Capability Machines

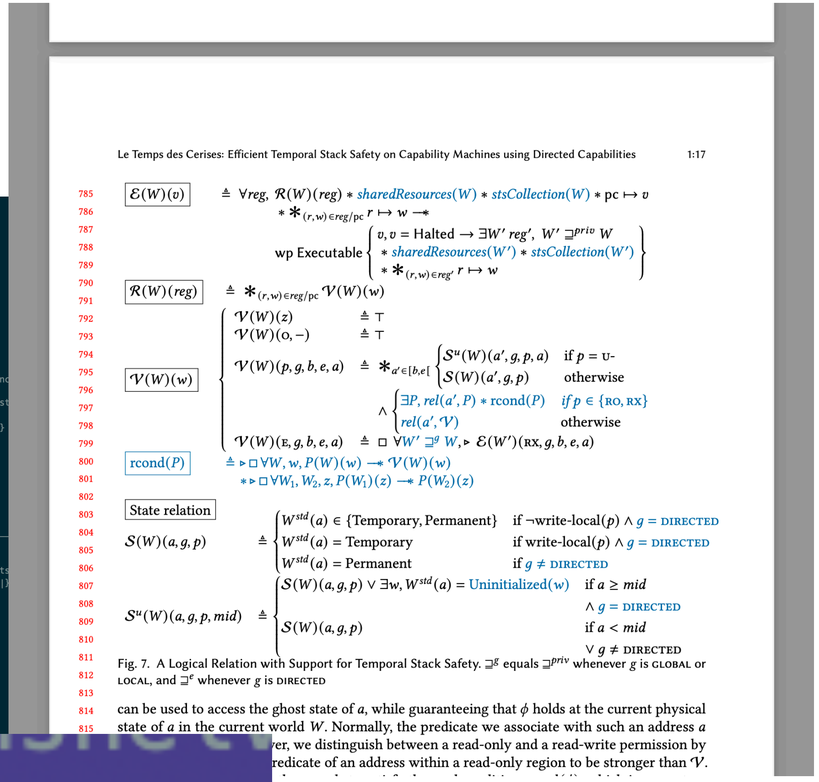
The future of secure processors

June Rousseau - February 2025
(thanks to Aina Linn Georges for the slides)

Who am I?

PhD student in the Logic and Semantics group

Mechanised Reasoning about Capability Machines



Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities 1:17

$$\mathcal{E}(W)(v) \triangleq \forall reg,\ \mathcal{R}(W)(reg) * sharedResources(W) * stsCollection(W) * \mathrm{pc} \mapsto v * \text{✻}_{(r,w)\in reg/\mathrm{pc}}\, r \mapsto w \multimap \text{wp Executable} \left\{ \begin{array}{l} v, v = \mathrm{Halted} \to \exists W'\, reg',\ W' \sqsupseteq^{priv} W \\ * sharedResources(W') * stsCollection(W') \\ * \text{✻}_{(r,w)\in reg'}\, r \mapsto w \end{array} \right\}$$

$$\mathcal{R}(W)(reg) \triangleq \text{✻}_{(r,w)\in reg/\mathrm{pc}}\, \mathcal{V}(W)(w)$$

$$\mathcal{V}(W)(w) \left\{ \begin{array}{ll} \mathcal{V}(W)(z) & \triangleq \top \\ \mathcal{V}(W)(\mathrm{o}, -) & \triangleq \top \\ \mathcal{V}(W)(p,g,b,e,a) & \triangleq \text{✻}_{a'\in[b,e[} \left\{ \begin{array}{ll} \mathcal{S}^u(W)(a',g,p,a) & \text{if } p = \mathrm{U}\text{-} \\ \mathcal{S}(W)(a',g,p) & \text{otherwise} \end{array} \right. \\ & \quad \wedge \left\{ \begin{array}{ll} \exists P, rel(a',P) * \mathrm{rcond}(P) & if\ p \in \{\mathrm{RO}, \mathrm{RX}\} \\ rel(a',\mathcal{V}) & \text{otherwise} \end{array} \right. \\ \mathcal{V}(W)(\mathrm{E},g,b,e,a) & \triangleq \square\, \forall W' \sqsupseteq^{g} W, \triangleright\, \mathcal{E}(W')(\mathrm{RX},g,b,e,a) \end{array} \right.$$

$$\mathrm{rcond}(P) \triangleq \triangleright \square \forall W, w, P(W)(w) \multimap \mathcal{V}(W)(w) * \triangleright \square \forall W_1, W_2, z, P(W_1)(z) \multimap P(W_2)(z)$$

State relation

$$\mathcal{S}(W)(a,g,p) \triangleq \left\{ \begin{array}{ll} W^{std}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \wedge g = \textsc{directed} \\ W^{std}(a) = \text{Temporary} & \text{if write-local}(p) \wedge g = \textsc{directed} \\ W^{std}(a) = \text{Permanent} & \text{if } g \neq \textsc{directed} \end{array} \right.$$

$$\mathcal{S}^u(W)(a,g,p,mid) \triangleq \left\{ \begin{array}{ll} \mathcal{S}(W)(a,g,p) \vee \exists w, W^{std}(a) = \mathrm{Uninitialized}(w) & \text{if } a \geq mid \\ \qquad \wedge\ g = \textsc{directed} & \\ \mathcal{S}(W)(a,g,p) & \text{if } a < mid \\ \qquad \vee\ g \neq \textsc{directed} & \end{array} \right.$$

Fig. 7. A Logical Relation with Support for Temporal Stack Safety. $\sqsupseteq^g$ equals $\sqsupseteq^{priv}$ whenever g is GLOBAL or LOCAL, and $\sqsupseteq^e$ whenever g is DIRECTED

can be used to access the ghost state of a, while guaranteeing that ϕ holds at the current physical state of a in the current world W. Normally, the predicate we associate with such an address a ver, we distinguish between a read-only and a read-write permission by redicate of an address within a read-only region to be stronger than $\mathcal{V}$.

Capability Machines

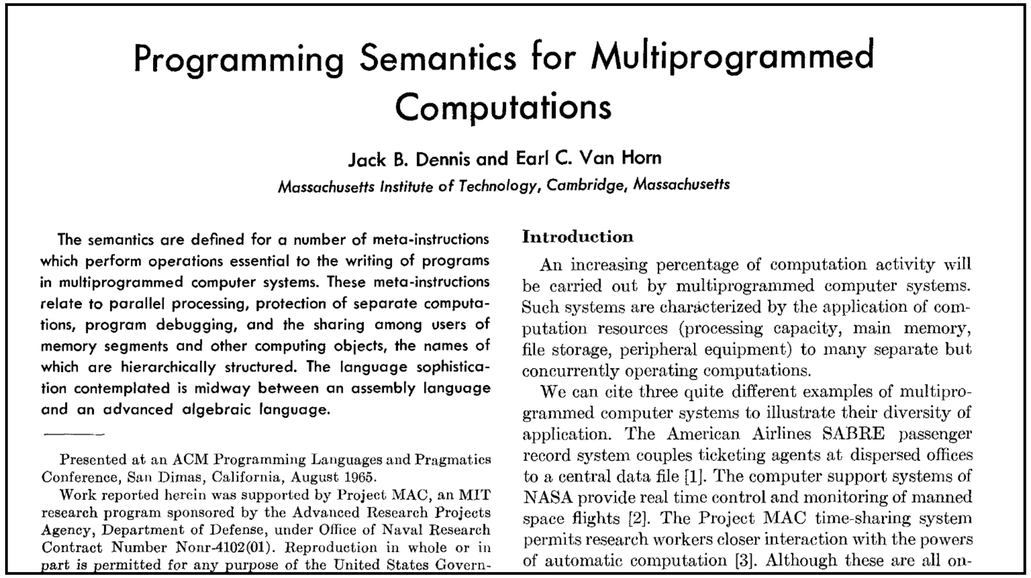
A capability at the lowest level of a machine:
Hardware capabilities

- Motivation and Design Principles
- Hardware Capability Operations
- Beyond Memory Safety

History

The concept of machine capabilities is old!

[Dennis & Van Horn - 1966]



Programming Semantics for Multiprogrammed Computations

Jack B. Dennis and Earl C. Van Horn
Massachusetts Institute of Technology, Cambridge, Massachusetts

The semantics are defined for a number of meta-instructions which perform operations essential to the writing of programs in multiprogrammed computer systems. These meta-instructions relate to parallel processing, protection of separate computations, program debugging, and the sharing among users of memory segments and other computing objects, the names of which are hierarchically structured. The language sophistication contemplated is midway between an assembly language and an advanced algebraic language.

Presented at an ACM Programming Languages and Pragmatics Conference, San Dimas, California, August 1965.

Work reported herein was supported by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Govern-

Introduction

An increasing percentage of computation activity will be carried out by multiprogrammed computer systems. Such systems are characterized by the application of computation resources (processing capacity, main memory, file storage, peripheral equipment) to many separate but concurrently operating computations.

We can cite three quite different examples of multiprogrammed computer systems to illustrate their diversity of application. The American Airlines SABRE passenger record system couples ticketing agents at dispersed offices to a central data file [1]. The computer support systems of NASA provide real time control and monitoring of manned space flights [2]. The Project MAC time-sharing system permits research workers closer interaction with the powers of automatic computation [3]. Although these are all on-

CHERI: **C**apability **H**ardware **E**nhanced **RI**SC **I**nstructions

UNIVERSITY OF CAMBRIDGE

[Woodruff et.al. - 2014]

[Watson et.al. - 2015]

The industry's growing interest in capability machines
or: "why this is an excellent time to be excited about capabilities"

Microsoft

2019

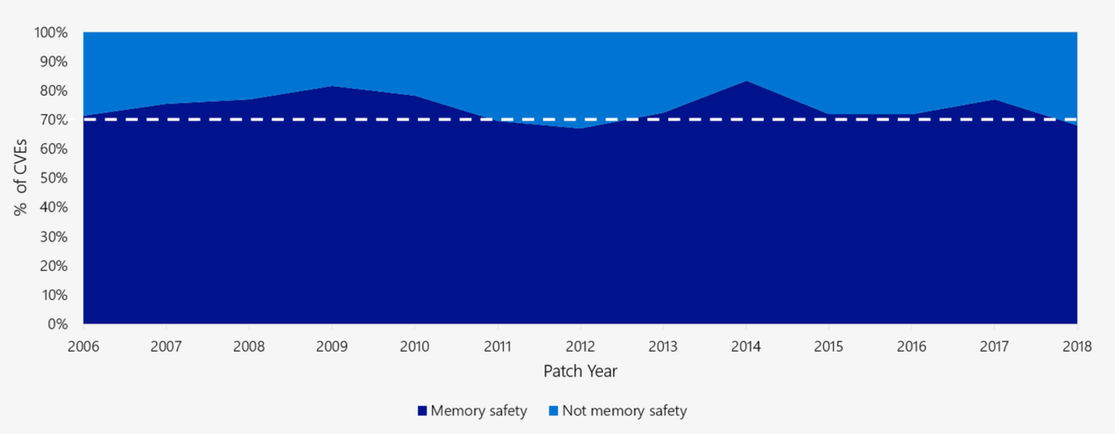


Figure 1: ~70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues

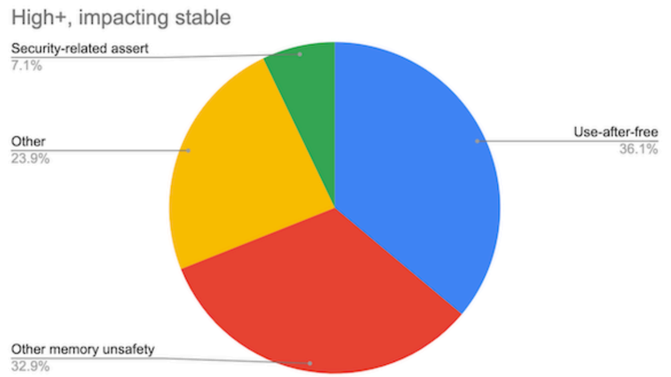
2020

We conservatively assessed the percentage of vulnerabilities reported to the Microsoft Security Response Center (MSRC) in 2019 and found that approximately 31% would no longer pose a risk to customers and therefore would not require addressing through a security update on a CHERI system based on the default configuration of the CheriBSD operating system. If we also assume that automatic initialization of stack variables (InitAll) and of heap allocations (e.g. pool zeroing) is present, the total number of vulnerabilities deterministically mitigated exceeds 43%. With additional features such as Cornucopia that help prevent temporal safety issues such as use after free, and assuming that it would cover 80% of all the UAFs, the number of deterministically mitigated vulnerabilities would be at least 67%. There is additional work that needs to be done to protect the stack and add fined grained CFI, but this combination means CHERI looks very promising in its early stages.

Google - Chromium

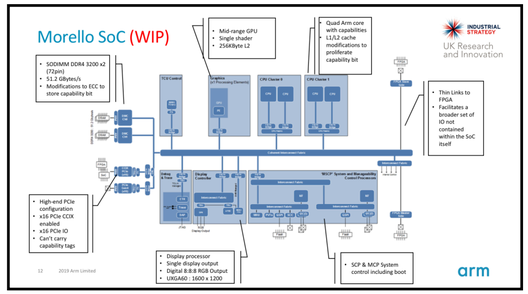
2020

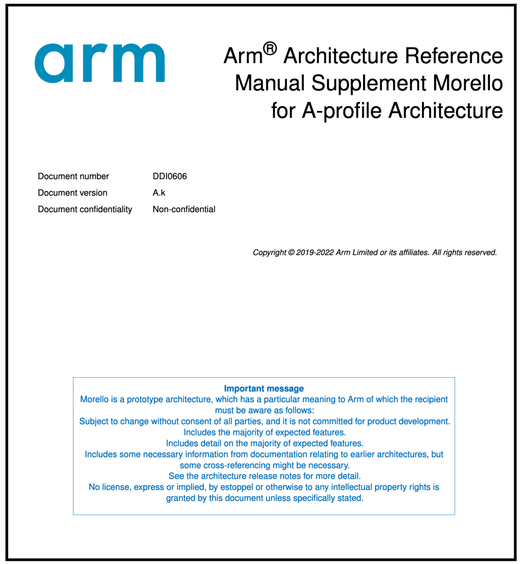
Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.



(Analysis based on 912 high or critical severity security bugs since 2015, affecting the Stable channel.)

arm : Morello




arm

Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture

Document number	DDI0606
Document version	A.k
Document confidentiality	Non-confidential

Important message
Morello is a prototype architecture, which has a particular meaning to Arm of which the recipient must be aware as follows:
Subject to change without consent of all parties, and it is not committed for product development.
Includes the majority of expected features.
Includes detail on the majority of expected features.
Includes some necessary information from documentation relating to earlier architectures, but some cross-referencing might be necessary.
See the architecture release notes for more detail.
No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

September 2020

Verified security for the Morello capability-enhanced prototype Arm architecture

THOMAS BAUEREISS, University of Cambridge, UK
BRIAN CAMPBELL, University of Edinburgh, UK
THOMAS SEWELL, University of Cambridge, UK
ALASDAIR ARMSTRONG, University of Cambridge, UK
LAWRENCE ESSWOOD, University of Cambridge, UK
IAN STARK, University of Edinburgh, UK
GRAEME BARNES, Arm Ltd., UK
ROBERT N. M. WATSON, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

Memory safety bugs continue to be a major source of security vulnerabilities in our critical infrastructure. The CHERI project has proposed extending conventional architectures with hardware-supported *capabilities* to enable fine-grained memory protection and scalable compartmentalisation, allowing historically memory-unsafe C and C++ to be adapted to deterministically mitigate large classes of vulnerabilities, while requiring only minor changes to existing system software sources. Arm is currently designing and building Morello, a CHERI-enabled prototype architecture, processor, SoC, and board, extending the high-performance Neoverse N1, to enable industrial evaluation of CHERI and pave the way for potential mass-market adoption. However, for such a major new security-oriented architecture feature, it is important to establish high confidence that it does provide the protections it intends to, and that cannot be done with conventional engineering techniques.

In this paper we put the Morello architecture on a solid mathematical footing from the outset. We define the fundamental security property that Morello aims to provide, reachable capability monotonicity, and prove that the architecture definition satisfies it. This proof is mechanised in Isabelle/HOL, and applies to a translation of the official Arm Morello specification into Isabelle. The main challenge is handling the complexity and scale of a production architecture: 62,000 lines of specification, translated to 210,000 lines of Isabelle. We do so by factoring the proof via a narrow abstraction capturing the essential properties of instruction execution in an arbitrary CHERI ISA, expressed above a monadic intra-instruction semantics. We also develop a model-based test generator, which generates instruction-sequence tests that give good specification coverage, used in early testing of the Morello implementation and in Morello QEMU development. We also use Arm's internal test suite to validate our internal model.

This gives us machine-checked mathematical proofs of whole-ISA security properties of a full-scale industry architecture, at design-time. To the best of our knowledge, this is the first demonstration that that is feasible, and it significantly increases confidence in Morello.

2022

“Memory Safety”

Design Principles behind CHERI

Spacial memory safety:

- Absence of certain undefined behaviours via memory accesses

```
int arr[4] = {0,1,2,3};
int *p = arr + 5;
p = 0;
int a = *p;
```

Design Principles behind CHERI

Spacial memory safety:

- Absence of certain undefined behaviours via memory accesses

Design Principles behind CHERI

Spacial memory safety:

- Absence of certain undefined behaviours via memory accesses

Design Principles behind CHERI

Spacial memory safety:

- Absence of certain undefined behaviours via memory accesses

undefined behaviour for indexing out of bounds

```
int arr[4] = {0,1,2,3};
int *p = arr + 5;
p = 0;
int a = *p;
```

undefined behaviour for dereferencing a null pointer

undefined behaviour of dereferencing an arbitrary cast

```
long f(long i){
    int* p = (int *)i;
    *p = 4;
    return ((long) p);
}
```

Design Principles behind CHERI

Spacial memory safety: **pointers as capabilities principle**

- Absence of the kinds undefined behaviours via memory accesses that pose a risk, i.e. an exploitable memory vulnerability
    - Dereferencing out of bounds
    - Dereferencing arbitrary pointer casts

Design Principles behind CHERI

Spacial memory safety: pointers as capabilities principle

- Absence of the kinds undefined behaviours via memory accesses that pose a risk, i.e. an exploitable memory vulnerability
    - Dereferencing out of bounds
    - Dereferencing arbitrary pointer casts

What about temporal memory safety?

- use-after-free

  interesting topic, but outside the scope of this lecture! (see Cornucopia [Filardo et.al. 2020])

Hardware Capabilities

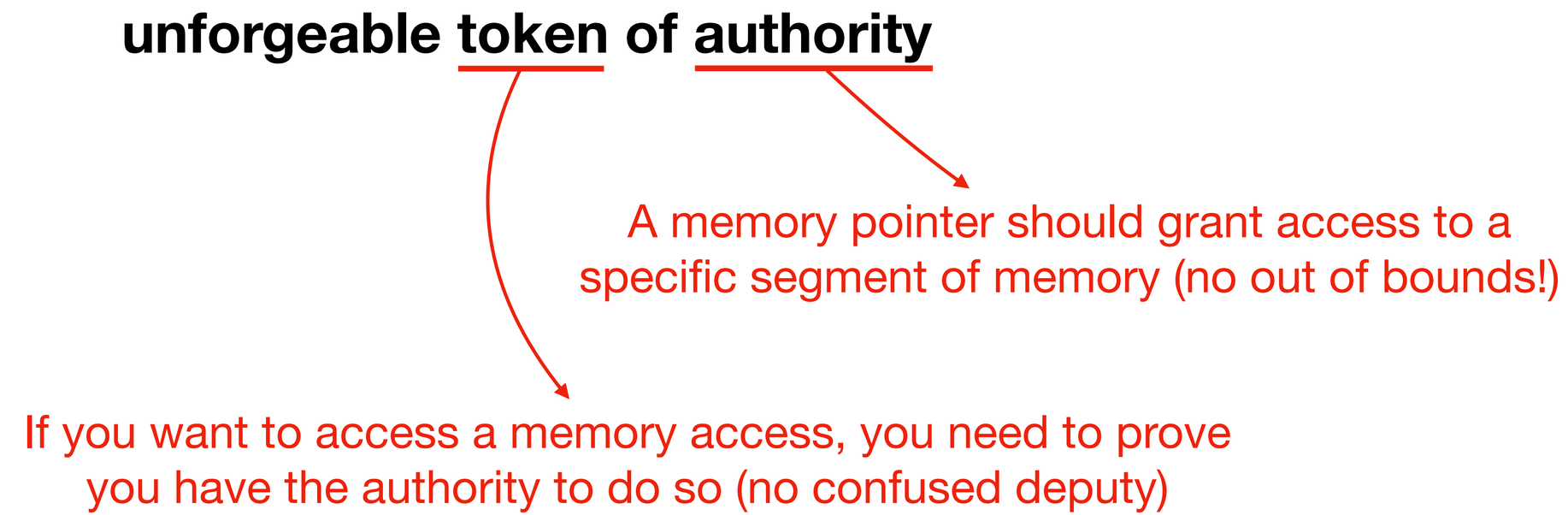
unforgeable token of authority

Hardware Capabilities

unforgeable token of authority

A memory pointer should grant access to a specific segment of memory (no out of bounds!)

Hardware Capabilities



Hardware Capabilities

A memory pointer should not appear out of thin air (no casting!)

unforgeable token of authority

A memory pointer should grant access to a specific segment of memory (no out of bounds!)

If you want to access a memory access, you need to prove you have the authority to do so (no confused deputy)

Hardware Capabilities

unforgeable token of authority

Hardware Capabilities

unforgeable token of authority

Authority over memory cannot be artificially increased or created

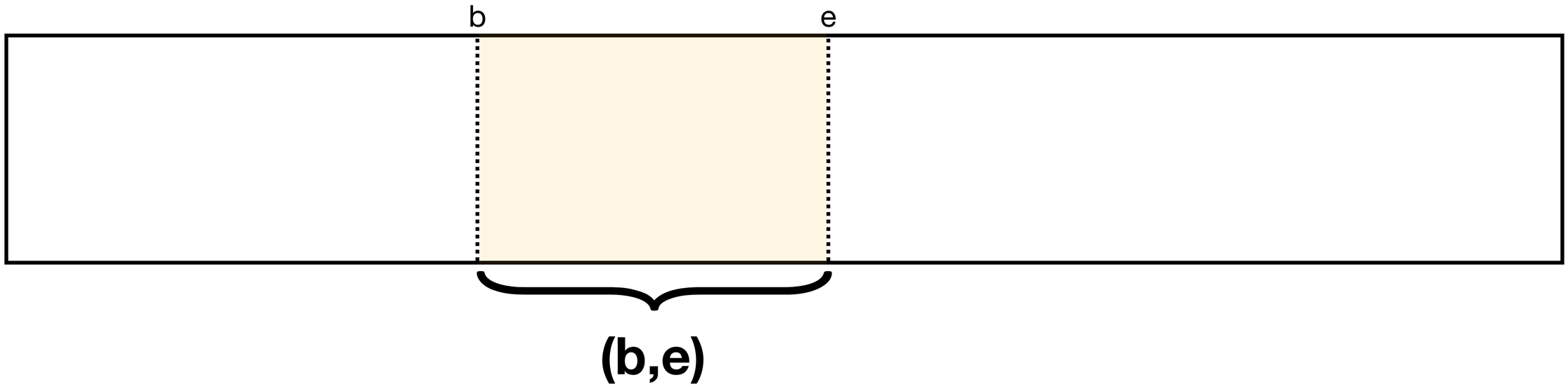
Hardware Capabilities

Pointers as Capabilities

Pointers are replaced by hardware capabilities

Pointers as Capabilities

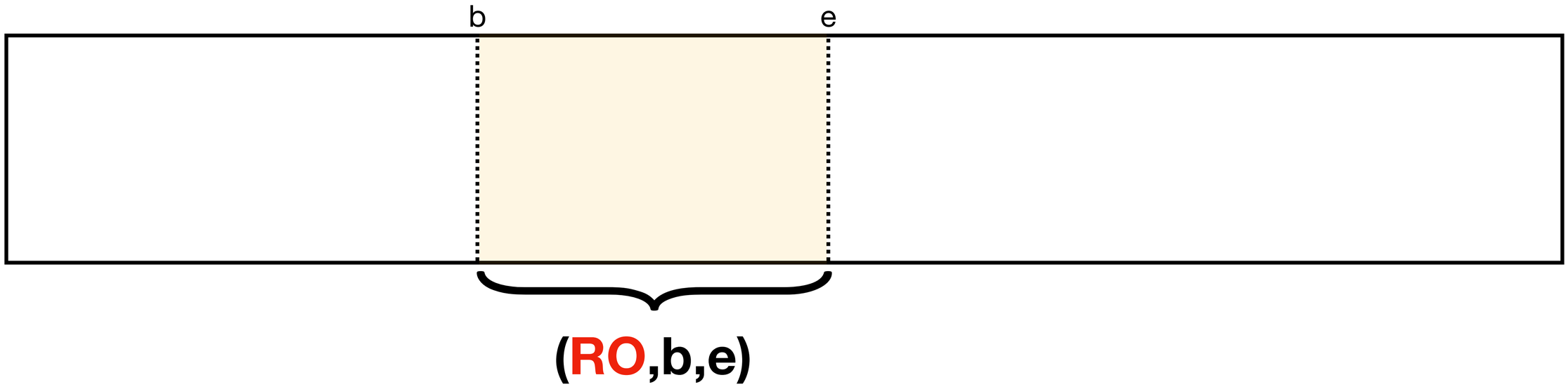
Pointers are replaced by hardware capabilities



- Bounds of authority

Pointers as Capabilities

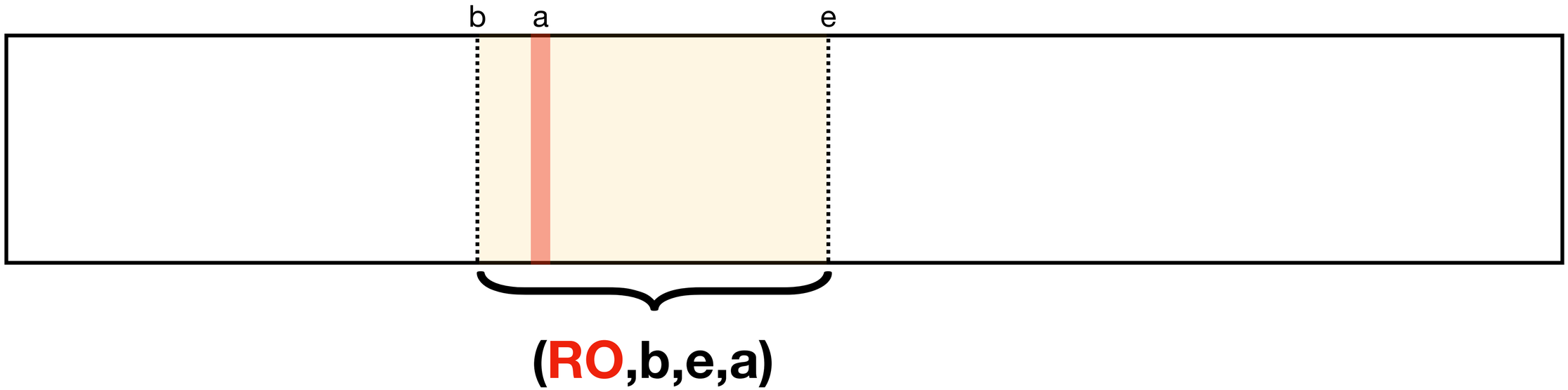
Pointers are replaced by hardware capabilities



- Bounds of authority
- Permission: RO/RW/etc

Pointers as Capabilities

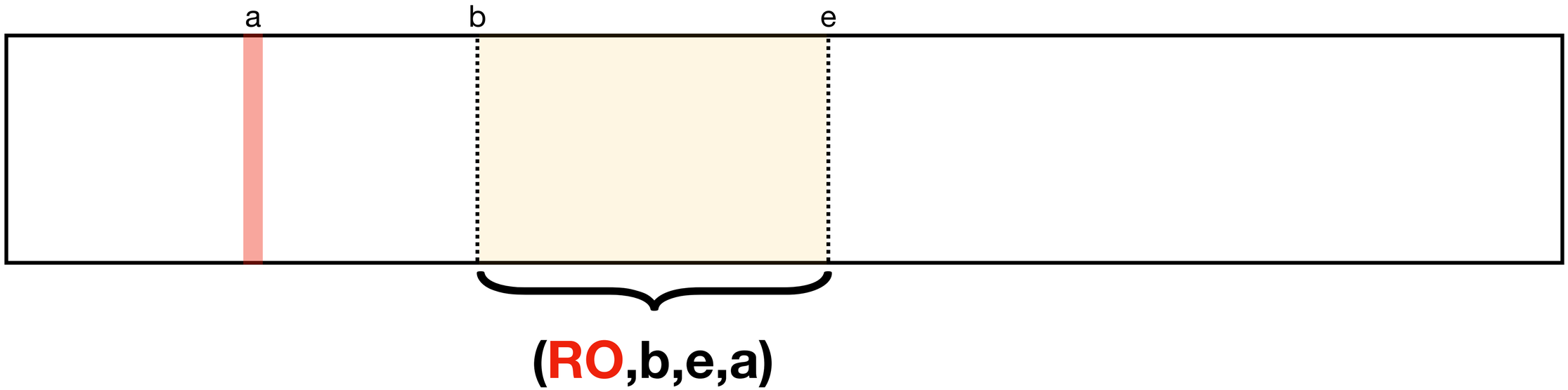
Pointers are replaced by hardware capabilities



- Bounds of authority
- Permission: RO/RW/etc
- Address

Pointers as Capabilities

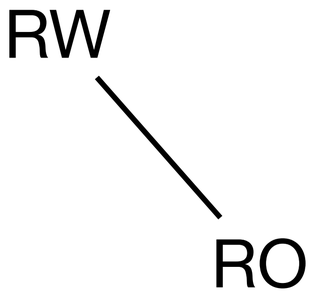
Pointers are replaced by hardware capabilities



- Bounds of authority
- Permission: RO/RW/etc
- Address

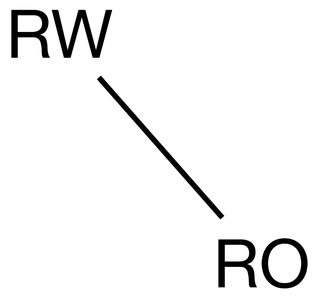
A Lattice of Permissions

A Lattice of Permissions



- RO : read-only
- RW : read-write

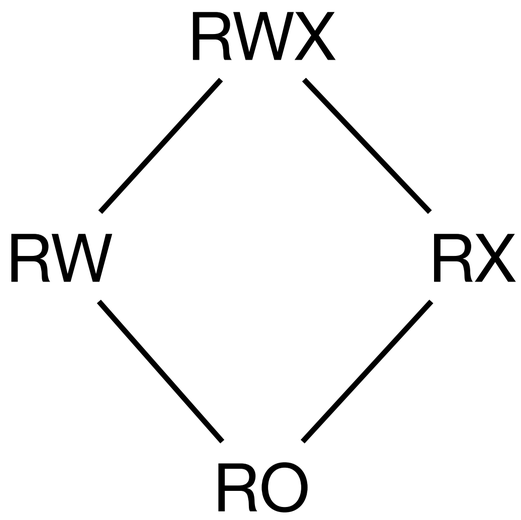
A Lattice of Permissions



- RO : read-only
- RW : read-write

What about the program counter?

A Lattice of Permissions

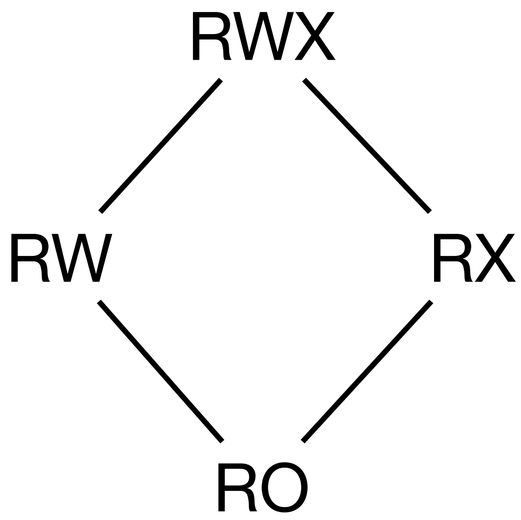


- RO : read-only
- RW : read-write

What about the program counter?

- RX : read execute
- RWX : read-write execute

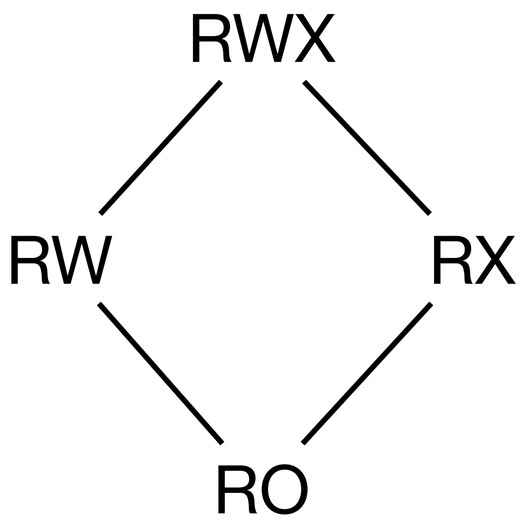
A Lattice of Permissions



What about function pointers?

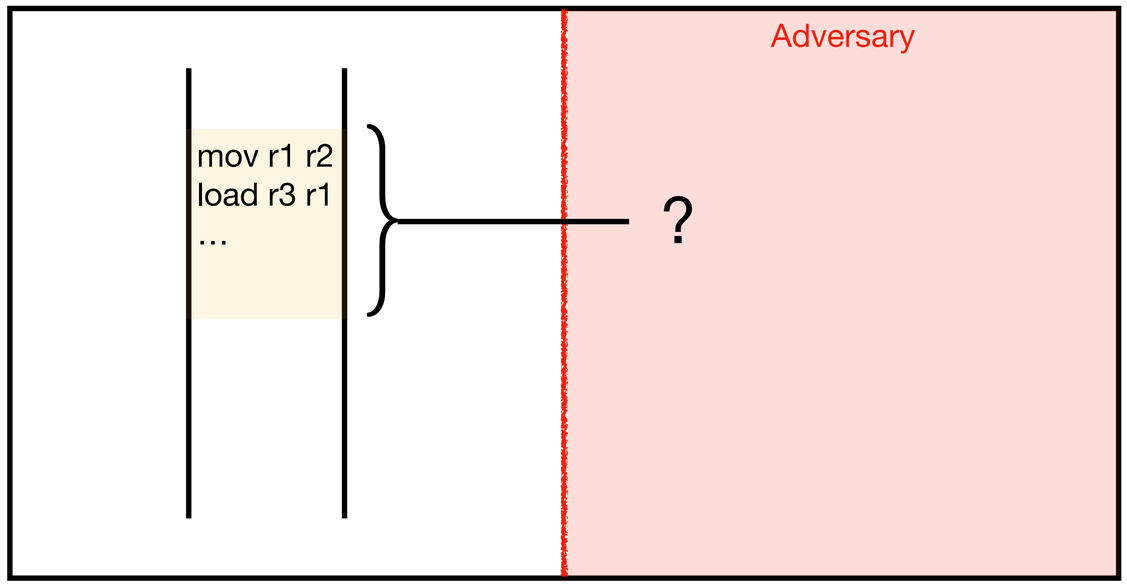
What authority does a function pointer grant?

A Lattice of Permissions



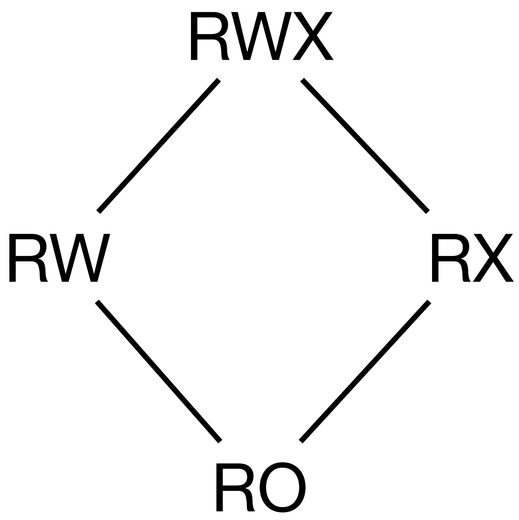
What about function pointers?

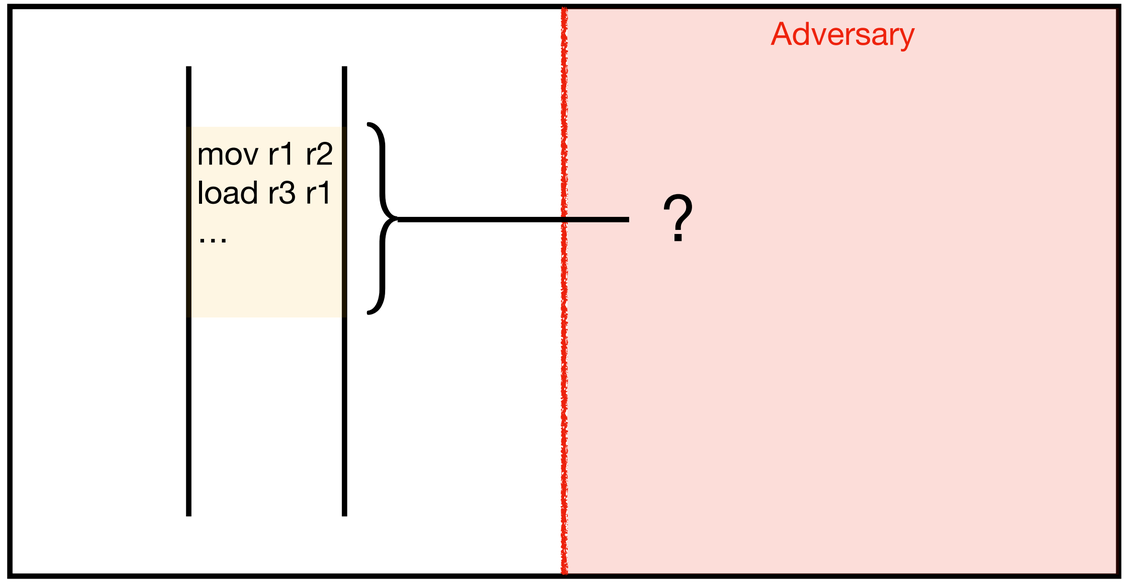
What authority does a function pointer grant?



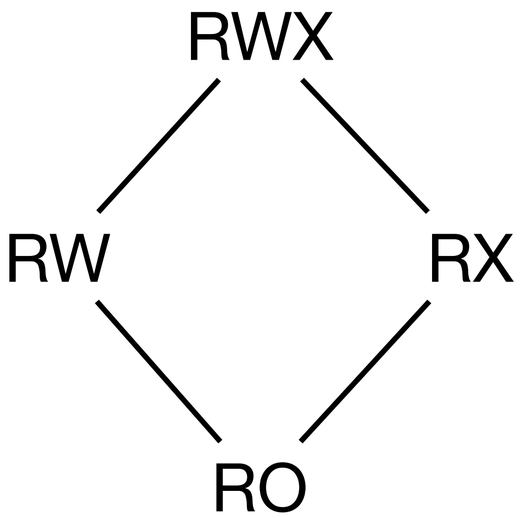
Attacker model: linking with arbitrary capability machine code

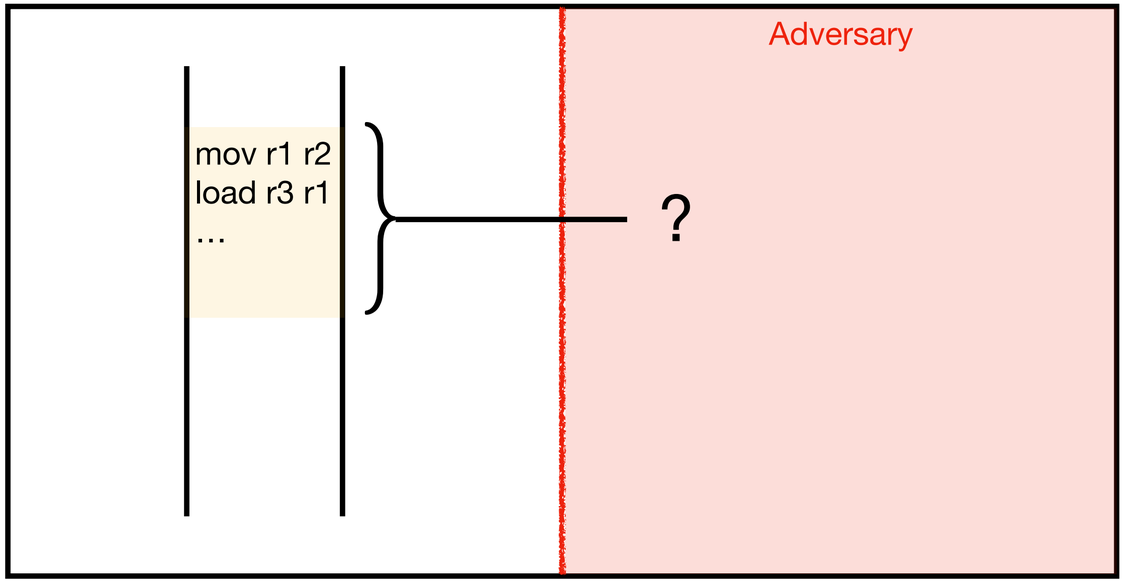
A Lattice of Permissions



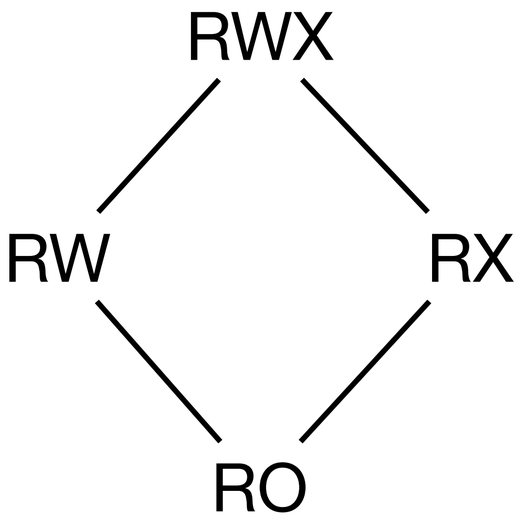


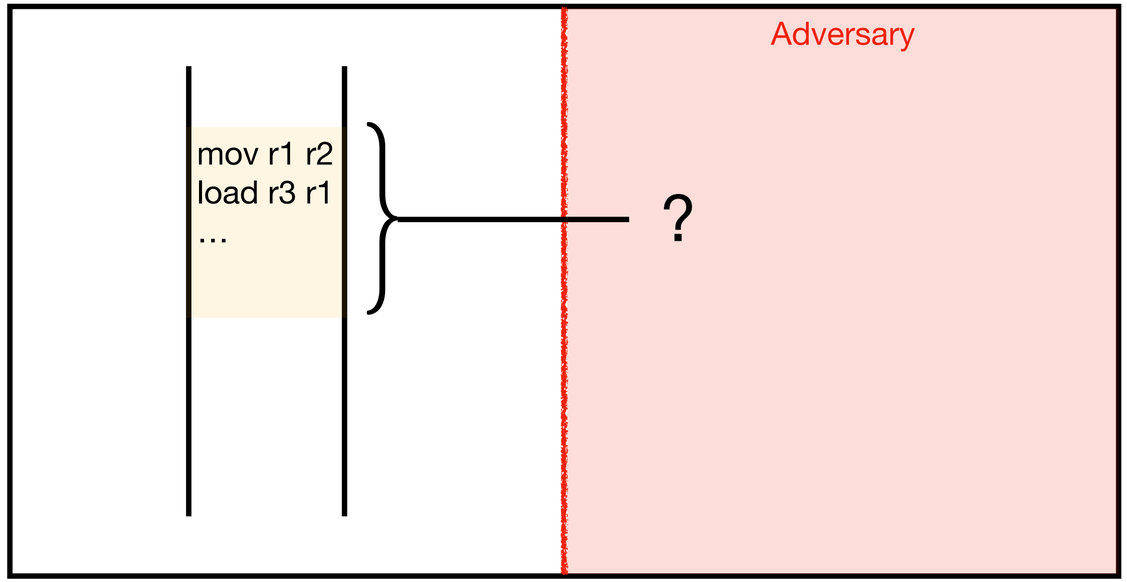
A Lattice of Permissions





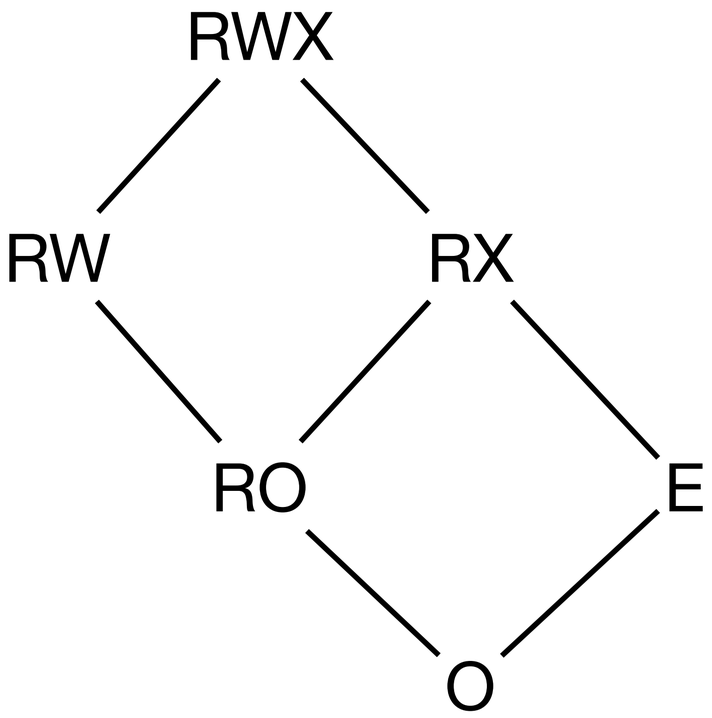
A Lattice of Permissions

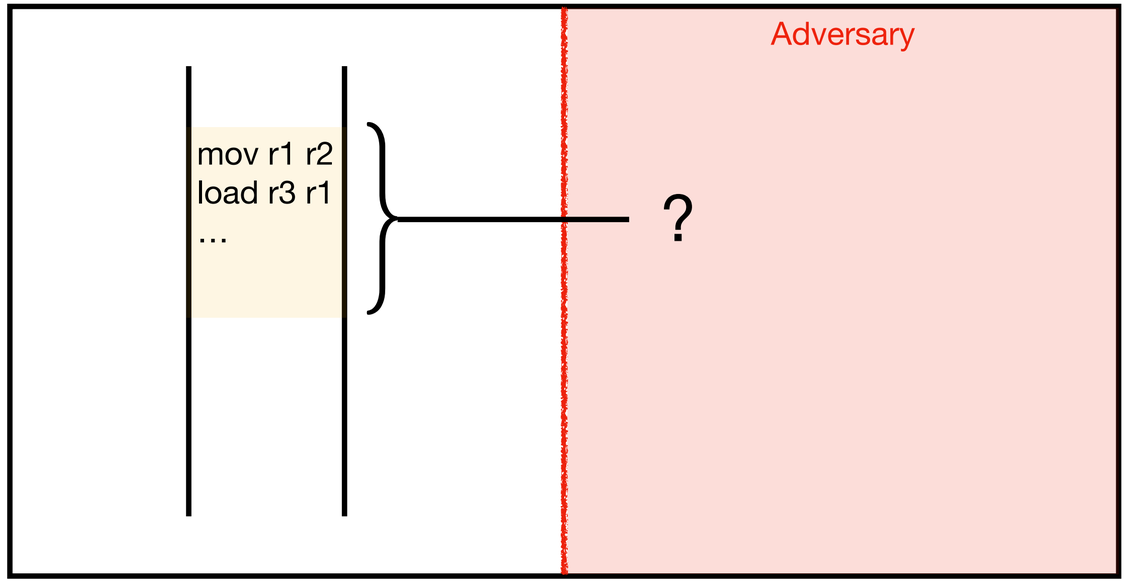




We need a permission for “a capability that can only be jumped”

A Lattice of Permissions





E : enter permission

(similar to CHERI's seals)

We need a permission for "a capability that can only be jumped"

Operations

Operations

State machine

$$(reg, mem) \longrightarrow (reg', mem')$$

reg : RegName → Word

mem : Addr → Word

RegName = {PC, r1, r2, r3}

Addr = [0,AddrMax)

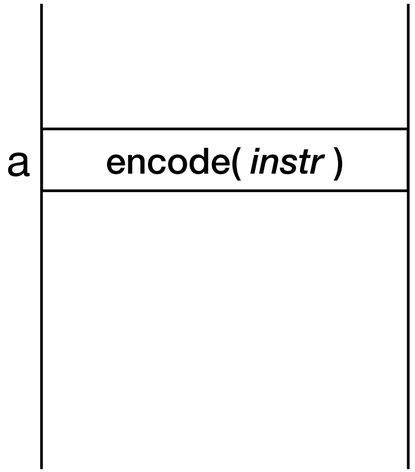
a word is either a capability or an integer

Operations

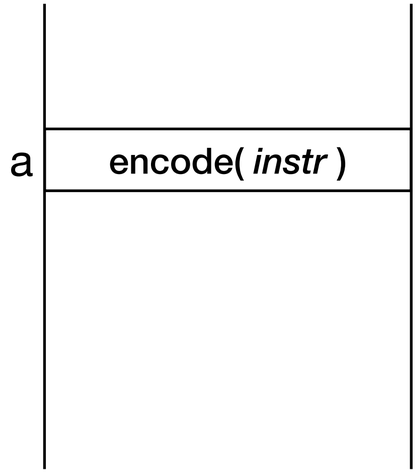
General shape of an instruction

With some dynamic checks in between,
starting with a check to see whether the PC is valid

PC	(RX,b,e,a)
r1	
r2	
r3	



PC	(RX,b,e,a+1)
r1	
r2	
r3	

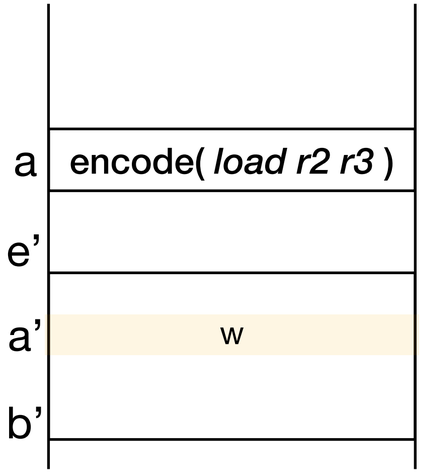


Operations

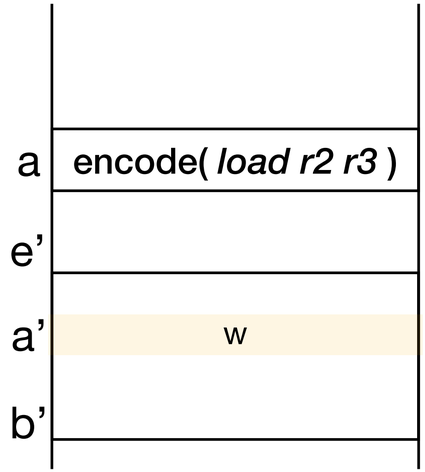
Load

PC	(RX,b,e,a)
r1	
r2	
r3	(p,b’,e’,a’)

PC	(RX,b,e,a+1)
r1	
r2	w
r3	(p,b’,e’,a’)



b’ ≤ a’ < e’
p is at least RO

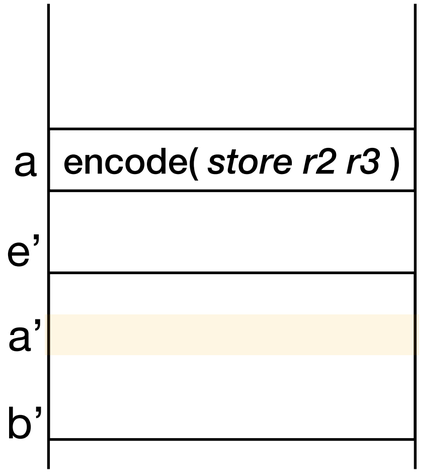


Operations

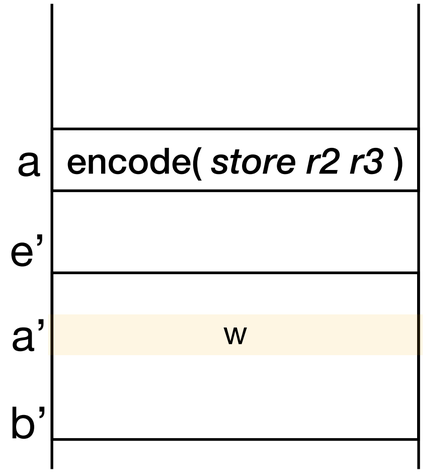
Store

PC	(RX,b,e,a)
r1	
r2	(p,b’,e’,a’)
r3	w

PC	(RX,b,e,a+1)
r1	
r2	(p,b’,e’,a’)
r3	w



b’ ≤ a’ < e’
p is at least RW

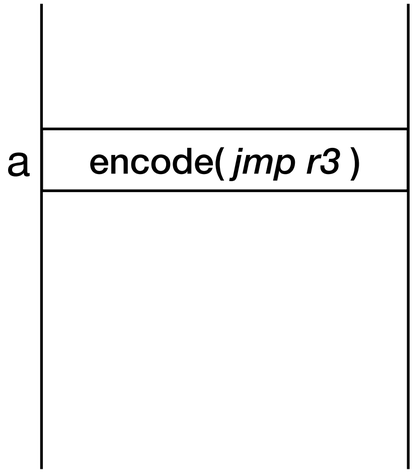


Operations

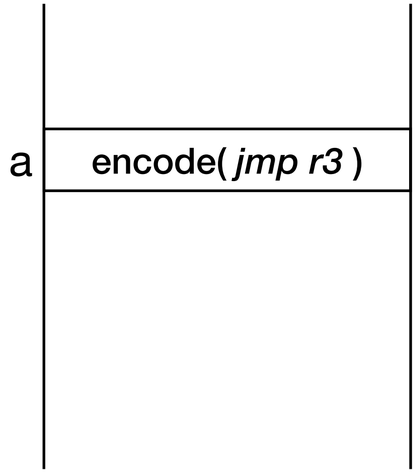
Jmp

PC	(RX,b,e,a)
r1	
r2	
r3	w

PC	w
r1	
r2	
r3	w



?

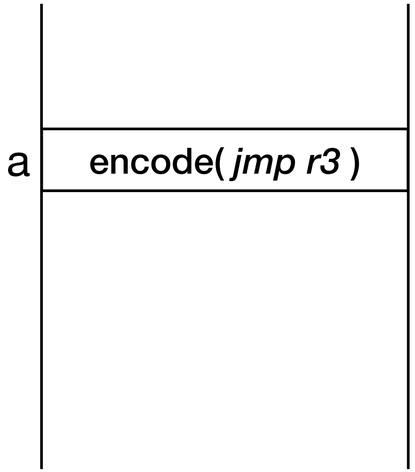


Operations

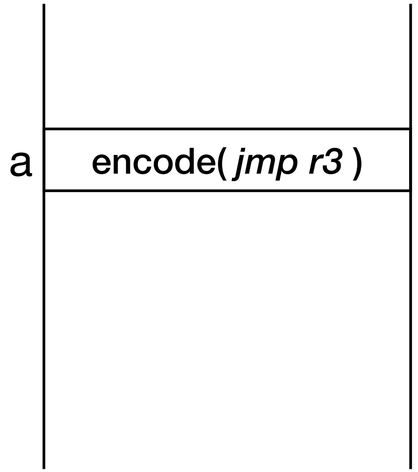
Jmp to an enter capability

PC	(RX,b,e,a)
r1	
r2	
r3	(E,b’,e’,a’)

PC	
r1	
r2	
r3	(E,b’,e’,a’)



?

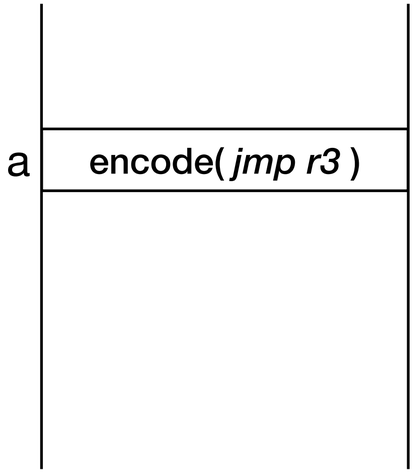


Operations

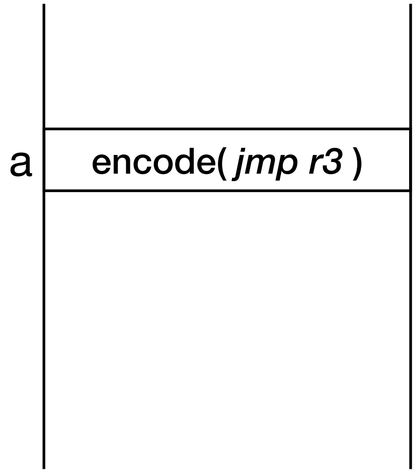
Jmp to an enter capability

PC	(RX,b,e,a)
r1	
r2	
r3	(E,b’,e’,a’)

PC	(RX,b’,e’,a’)
r1	
r2	
r3	(E,b’,e’,a’)



?

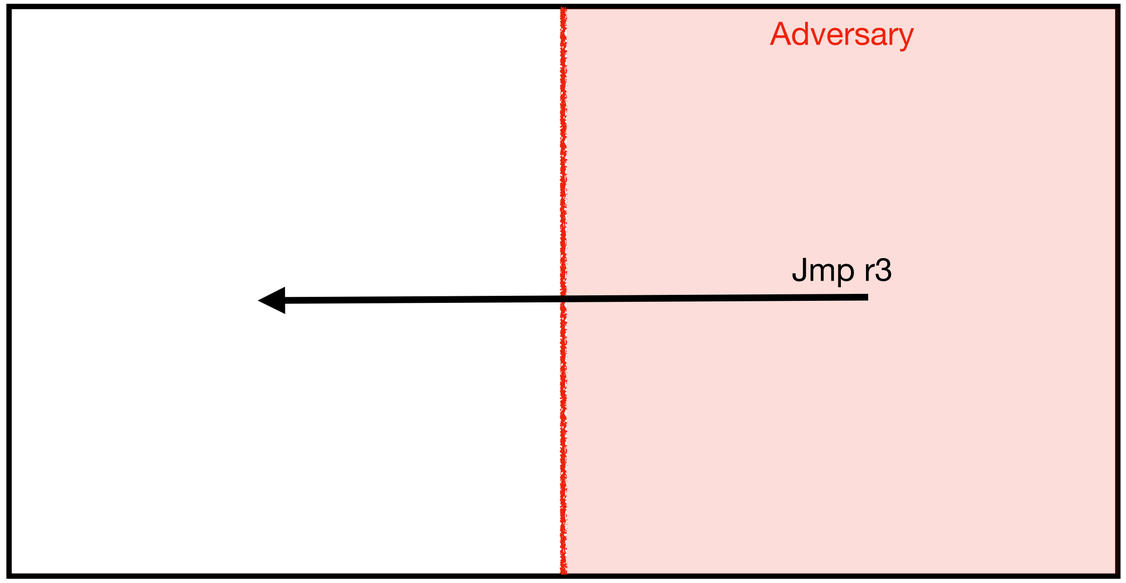


Operations

Jmp to an enter capability

PC	(RX,b,e,a)
r1	
r2	
r3	(E,b’,e’,a’)

PC	(RX,b’,e’,a’)
r1	
r2	
r3	(E,b’,e’,a’)



Operations

Is it safe to have instructions that can change capabilities?

Operations

Is it safe to have instructions that can change capabilities?

- Restrict : **lowers** the permission of a capability
- Subseg : **tightens** the bounds of a capability

Operations

Is it safe to have instructions that can change capabilities?

- Restrict : **lowers** the permission of a capability
- Subseg : **tightens** the bounds of a capability

What about pointer arithmetic?

Operations

Is it safe to have instructions that can change capabilities?

- Restrict : **lowers** the permission of a capability
- Subseg : **tightens** the bounds of a capability

What about pointer arithmetic?

- Lea : produces pointer arithmetic on the address of a **non-enter** capability

Operations

What should happen if a dynamic check fails?

Operational Semantics

Exercise: symbolically execute the following program

```
0:   mov r1 PC
     lea r1 5
     load r1 r1
     subseg r1 10 11
     jmp r2
5:   (RW,9,12,9)
6:   lea r1 1
7:   store r1 42
8:   halt
9:   0
10:  0
11:  0
```

From the following starting register state:

PC	(RWX,0,6,0)
r1	0
r2	(E,6,9,6)
r3	0

Operational Semantics

Exercise: symbolically execute the following program

```
0:   mov r1 PC
     lea r1 5
     load r1 r1
     subseg r1 10 11
     jmp r2
5:   (RW,9,12,9)
6:   lea r1 1
7:   store r1 42
8:   halt
9:   0
10:  0
11:  0
```

From the following starting register state:

PC	(RWX,0,6,0)
r1	0
r2	(E,6,9,6)
r3	0

instr dst src

lea dst z adds z to the address of the capability in dst

subseg dst z1 z2 changes the bounds of the capability in dst to [z1,z2)

Operational Semantics

Exercise: symbolically execute the following program

```
0:   mov r1 PC
     lea r1 5
     load r1 r1
     subseg r1 10 11
     jmp r2
5:   (RW,9,12,9)
6:   lea r1 1
7:   store r1 42
8:   halt
9:   0
10:  0
11:  0
```

From the following starting register state:

PC	(RWX,0,6,0)
r1	0
r2	(E,6,9,6)
r3	0

Operational Semantics

Exercise: symbolically execute the following program

```
0:   mov r1 PC
     lea r1 5
     load r1 r1
     subseg r1 10 11
     jmp r2
5:   (RW,9,12,9)
6:   lea r1 1
7:   store r1 42
8:   halt
9:   0
10: 0
11: 0
```

From the following starting register state:

PC	(RWX,0,6,0)
r1	0
r2	(E,6,9,6)
r3	0

PC	(RX,6,9,8)
r1	(RW,10,11,10)
r2	(E,6,9,6)
r3	0

10: 42

Capability Safety (and going further)

Capability Safety

- Often we talk about memory safety, but as discussed, memory safety is not exactly a precise definition
- Instead, we can formally talk about “capability safety”
  - What is capability safety?

    “A single call must end with less (or equal) authority than it started out with”

  - **Capability monotonicity**
  - **Software compartmentalisation**

Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process

Kyndylan Nienhuis*, Alexandre Joannou*, Thomas Bauereiss*, Anthony Fox†, Michael Roe*, Brian Campbell‡, Matthew Naylor*, Robert M. Norton*, Simon W. Moore*, Peter G. Neumann§, Ian Stark‡, Robert N. M. Watson*, and Peter Sewell*
*University of Cambridge †ARM Limited ‡University of Edinburgh §SRI International

Goal of Fault Isolation

- Execute untrusted code in isolation
- Confine errors within distrusted module
- Call media player to codec
- Call kernel and device drivers

Isolation mechanisms

- Hardware process isolation (MMU)

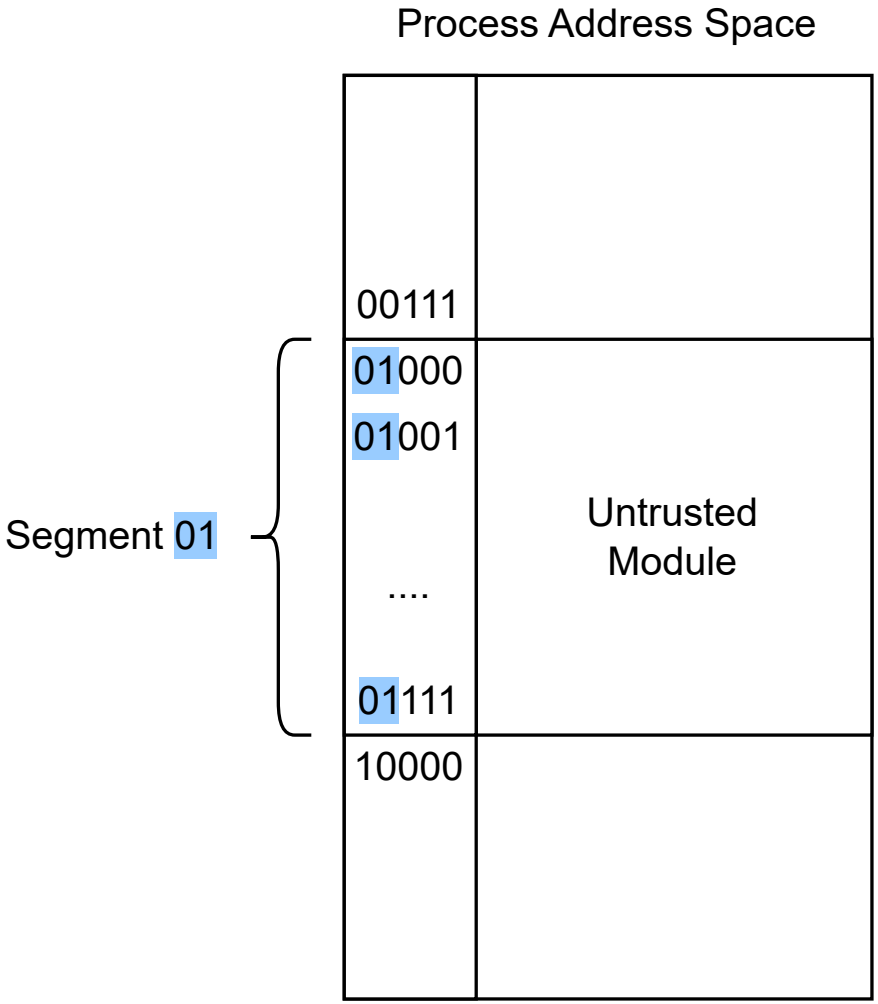
Isolation mechanisms

- Hardware process isolation (MMU)
- Context switching is heavy:
  1. trap into the OS level
  2. copy arguments caller-to-callee
  3. saving (and restoring) registers
  4. switching address space
  5. trap back to user level

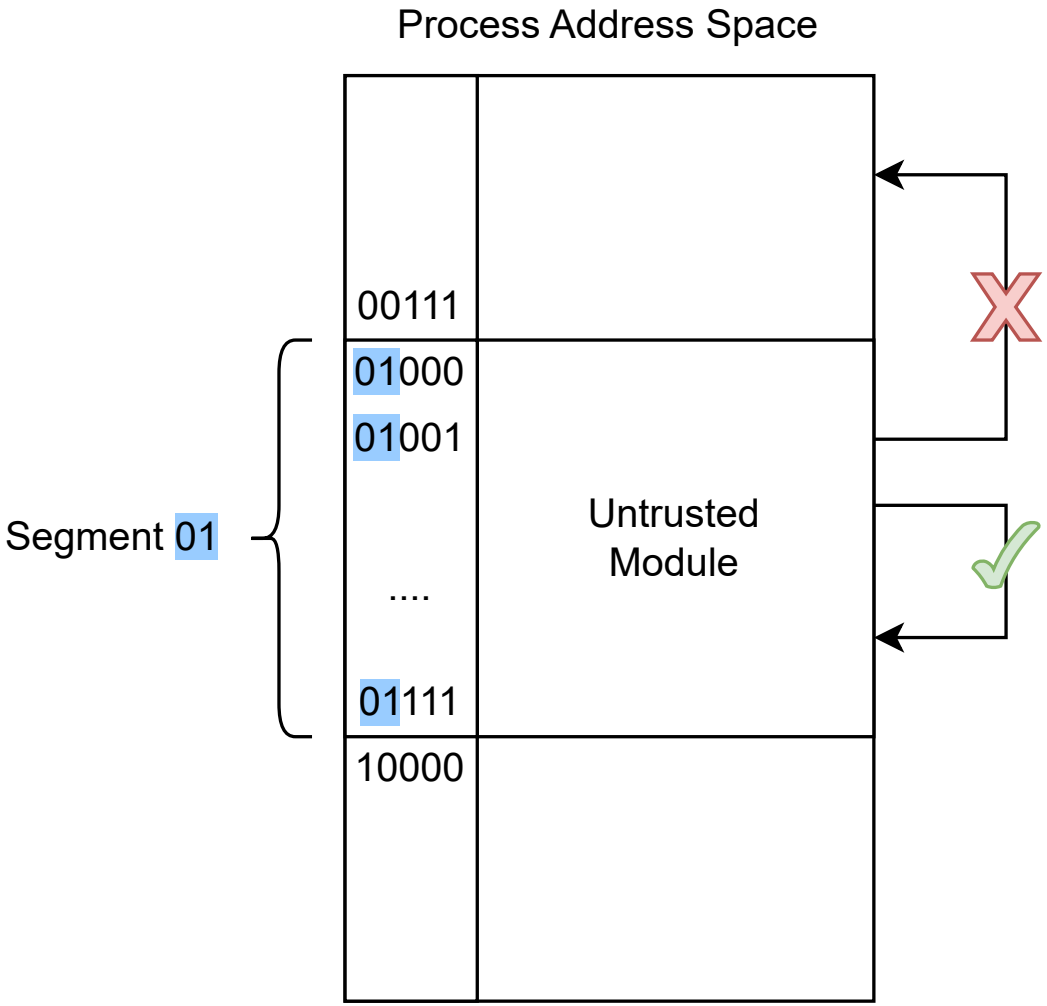
Isolation mechanisms

- Hardware process isolation (MMU)
- Context switching is heavy:
  1. trap into the OS level
  2. copy arguments caller-to-callee
  3. saving (and restoring) registers
  4. switching address space
  5. trap back to user level
- Software isolation: within the same address space!
- Slow down common case (normal execution) to speedup uncommon case (cross-domain communication)

Software Fault Isolation (SFI)



Software Fault Isolation (SFI)



Transform code of untrusted modules

Unsafe instructions

What are instruction possibly unsafe?

Transform code of untrusted modules

Unsafe instructions

What are instruction possibly unsafe?

- static jumps are easy to verify statically
- insert instructions to prevent unsafe instruction to escape

Transform code of untrusted modules

Unsafe instructions

What are instruction possibly unsafe?

- static jumps are easy to verify statically
- insert instructions to prevent unsafe instruction to escape

Encapsulation

- Segment matching $\rightarrow$ checks that memory accesses stay within the fault domain, fault otherwise
- Sandboxing $\rightarrow$ masks the upper bits of the address of memory access with segment identifiers

Safety and limitation

Safety

- checking/masking instructions must run always run to ensure safety (CFI)
- dedicated registers not writable by unstrusted module

Limitations

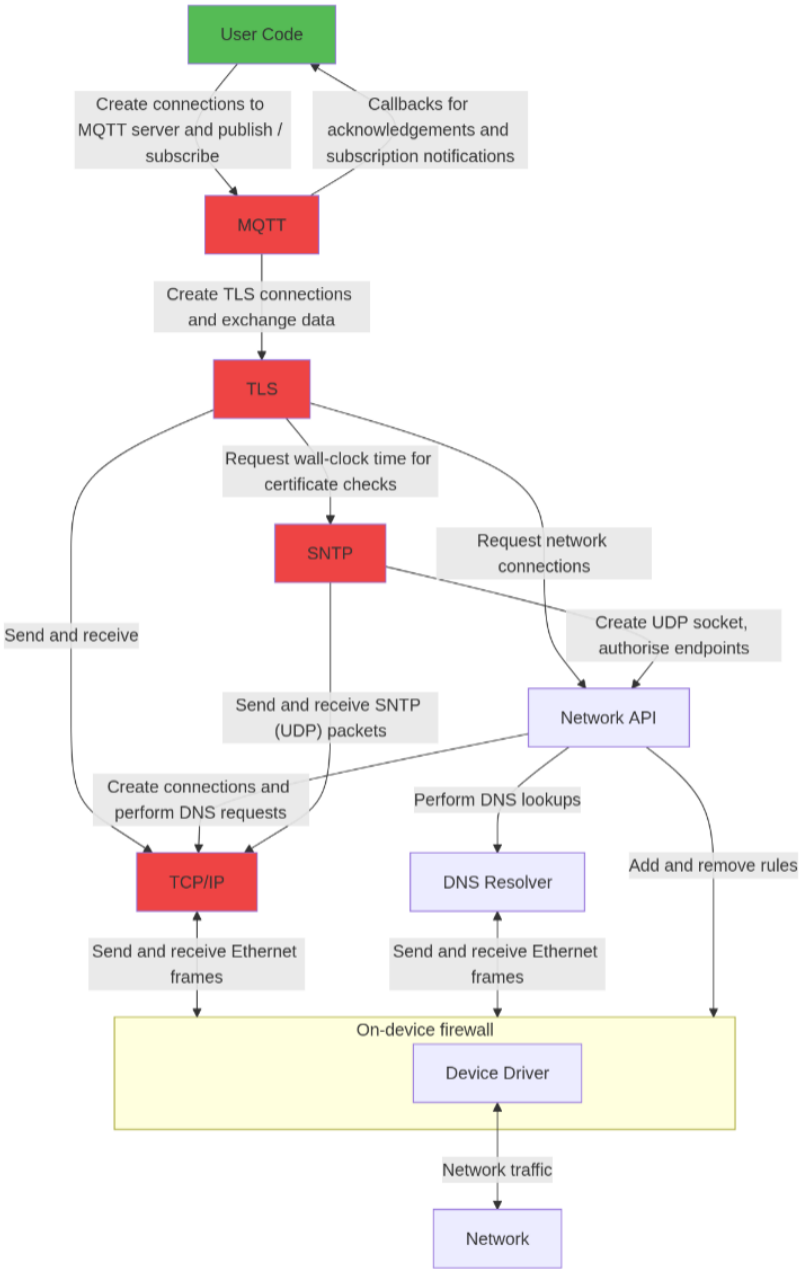
- memory sharing? can be done relying on hardware page table aliasing
- sharing pointers? no virtual memory (IoT devices)

Compartmentalisation with capabilities

CHERIoT

- CHERI for IoT devices (no virtual memory)
- Use capabilities to compartmentalise: pointer sharing!
- Build compartments on top of CHERI memory safety

CHERIoT network stack



- Relies on third parties (TLS, TCP/IP)
- strong guarantees: f.x. only encrypted messaged can be sent on the network

Capability Safety and Security

What other security properties can we enforce using capabilities?

In particular, what properties of **high level languages** can we now enforce?

$$
\begin{array}{l}
\mathit{let}\ x := \mathit{ref}(0)\ \mathit{in} \\
\lambda\, f : () \to (),\ \ x \leftarrow 42;\ f();\ \mathit{assert}(!x = 42)
\end{array}
$$

Capability Safety and Security

What other security properties can we enforce using capabilities?

In particular, what properties of **high level languages** can we now enforce?

$$let\ x := ref(0)\ in$$
$$\lambda\, f : () \to (),\ \ x \leftarrow 42;\ f();\ assert(!x = 42)$$

Local state encapsulation

Capability Safety and Security

What security properties can enforce using capabilities?

In particular, what properties of **high level languages** can we now enforce?

$$
\begin{array}{l}
\mathit{let}\ x := \mathit{ref}(0)\ \mathit{in} \\
\lambda\, f : () \to (),\ \ x \leftarrow 42;\ f();\ x \leftarrow 43;\ f();\ \mathit{assert}(!x = 43)
\end{array}
$$

Capability Safety and Security

What security properties can enforce using capabilities?

In particular, what properties of **high level languages** can we now enforce?

$$let\ x := ref(0)\ in$$
$$\lambda\, f : () \to (),\ \ x \leftarrow 42;\ f();\ x \leftarrow 43;\ f();\ assert(!x = 43)$$

Well bracketed control flow

Well bracketed control flow

What is missing from our simple machine to be able to reason about well bracketed calls?

What is missing from our machine if you were to implement a compiler that targets a capability machine?

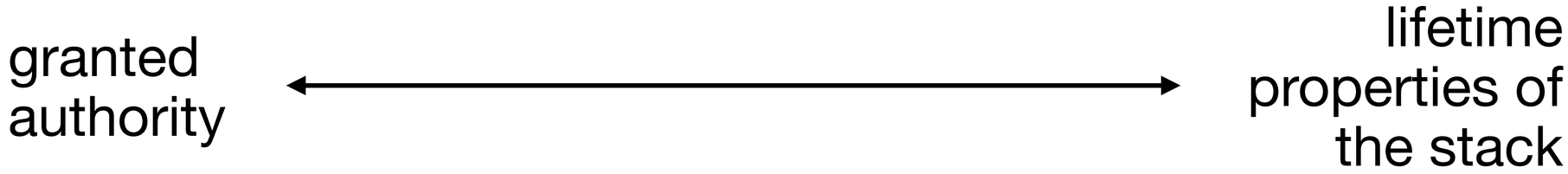
Well bracketed control flow

What is missing from our simple machine to be able to reason about well bracketed calls?

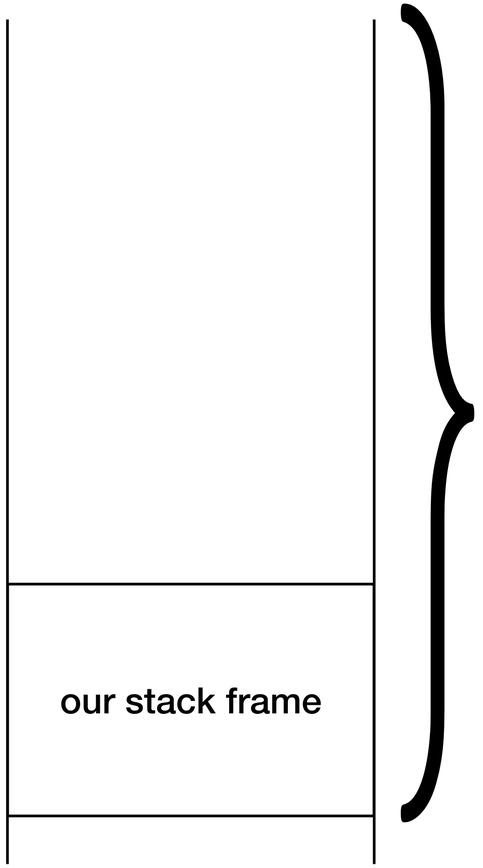
What is missing from our machine if you were to implement a compiler that targets a capability machine?

A stack!

Difficulties of a stack

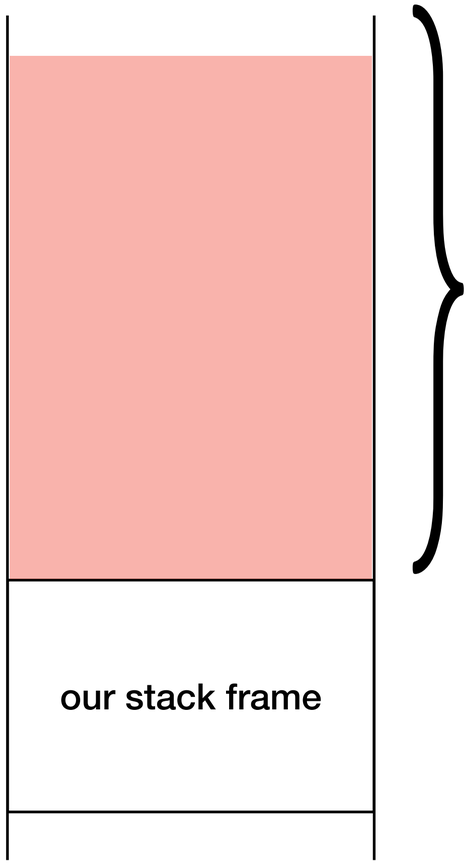


Difficulties of a stack



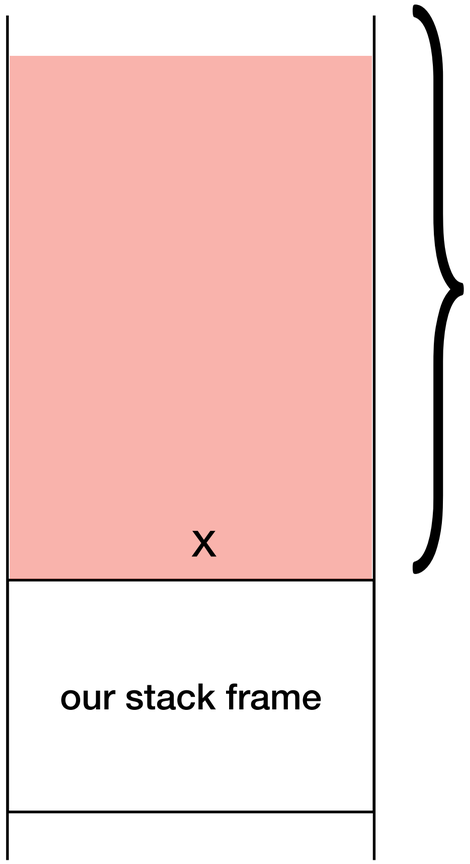
$$\cdots; f(x); \cdots; f(y); \cdots;$$

Difficulties of a stack



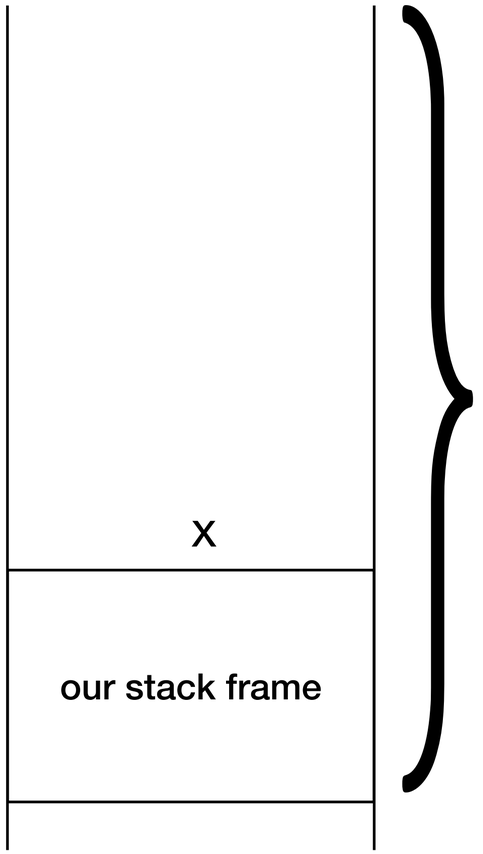
$$\cdots; f(x); \cdots; f(y); \cdots;$$

Difficulties of a stack



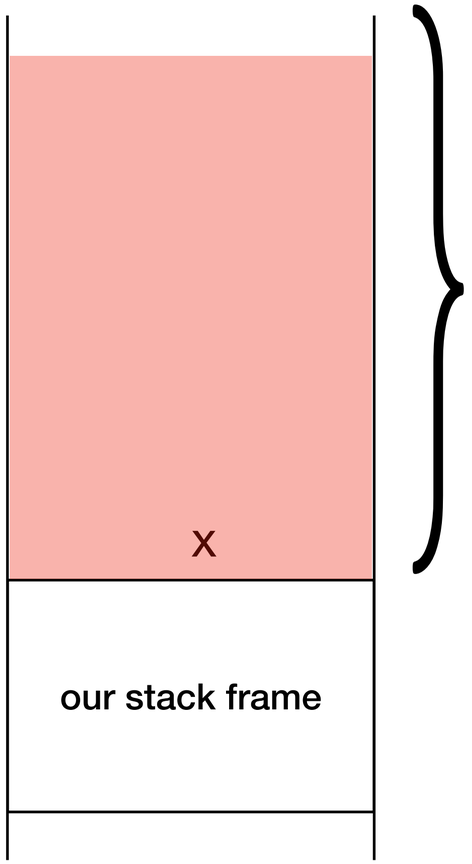
$$\cdots; f(x); \cdots; f(y); \cdots;$$

Difficulties of a stack



$$\cdots; f(x); \cdots; f(y); \cdots;$$

Difficulties of a stack



$$\cdots; f(x); \cdots; f(y); \cdots;$$

Difficulties of a stack

- This showed an example of local state encapsulation
- Other properties include:
  - Well bracketed control flow (frames are pushed and popped in a first in first out order)
  - Temporal stack safety (popped frames cannot be read by caller)

Difficulties of a stack

- How can we solve this issue?
  - Using a calling convention
- How can we solve this fast?
  - Using a calling convention and FANCY capabilities!