Enforcing Well-Bracketed Control Flow on a Capability Machine using Local Capabilities

Lau Skorstengaard\textsuperscript{1} Dominique Devriese\textsuperscript{2} Lars Birkedal\textsuperscript{1}

\textsuperscript{1}Aarhus University
\textsuperscript{2}imec-DistriNet, KU Leuven

SCM, January 15, 2017
Why capability machines?

- Interesting compilation target
  - C-like calling convention
  - Enforcement of well-bracketed calls
- Subject of systems research
  - CHERI
Talk outline

A simple capability machine

Applications
   Enforcing well-bracketedness

Semantic model
   Kripke worlds
   Logical relation
   Interesting properties

Conclusion
Road map

A simple capability machine

Applications
Enforcing well-bracketedness

Semantic model
Kripke worlds
Logical relation
Interesting properties

Conclusion
Memory capabilities

Challenge:

- Low-level machines provide no means to enforce fine-grained access control.
Memory capabilities

Challenge:
- Low-level machines provide no means to enforce fine-grained access control.

Solution:
- Assembly language that uses capabilities instead of pointers

Capabilities (perm, base, end, a)
- Permission e.g. read (r), write (w), execute (x)
- Range of authority
- Pointer
- Capability aware instructions enforce capability permissions
Memory capabilities

Challenge:
▶ Low-level machines provide no means to enforce fine-grained access control.

Solution:
▶ Assembly language that uses capabilities instead of pointers
▶ Tagged memory
Memory capabilities

Challenge:

▶ Low-level machines provide no means to enforce fine-grained access control.

Solution:

▶ Assembly language that uses capabilities instead of pointers
▶ Tagged memory
▶ Capabilities \((perm, base, end, a)\)
  ▶ Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  ▶ Range of authority
  ▶ Pointer
Memory capabilities

Challenge:
- Low-level machines provide no means to enforce fine-grained access control.

Solution:
- Assembly language that uses capabilities instead of pointers
- Tagged memory
- Capabilities \((perm, base, end, a)\)
  - Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  - Range of authority
  - Pointer
- Capability aware instructions enforce capability permissions

\[
\text{[[store } r_1 \text{ } r_2 \text{]}(\Phi) =
\]
Memory capabilities

Challenge:
- Low-level machines provide no means to enforce fine-grained access control.

Solution:
- Assembly language that uses capabilities instead of pointers
- Tagged memory
- Capabilities \((perm, base, end, a)\)
  - Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  - Range of authority
  - Pointer
- Capability aware instructions enforce capability permissions

\[
w = \Phi.\text{reg}(r_2)
\]

\[
\text{[store } r_1 \ r_2\text{]}(\Phi) = \Phi[\text{mem.? } \mapsto w]
\]
Memory capabilities

Challenge:
- Low-level machines provide no means to enforce fine-grained access control.

Solution:
- Assembly language that uses capabilities instead of pointers
- Tagged memory
- Capabilities \((perm, base, end, a)\)
  - Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  - Range of authority
  - Pointer
- Capability aware instructions enforce capability permissions

\[
w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = (perm, base, end, a)
\]

\[
[\text{store } r_1 \ r_2](\Phi) = \Phi[\text{mem}.a \rightarrow w]
\]
Memory capabilities

Challenge:
- Low-level machines provide no means to enforce fine-grained access control.

Solution:
- Assembly language that uses capabilities instead of pointers
- Tagged memory
- Capabilities \((perm, base, end, a)\)
  - Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  - Range of authority
  - Pointer
- Capability aware instructions enforce capability permissions
  \[
  w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = (perm, base, end, a) \\
  \text{perm} \in \{rw, rwx\} \\
  \left[\text{store } r_1 \ r_2\right](\Phi) = \Phi[\text{mem} \cdot a \mapsto w]
  \]
Memory capabilities

Challenge:

- Low-level machines provide no means to enforce fine-grained access control.

Solution:

- Assembly language that uses capabilities instead of pointers
- Tagged memory
- Capabilities \((perm, base, end, a)\)
  - Permission e.g. read \((r)\), write \((w)\), execute \((x)\)
  - Range of authority
  - Pointer
- Capability aware instructions enforce capability permissions

\[
\begin{align*}
  w &= \Phi.\text{reg}(r_2) \\
  \Phi.\text{reg}(r_1) &= (perm, base, end, a) \\
  perm &\in \{rw, rwx\} \quad base \leq a \leq end \\
  [\text{store } r_1 \ r_2](\Phi) &= \Phi[\text{mem}.a \mapsto w]
\end{align*}
\]
Enter capabilities

Challenge:

- Execute capabilities provide no encapsulation: how can we give the callee more authority than caller?
Enter capabilities

Challenge:
- Execute capabilities provide no encapsulation: how can we give the callee more authority than caller?

Solution (from M-Machine):
- Enter capability:
  - Completely opaque, you can only jump to it
  - Becomes $rx$ when jumped to
- $\sim$ encapsulated closure
- Security boundaries
- Modularisation
Local capabilities

Challenge:

- Capabilities are irrevocable.
Local capabilities

Challenge:
- Capabilities are irrevocable.

Solution (from CHERI):
- Local capabilities (form of temporal information-flow control)
- Capabilities extended with a *local* tag and a *permit write local* permission (*wl*)
- Local capabilities can only be written to memory through a *wl* capability
- (to make it useful:) *wl*-capabilities must be local themselves
Local capabilities

Challenge:

- Capabilities are irrevocable.

Solution (from CHERI):

- Local capabilities (form of temporal information-flow control)
- Capabilities extended with a local tag and a permit write local permission (wl)
- Local capabilities can only be written to memory through a wl capability
- (to make it useful:) wl-capabilities must be local themselves

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]

\[ \text{perm} \in \{\text{rw}, \text{rwx}, \} \]

\[ \text{base} \leq a \leq \text{end} \]

\[ [\text{store } r_1 r_2](\Phi) = \Phi[\text{mem.a} \mapsto w] \]
Local capabilities

Challenge:

- Capabilities are irrevocable.

Solution (from CHERI):

- Local capabilities (form of temporal information-flow control)
- Capabilities extended with a local tag and a permit write local permission (wl)
- Local capabilities can only be written to memory through a wl capability
- (to make it useful:) wl-capabilities must be local themselves

\[
w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a)
\]

\[
\text{perm} \in \{\text{rw, rwx, rwl, rwxl}\}
\]

\[
\text{base} \leq a \leq \text{end}
\]

\[
\left[\text{store } r_1 \ r_2\right](\Phi) = \Phi[\text{mem.a} \mapsto w]
\]
Local capabilities

Challenge:

▶ Capabilities are irrevocable.

Solution (from CHERI):

▶ Local capabilities (form of temporal information-flow control)
▶ Capabilities extended with a local tag and a permit write local permission (wl)
▶ Local capabilities can only be written to memory through a wl capability
▶ (to make it useful:) wl-capabilities must be local themselves

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm, g}), \text{base, end, a}) \]

\[ \text{perm} \in \{\text{rw, rwx, rwl, rwlx}\} \]

\[ \text{base} \leq a \leq \text{end} \quad w = (((-, \text{local}), -, -, -)) \]

\[ \left[\text{store } r_1 \ r_2\right](\Phi) = \Phi[\text{mem.a} \mapsto w] \]
Local capabilities

Challenge:
▶ Capabilities are irrevocable.

Solution (from CHERI):
▶ Local capabilities (form of temporal information-flow control)
▶ Capabilities extended with a local tag and a permit write local permission (wl)
▶ Local capabilities can only be written to memory through a wl capability
▶ (to make it useful:) wl-capabilities must be local themselves

\[ w = \Phi.\text{reg}(r_2) \quad \Phi.\text{reg}(r_1) = ((\text{perm}, g), \text{base}, \text{end}, a) \]
\[ \text{perm} \in \{\text{rw, rwx, rwl, rwlx}\} \]
\[ \text{base} \leq a \leq \text{end} \quad w = (((-, \text{local}), -, -, -) \Rightarrow \text{perm} \in \{\text{rwl, rwlx}\}) \]

\[ [\text{store } r_1 \ r_2](\Phi) = \Phi[\text{mem} \cdot a \mapsto w] \]
Road map

A simple capability machine

Applications

  Enforcing well-bracketedness

Semantic model

  Kripke worlds
  Logical relation
  Interesting properties

Conclusion
Enforcing well-bracketedness (without a trusted stack)

Basic idea:
- Return pointer as local enter-capability
- Stack pointer as local `rwlx`-capability
- Only place one can store local capabilities

Many details to get right:
- Clear non-argument registers before jumps to untrusted code
- Clear part of the stack the callee gains control over
- Adversary callbacks must be global

Results:
- Provably enforce well-bracketed control flow and local state encapsulation, without a trusted stack!
- (Even with a trusted stack, some points above still needed.)
Enforcing well-bracketedness (without a trusted stack)

Basic idea:

- Return pointer as local enter-capability
- Stack pointer as local \texttt{rwx}-capability
  - Only place one can store local capabilities

Many details to get right:

- Clear non-argument registers before jumps to untrusted code
- Clear part of the stack the callee gains control over
- Adversary callbacks must be global

Results:

- Provably enforce well-bracketed control flow and local state encapsulation, without a trusted stack!
- (Even with a trusted stack, some points above still needed.)
Enforcing well-bracketedness (without a trusted stack)

Basic idea:
- Return pointer as local enter-capability
- Stack pointer as local rwx-capability
  - Only place one can store local capabilities

Many details to get right:
- Clear non-argument registers before jumps to untrusted code
- Clear part of the stack the callee gains control over
- Adversary callbacks must be global

Results:
- Provably enforce well-bracketed control flow and local state encapsulation, without a trusted stack!
- (Even with a trusted stack, some points above still needed.)
Enforcing well-bracketedness (without a trusted stack)

Basic idea:
- Return pointer as local enter-capability
- Stack pointer as local rwx-capability
  - Only place one can store local capabilities

Many details to get right:
- Clear non-argument registers before jumps to untrusted code
- Clear part of the stack the callee gains control over
- Adversary callbacks must be global

Results:
- Provably enforce well-bracketed control flow and local state encapsulation, without a trusted stack!
- (Even with a trusted stack, some points above still needed.)
Road map

A simple capability machine

Applications

  Enforcing well-bracketedness

Semantic model

  Kripke worlds
  Logical relation
  Interesting properties

Conclusion
Kripke worlds

- Recursive Kripke world
  - Collection of regions
- Regions model evolvable invariants (protocols) on memory
  - State machines with public and private transitions
Kripke worlds

- Recursive Kripke world
  - Collection of regions
- Regions model evolvable invariants (protocols) on memory
  - State machines with public and private transitions
Kripke worlds

- Recursive Kripke world
  - Collection of regions
- Regions model evolvable invariants (protocols) on memory
  - State machines with public and private transitions
- A future world is an extension of a world
Kripke worlds

Challenge: we want to reuse memory e.g., on the stack
Kripke worlds

Permanent:

Temporary:

Challenge: we want to reuse memory e.g., on the stack

- *Permanent* regions, remain present in any future world
- *Temporary* regions, may be revoked in private future worlds
Kripke worlds

Permanent:

Temporary:

Challenge: we want to reuse memory e.g., on the stack

- *Permanent* regions, remain present in any future world
- *Temporary* regions, may be revoked in private future worlds
- *Revoked* regions
Kripke worlds

Permanent:
Temporary:
Revoked:

Challenge: we want to reuse memory e.g., on the stack

- *Permanent* regions, remain present in any future world
- *Temporary* regions, may be revoked in private future worlds
- *Revoked* regions
Kripke worlds

Permanent:

Temporary:

Revoked:

Relation to local capabilities
Kripke worlds

Relation to local capabilities

- Local capabilities
  - Can depend on temporary and permanent regions
Kripke worlds

- Permanent:
- Temporary:
- Revoked:
- Local capability:
- Global capability:

Relation to local capabilities
- Local capabilities
  - Can depend on temporary and permanent regions
- Global capabilities
  - Can only depend on permanent regions
Kripke worlds

Permanent:

Temporary:

Revoked:

Local capability:

Global capability:

Relation to local capabilities

- Local capabilities
  - Can depend on temporary and permanent regions

- Global capabilities
  - Can only depend on permanent regions
Logical relation

- Semantic model of well-behaved programs
- Captures the safe behaviour of the system
  - e.g., no global permit-write-local capabilities
Logical relation

\[ \mathcal{V}(W) \stackrel{\text{def}}{=} \{(n, i) \mid i \in \mathbb{Z}\} \]
\[ \cup \left\{ (n, ((r, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \right\} \]
\[ \cup \ldots \]

- Semantic model of well-behaved programs
- Captures the safe behaviour of the system
  - e.g., no global permit-write-local capabilities
Logical relation

\[ \mathcal{V}(W) \overset{\text{def}}{=} \{(n, i) \mid i \in \mathbb{Z}\} \]
\[ \cup \left\{ (n, ((r, g), \text{base, end}, a)) \mid (n, \text{base, end}) \in \text{readCondition}(g)(W) \right\} \]
\[ \cup \ldots \]

\[ \mathcal{E}(W) \overset{\text{def}}{=} \left\{ (n, pc) \mid \forall n' \leq n, (n', \text{reg}) \in \mathcal{R}(W), ms : n' \ W. (n', (\text{reg}[pc \mapsto pc], ms)) \in \mathcal{O}(W) \right\} \]

- Semantic model of well-behaved programs
- Captures the safe behaviour of the system
  - e.g., no global permit-write-local capabilities
Logical relation

\[ \mathcal{V}(W) \overset{\text{def}}{=} \{(n, i) \mid i \in \mathbb{Z}\} \]
\[ \cup \left\{ \left(n, ((r, g), \text{base}, \text{end}, a)) \mid (n, (\text{base}, \text{end})) \in \text{readCondition}(g)(W) \right\} \]
\[ \cup \ldots \]
\[ \mathcal{R}(W) \overset{\text{def}}{=} \{(n, \text{reg}) \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\}. (n, \text{reg}(r)) \in \mathcal{V}(W)\} \]
\[ \mathcal{E}(W) \overset{\text{def}}{=} \left\{(n, \text{pc}) \mid \forall n', (n', \text{reg}) \in \mathcal{R}(W), ms : n', W. \left(n', (\text{reg}[\text{pc} \mapsto pc], ms) \right) \in \mathcal{O}(W) \right\} \]

- Semantic model of well-behaved programs
- Captures the safe behaviour of the system
  - e.g., no global permit-write-local capabilities
Logical relation

\[ \mathcal{V}(W) \overset{\text{def}}{=} \{ (n, i) \mid i \in \mathbb{Z} \} \]
\[ \cup \left\{ (n, ((r, g), \text{base, end}, a)) \mid (n, (\text{base, end})) \in \text{readCondition}(g)(W) \right\} \]
\[ \cup \ldots \]

\[ \mathcal{R}(W) \overset{\text{def}}{=} \{ (n, reg) \mid \forall r \in \text{RegisterName} \setminus \{ \text{pc} \}. (n, reg(r)) \in \mathcal{V}(W) \} \]

\[ \mathcal{E}(W) \overset{\text{def}}{=} \left\{ (n, pc) \mid \forall n' \leq n, (n', reg) \in \mathcal{R}(W), ms :_{n'} W. (n', (\text{reg}[pc \mapsto pc], ms)) \in \mathcal{O}(W) \right\} \]

- Semantic model of well-behaved programs
- Captures the safe behaviour of the system
  - e.g., no global permit-write-local capabilities
- Uses PL techniques known from high-level languages
Interesting properties

Lemma (Revoke temporary memory satisfaction)

If $ms :_n W$, then $ms = ms' \uplus ms_r$ and $ms' :_n revokeTemp(W)$
Interesting properties

Lemma (Revoke temporary memory satisfaction)
If $ms : n W$, then $ms = ms' \uplus ms_r$ and $ms' : n revokeTemp(W)$

Lemma (Double monotonicity of value relation)

- If $(n, w) \in \mathcal{V}(W)$ and $W' \sqsupseteq^{pub} W$ then $(n, w) \in \mathcal{V}(W')$.
- If $(n, w) \in \mathcal{V}(W)$ and $W' \sqsupseteq^{priv} W$ and $w$ is not a local capability, then $(n, w) \in \mathcal{V}(W')$. 
Fundamental theorem of logical relations

- General statement of the guarantees provided by the capability machine.
- Intuitively: any program is safe as long as it only has access to safe values.

**Theorem (FTLR)**

If

\[ perm = rx \land (n, (base, end)) \in readCondition(g)(W) \]

(or similarly for \(rwx\) and \(rwlx\)),

then

\[ (n, ((perm, g), base, end, a)) \in E(W) \]
Road map

A simple capability machine

Applications
  Enforcing well-bracketedness

Semantic model
  Kripke worlds
  Logical relation
  Interesting properties

Conclusion
Conclusion

- Reasoning about a capability machine
  - Logical relation with some interesting novel aspects
    - local capabilities require public/private future worlds, used in new way
  - Provably enforce well-bracketed control flow using (just) local capabilities
    - Several details to get right
Questions/discussion
Recursive domain equation (simplified)

Wor \approx \text{Region}^*
\text{Region} ::= \text{revoked}
| (\text{temp}, s, (\phi_{pub}, \phi), H) \quad \text{with } H \in \text{State} \rightarrow (\text{Wor} \xrightarrow{\text{mon, ne}} \text{UPred(MemSegment)})^\text{pub}
| (\text{perm}, s, (\phi_{pub}, \phi), H) \quad \text{with } H \in \text{State} \rightarrow (\text{Wor} \xrightarrow{\text{mon, ne}} \text{UPred(MemSegment)})^\text{priv}