Temporal safety for stack allocated memory on capability machines

Stelios Tsampas Dominique Devriese Frank Piessens

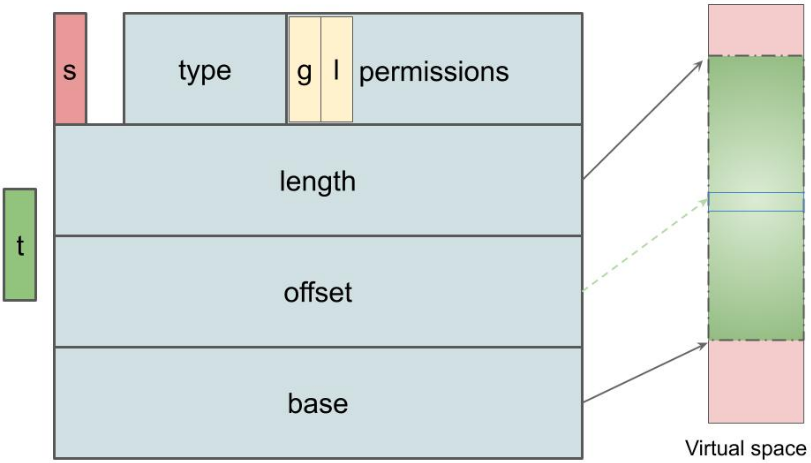
stelios.tsampas@cs.kuleuven.be

imec-DistriNet, KU Leuven

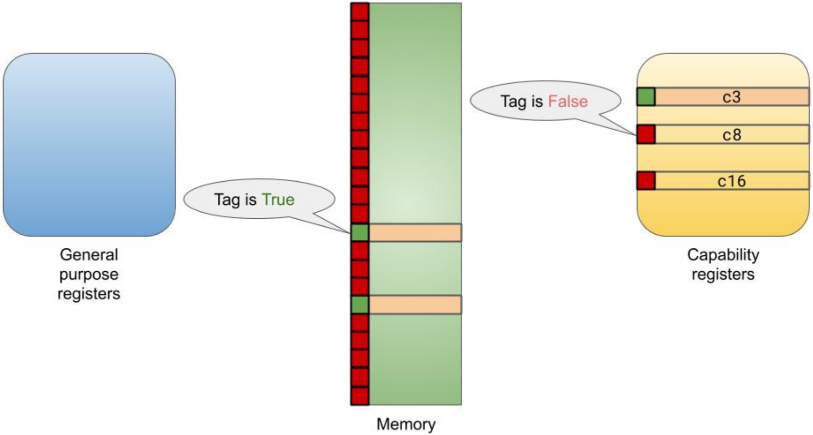
IEEE Computer Security Foundations Symposium, June 27, 2019

Capability machine	a secure architecture
(Data/code) capability	memory access token
Object capability	representation of sandboxes
CHERI[1]	a prominent capability machine

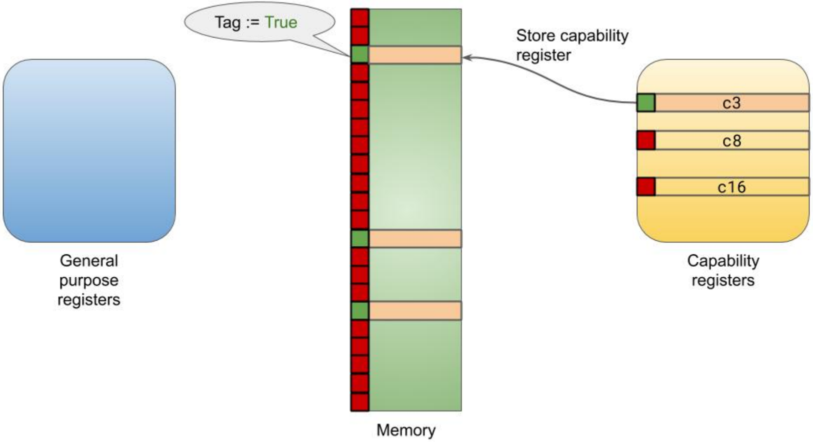
[1] R. N. Watson, Woodruff, Neumann, S. W. Moore, Anderson, Chisnall, Dave, Davis, Gudka, Laurie, *et al.*, “CHERI: A hybrid capability-system architecture for scalable software compartmentalization”.


s
type
g
l
permissions
length
t
offset
base
Virtual space

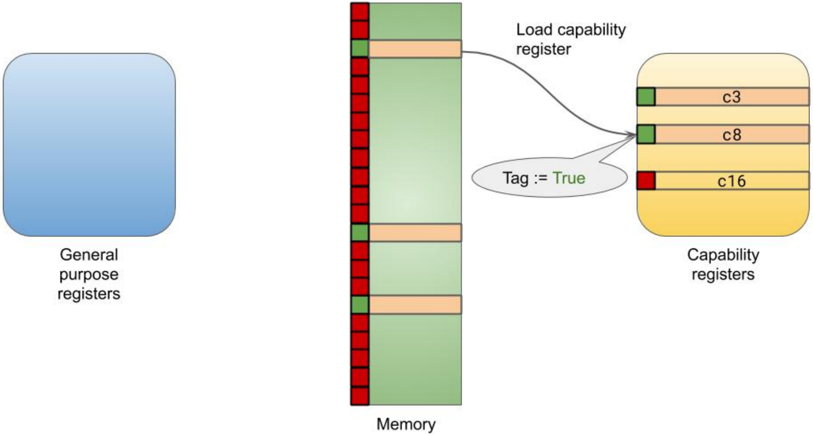
Unforgeability



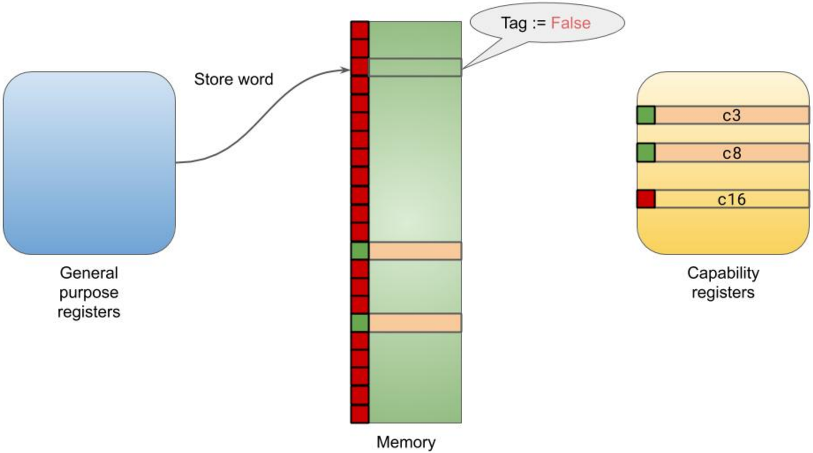
Unforgeability



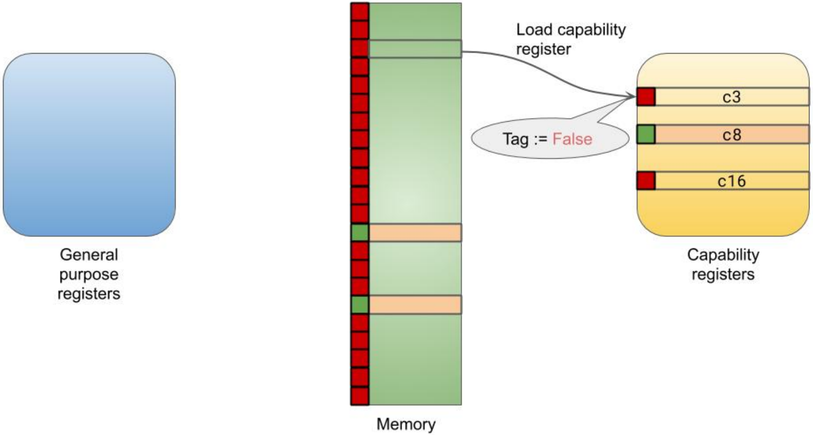
Unforgeability



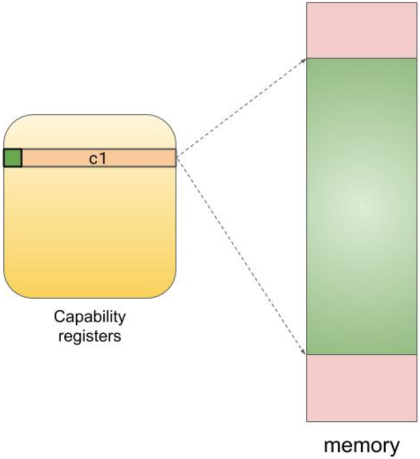
Unforgeability



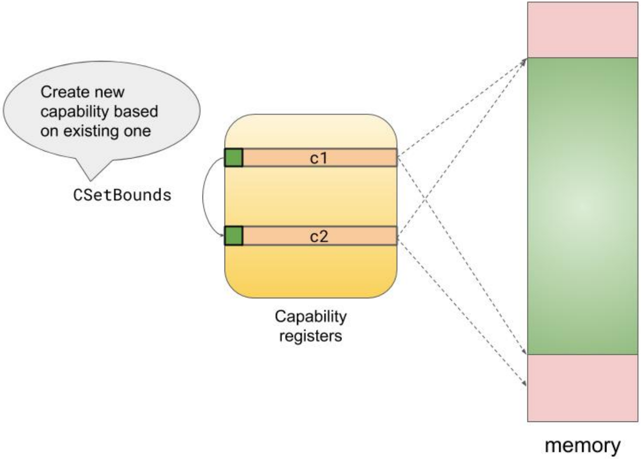
Unforgeability



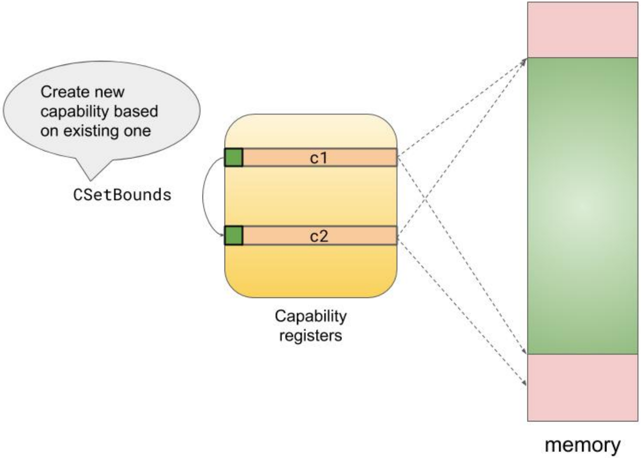
Monotonicity



Monotonicity



Monotonicity



Create new capability based on existing one

CSetBounds

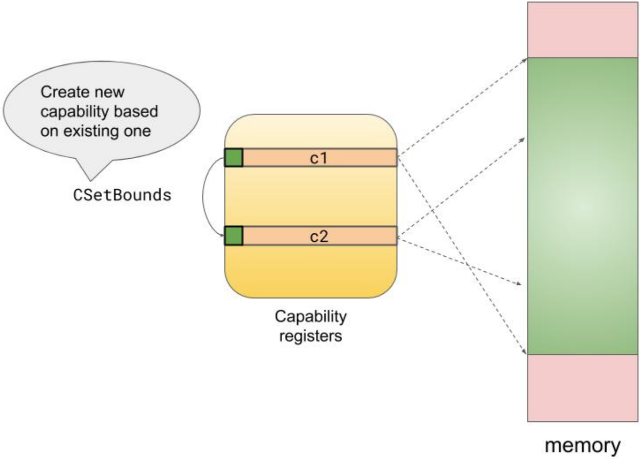
c1

c2

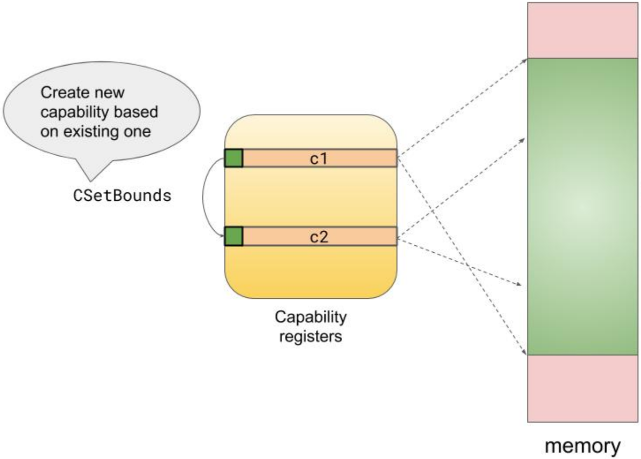
Capability registers

memory

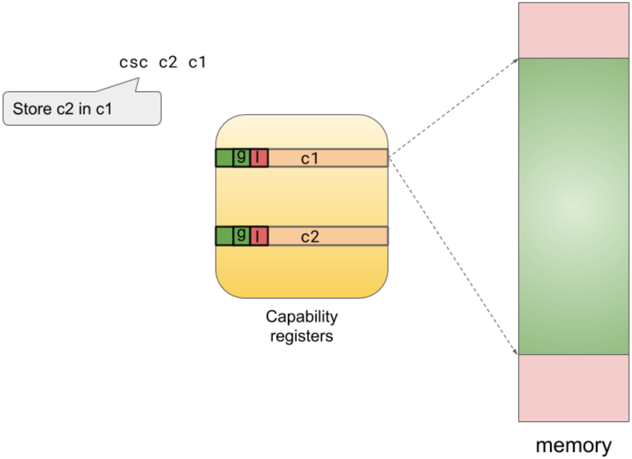
Monotonicity



Monotonicity



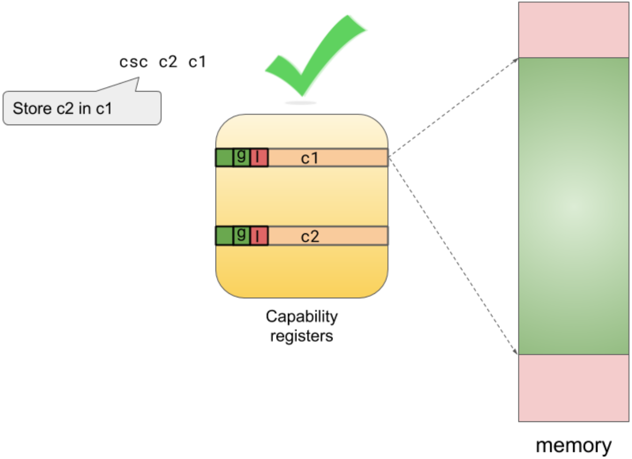
The *g* bit



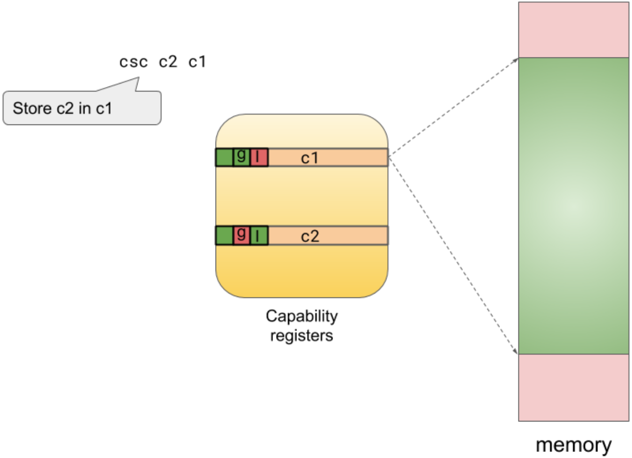
```
csc c2 c1
```

Store c2 in c1

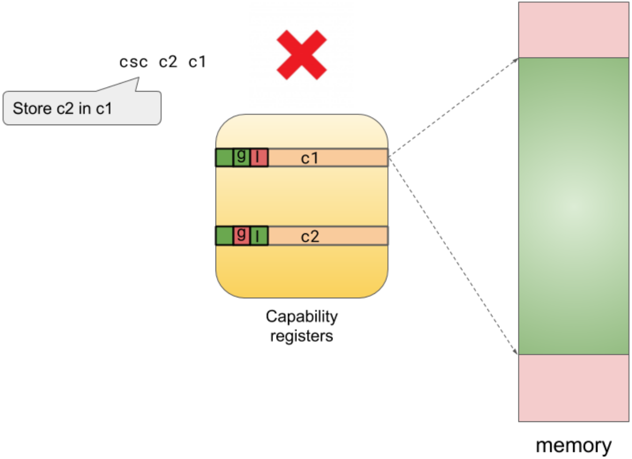
The *g* bit


csc c2 c1
Store c2 in c1
g
l
c1
g
l
c2
Capability registers
memory

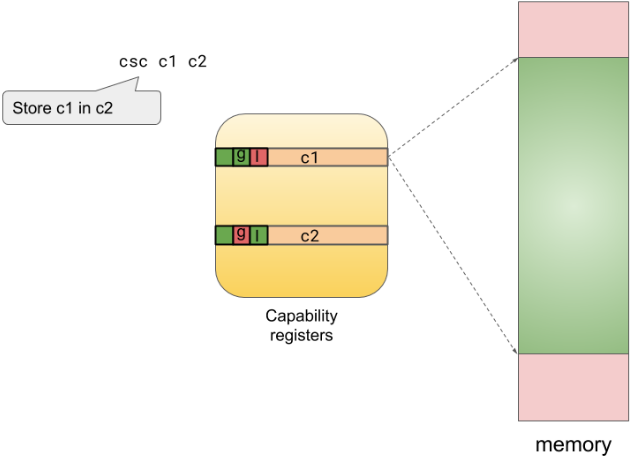
The *g* bit


csc c2 c1
Store c2 in c1
g
l
c1
g
l
c2
Capability registers
memory

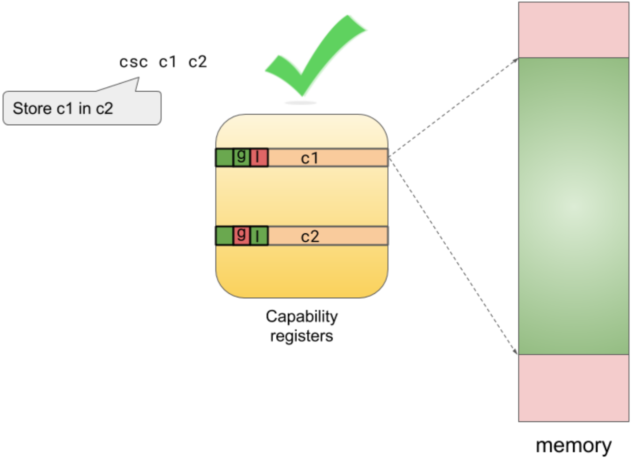
The *g* bit


csc c2 c1
Store c2 in c1
g
l
c1
g
l
c2
Capability registers
memory

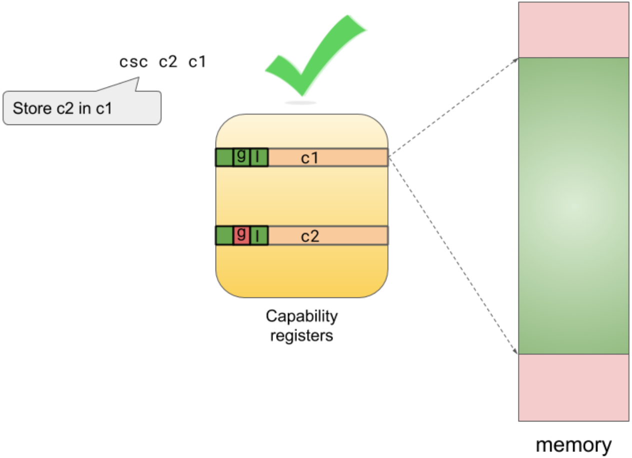
The *g* bit


csc c1 c2
Store c1 in c2
g
l
c1
g
l
c2
Capability registers
memory

The *g* bit


csc c1 c2
Store c1 in c2
g
l
c1
g
l
c2
Capability registers
memory

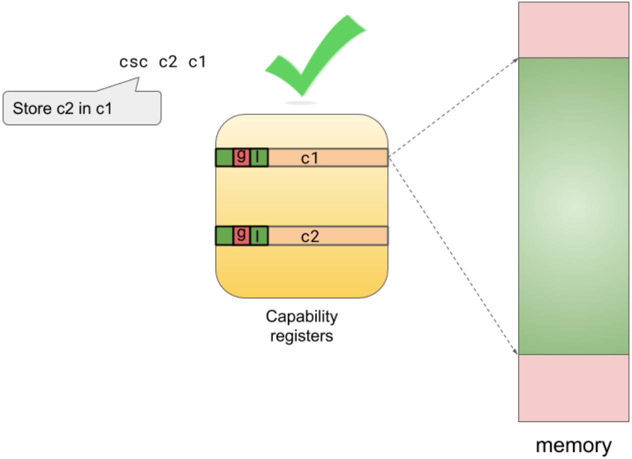
The *g* bit



```
csc c2 c1
```

Store c2 in c1

The *g* bit



csc c2 c1
Store c2 in c1
g
l
c1
g
l
c2
Capability registers
memory

CHERI does not allow local capabilities passed around sandboxes.

```c
// fetch and sort are exported from a different sandbox
void fetch(int *r);
void sort(int *r);

void main(void) {
  int q[100]; // compiled as a local capability
  fetch(q); // not allowed
  sort(q); // not allowed
}
```

The above code will cause a runtime exception.

What if the restriction was lifted...

```c
void ally(int** p) {
  int x;
  *p = &x; // Unsafe assignment
}
void main() {
  int *q;
  ally(&q); // q points at unused stack memory
  victim(q);
}
void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

[2] Song, Lettner, Rajasekaran, Na, Volckaert, Larsen, and Franz, “SoK: Sanitizing for Security”.

[3] *CVE-2015-1730. CVE-2017-7756.*

What if the restriction was lifted...

```
void ally(int** p) {
  int x;
  *p = &x; // Unsafe assignment
}
void main() {
  int *q;
  ally(&q); // q points at unused stack memory
  victim(q);
}
void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

- Attack in a sandboxed environment

[2] Song, Lettner, Rajasekaran, Na, Volckaert, Larsen, and Franz, “SoK: Sanitizing for Security”.
[3] *CVE-2015-1730. CVE-2017-7756.*

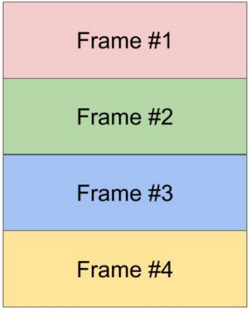
What if the restriction was lifted...

```c
void ally(int** p) {
  int x;
  *p = &x; // Unsafe assignment
}
void main() {
  int *q;
  ally(&q); // q points at unused stack memory
  victim(q);
}
void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

- Attack in a sandboxed environment
- ...Also a bug in a single-sandbox application[2,3]

[2] Known as *stack-based use-after-free* or *use-after-return* (Song, Lettner, Rajasekaran, Na, Volckaert, Larsen, and Franz, “SoK: Sanitizing for Security”)

[3] *CVE-2015-1730. CVE-2017-7756.*



Frame #1

Frame #2

Frame #3

Frame #4

- The number of active stack frames defines a hierarchy of various lifetimes
- The more recent the frame, the less its variables will live
- 2^n stack frames require n bits to accurately represent the lifetimes of their objects
- The 1-bit information flow model is not adequate

Contributions

What? Reserve additional bits for a hierarchy of localities

Contributions

What? Reserve additional bits for a hierarchy of localities

How? Expand policy for multiple levels of locality

Contributions

What? Reserve additional bits for a hierarchy of localities

How? Expand policy for multiple levels of locality

Really? Formalization and proof of correctness in Agda

Contributions

What? Reserve additional bits for a hierarchy of localities

How? Expand policy for multiple levels of locality

Really? Formalization and proof of correctness in Agda

So? Propose an implementation in CHERI

- Capabilities have an extra *n*-bit field to represent locality
  - The higher the value, the more ephemeral the region
  - Local/global no longer a meaningful distinction
- Storing a capability in a region requires:
  - Original boundary checks
  - $\texttt{source.locality} \leq \texttt{destination.locality}$
- Sandbox capability restriction is now lifted

Formal methodology

ImpR High level language with local variables and functions

Ideal Idealized dependently typed machine that runs ImpR “as intended”

Cap Unmodified capability semantics

Cap+ Extended capability semantics

ImpR | Ideal

- Pointer values are always in bounds
- Pointers in the `Store` may only point to current or parent stack frame
- Assignment is restricted by the definition of `Store`
- Local pointers can be used as arguments

ImpR | Cap

- A capability may point to an out of bounds address
- `Memory` is simply an array of values
- No restrictions on assignments
- Capabilities cannot be used as arguments

ImpR | Ideal

- Pointer values are always in bounds
- Pointers in the `Store` may only point to current or parent stack frame
- Assignment is restricted by the definition of `Store`
- Local pointers can be used as arguments

ImpR | Cap

- A capability may point to an out of bounds address
- `Memory` is simply an array of values
- No restrictions on assignments
- Capabilities cannot be used as arguments

We show that the capability semantics cannot simulate the ideal ones.

ImpR | Ideal

- Pointer values are always in bounds
- Pointers in the `Store` may only point to the current or a parent stack frame
- Assignment is restricted by the definition of `Store`
- Local pointers can be used as arguments

ImpR | Cap+

- A capability consists of an address **and a locality counter**
- `Memory` is still just an array of values
- Assigning a capability value `c` to the location referenced by cap/ty `d` **requires** `c.locality` ≤ `d.locality`
- Local capabilities **can** be used as arguments

ImpR | Ideal

- Pointer values are always in bounds
- Pointers in the `Store` may only point to the current or a parent stack frame
- Assignment is restricted by the definition of `Store`
- Local pointers can be used as arguments

ImpR | Cap+

- A capability consists of an address **and a locality counter**
- `Memory` is still just an array of values
- Assigning a capability value `c` to the location referenced by cap/ty `d` **requires** `c.locality` ≤ `d.locality`
- Local capabilities **can** be used as arguments

We show that the extended capability semantics can simulate the ideal ones and prove that the identity compiler is fully abstract.

Proof

Proof

It’s inductive

Proof

It’s inductive

– Available online [4]

[4] https://github.com/solidsnk/cap-extensions.git

```c
// fetch and sort are exported from a different sandbox
void fetch(int *r);
void sort(int *r);

void main(void) {
  int q[100]; // compiled as a local capability
  fetch(q); // allowed
  sort(q); // allowed
}
```

```
// fetch and sort are exported from a different sandbox
void fetch(int *r);
void sort(int *r);

void main(void) {
  int q[100]; // compiled as a local capability
  fetch(q); // allowed
  sort(q); // allowed
}
```

The above code will **not** cause a runtime exception.

```c
void ally(int** p) {
  int x; // &x.locality = 1
  *p = &x; // &x.locality > q.locality
}
void main() {
  int *q; // q.locality = 0
  ally(&q);
  victim(q);
}
void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

```
void ally(int** p) {
  int x; // &x.locality = 1
  *p = &x; // &x.locality > q.locality
}
void main() {
  int *q; // q.locality = 0
  ally(&q);
  victim(q);
}
void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

This **will** cause an exception at the unsafe assignment

Notes on CHERI implementation

- Use reserved bits for locality counter
- Adequate (est.) number in 256-bit version
- Locality bottoms out if bits are exhausted
- New compression schemes[5] allow for 128-bit implementation
- We require automatic cleanup of stack on sandbox entry
- Few necessary adjustments in stack allocator

[5] Woodruff, Joannou, Xia, Davis, Neumann, R. N. M. Watson, S. Moore, Fox, Norton, Chisnall, and Fox, “CHERI Concentrate: Practical Compressed Capabilities”.

Thank you :-)