RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Le Temps des Cerises: Efficient stack safety using uninitialized and directed capabilities

Alix Trieu

Based on joint work with A. L. Georges[†], A. Guéneau[🦇], T. van Strydonck[✝], A. Timany[†], D. Devriese[✝], L. Birkedal[†]

[†]Aarhus University, [🦇]Inria, [✝]KU Leuven

March 31, 2023

Disclaimer

The views, opinions and/or findings contained in this presentation are those of the authors and should not be interpreted as representing the official views or policies of the French National Cybersecurity Agency or the French Government.

Stack Safety

- At a low level, functions must manage their own data, e.g., local variables and information on how to return to their callers.
- The call stack is a fundamental data structure used by many programming languages to implement function calls efficiently.
- It is thus the target of many attacks.
- Is it protected by capabilities? Can it be more protected?

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Stack Safety

- In pure-capability mode, the most basic usage of CHERI capabilities is to enforce spatial memory safety, e.g., protect against buffer overflows.

```
void f(void) {
  int ch;
  char buf[512];
  char *p = buf;
  while ((ch = getchar ()) != EOF) {
    *p++ = (char)ch;
  }
  return;
}
```

Stack Safety

- Current software development practices are such that projects may have many dependencies that are difficult to completely audit.
- A gross overapproximation would be to basically consider external library as unknown code.

```
void adv(void);
void f(void) {
  int *x = 1; // Allocated on the stack.
  adv();      // Call some arbitrary code.
  assert (x == 1); // Can we be sure this will not fail?
}
```

Spatial Stack Safety

- The issue is that the stack pointer is not protected and is shared accross all functions.

```
f:                                      # @f
        cincoffset      csp, csp, -64           # reserve stack frame
        csc     cra, 48(csp)                    # save return address
        csc     cs0, 32(csp)                    # save frame pointer
        cincoffset      cs0, csp, 64            # \
        cincoffset      ca0, cs0, -48           # build x in ca1
        csetbounds      ca1, ca0, 16            # /
        csc     ca1, -64(cs0)                   #
        cmove   ca0, cnull                      # ca0 = 0
        cincoffset      ca0, ca0, 1             # ca0 = 1
        csc     ca0, 0(ca1)                     # *x = 1
        ccall   adv
        ...
```

https://cheri-compiler-explorer.cl.cam.ac.uk/z/xqj7WY

Spatial Stack Safety

- The issue is that the stack pointer is not protected and is shared accross all functions.
- This is not unexpected since that's what the ABI mandates.
- But can we define a new calling convention that protects the stack pointer?
- How about not sharing the part of the stack that is used and restore it when we need it, how can we do that?

CInvoke

Format

CInvoke cs1, cs2

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
0x7e	cs2	cs1	0x0	0x1	0x5b

Description

PCC is set equal to capability register *cs1* and unsealed with the 0th bit of its **address** set to 0, whilst **C31** is set equal to capability register *cs2* and unsealed. This provides a constrained form of non-monotonicity, allowing for fast jumps between protection domains, with *cs1* providing the target domain's code and *cs2* providing the target domain's data. The capabilities must have a matching **otype** to ensure the right data is provided for the given jump target.

Sealing

2.3.6 Sealed Capabilities

Capability *sealing* allows capabilities to be marked as *immutable* and *non-dereferenceable*, causing hardware exceptions to be thrown if attempts are made to modify, dereference, or jump to them. This enables capabilities to be used as unforgeable tokens of authority for higher-level software constructs grounded in *encapsulation*, while still allowing them to fit within the pointer-centric framework offered by CHERI capabilities. There are two forms of capability sealing: pairs of capabilities sealed using a common *object type*, and stand-alone *sealed entry capabilities* (sentry capabilities).

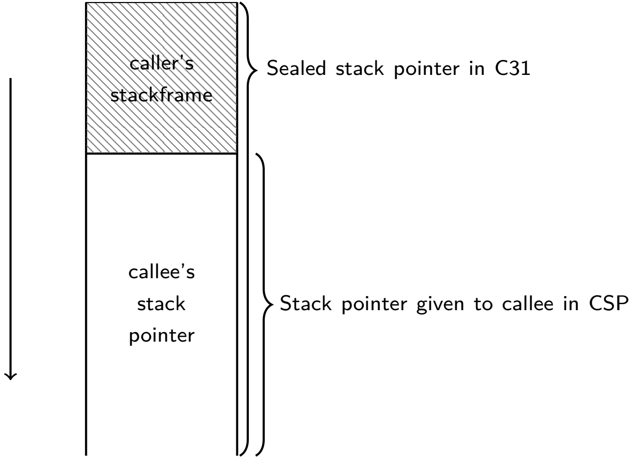
A Calling Convention

A naive idea would be the following.

- Before calling a function:
    - Copy and seal stack pointer.
    - Resize the stack pointer to remove own frame.
    - Clear registers if needed.
    - Provide sealed pair of return pointer and own frame to callee.
    - When callee returns, restore own frame.
- When returning from a call:
    - Clear own stackframe and registers if needed.
    - Use `CInvoke` on sealed pair to return to caller.

The live stack pointer now only gives access to unused parts of the stack.

A Calling Convention



Another example

```c
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv();
  x = 1;
  adv();
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- "High-level" languages have a structured control-flow, and expect function calls to be "well-bracketed".

```
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv();
  x = 1;
  adv();
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- “High-level” languages have a structured control-flow, and expect function calls to be “well-bracketed”.

```
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv(); // <- Normal return.
  x = 1;
  adv();
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- "High-level" languages have a structured control-flow, and expect function calls to be "well-bracketed".

```c
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv();
  x = 1;
  adv(); // <- Stash the return capability away and calls f.
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- "High-level" languages have a structured control-flow, and expect function calls to be "well-bracketed".

```
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv(); // <- x is set to 0, and use the stashed return capability.
  x = 1;
  adv();
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- "High-level" languages have a structured control-flow, and expect function calls to be "well-bracketed".

```
void adv(void);
void f(void) {
  static int x; // Variable persists accross calls.
  x = 0;
  adv();
  x = 1;
  adv(); // Returned here with x set to 0.
  assert (x == 1); // This should not fail.
}
```

Well-Bracketed Control-Flow

- In order to ensure proper local state encapsulation, we need to enforce well-bracketedness of control-flow.
- The issue is that the return pointer can be stashed anywhere.
- Is there a way to restrict that?

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Local Capabilities

Bit	Name	Tag?	Seal?	Bounds?
0	GLOBAL	✓	-	-
1	PERMIT_EXECUTE	✓	Unsealed	Address
2	PERMIT_LOAD	✓	Unsealed	Address
3	PERMIT_STORE	✓	Unsealed	Address
4	PERMIT_LOAD_CAPABILITY	✓	Unsealed	-
5	PERMIT_STORE_CAPABILITY	✓	Unsealed	-
6	PERMIT_STORE_LOCAL_CAPABILITY	✓	Unsealed	-
7	PERMIT_SEAL	✓	Unsealed	Object Type
8	PERMIT_CINVOKE	✓	Sealed	-
9	PERMIT_UNSEAL	✓	Unsealed	Object Type
10	PERMIT_ACCESS_SYSTEM_REGISTERS	✓	Unsealed	-
11	PERMIT_SET_CID	✓	Unsealed	CID

Local Capabilities

The PERMIT_STORE_LOCAL_CAPABILITY permission bit is used to limit capability propagation via software-defined policies: local capabilities (i.e., those without the GLOBAL permission set) can be stored only via capabilities that have PERMIT_STORE_LOCAL_CAPABILITY set. Normally, this permission will be set only on capabilities that, themselves, have the GLOBAL bit cleared. This allows higher-level, software-defined policies, such as "Disallow storing stack references to heap memory" or "Disallow passing local capabilities via cross-domain procedure calls," to be implemented. We anticipate both generalizing and extending this model in the future in order to support more complex policies – e.g., relating to the propagation of garbage-collected pointers, or pointers to volatile vs. non-volatile memory.

Local Capabilities

We follow the CHERI ISA Tech Report's suggestion:

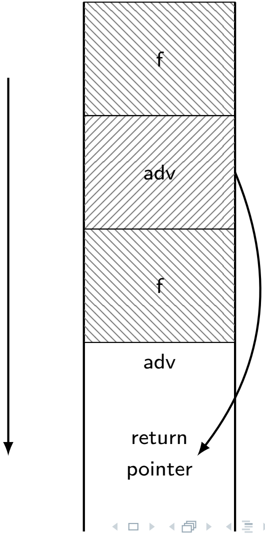
- The stack pointer is made local (GLOBAL bit unset) and is the only capability with the PERMIT STORE LOCAL bit set.
- Thus all stack derived pointers are local and can only be stored on the stack.
- We can make return pointers local and can thus only be stored on the stack!

Is it enough?

Before calling a function:

- Copy and seal stack pointer.
- Resize the stack pointer to remove own frame.
- Clear registers if needed.
- Provide sealed **local** pair of return pointer and own frame to callee.
- When callee returns, restore own frame.

The return pointer can only be kept on the stack now!

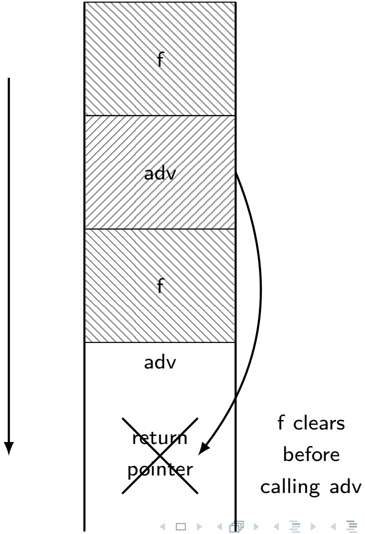


Is it enough?

Before calling a function:

- Copy and seal stack pointer.
- Resize the stack pointer to remove own frame.
- Clear registers if needed.
- **Clear the whole stack.**
- Provide sealed **local** pair of return pointer and own frame to callee.
- When callee returns, restore own frame.

The return pointer can only be kept on the stack now!



RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

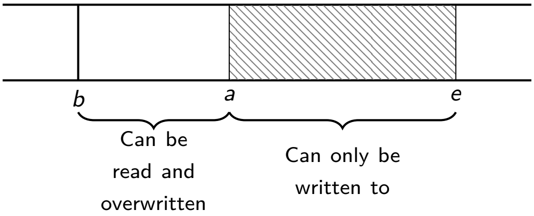
An (Improved) Calling Convention

- Before calling a function:
  - Copy and seal stack pointer.
  - Resize the stack pointer to remove own frame.
  - Clear registers if needed.
  - **Clear the whole stack.**
  - Provide sealed **local** pair of return pointer and own frame to callee.
  - When callee returns, restore own frame.
- When returning from a call:
  - Clear **whole stack** and registers if needed (to prevent caller and callee collaborating).
  - Use `CInvoke` on sealed pair to return to caller.

Clearly very inefficient!

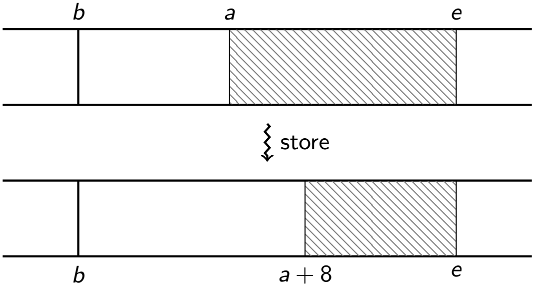
Uninitialized Capabilities

- Clearing the whole stack is clearly an expensive operation.
- We introduce uninitialized capabilities to remedy that.



Unitialized Capabilities

- Moving the cursor can only be accomplished through writing at the cursor boundary.



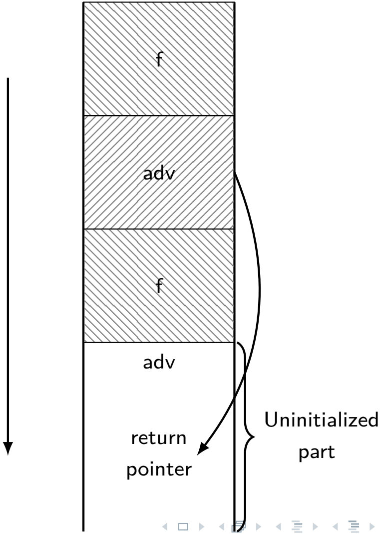
Uninitialized Capabilities

- How can we use uninitialized capabilities to avoid clearing the stack?
- To protect against the callee, one must first clear the part of the stack handed to it.
- We can instead just give an uninitialized stack pointer to the callee, avoiding the need for clearing.

No Clearing!

Before calling a function:

- Copy and seal the local stack pointer.
- Resize the stack pointer to remove own frame.
- Clear registers if needed.
- ~~Clear the whole stack.~~ **Make sure the stack pointer is uninitialized.**
- Provide sealed local pair of return pointer and own frame to callee.
- When callee returns, restore own frame.

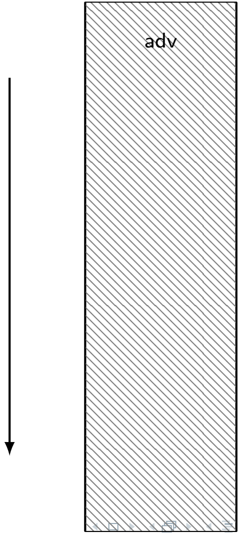


A (More Improved) Calling Convention

- Before calling a function:
  - Copy and seal stack pointer.
  - Resize the stack pointer to remove own frame.
  - Clear registers if needed.
  - ~~Clear the whole stack.~~ **Make sure the stack pointer is uninitialized.**
  - Provide sealed local pair of return pointer and own frame to callee.
  - When callee returns, restore own frame.
- When returning from a call:
  - Clear **whole stack** and registers if needed (to prevent caller and callee collaborating).
  - Use `CInvoke` on sealed pair to return to caller.

Can we avoid the stack clearing when returning?

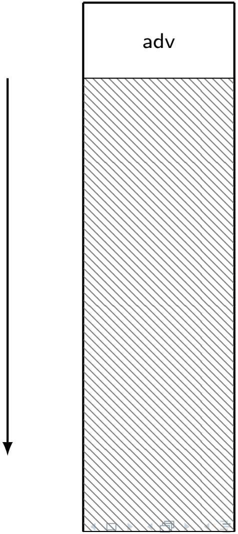
What's the issue?



adv needs to overwrite the stack to reserve its stackframe.

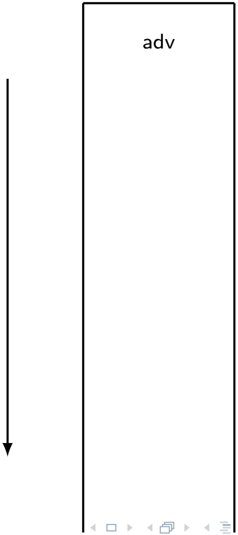
What's the issue?

adv needs to overwrite the stack to reserve its stackframe.



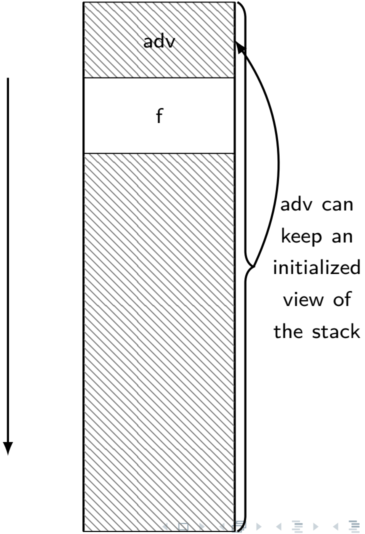
What's the issue?

There is nothing keeping adv from overwriting the whole stack!



What’s the issue?

adv can keep a completely initialized stack pointer and read leftover capabilities on the stack!



Temporal Safety and Dangling Pointers

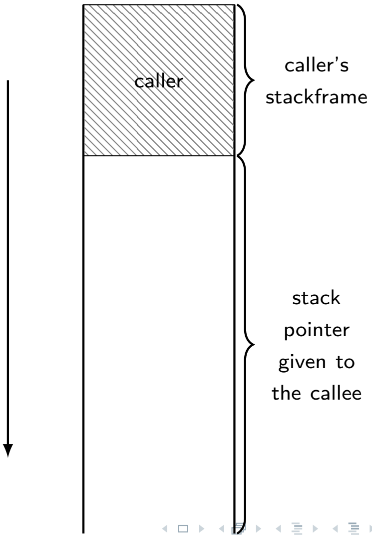
```c
int N, K;
void h(int* x) { *x = 0; }
void g(int* x) {
  char* t[K];
  h(x); }
void f(int** x) {
  char* t[N];         // Example illustrating
  int z;              // use after reallocate
  *x = &z; }          // issue
int main(void) {
  int* x;
  f(&x);
  g(x);
  return 0; }
```

Temporal Safety and Dangling Pointers

- Functions should not be able to read leftover data on the stack.
- Functions should not be able to pass up capabilities that are becoming stale.
- How can we prevent this?

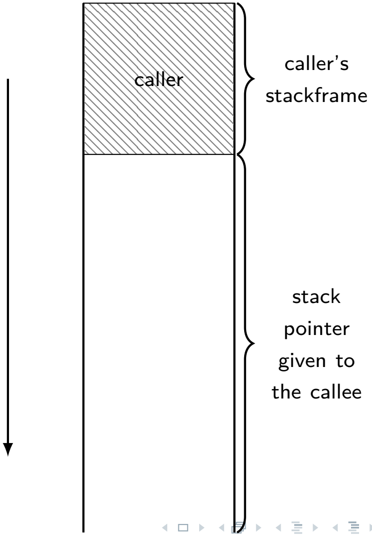
How the stack evolves

- The stack evolves in a specific way and maybe we can take advantage of this.



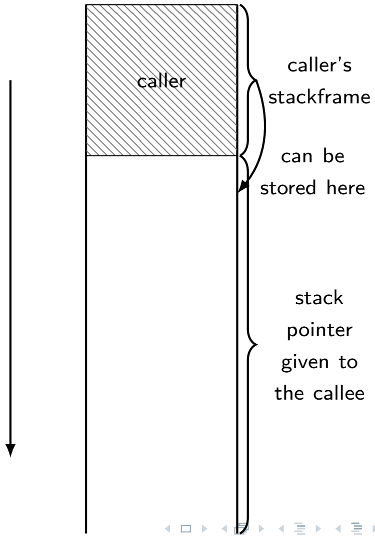
How the stack evolves

- The stack evolves in a specific way and maybe we can take advantage of this.
- We want to prevent the caller to be able to read what's left on the stack by the callee.
- The unsealed stackframe capability should not have read authority over the area of the stack given to the callee.
- This stackframe capability is passed to the callee, so it doesn't need to be able to be kept within its own read authority.



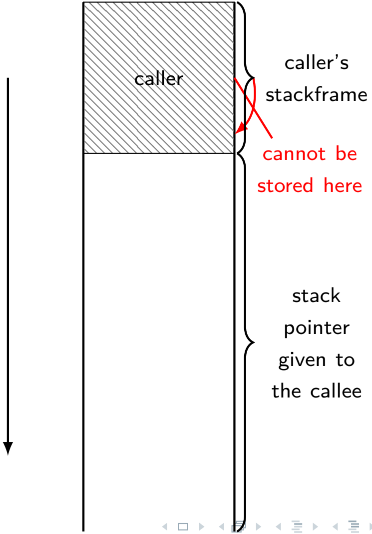
Restricting where capabilities can be stored

- What if we restricted where the sealed stackframe capability can be stored?
- What if it could only be stored outside of its read authority?
- Since a capability range is contiguous, this would allow the callee to ensure that its caller does not have read authority over the given part of the stack.



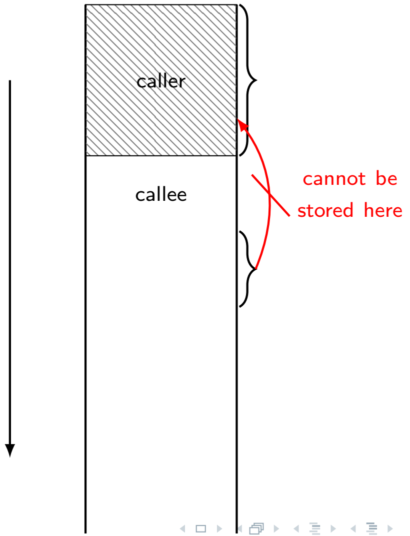
Restricting where capabilities can be stored

- What if we restricted where the sealed stackframe capability can be stored?
- What if it could only be stored outside of its read authority?
- Since a capability range is contiguous, this would allow the callee to ensure that its caller does not have read authority over the given part of the stack.



Restricting where capabilities can be stored

- We also want to prevent a callee to pass up a stack derived capability that is going to be stale at the end of the call.



Restricting where capabilities can be stored

This gives us some constraints:

- Stack derived capabilities should not be stored where they have read authority.
- Stack derived capabilities should not be passed up.
- Therefore, stack derived capabilities can only be stored down.

Directed Capabilities

- We propose directed capabilities such that a directed capability c can only be stored at some address a such that $\text{readUpTo}(c) \leq a$.
- $\text{readUpTo}(UR\_, b, e, a) = \min(e, a)$
- $\text{readUpTo}(R\_, b, e, a) = e$

- Simple check similar to existing ones.

A Final Calling Convention

Parameters are now passed on the stack.

- Before calling a function:
  - Copy and seal **the directed** stack pointer.
  - Resize the stack pointer to remove own frame.
  - Clear registers if needed.
  - **Make sure the stack pointer is uninitialized.**
  - Provide sealed local pair of return pointer and own frame to callee by **storing it on the stack**.
  - When callee returns, restore own frame.
- When returning from a call:
  - Clear the registers if needed.
  - Use `CInvoke` on sealed pair to return to caller.

Conclusion

- We proposed a new calling convention for ensuring spatial and temporal stack safety.
- We prove (formally) in the following publications that the properties are indeed properly enforced for an idealized core capability machine.
  - **Efficient and Provable Local Capability Revocation using Uninitialized Capabilities.**
    Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, Lars Birkedal.
    48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), 2021.
  - **Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities.**
    Aïna Linn Georges, Alix Trieu, Lars Birkedal.
    Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2022.

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Future Work

- This is all theoretical currently, implement and test whether this calling convention is practical.
- Is it compatible with CHERIoT? Are there enough free bits in their compression scheme?
- Heap safety: Can we revoke shared heap capabilities?
  - Make heap capabilities local before sharing: capabilities loaded using these capabilities may be global.
  - Need some sort of recursive “load-global” permission as proposed in CHERIoT.