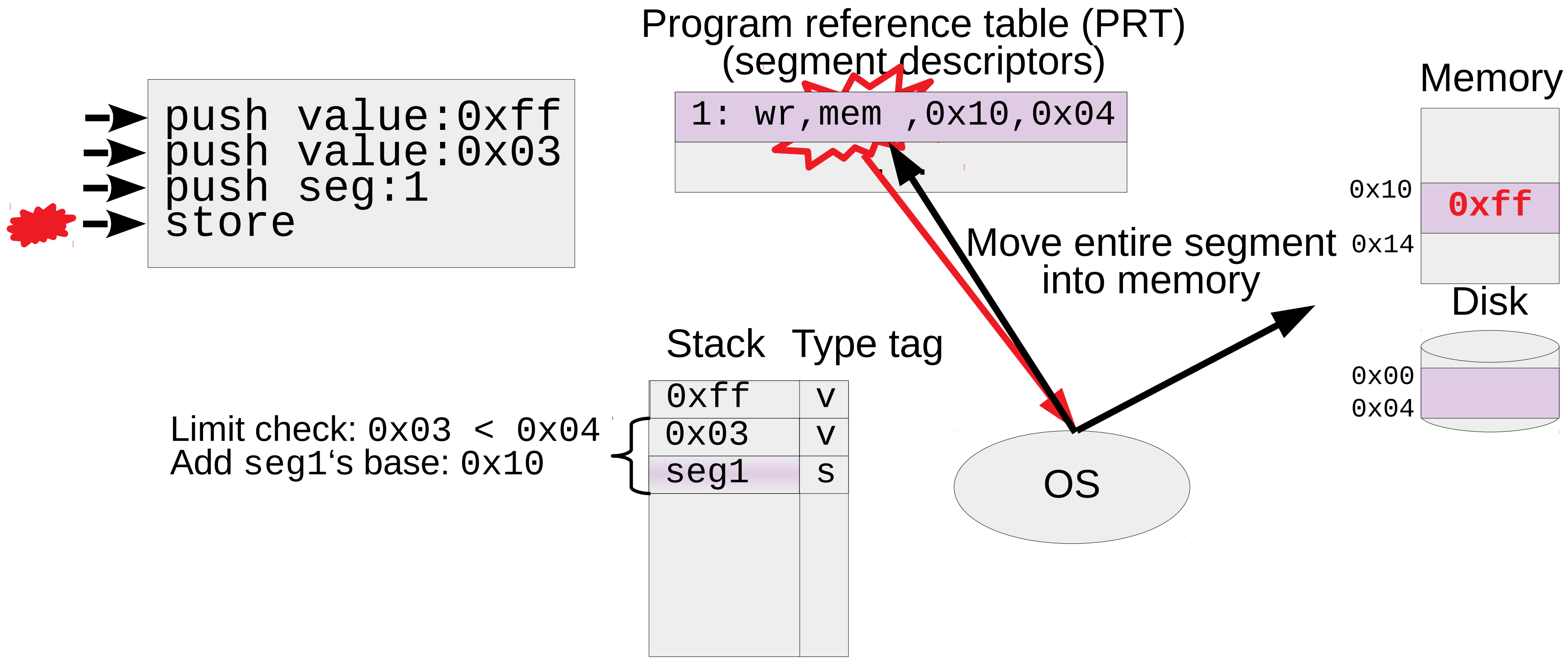
- Burroughs Corporation (1961)
  - First commercial system with virtual memory
- One-level storage system (*same objective*)
  - On-demand memory / disk transfer
- *Segment:* contiguous region of memory representing a logical entity (e.g., routine or array)
  - We’re getting close to capabilities

Physical address | Size | {Read,Write,Execute}

Virtual Memory Segmentation: The B5000 *[2/2]*

Source: Burroughs Corporation

Stack machine:
Store value `0xff` into segment 1 (offset 3)



```
push value:0xff
push value:0x03
push seg:1
store
```

Virtual Memory Segmentation: Segment Descriptor Protection

B500 (‘61) $\longrightarrow$

- Global table
- Descriptors can be stored in (tagged) stack
- Code segments: can only be entered at known points, and are not writable

Rice Univ. (‘59) $\longrightarrow$

- Global *root* table
- Can build a «tree» of descriptors (e.g., matrix)
- Full traversal on every operation
- Descriptors passed as register arguments

BLM (‘68)

- Descriptors can be stored in arbitrary (tagged) memory
- Needs garbage collection
  - Segment freed only when nobody holds a reference

Virtual Memory Segmentation: Properties

- **The good**
  - Conceptually simple
    - Range check
  - Arbitrary granularity
  - Sharing of logical entities (e.g., array, procedure)
  - Can grow/shrink segments
    - Addressed with offsets
    - Indirected to descriptors
    - Relocated by the OS
- **The bad**
  - External memory fragmentation
    - OS does compaction
  - Segment descriptor indirection chains
  - Memory tagging
- **The ugly**
  - Relocation must locate affected descriptors
  - Creating new segment descriptors needs OS intervention

Capability Addressing Architectures

- Implemented by hardware (like segments)
  - «Fat pointer» with permission bits
- Can be ***safely*** manipulated by ***user*** software iff we never «upgrade» a capability
  - Copy
  - Shrink range
  - Remove permissions

Capability Addressing: Historical Perspective

- **Chicago Magic Number Machine *('67–cancelled)***
  - **(1)** Split data/capability registers and segments
  - **(2)** Uniform naming for user objects and system operations
- **Plessey System 250 *('70)***
  - (1) (2)
  - **(3)** 2-way protected procedures; can represent object methods
  - **(4)** No privileged software
  - Capabilities point to central table
    - Simpler segment relocation
    - Table entries are garbage-collected
- **Cambridge CAP Computer *('76)***
  - (1) (2) (3) (4)
  - All capabilities (indirectly) point to capabilities in parent process
    - User-managed capabilities
    - Traversed on every access
    - Cached in «*capability unit*»
- **IBM System/38 *('79)***
  - (2) (3)
  - Capabilities instead of pointers
    - Memory tagging
  - 40-bit segment space
    - Never reused (no need for garbage collection)
  - Capabilities on top of paging

Review of Recent Security Vulnerabilities

- **Heartbleed**
  - «*Buffer bounds check* in OpenSSL»
  - Capabilities protect arbitrary buffer bounds
- **Meltdown**
  - «Read ***arbitrary*** memory by exploiting *speculative memory accesses* and *cache access timing* together»
  - A simple address range check can be executed before the memory access

Capability Addressing: Properties

- **The good**
  - Arbitrary granularity
  - (Safely) User-managed
  - Uniform protection mechanism
    - Protect from array access to method invocation
- **The bad**
  - Pervasive memory tagging
    - Extra DRAM bandwidth consumption
  - Compatibility with existing languages
- **The ugly**
  - Capability revocation (for object deletion) is costly
    - Indirection → overheads on every access
    - Virtual address non-reuse → internal memory fragmentation
    - Garbage collection → overheads, not part of all languages

Modern Proposals for Capability Addressing

Language Compatiblity: *[1/2]* Capabilities-as-Pointers in C

[Chisnall et al. «*Beyond the PDP-11: Architectural support for a memory-safe C abstract machine*» ASPLOS'15]

- ***Objective:*** use capabilities in all C pointers
- Add an «`offset`» field:

```
Base | Size | Offset | Permissions
```

  - Memory access check: `0 ≤ offset < size`
  - Pointer arithmetic modifies «`offset`»
  - Allows common idioms in low-level C code
- Operations:
  - Get/Set/Add/Sub offset
  - Compare capabilities
  - Capability to pointer
  - Pointer to capability (from default capability)

Language Compatiblity: *[2/2]* Capabilities-as-Pointers in C

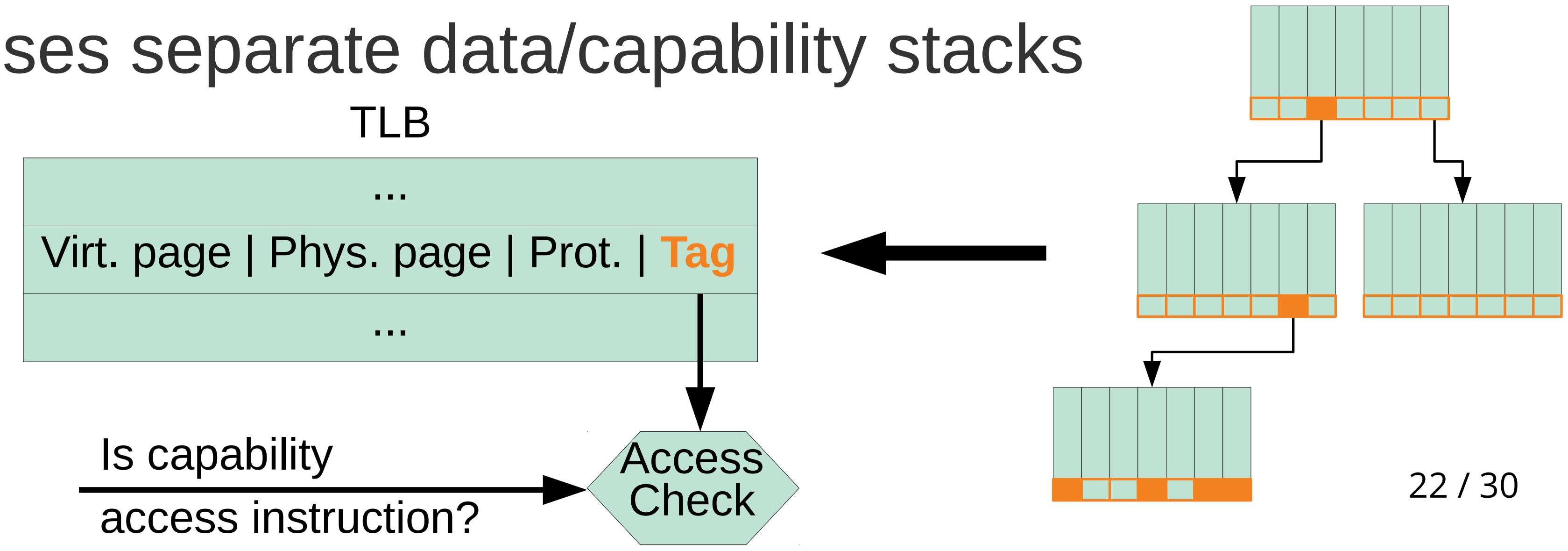
[Chisnall et al. «*Beyond the PDP-11: Architectural support for a memory-safe C abstract machine*» ASPLOS'15]

- Use existing/new attributes to identify permissions
  - `__capability int*`: Read-write
  - `__capability const int*`: Read-only
  - `__capability int (*)(int)`: Call-only
  - `intptr_t ptr = cap`: Arithmetic on `offset`
- **Need `__capability` only in library interfaces that cross capability and non-capability worlds**
  - Otherwise compiler can use capabilities instead of pointers (i.e., malloc returns a capability)
- Negligible performance overheads in most cases

Efficient Memory Tagging: *[1/2]* Eliminate Word Tagging

[Vilanova et al. «*CODOMs: Protecting Software with Code-centric Memory Domains*» ISCA'14]

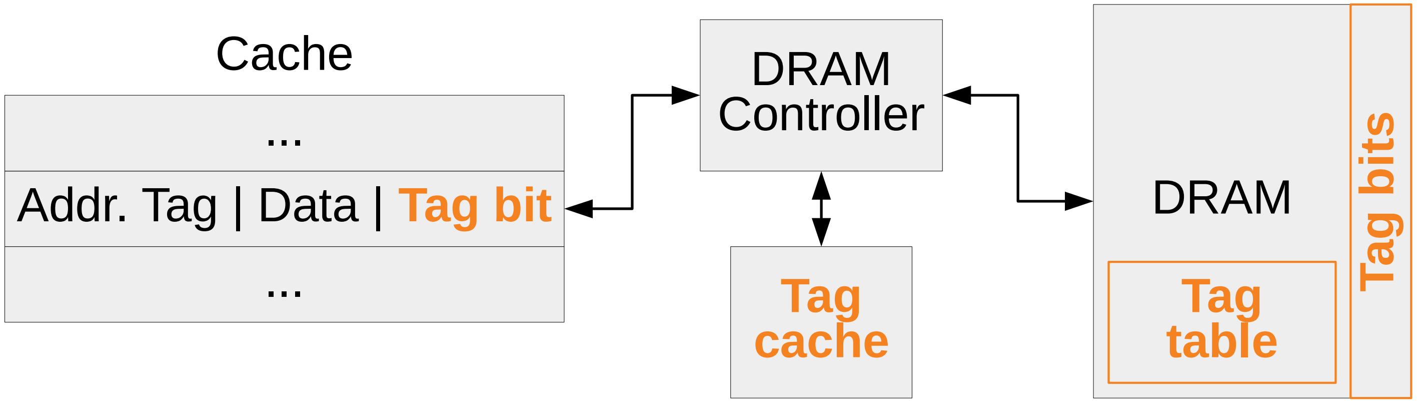
- ***Objective:*** eliminate DRAM traffic for tags
- Repurpose one bit in the page table
  - Very efficient checks and storage, no traffic
- Both page types can be mixed
  - Structures with capabilities need to be split
  - Uses separate data/capability stacks



Efficient Memory Tagging: *[2/2]* Optimize Word Tagging

[Joannou et al. «*Efficient Tagged Memory*» ICCD’17]

- ***Objective:*** decrease DRAM traffic for tags



- Tag cache has very good spatial locality
  - < 8% DRAM traffic increase
- Traffic «compression» in 2-level tag table
  - <1% – 4% DRAM traffic increase
- Ellide same-value writes: 2x–20x write reduction

Capability Revocation: Scope-Based Revocation *[1/2]*

[Vilanova et al. «*CODOMs: Protecting Software with Code-centric Memory Domains*» ISCA'14]

- ***Objective:*** minimize need for revocation
- Arguments often ignored by callee after return:

```
__capability int *array
int index = 0x1
func(array, index)
```

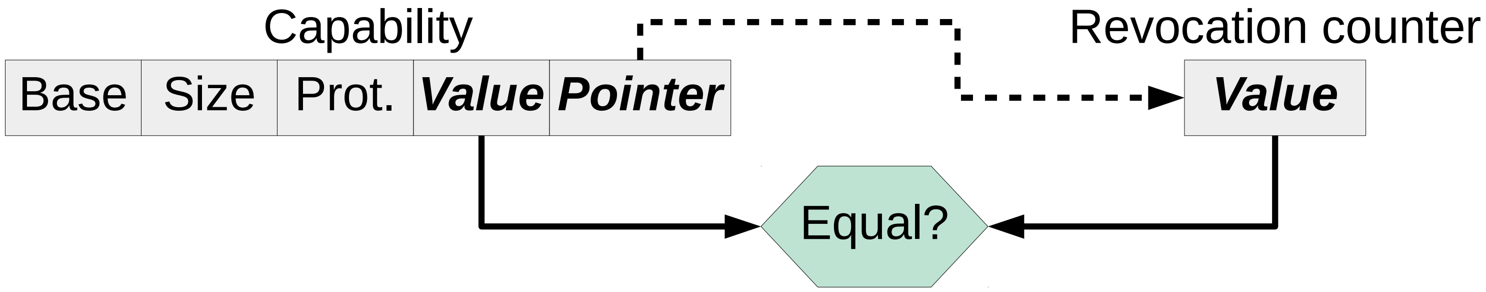
```
int secret = ...
func(array, index):
  return array[index*secret]
```

  - True for >95% of memory references in Linux kernel modules when mutually isolated
- «*Synchronous*» (scope-revocable) vs. «*Asynchronous*» (arbitrary-revocable) capabilities
  - Synchronous caps. only in registers or cap. stack
  - Capability stack frame inaccessible after return

Capability Revocation: Efficient Revocation Control [2/2]

[Vilanova et al. «*CODOMs: Protecting Software with Code-centric Memory Domains*» ISCA'14]

- ***Objective:*** make revocation efficient
  - ***Reuse*** addresses → avoid internal fragmentation
  - ***Avoid*** garbage collection → performance overheads & not part of all languages
- Add 46-bit «*revocation counter*» (reusable 2^{46} times)

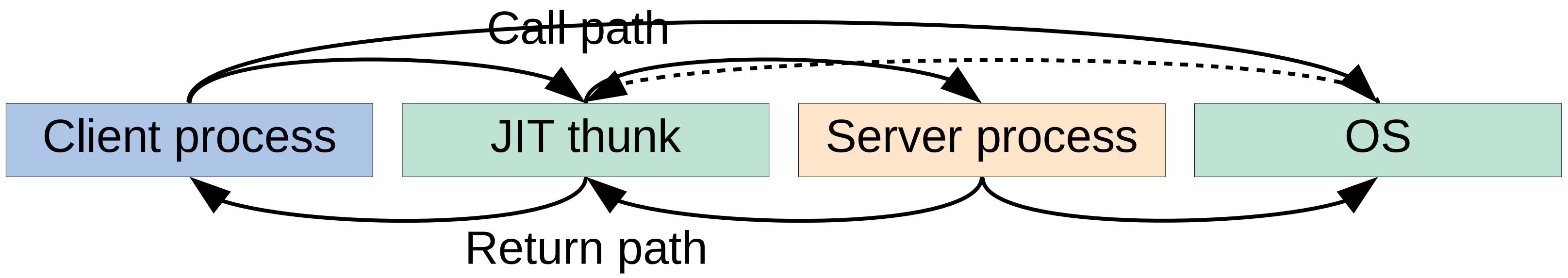


  - Checked when a capability is loaded into a register
  - Revocation: increment the counter and propagate to capability registers (immediate invalidation)

Inter-Process Communication Without OS Intervention

[Vilanova et al. «Direct Inter-Process Communication (dIPC): *Repurposing the CODOMs Architecture to Accelerate IPC*» EuroSys'17]

- ***Objective:*** protected procedure calls ***across*** existing Linux processes without involving the OS
  - Processes in a shared page table, but isolated



1) Exchange rights and policies through OS
2) OS generates specialized code from policies
3) Processes use the «*JIT thunk*» to communicate

- Full-stack web server
  - Up to 5.12x speedup

Further Proposals

Pico-Para-Virtualization

- ***Problem:*** many VMs rely on *trap-and-emulate*
  - A memory access or instruction traps into VMM
  - *Examples:* I/O devices, specific HW registers
  - Allows migration between heterogeneous hosts
- ***Solution:*** *Access low-level HW with protected procedure calls (i.e., through a capability)*
  - Close to original HW, but has opaque implementation
  - *Example:* memory mapped control register
    - Native: simply write into the register
    - VM: handle device emulation

Capabilities for Memory Translation

- ***Problem:*** Lots of TLB misses in big data apps
  - Up to 50% of cycles spent servicing misses

    [Basu et al. «Efficient Virtual Memory for Big Memory Servers», ISCA'13]
- ***Solution:*** Use base address in capabilities to directly index physical memory (bypass TLB)
  - A single bit in a capability identifies TLB bypass
  - Can use revocation counters in CODOMs to support paging of entire segments

Conclusions:
Security *and* Performance

- SW and HW design are now much more mature
  - Early commercial attemps at capabilities were often too complex for their time
- Current security interests bring a renewed push
  - Thwarts attacks like *heartbleed* and *meltdown*
  - Optimized for compatibility
  - Improved performance
    - Solve or improve many of the open problems
    - Can provide full-stack application speedups

Lluís Vilanova
vilanova@technion.ac.il