Toward Complete Stack Safety for Capability Machines

March 16, 2021

Alix Trieu

Aarhus University

Based on joint work with A. L. Georges, A. Guéneau, T. van Strydonck, A. Timany, D. Devriese, L. Birkedal

Journée “Méthodes Formelles” 2021 du GdR Sécurité Informatique

Secure Compilation

- Good programming languages provide helpful abstractions for writing more secure code: such as structured control-flow, modules, interfaces, etc.
- However, those abstractions are not enforced when interacting with adversarial low-level code: all source level guarantees are lost.
    - Internet browsers execute arbitrary Javascript code.
    - Software development nowadays depend on pulling libraries and their dependencies, they may be buggy or even compromised.
    - Machines in the cloud are shared.

Secure Compilation

- We need secure compilation: compilers that protect source-level abstractions even in presence of linked low-level machine code.
- This allows security reasoning at the source level which is arguably easier than at the machine code level.
  - Low level linked code cannot break the security of the compiled program any more than source level linked code.
- This seems difficult to achieve without help from the hardware, how do we enforce this ?

Outline of the talk

Capability machines are a kind of CPUs with fine-grained management of memory, I will show how we can leverage them to enforce a hierarchy of stack safety properties.

1. What are capability machines ?
2. How can we prove that some simple properties are enforced no matter the context ?
3. How can we enforce WBCF and LSE on capability machines ?
4. Can we go further and achieve complete stack safety ?

Capability Machines

Capability machines are a kind of CPUs with fine-grained management of memory.

- Fairly old idea, starting in the ~ 1960s.
- CHERI is a recent implementation (started ~ 2010) developed at the University of Cambridge and SRI.
- Recent commitment from Arm to develop an experimental CHERI-extended processor[1], and interest from Microsoft[2].

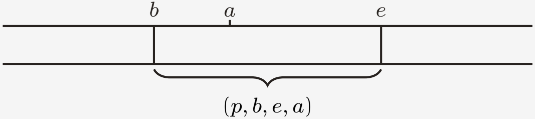
[1] https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html

[2] https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/

Capability Machines

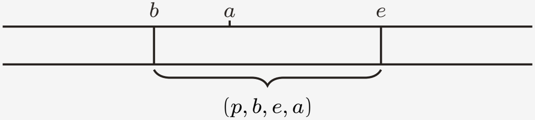
Capability machines are a kind of CPUs with fine-grained management of memory that support two kinds of machine words:

- Regular machine integers (e.g., 64 bits integers).
- Capabilities: unforgeable tokens of authority over a range of memory.



p represents a permission, e.g., RX, RWX, RO, O, etc.
$[b, e[$ are the bounds of the capability.
a is the current address the capability points to.

Capabilities



Memory operations dynamically check at runtime that

- the access is within bounds;
- the permission is sufficient.

For instance, `store rdst rsrc` checks that

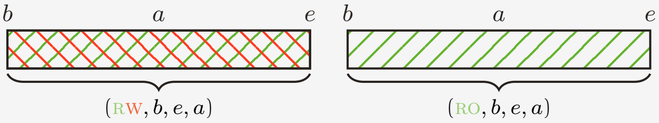
- `rdst` contains a capability (p, b, e, a);
- p has write permission;
- a is within $[b, e[$.

Capabilities

There are some special instructions for manipulating capabilities:

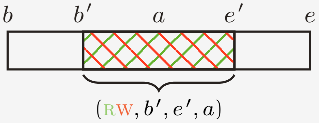
- `restrict` to decrease the permission of a capability.

  `restrict` (RW, b, e, a) RO



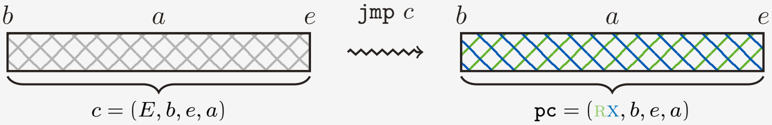
- `subseg` to decrease the range of authority of a capability.

  `subseg` $(\mathrm{RW}, b, e, a)\ b'\ e'$



Enter Capabilities

Capability machines provide a mean to “encapsulate” data and code (code is data!) through enter capabilities.

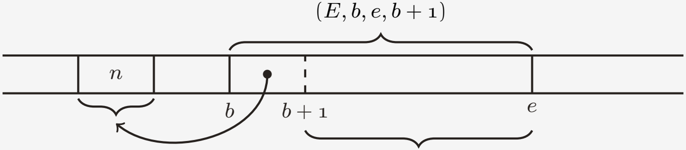


Enter capabilities (called sealed entry capabilities in CHERI) are completely opaque and cannot be modified in any way, except being copied and jumped to.

Local State Encapsulation

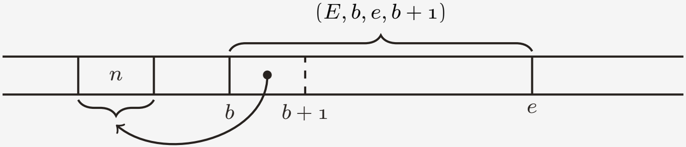
We can use enter capabilities to enforce state encapsulation.
For instance, we can implement closures with them:

```
let n = ref 0 in
(fun () -> n := !n + 1; !n)
```



```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
```

Reasoning on a Capability Machine



We want to prove that $n \geq 0$ no matter what the context is.

How do we even state this property ?

Reasoning on a Capability Machine

Our methodology is:

- A program logic to describe such a specification and step through the known part of a program.
- A logical relation to define a notion of “capability safety” and give a specification to unknown code.

A Core Capability Machine

We consider a capability machine with registers and finite memory.

$$\begin{aligned}
\mathsf{RegName} &::= \mathsf{PC} \mid r_0 \mid \cdots \mid r_{31} \\
\mathsf{Addr} &::= [0, \mathsf{AddrMax}[\\
\mathsf{Word} &::= n \mid (p, b, e, a) \\
\mathsf{Reg} &::= \mathsf{RegName} \rightarrow \mathsf{Word} \\
\mathsf{Mem} &::= \mathsf{Addr} \rightarrow \mathsf{Word}
\end{aligned}$$

A state of the machine is just a pair of registers and memory state.

Syntax

We consider the following instructions.

$$\begin{array}{rcl}
\rho & \in & \mathbb{Z} + \mathsf{RegName} \\
i & ::= & \mathsf{jmp}\ r \mid \mathsf{jnz}\ r\ r \mid \mathsf{move}\ r\ \rho \mid \\
 & & \mathsf{load}\ r\ r \mid \mathsf{store}\ r\ \rho \mid \mathsf{add}\ r\ \rho\ \rho \mid \mathsf{sub}\ r\ \rho\ \rho \mid \\
 & & \mathsf{lt}\ r\ \rho\ \rho \mid \mathsf{lea}\ r\ \rho \mid \mathsf{restrict}\ r\ \rho \mid \\
 & & \mathsf{subseg}\ r\ \rho\ \rho \mid \mathsf{isptr}\ r\ r \mid \mathsf{getP}\ r\ r \mid \\
 & & \mathsf{getB}\ r\ r \mid \mathsf{getE}\ r\ r \mid \mathsf{getA}\ r\ r \mid \mathsf{fail} \mid \mathsf{halt}
\end{array}$$

Informal Semantics

The informal semantics of the machine is to

- Check that PC contains a capability (p, b, e, a) such that p has execute permission and $b \leq a < e$;
- Load word w stored at address a, and decode it into an instruction i and execute it (capabilities cannot be decoded into instructions).

$$\frac{\varphi.\mathsf{pc} = (p, b, e, a) \qquad \mathsf{executable}(p) \qquad a \in [b, e[\qquad \varphi.\mathsf{mem}(a) = w \qquad \mathsf{decode}(w) = \mathsf{instr}}{\varphi \rightarrow [\![\mathsf{instr}]\!](\varphi)}$$

Program Logic for Capability Machines

We use a program logic based on separation logic. We have points-to resources for registers and memory.

Registers	$r \Mapsto w$
Memory	$a \mapsto w$

The Hoare triples are of the following form:

$$
\begin{aligned}
&\mathsf{decode}(w) = i \rightarrow \\
&\mathsf{ValidPC}(p, b, e, a) \rightarrow \\[1em]
&\{\, \mathsf{PC} \Mapsto (p, b, e, a) * a \mapsto w * \cdots \,\} \\
&\mathsf{SingleStep} \\
&\{\, \mathsf{PC} \Mapsto (p, b, e, a + 1) * a \mapsto w * \cdots \,\}
\end{aligned}
$$

Program Logic for Capability Machines

Executing a sequence of instructions

$$\mathsf{map\ decode}\ l = \mathit{prog} \rightarrow$$

$$\mathsf{ValidPCRange}(p, b, e, -)(a_1, a_n) \rightarrow$$

$$\{\ \mathsf{PC} \Mapsto (p, b, e, a_1) * [a_1 - a_n] \mapsto l * \cdots\ \}$$

$$\mathsf{Repeat\ SingleStep}$$

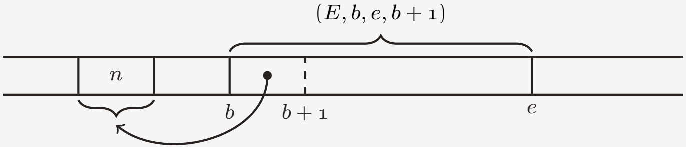
$$\{\ \mathsf{PC} \Mapsto (p, b, e, a_n) * [a_1 - a_n] \mapsto l * \cdots\ \}$$

Program Logic for Capability Machines

Executing a macro, or a sequence of instructions within a program

$$\begin{array}{l}
\quad \mathsf{map\ decode}\ l = \mathit{prog} \rightarrow \\
\quad \mathsf{ValidPCRange}(p, b, e, -)(a_1, a_n) \rightarrow \\[1ex]
\{\ \mathsf{PC} \Mapsto (p, b, e, a_1) * [a_1 - a_n] \mapsto l\ \ * \cdots * \\
\quad \triangleright\, (\mathsf{PC} \Mapsto (p, b, e, a_n) * [a_1 - a_n] \mapsto l * \cdots \twoheadrightarrow \Phi)\ \} \\
\quad \mathsf{Repeat\ SingleStep} \\
\{\ \Phi\ \}
\end{array}$$

Functional Specification for the Counter Example



map decode $l = \mathit{counter\_instrs} \to$

ValidPCRange$(\textsc{rx}, b, e, -)(b+1, e) \to$

$\{$ PC $\Mapsto (\textsc{rx}, b, e, b+1) * [b+1-e] \mapsto l \; * b \mapsto n * r_{ret} \Mapsto c * r_1 \Mapsto w_1 * \cdots *$

$\triangleright$ (PC $\Mapsto c * [b+1-e] \mapsto l \; * b \mapsto (n+1) * r_{ret} \Mapsto c * r_1 \Mapsto (n+1) * \cdots \mathrel{-\!\!*} \Phi) \}$

Repeat SingleStep

$\{ \Phi \}$

Logical Relation

The logical relation defines a contract that capability machine programs must follow. We use this contract as the interface between known secure code, and unknown arbitrary code, when reasoning about the full program.

The logical relation defines what it means to be "capability safe", or more intuitively, what is safe to share to the adversary.

Logical Relation

- Expression relation
  - The execution does not get stuck: validity of the registers is sufficient for executing the program.
  - All registered invariants hold at every step of execution.
- Value relation

$$\boxed{\mathcal{E}(w)} \quad \triangleq \forall \mathsf{reg},\ \left\{\mathsf{PC} \Mapsto w * \mathop{\Large\ast}_{(r,v)\in\mathsf{reg}, r\neq\mathsf{PC}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z) & \triangleq \textsc{True} \\ \mathcal{V}(\mathrm{E}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\mathrm{RX}, b, e, a) \\ \mathcal{V}(\mathrm{RO}/\mathrm{RX}, b, e, -) & \triangleq \mathop{\Large\ast}_{a\in[b,e[} \exists P, \boxed{\exists w,\ a \mapsto w * P(w)} * \\ & \qquad \triangleright \Box \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w) \\ \mathcal{V}(\mathrm{RW}/\mathrm{RWX}, b, e, -) & \triangleq \mathop{\Large\ast}_{a\in[b,e[} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \end{cases}$$

Logical Relation

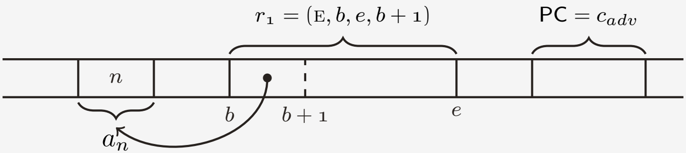
$$\boxed{\mathcal{E}(w)} \quad \triangleq \forall \mathsf{reg},\ \left\{\mathsf{PC} \Mapsto w * \mathop{\LARGE\ast}_{(r,v)\in\mathsf{reg}, r\neq\mathsf{PC}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \quad \begin{cases} \mathcal{V}(z) & \triangleq \textsc{True} \\ \mathcal{V}(\textsc{e}, b, e, a) & \triangleq \triangleright \Box\, \mathcal{E}(\textsc{rx}, b, e, a) \\ \mathcal{V}(\textsc{ro}/\textsc{rx}, b, e, -) & \triangleq \mathop{\LARGE\ast}_{a\in[b,e[} \exists P, \boxed{\exists w,\ a \mapsto w * P(w)} * \\ & \qquad\qquad \triangleright \Box\, \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w) \\ \mathcal{V}(\textsc{rw}/\textsc{rwx}, b, e, -) & \triangleq \mathop{\LARGE\ast}_{a\in[b,e[} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \end{cases}$$

Theorem (FTLR)

Let $p \in \mathsf{Perm}, b, e, a \in \mathsf{Addr}$. *If* $\mathcal{V}(p, b, e, a)$, *then* $\mathcal{E}(p, b, e, a)$.

Back to the example



We want to show that no matter the adversary code, the value n stored at address a_n is such that $n \geq 0$.

We can show the following spec.

$$\boxed{\exists n, a_n \mapsto n * n \geq 0}$$

$$\vdash \left\{ (\mathrm{RX}, b, e, b+1);\ \begin{array}{l} \mathop{\LARGE\ast}_{(r,v)\in \mathsf{reg}, r\notin\{\mathsf{PC}, r_{ret}\}} r \Mapsto v \\ * \; r_{ret} \Mapsto w_{ret} * \mathcal{V}(w_{ret}) \\ * \; [b+1, e) \mapsto instrs \end{array} \right\} \rightsquigarrow \bullet$$

Back to the example

$$\boxed{\exists n, a_n \mapsto n * n \geq 0}$$

$$\vdash \left\{ \begin{array}{ll} & \mathop{\LARGE *}_{(r,v)\in \mathsf{reg}, r \notin \{\mathsf{PC}, r_{ret}\}} r \Mapsto v \\ (\mathrm{RX}, b, e, b+1); & * \ r_{ret} \Mapsto w_{ret} * \mathcal{V}(w_{ret}) \\ & * \ [b+1, e) \mapsto instrs \end{array} \right\} \rightsquigarrow \bullet$$

```
mov r2 pc
lea r2 (-1)
load r2 r2
load r1 r2
add r1 r1 1
store r2 r1
mov r2 0
jmp rret
```

Using the adequacy of the program logic, we can then show that for an initial state (reg, m) where

- $reg(\mathsf{PC}) = c_{adv}$, $reg(r_0) = (\mathrm{E}, b, e, b+1)$ and $reg(r) \in \mathbb{Z}$ otherwise
- m has been initialized with the code of the program and unknown adversarial code (pointed by c_{adv})
- $m(a_n) = n_0$ and $n_0 \geq 0$

Then for all (reg', m') such that $(reg, m) \rightarrow^* (reg', m')$ then $m'(a_n) \geq 0$.

WBCF and LSE

Consider the following code known as the “awkward example”:

```
void adv(void);        // OCaml equivalent
void f(void) {         // let x = ref 0 in
  static int x = 0;    // fun adv ->
  x = 0;               //    x := 0;
  adv();               //    adv ();
  x = 1;               //    x := 1;
  adv();               //    adv ();
  assert (x == 1);     //    assert (!x = 1)
}
```

We need revocation !

Local Capabilities

Recent capability machines such as CHERI provide so-called local capabilities, capabilities that can only be stored in a restricted way.
Concretely, capabalities have now a locality field ℓ:

$$(p, \ell, b, e, a)$$

The set of permissions is enriched with write local permissions: RWLX, RWL.

WBCF and LSE

We can combine local capabilities and enter capabilities to enforce well-bracketed control flow and local state encapsulation. The basic idea is to

- Make the stack capability RWLX and local.
- Make return capabilities local.

In this way, we know that any capability pointing to the stack must necessarily be stored on the stack.

A Secure Calling Convention

When calling a function, the caller must:

- clear the part of the stack capability it intends to pass to the callee;
- pass a local enter return capability.

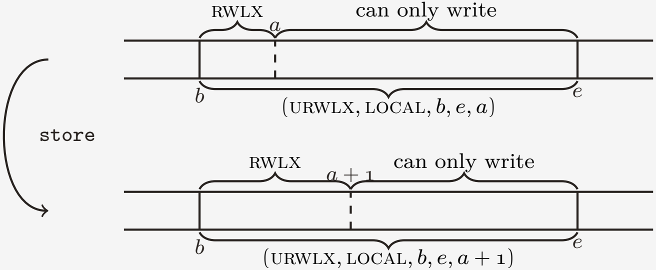
When returning, the callee must clear its own stackframe.

We have shown that using this calling convention, the assertion in the awkward example *cannot be violated.*

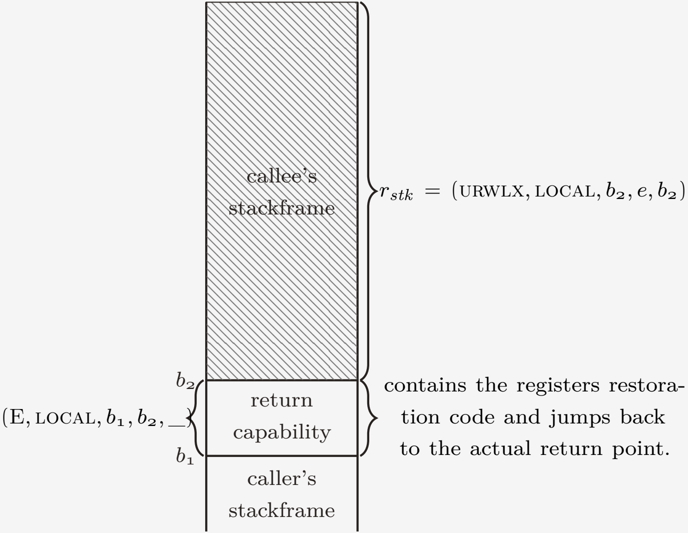
Uninitialized Capabilities

The presented calling convention is quite inefficient, requiring to clear a large amount of memory before each call.

We solve this issue by introducing uninitialized capabilities, a new form of capabilities that is plausibly implementable.


RWLX
can only write
a
b
e
$(\text{URWLX}, \text{LOCAL}, b, e, a)$
store
RWLX
can only write
$a+1$
b
e
$(\text{URWLX}, \text{LOCAL}, b, e, a+1)$

Secure Calling Convention Using Uninitialized Capabilities



callee's stackframe
$r_{stk} = (\textsc{urwlx}, \textsc{local}, b_2, e, b_2)$
b_2
return capability
$(\mathrm{E}, \textsc{local}, b_1, b_2, \_)$
b_1
caller's stackframe

contains the registers restoration code and jumps back to the actual return point.

Toward Complete Stack Safety

The calling convention enforces WBCF and LSE efficiently, what are we missing?

```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; }
```

Use After Free

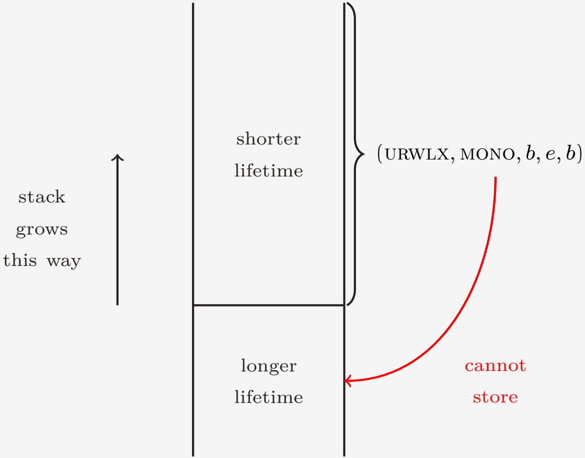
```
int compare(char* x, char* y)
int compare_secret(char* in) {
  static char[] secret = ...;
  int x = compare(secret, in);
  return x; }
```

Even by passing a local capability to a callee, we need to clear the whole stack in order to make sure that confidential data are not leftover on the stack, and can be subsequently read by others.

What’s the issue?

- Pointers/capabilities are used outside of their lifetime.
- We remark that the stack evolve in a specific way, i.e., it grows in one direction.
- We can exploit this: the address on the stack represents implicitly the lifetime of its pointer.

Monotone Capabilities



Monotone capabilities enforce that capabilities with shorter lifetime cannot be stored using capabilities with longer lifetime.

Secure Calling Convention

We can now use instead uninitialized monotone capabilities as stack capability, and we don't need to clear anything anymore!

Dead store elimination is secure for the stack!

How do we prove it?

Relational Model

Before, we only showed that an adversary could not overwrite some memory.

Now, we need to show it cannot read it!.

This asks for a relational model, we need to show the following programs are (morally) contextually equivalent for instance.

```
int f(void)
int secret = ...;
...;
return;
```

```
int f(void)
int secret = ...;
...;
secret = 0;
return;
```

Conclusion

- There is a “hierarchy” of stack safety properties from simple encapsulation, followed by well-bracketed control-flow, to finally include temporal properties, and these can be enforced using a capability machine.
- I have sketched how one can prove that “simple” encapsulation is enforced in presence of arbitrary code using a program logic and logical relations. This approach can scale to more sophisticated properties.
- This forms a foundation on which a compiler can be built, and transfer source-level guarantees down to compiled code.

References

- **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.
  https://cs.au.dk/~trieu/publications/POPL21.pdf
- **Cap' ou pas cap' ? Preuve de programmes pour une machine à capacités en présence de code inconnu.**
  Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, Lars Birkedal.
  Trente-deuxièmes Journées Francophones des Langages Applicatifs (JFLA), 2021.
  https://cs.au.dk/~trieu/publications/JFLA21.pdf
- **Toward Complete Stack Safety for Capability Machines.**
  Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Lars Birkedal.
  5th Workshop on Principles of Secure Compilation (PriSC), 2021.
  https://cs.au.dk/~trieu/publications/PRISC21.pdf