Reasoning About a Machine with Local Capabilities Provably Safe Stack and Return Pointer Management Technical Appendix Including Proofs and Details

Lau Skorstengaard
Aarhus University
lask@cs.au.dk

Dominique Devriese
imec-DistriNet, KU Leuven
dominique.devriese@cs.kuleuven.be

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

May 11, 2018

Contents

1 Capability Machine Definition and Operational Semantics

1.1 Domains and Notation

$$
\begin{aligned}
\text{Addr} &\stackrel{\text{def}}{=} \mathbb{N} \\
\text{Word} &\stackrel{\text{def}}{=} \text{Cap} + \mathbb{Z} \\
\text{Reg} &\stackrel{\text{def}}{=} \text{RegisterName} \to \text{Word} \\
\text{Mem} &\stackrel{\text{def}}{=} \text{Addr} \to \text{Word} \\
\text{Perm} &::= \textsc{o} \mid \textsc{ro} \mid \textsc{rw} \mid \textsc{rwl} \mid \textsc{rx} \mid \textsc{e} \mid \textsc{rwx} \mid \textsc{rwlx} \\
\text{ExecConf} &\stackrel{\text{def}}{=} \text{Reg} \times \text{Mem} \\
\text{Global} &::= \textsc{global} \mid \textsc{local} \\
\text{Cap} &\stackrel{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times (\text{Addr} + \{\infty\}) \times \text{Addr} \\
\text{Conf} &\stackrel{\text{def}}{=} \text{ExecConf} + \{\textit{failed}\} + \{\textit{halted}\} \times \text{Mem} \\
\text{MemSegment} &\stackrel{\text{def}}{=} \text{Addr} \rightharpoonup \text{Word}
\end{aligned}
$$

Local capabilities have been added by adding a new domain Global which represents whether a capability is local or global. There are two new permissions RWL and RWLX that permits writing local capabilities. They are otherwise the same as their non-"permit write local" counterparts.

As we have ∞ as a possible address, but our words cannot express ∞. We pick -42 as a representative for ∞ when it is in memory (we could have picked any negative noumber). Note that -42 is not an address, so for address operations -42 only represents ∞. It is the responsible of the programmer to keep track of what represents addresses (and take necessary precautions).

Define the following predicate:

Definition 1. *We say word w "w is non-local" iff either*

- $w = ((perm, g), base, end, a)$ $(perm, \text{GLOBAL})$ *for some perm, a, base, and end; or*
- $w \in \mathbb{Z}$

■

Figure 1: Locality hierarchy

Things to note:

- RegisterName contains pc, but is otherwise a sufficiently large finite set.
- Table 1 describes what all the permissions grant access to.
- Figure 2 shows the ordering of the permissions, i.e, the elements of Perm.
- Figure 1 shows the ordering of LOCAL and GLOBAL, i.e., the elements of Global.
- The ordering of Perm $\times$ Global is pointwise.

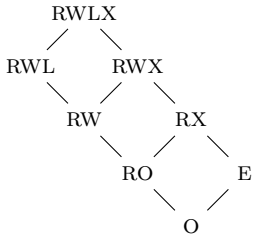


Figure 2: Permission hierarchy

O	No permissions. Grants no permissions
RO	Read only. Grants read permission
RW	Read-write. Grants read and write permission. Storage of local capabilities prohibited.
RWL	Read-write, permit write local. Grants read and write permission. Storage of local capabilities possible.
RX	Execute permission. Grants execute and read permissions.
E	Enter permission. This permission grants no access, but when jumped to, it will turn into an RX permission.
RWX	Read-write-execute permission. Grants read, write, and execute permissions. Storage of local capabilities prohibited.
RWLX	Read-write-execute, permit write local. Grants read, write, and execute permissions. Storage of local capabilities possible.

Table 1: The permissions in this capability system

Notation:

$$
\begin{aligned}
i &\in \text{Instructions} \\
r &\in \text{RegisterName} \\
pc &\in \text{Cap} \\
\text{pc} &\in \text{RegisterName} \\
\Phi &\in \text{ExecConf} \\
m, \Phi.\text{mem} &\in \text{Mem} \\
\Phi.\text{reg} &\in \text{Reg} \\
a &\in \text{Addr} \\
perm &\in \text{Perm} \\
((perm, g), base, end, a) &\in \text{Cap} \\
n &\in \mathbb{Z} \\
ms &\in \text{MemSegment}
\end{aligned}
$$

Words and instructions:

$$
\begin{aligned}
lv &::= \lfloor r \rfloor \\
hv &::= \langle r \rangle_m \\
rv &::= n \mid lv \\
i &::= \texttt{jmp}\ lv \mid \texttt{jnz}\ lv\ lv \mid \texttt{move}\ lv\ rv \mid \texttt{load}\ lv\ hv \mid \texttt{store}\ hv\ rv \mid \\
&\quad \texttt{plus}\ lv\ rv\ rv \mid \texttt{minus}\ lv\ rv\ rv \mid \texttt{lt}\ lv\ rv\ rv \mid \texttt{lea}\ lv\ rv \mid \texttt{restrict}\ lv\ rv \mid \texttt{subseg}\ lv\ rv\ rv \mid \\
&\quad \texttt{isptr}\ lv\ rv \mid \texttt{getp}\ lv\ lv \mid \texttt{getl}\ lv\ lv \mid \texttt{getb}\ lv\ lv \mid \texttt{gete}\ lv\ lv \mid \texttt{geta}\ lv\ lv \mid \\
&\quad \texttt{fail} \mid \texttt{halt}
\end{aligned}
$$

Further define $reg_0 \in \text{Reg}$ such that

$$\forall r \in \text{RegisterName}.\ reg_0(r) = 0$$

1.2 Operational Semantics

Assume a *decode* function that decodes words to instructions:

$$\mathit{decode} : \text{Word} \to \text{Instructions}$$

Assume an *encodePerm*, *encodeLoc*, and *encodePermPair* function that encodes a permissions, locality, and permission pair, respectively, as an integer:

$$\begin{aligned} \mathit{encodePerm} &: \text{Perm} \to \mathbb{Z} \\ \mathit{encodeLoc} &: \text{Global} \to \mathbb{Z} \\ \mathit{encodePermPair} &: (\text{Perm} \times \text{Global}) \to \mathbb{Z} \end{aligned}$$

Further, assume a left inverse function, *decodePermPair*, that decodes permissions

$$\mathit{decodePermPair} : \mathbb{Z} \to (\text{Perm} \times \text{Global})$$

We define the operational semantics as follows:

$$\begin{array}{ll} \Phi \to [\![\mathit{decode}(\Phi.\text{mem}(a))]\!](\Phi) & \text{if } \Phi.\text{reg}(\text{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ & \quad \text{and } \mathit{base} \le a \le \mathit{end} \\ & \quad \text{and } \mathit{perm} \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \\ \Phi \to \mathit{failed} & \text{otherwise} \end{array}$$

A number of functions and predicates used in the definition of $[\![-]\!]$ (defined later). Notice all of them are total.

$$\mathit{readAllowed}(\mathit{perm}) = \begin{cases} \mathit{true} & \text{if } \mathit{perm} \in \{\text{RWX}, \text{RWLX}, \text{RX}, \text{RW}, \text{RWL}, \text{RO}\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{writeAllowed}(\mathit{perm}) = \begin{cases} \mathit{true} & \text{if } \mathit{perm} \in \{\text{RWX}, \text{RWLX}, \text{RW}, \text{RWL}\} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{updatePcPerm}(w) = \begin{cases} ((\text{RX}, g), \mathit{base}, \mathit{end}, a) & \text{if } w = ((\text{E}, g), \mathit{base}, \mathit{end}, a) \\ w & \text{otherwise} \end{cases}$$

$$\mathit{nonZero}(w) = \begin{cases} \mathit{true} & \text{if } w \in \text{Cap or } w \in \mathbb{Z} \text{ and } w \neq 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{withinBounds}((\_, \mathit{base}, \mathit{end}, a)) = \begin{cases} \mathit{true} & \text{if } \mathit{base} \le a \le \mathit{end} \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\mathit{updatePc}(\Phi) = \begin{cases} \Phi[\text{reg.pc} \mapsto \mathit{newPc}] & \text{if } \Phi.\text{reg}(\text{pc}) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ & \quad \text{and } \mathit{newPc} = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a + 1) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{fail}]\!](\Phi) = \mathit{failed}$$

$$[\![\texttt{halt}]\!](\Phi) = (\mathit{halted}, \Phi.\mathrm{mem})$$

$$[\![\texttt{jmp}\ lv]\!](\Phi) = \Phi[\mathrm{reg.pc} \mapsto \mathit{updatePcPerm}(\Phi.\mathrm{reg}(lv))]$$

$$[\![\texttt{jnz}\ lv\ rv]\!](\Phi) = \begin{cases} \Phi[\mathrm{reg.pc} \mapsto \mathit{updatePcPerm}(\Phi.\mathrm{reg}(lv))] & \text{if } \mathit{nonZero}(\Phi.\mathrm{reg}(rv)) \\ \mathit{updatePc}(\Phi) & \text{if not } \mathit{nonZero}(\Phi.\mathrm{reg}(rv)) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{load}\ \lfloor r_1 \rfloor\ \langle r_2 \rangle_m]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto w]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) = c \\ & \quad \text{and } \mathit{readAllowed}(\mathit{perm}) \text{ and } \mathit{withinBounds}(c) \\ & \quad \text{and } w = \Phi.\mathrm{mem}(a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{store}\ \langle r_1 \rangle_m\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{mem}.a \mapsto w]) & \text{if } \Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) = c \\ & \quad \text{and } \mathit{writeAllowed}(\mathit{perm}) \text{ and } \mathit{withinBounds}(c) \\ & \quad \text{and } w = \Phi.\mathrm{reg}(r_2) \\ & \quad \text{and if } w = ((\_, \textsc{local}), \_, \_, \_), \\ & \quad\ \text{then } \mathit{perm} \in \{\textsc{rwlx}, \textsc{rwl}\} \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{move}\ \lfloor r_1 \rfloor\ rv]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto rv]) & rv \in \mathbb{Z} \\ \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto \Phi.\mathrm{reg}(rv)]) & \text{otherwise} \end{cases}$$

$$[\![\texttt{lea}\ \lfloor r_1 \rfloor\ rv]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto c]) & \text{if either } n = rv \text{ or } rv = \lfloor r_2 \rfloor \text{ and } n = \Phi.\mathrm{reg}(r_2) \\ & \quad \text{and in either case } n \in \mathbb{Z} \\ & \quad \text{and } \Phi.\mathrm{reg}(r_1) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ & \quad \text{and } \mathit{perm} \neq \textsc{e} \\ & \quad \text{and } c = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a + n) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{restrict}\ \lfloor r \rfloor\ rv]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r \mapsto c]) & \text{if } \Phi.\mathrm{reg}(r) = (\mathit{permPair}, \mathit{base}, \mathit{end}, a) \\ & \quad \text{and either } rv = n \text{ or } \Phi.\mathrm{reg}(rv) = n \\ & \quad \text{and in either case } n \in \mathbb{Z} \\ & \quad \text{and } \mathit{decodePermPair}(n) \sqsubseteq \mathit{permPair} \\ & \quad \text{and } c = (\mathit{decodePermPair}(n), \mathit{base}, \mathit{end}, a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{plus}\ \lfloor r_1 \rfloor\ rv_1\ rv_2]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto n_1 + n_2]) & \text{if for } i \in \{1,2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \quad \text{and in either case } n_i \in \mathbb{Z} \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{minus}\ \lfloor r_1 \rfloor\ rv_1\ rv_2]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto n_1 - n_2]) & \text{if for } i \in \{1,2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \quad \text{and in either case } n_i \in \mathbb{Z} \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{lt}\ \lfloor r_1 \rfloor\ rv_1\ rv_2]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto 1]) & \text{if for } i \in \{1,2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \quad \text{and in either case } n_i \in \mathbb{Z} \\ & \quad \text{and } n_1 < n_2 \\ \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto 0]) & \text{if for } i \in \{1,2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \quad \text{and in either case } n_i \in \mathbb{Z} \\ & \quad \text{and } n_1 \not< n_2 \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{subseg}\ \lfloor r \rfloor\ rv_1\ rv_2]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r \mapsto c]) & \text{if } \Phi.\mathrm{reg}(r) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ & \quad \text{and for } i \in \{1,2\} \\ & \quad n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \quad \text{and in either case } n_1 \in \mathbb{N} \\ & \quad \text{and } \mathit{base} \leq n_1 \\ & \quad \text{and } n_2 \leq \mathit{end} \text{ where } n_2 \in \mathbb{N} \\ & \qquad \text{or } n_2 = -42 \text{ and } \mathit{end} = \infty \\ & \quad \text{and } \mathit{perm} \neq \mathrm{E} \\ & \quad \text{and } c = ((\mathit{perm}, g), n_1, n_2, a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{geta}\ \lfloor r_1 \rfloor\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto a]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\_, \_), \_, \_, a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{getb}\ \lfloor r_1 \rfloor\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto \mathit{base}]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\_, \_), \mathit{base}, \_, \_) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{gete}\ \lfloor r_1 \rfloor\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto \mathit{end}]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\_, \_), \_, \mathit{end}, \_) \text{ and } \mathit{end} \neq \infty \\ \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto -42]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\_, \_), \_, \infty, \_) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{getp}\ \lfloor r_1 \rfloor\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto \mathit{encodePerm}(\mathit{perm})]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\mathit{perm}, \_), \_, \_, \_) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{getl}\ \lfloor r_1 \rfloor\ \lfloor r_2 \rfloor]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto \mathit{encodeLoc}(g)]) & \text{if } \Phi.\mathrm{reg}(r_2) = ((\_, g), \_, \_, \_) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

$$[\![\texttt{isptr}\ \lfloor r \rfloor\ rv]\!](\Phi) = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto 1]) & \text{if } \Phi.\mathrm{reg}(rv) \in \mathrm{Cap} \\ \mathit{updatePc}(\Phi[\mathrm{reg}.r_1 \mapsto 0]) & \text{otherwise} \end{cases}$$

Define the following macros: `restrict`, `subseg`, and `lea` that does not overwrite the source register. A `store` that allows integers to be stored directly. `store` requires a register r_t for storage of temporary values to be available.

$$\begin{aligned}
\texttt{restrict}\ r_1\ r_2\ r_3\ r_4 &\stackrel{def}{=} \texttt{move}\ r_1\ r_2 \\
&\quad\ \texttt{restrict}\ r_1\ r_3\ r_4 \\
\texttt{subseg}\ r_1\ r_2\ r_3\ r_4 &\stackrel{def}{=} \texttt{move}\ r_1\ r_2 \\
&\quad\ \texttt{subseg}\ r_1\ r_3\ r_4 \\
\texttt{lea}\ r_1\ r_2\ r_3 &\stackrel{def}{=} \texttt{move}\ r_1\ r_2 \\
&\quad\ \texttt{lea}\ r_1\ r_3 \\
\texttt{store}\ r\ n &\stackrel{def}{=} \texttt{move}\ r_t\ n \\
&\quad\ \texttt{store}\ r\ r_t
\end{aligned}$$

Lemma 1 (Determinacy)**.** *If* $\Phi \to \Phi'$ *and* $\Phi \to \Phi''$*, then* $\Phi' = \Phi''$*. If* $\Phi \to_n \Phi'$ *and* $\Phi \to_n \Phi''$*, then* $\Phi' = \Phi''$*. If* $\Phi \to_n \Phi'$ *and* $\Phi \to_{n'} (halted, mem'')$*, then* $n \leq n'$ *and* $\Phi' \to_{n'-n} (halted, mem'')$*.* ■

Proof. By easy inspection of the definition of the operational semantics. □

2 Malloc specification

Specification 1 (Malloc Specification)**.** *c_{malloc} satisfies the specification for malloc iff*

$$
\begin{aligned}
&c_{malloc} = ((\text{E}, \text{GLOBAL}), \_, \_, \_) \wedge \\
&\exists \iota_{malloc,0}. \\
&\quad (\forall \iota' \sqsupseteq^{priv} \iota_{malloc,0}. \forall W, i.\, W(i) = \iota' \Rightarrow \iota'.H(\iota'.s)(\xi^{-1}(W)) = \iota'.H(\iota'.s)(\xi^{-1}([i \mapsto W(i)]))) \wedge \\
&\quad \iota_{malloc,0}.v = \text{perm} \wedge \\
&\quad (\forall \Phi \in \text{ExecConf}.\, \forall ms_{footprint}, ms_{frame} \in \text{MemSegment}. \\
&\qquad \forall i, n, size \in \mathbb{N}.\, \forall w_{ret} \in \text{Word}. \\
&\qquad\quad \forall \iota_{malloc} \sqsupseteq^{priv} \iota_{malloc,0} \wedge \\
&\qquad\quad \Phi.\text{mem} = ms_{footprint} \uplus ms_{frame} \wedge ms_{footprint} :_n [i \mapsto \iota_{malloc}] \wedge \\
&\qquad\quad \Phi.\text{reg}(r_1) = size \wedge size \geq 0 \wedge \Phi.\text{reg}(r_0) = w_{ret} \wedge \\
&\qquad\quad \Phi.\text{reg}(\text{pc}) = updatePcPerm(c_{malloc}) \\
&\qquad\quad \Rightarrow \\
&\qquad\qquad \exists \Phi' \in \text{ExecConf}.\, \exists ms'_{footprint}, ms_{alloc} \in \text{MemSegment}. \\
&\qquad\qquad\quad \exists j \in \mathbb{N}.\, j > 0 \wedge \exists b', e' \in \text{Addr}.\, \exists \iota'_{malloc} \in \text{Region}. \\
&\qquad\qquad\qquad \Phi \rightarrow_j \Phi' \wedge \\
&\qquad\qquad\qquad \Phi'.\text{mem} = ms'_{footprint} \uplus ms_{alloc} \uplus ms_{frame} \wedge \\
&\qquad\qquad\qquad \iota'_{malloc} \sqsupseteq^{pub} \iota_{malloc} \wedge \\
&\qquad\qquad\qquad ms'_{footprint} :_{n-j} [i \mapsto \iota'_{malloc}] \wedge \\
&\qquad\qquad\qquad \text{dom}(ms_{alloc}) = [b', e'] \wedge \forall a \in [b', e'].\, ms_{alloc}(a) = 0 \wedge \\
&\qquad\qquad\qquad \Phi'.\text{reg} = \Phi.\text{reg}[\text{pc} \mapsto updatePcPerm(w_{ret})][r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')] \wedge \\
&\qquad\qquad\qquad size - 1 = e' - b') \wedge \\
&\quad (\forall \Phi \in \text{ExecConf}.\, (\Phi.\text{reg}(r_1) \notin \mathbb{Z} \vee \Phi.\text{reg}(r_1) < 0) \wedge \Phi.\text{reg}(\text{pc}) = updatePcPerm(c_{malloc}) \Rightarrow \exists j \in \mathbb{N}.\, \Phi \rightarrow_j \mathit{failed})
\end{aligned}
$$

■

In the specification above ι'_{malloc} is a future region of the initial region that governs malloc.

3 Macros

In order to write readable example programs, we provide macros (macro-instructions) that can be implemented in terms of the instruction set given in the formalisation.

In order to compute offsets and the like, the macros need registers to keep temporary computations in. We assume such a small set of registers $\text{RegisterName}_t \subseteq \text{RegisterName}$ is available and that RegisterName_t does not contain registers explicitely named in a program nor r_0, r_{stk}, or pc (but clearing all registers still clears the temporary registers).

3.1 Linking and ABI

In order to make capabilities to trusted code (and possibly untrusted code) available, we assume that some sort of linker has made these available. This is done in the following way: For every function, the first memory cell the capability for that function governs contains a capability for

the linking table. Each function name in a program corresponds to an offset in the table, e.g., `malloc` could be at offset 0. When a name is used in a program, it indicates what entry from the linking table to pick. The table should always be accessible by taking a copy of the capability in the pc-register and adjusting it to point to the first cell it governs.

The capability linking table can be shared between multiple functions that are linked to the same capabilities as it is accessed through read-only capabilities.

3.2 Flag table

A function may use flags to signal failure. We use the convention that a flag table is available in the second memory cell of a functions code (so just after the linking table). The flag table is accessed through a read-write capability and initially it contains all zero. Like the linking table, each entry is associated with a name which may appear in the macros.

The flag table should never be shared between distrusting parties.

We will often want to make room in memory for a linking-table capability and a flag-table capability. We therefore define a constant that represents the offset of the actual code of a function caused by these two capabilities:

$$\mathit{offsetLinkFlag} \stackrel{def}{=} 2$$

3.3 Macro definitions

In the following, we describe each of the macros. The descriptions are so detailed that it should be a simple matter to implement the macros. We provide a proposed implementation for each of the macros in order to install some confidence in the fact that it is possible to implement each of the macro.

`fetch` r f load the entry of the linking table corresponding to f to register r.

One possible fetch implementation (r_t1 and r_t2 are registers in RegName_t).

```
move r pc
getb r_t1 r
geta r_t2 r
minus r_t1 r_t1 r_t2 // Offset to first address, i.e., linking table (b-a)
lea r r_t1
load r r
lea r ... // ... replaced with offset to f in the linking table
move r_t1 0
move r_t2 0
load r r // f capability loaded to register r
```

`call` $r(\bar{r}_{args}, \bar{r}_{priv})$

$\bar{r}_{args}$ and $\bar{r}_{priv}$ are lists of registers. An overview of this call:

- Set up activation record
- Create local enter capability for activation (protected return pointer)
- Clear unused registers
- Jump
- Upon return: Run activation code

A more detailed description of each of the above steps:

Set up activation record

- Run malloc to get a piece of memory with space for:
  - Words in $\bar{r}_{priv}$
  - Code return capability (opc)
  - Activation code
- Store the words in $\bar{r}_{priv}$ to the activation record.
- Adjust a copy of the current pc to point to the return address in code and save it to the activation record.
- Write the activation code to the activation record.

Create local enter capability for activation Adjust the capability for the activation record to point to the beginning of the activation record and restrict it to a local enter-capability. Place this capability in r_0.

Clear unused registers Clear all the register that are not pc, r, r_0 or in $\bar{r}_{args}$.

Jump Jump to register r

Activation code The activation code does the following:

- Move the stored "private" words in to their respective $\bar{r}_{priv}$ registers.
- Load the return capability to pc

Possible implementation. We will use `malloc` r n and `rclear` $\bar{r}$ (defined below). Assume $r_{\bar{priv}} = r_{priv,1}, \ldots, r_{priv,n}$

```
  malloc r_t ... // ... is the size of activation record
// store private state in activation record
  store r_t r_priv,1
  lea r_t 1
  store r_t r_priv,2
  lea r_t 1
  ...
  lea r_t 1
  store r_t r_priv,n
  lea r_t 1
// store old pc
  move r_t1 pc
  lea r_t1 ... // ... is the offset to return address
  store r_t r_t1
  lea r_t1 1
// store activation record
  store r_t encode(i_1)
  lea r_t1 1
  ...
  lea r_t1 1
  store r_t encode(i_m)
  lea r_t1 k //  k is m-1, i.e. the offset to the first instruction of the activation code.
  restrict r_t1 encodePermPair((Local,e))
  move r_0 r_t1
```

```
rclear R // R = RegisterName - {r,pc,r_0,r_args}
jmp r
```

Activation record. The instructions correspond to $i_1, \ldots, i_m$ in the above.

```
move r_t pc
getb r_t1 r_t
geta r_t2 r_t
minus r_t1 r_t1 r_t2
// load private state
lea r_t r_t1
load r_priv,1 r_t
lea r_t 1
load r_priv,2 r_t
lea r_t 1
...
lea r_t 1
load r_priv,n r_t
lea r_t 1
// load old pc
load pc r_t
```

`malloc` r n Calls malloc to allocates a piece of memory of size n. The capability will be stored in register r. One possible malloc implementation (r_t1 is a register in RegName_t) and r_1 is the register from the malloc specification.

```
fetch r malloc
move r_1 n
// save return pointer
move r_t1 r_0
// setup new return pointer
move r_0 pc
lea r_0 4 // 4 is the offset to just after jmp r
restrict r_0 encodePerm(e)
jmp r
move r r_1
move r_0 r_t1 // restore return pointer
move r_1 0
move r_t1 0
```

`assert`$_{flag}$ r_1 r_2 Compares the words in register r_1 and r_2 (if one of them is an integer, then use that in the comparison). If they are equal, then execution continues. If they are unequal, then the assertion flag named *flag* in the flag list is set to 1 and execution halts (if no flag is specified, then the first flag in the list is set to 1).
There are four different asserts based on whether r_1 and r_2 are registers or numbers. If r_1 and r_2 are registers:

```
// setup pointer to fail.
move r_t3 pc
lea r_t3 ... // ... is the offset to fail
// make sure both registers contain either capability or integer
```

```
            isptr r_t1 r_1
            isptr r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            // set up capability for cap case:
            move r_t4 pc
            lea r_t4 ... // ... is the offset to caps
            jnz r_t4 r_t2 // jump to caps if r_t2 contains a capability
            // the two registers contain an integer
            minus r_t1 r_1 r_2
            jnz r_t3 r_t1
            // the two integers in the registers are equal
            move r_t4 pc
            lea r_t4 ... // .. offset to success
caps:
            geta r_t1 r_1
            geta r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            getb r_t1 r_1
            getb r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            gete r_t1 r_1
            gete r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            getp r_t1 r_1
            getp r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            getl r_t1 r_1
            getl r_t2 r_2
            minus r_t1 r_t1 r_t2
            jnz   r_t3 r_t1
            // the two capabilities in the registers are equal
            move r_t4 pc
            lea r_t4 ... // .. offset to success
fail:
            // get the flag capability
            move r_t3 pc
            getb r_t1 pc
            geta r_t2 pc
            minus r_t1 r_t1 r_t2
            lea r_t3 r_t1
            lea r_t3 1 // the flag table capability is at the second address of cap.
            load r_t1 r_t3
            lea r_t1 ... // ... is the offset of flag in the table
            store r_t1 1
```

```
          halt
success:
          // clean up
          move r_t1 0
          move r_t2 0
          move r_t3 0
          move r_t4 0
```

If r_1 is a register, but r_2 is a constant:

```
          // setup pointer to fail.
          move r_t3 pc
          lea r_t3 ... // ... is the offset to fail
          // make sure both registers contain either capability or integer
          isptr r_t1 r_1
          jnz   r_t3 r_t1
          minus r_t1 r_1 r_2
          jnz r_t3 r_t1
          // the two integers in the registers are equal
          move r_t3 pc
          lea r_t3 ... // .. offset to success
fail:
          // get the flag capability
          move r_t3 pc
          getb r_t1 pc
          geta r_t2 pc
          minus r_t1 r_t1 r_t2
          lea r_t3 r_t1
          lea r_t3 1 // the flag table capability is at the second address of cap.
          load r_t1 r_t3
          lea r_t1 ... // ... is the offset of flag in the table
          store r_t1 1
          halt
success:
          // clean up
          move r_t1 0
          move r_t3 0
```

The case where r_1 is a constant and r_2 is a register is omitted. The case where both are constant is also omitted - if the constants are the same, then the macro is nothing. If they are different, then it corresponds to the failed part of both of the above implementations.

`mclear` r Stores 0 to all the memory cells the capability r governs.[1]

Possible implementation:

```
        move r_t r
        getb r_t1 r_t
        geta r_t2 r_t
        minus r_t2 r_t1 r_t2
```

[1]This may in some cases seem like an unreasonable slow instruction. In a real system it would probably be implemented as a vector operation which allows modification of continuous segments of memory rather fast.

```
        lea r_t r_t2
        gete r_t2
        minus r_t1 r_t2 r_t1
        plus r_t1 r_t1 1
        move r_t2 pc
        lea r_t2 ... // ... is the offset to end
        move r_t3 pc
        lea r_t3 ... // ... is the offset to iter
iter:
        jnz r_t2 r_t1
        store r_t 0
        lea r_t 1
        plus r_t1 r_t1 1
        jmp r_t3
end:
        move r_t 0
        move r_t1 0
        move r_t2 0
        move r_t3 0
```

`rclear` $\bar{r}$ Moves 0 to all the registers in the list $\bar{r}$.
Possible implementation: Say $\bar{r} = r_1, \ldots, r_n$

```
move r_1 0
move r_2 0
// ...
move r_n 0
```

Note:

- `call` will fail if we have local capabilities in one of the registers of the "private" register list as it relies on a capability returned by malloc which will not be permit-write-local. This severely limits how `scall` can be used and it provides very little in terms of control-flow integrety when nested. Below, we introduce `scall` which can handle local capabilities in the "private" state.

3.4 Stack

Some programs will assume access to a stack which will be in part indicated by the program macros but also in the correctness lemma. The stack is accessed through a local RWLX-capability. Programs will assume that the stack resides in some register, say r_{stk}.

The stack resides entirely in memory. There is no separation between the memory and the stack, so when we talk about the stack it is as a conceptual thing.

Even though the memory is infinite, we will only use a finite part for the stack. If we have allocated too little memory for the stack, and we try to push something anyway, then the execution will fail. As we consider failing admissible, we are okay with this.

When not in the middle of a push or a pop, the stack capability points to the top word of the stack. For an empty stack, the stack capability points to the address just below of the range of authority for the stack capability.

The stack grows upwards

`push` r Pushes the word in register `r` to the stack by incrementing the address of the stack capability by one and storing the word through the stack capability.
Possible implementation:

```
lea r_stk 1
store r_stk r
```

`pop` r Pops the top word of the stack by loading it to register r, and decrementing the address of the stack capability.

```
load r r_stk
minus r_t1 0 1
lea r_stk r_t1
```

`scall` $r(\bar{r}_{args}, \bar{r}_{priv})$
$\bar{r}_{args}$ and $\bar{r}_{priv}$ are lists of registers. This call assumes r_{stk} contains a stack capability. An overview of this call:

- Push "private" registers to the stack.
- Push the restore code to the stack.
- Push return address capability
- Push stack capability
- Create protected return pointer
- Restrict stack capability to unused part
- Clear the part of the stack we release control over
- Clear unused registers
- Jump
- Upon return: Run the on stack restore code
- Return address in caller-code: Restore "private" state

A more detailed description of the above steps:

Push "private" registers to the stack Push all the words in the registers in $\bar{r}_{priv}$ to the stack.

Push the restore code to the stack Push the restore code to the stack (described later). This code needs to be on the stack to make sure the stack capability can be restored. We keep the restore code on the stack minimal. The caller code does the rest of the restoration.

Push return address capability Push a capability for the return address (in the memory) to the stack.

Push stack capability Push the full stack capability to the stack.

Create protected return pointer Make a new version of the stack pointer that points to the beginning of the restoration code. Restrict it to a local enter-capability and put it in r_0.

Restrict stack capability to unused part Make the stack capability only govern the unused part.

Clear the part of the stack we release control over Store 0 to all the memory cells the restricted stack pointer has authority over.

Clear unused registers Clear all registers but pc, r, r_0, r_{stk}, and $\bar{r}_{args}$.

Jump Jump to register r.

Run the on stack restore code Load the stack capability to r_{stk}. Pop the old program counter (the return address in caller-code) from the stack to pc.

Return address in caller-code: Restore "private" state

- Pop the restore code of the stack
- Pop the private state on the stack into their respective $\bar{r}_{priv}$ registers.

Possible implementation, say $\bar{r}_{args} = r_{args,1}, \ldots, r_{args,m}$ and $\bar{r}_{priv} = r_{priv,1}, \ldots, r_{priv,n}$:

```
// push private state
  push r_priv,1
  ...
  push r_priv,n
// push activation code
  push encode(i_1)
  ...
  push encode(i_4)
// push old pc
  move r_t1 pc
  lea r_t1 ... // ... is the offset to after
  push r_t1
// push stack pointer
  push r_stk
// set up protected return pointer
  move r_0 r_stk
  lea r_0 -5 // -5 is the offset to the first instruction of the activation code
  restrict r_0 encodePermPair((Local,e))
// restrict stack capability
  geta r_t1 r_stk
  plus r_t1 r_t1 1
  getb r_t2 r_stk
  subseg r_stk r_t1 r_t2
// clear unused part of the stack
  mclear r_stk
// clear non-argument registers
  rclear R // where R = RegisterName - {pc,r_stk,r_0,r,r_args}
  jmp r
after:
// pop the restore code
  pop r_t1
  pop r_t1
  pop r_t1
  pop r_t1
```

```
// pop the private state into approriate registers
  pop r_priv,1
  ...
  pop r_priv,n
```

where the restore code is as follows:

```
 i_1 = move r_t1 pc
 i_2 = lea r_t1 5 // 5 is the offset to the address where the old stack pointer is located
 i_3 = load r_stk r_t1
 i_4 = pop pc
```

Note:

- If we want to have local capabilities as part of our private state, then we need to have a stack and use `scall`. If we do not have any local capabilities we want to keep around, then we can use `call`, but it will incur a small memory leak as the activation records cannot be recycled! It is also possible to use a combination of `scall` and `call`, but when `call` is used, then we have no way to store the stack, so we cannot use `scall` after that.
- As a rule of thumb: If you have provided an untrusted entity access to part of the stack, then it needs to be cleared before it is passed to an untrusted party.
- As a rule of thumb: If you receive a stack from an untrusted source, then you need to check that it is a local RWLX-capability and clear it! If any callbacks are provided, then they need to be global.

`crtcls` $[(x_1, r_1), \ldots (x_n, r_n)]$ r_{code}

$[(x_1, r_1), \ldots (x_n, r_n)]$ is a list of variable bindings. If an instruction refers to a variable, then it will assume that an environment is available in a designated register (say r_{env}). The register r_{code} should contain a capability governs the code of the closure and that is executable when jumped to.

Allocate memory for variable environment

Store register contents to environment

Allocate memory for record with environment capability, code capability, and activation code

Store capabilities and activation code to record

Restrict the capability for the "closure pair" to an enter capability

Activation code:

- Load the environment capability to a designated register
- Load the code capability.
- Jump to the code.

A more detailed description of each step:

Allocate memory for variable environment Have malloc allocate a piece of memory of size n (the size of the variable environment).

Store register contents to environment Store the contents of each of the registers $r_1, \ldots, r_n$ to the newly allocated memory.

Allocate memory for record with environment capability, code capability, and activation code Allocate a new piece of memory with room for a capability for the environment.

Store capabilities and activation code to record Store the environment capability and code capability in the record followed by the activation code.

Restrict the capability for the "closure pair" to an enter capability Adjust the capability to point to the start of the activation code and restrict it to a global enter-capability.

Activation code:

- Load the environment capability to a designated register.
- Load the code capability.
- Jump to the code.

Possible implementation of `crtcls` $\overline{(x, r_v)}$ r_{code} where $|\overline{(x, r_v)}| = n$ (i_1,...,i_6, i.e. the activation code, is defined later):

```
malloc r_t1 n
store r_t1 r_v1
lea r_t1 1
store r_t1 r_v2
lea r_t1 1
...
lea r_t1 1
store r_t1 r_vn
lea r_t1 -n
restrict r_t1 encodePermPair((Global,rw))
malloc r_1 8 //length of activation record
store r_1 r_code // code capability
lea r_1 1
store r_1 r_t1 // environment capability
move r_t1 0
lea r_1 1
store r_1 encode(i_1)
lea r_1 1
store r_1 encode(i_2)
lea r_1 1
...
lea r_1 1
store r_1 encode(i_6)
lea r_1 -5 //offset to first instruction
restrict r_1 encodePerm(e)
```

Activation code (i_1,...,i_6):

```
i_1 = move r_t1 pc
i_2 = lea r_t1 -2
i_3 = load r_env r_t1
i_4 = lea r_t1 1
i_5 = load r_t1 r_t1
i_6 = jmp r_t1
```

`load` r x Assumes environment capability available in register r_{env}. Loads the word at the index associated with x in the environment list. Loads from this capability into r.
Possible implementation:

```
move r_t1 r_env
lea r_t1 ... // ... corresponds to offset of x in environment
load r r_t1
move r_t1 0
```

`store` x r Assumes environment capability available in register r_{env}. Loads the word at the index associated with x in the environment list. Stores the contents of register r through this capability.

```
move r_t1 r_env
lea r_t1 ... // ... corresponds to offset of x in environment
store r_t1 r
move r_t1 0
```

`reqglob` r Tests if register r contains a GLOBAL capability. If not fail, otherwise continue execution.
Possible implementation:

```
getl r_t1 r
minus r_t1 r_t1 encodeLoc(Global)
move r_t2 pc
lea r_t2 4 // 4 is the offset to just after fail
jnz r_t1 r_t2
fail
move r_t1 0
move r_t2 0
```

`reqperm` r n Tests if register r contains a capability with permission $decodePerm(n)$. If not fail, otherwise continue execution.
Possible implementation:

```
getp r_t1 r
minus r_t1 r_t1 n
move r_t2 pc
lea r_t2 4 // 4 is the offset to just after fail
jnz r_t1 r_t2
fail
move r_t1 0
move r_t2 0
```

`prepstack` r Tests if register r contains a capability with permission RWLX. If not fail, otherwise assume r points to $((\text{RWLX}, g), base, end, a)$ adjust it to $((\text{RWLX}, g), base, end, base - 1)$.
Possible implementation

```
reqperm r encodePerm(rwlx)
getb r_t1 r
geta r_t2 r
```

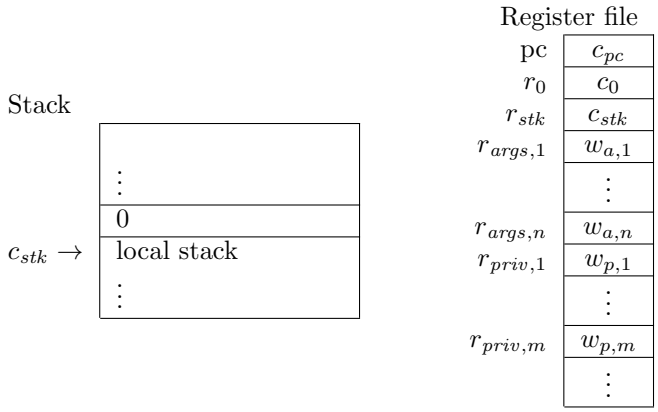


Figure 3: This is the first figure of 6 that illustrates how `scall` works. In this example, the call `scall` $r([r_{args,1}, \ldots, r_{args,n}], [r_0, r_{priv,1}, \ldots, r_{priv,m}])$. `In this example the two lists of registers are disjoint even though that does not have to be the case.`

```
minus r_t1 r_t1 r_t2
lea r r_t1
minus r_t1 0 1
lea r r_t1
move r_t1 0
move r_t2 0
```

Note:

- In a real setting due to a limited number of registers, some of the arguments might be spilled to the stack. It would be possible to do something similar here, but to keep matters simple, we opt not to do so.
- `reqperm` can be used to test whether something can pass as a stack.
- `reqglob` can be used to test whether a callback is admissible in the presence of a stack.
- The code of a closure will often be found in conjunction with the code that creates it.
- `prepstack` as "prepare stack". This ensures that the register contains something that looks like a stack and it is prepared for our stack convention.

3.5 Labels

`l:` is a meta level label that can be used to refer to a specific address. When placed on the line of a macro, it refers to the first instruction of this macro.

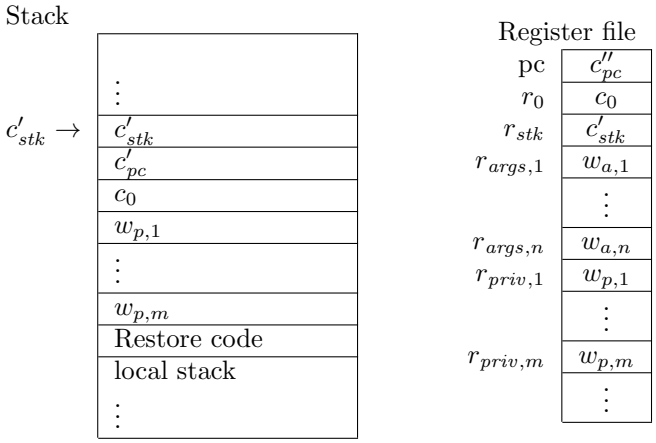


Figure 4: Stack and register-file after the restore code, "private" registers (remember r_0 is here private.), return address (c'_{pc}), and stack capability (c'_{stk}) have been pushed to the stack.

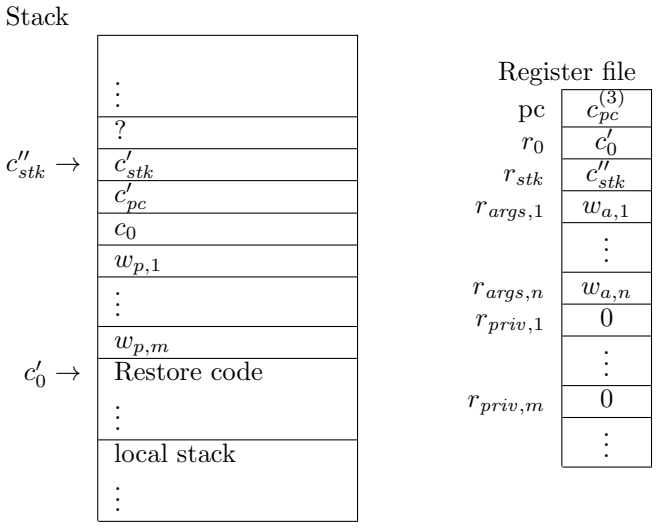


Figure 5: Stack and register-file after the c'_{stk} has been limited to only give authority over the empty part of the stack (the new capability is c''_{stk}). The empty part of the stack has been cleared. c'_0 is made from c'_{stk} by setting it to point to the restore code and restricting it to a local enter-capability. The "private" registers have been cleared.

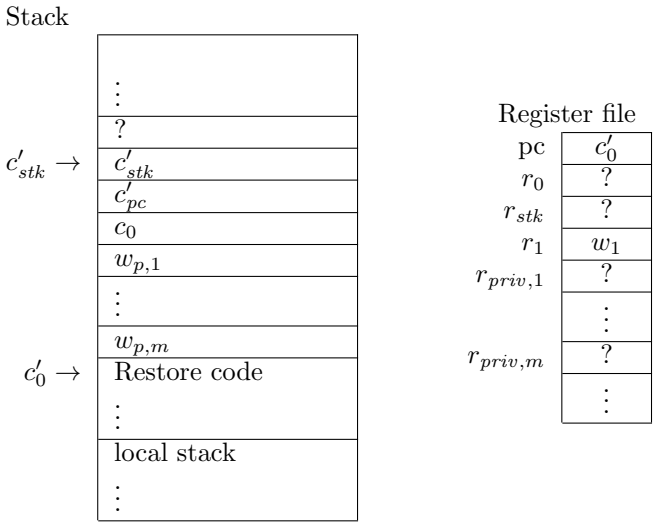


Figure 6: Stack and register-file upon return from f. At this point we have no idea what is in the register-file apart from the pc which we know points to the restore code. The contents of the stack we released access to is also unknown. (Notice that we have changed the order of the registers as we are no longer interested in the argument registers. By convention we expect a return value to be in r_1, which is why we have named that word, but the words in the remaining non-special-purpose registers could also be considered return values.)

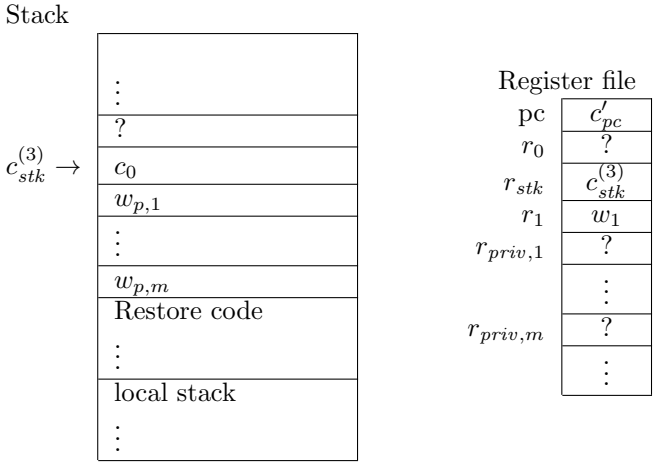


Figure 7: Stack and register-file after executing the restore code. The old stack capability has been restored and the pc-register now points to the return address in memory.

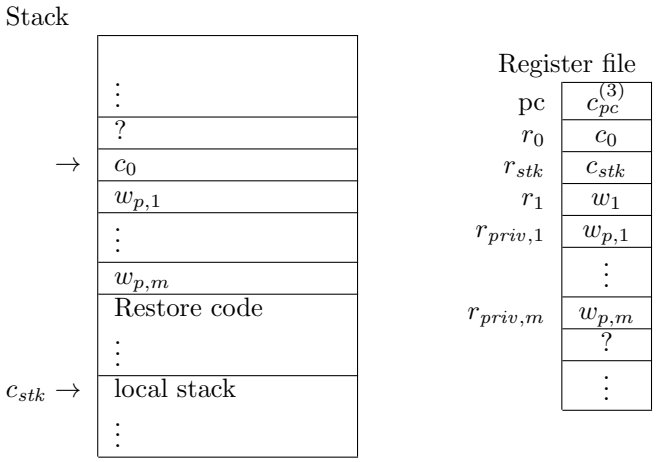


Figure 8: Stack and register-register file after the clean up code has been run. The "private" words have been popped to their respective registers. The restore code has been popped off the stack.

4 Examples

4.1 Encapsulation of Local State

Assembly program not using stack. Assume that $\mathtt{r_1} \notin \{\mathrm{pc}, r_0\}$ is a register.

```
f1: malloc r_l 1
    store r_l 1
    fetch r_adv adv
    call r_adv([],[r_l])
    assert r_l 1
1f: halt
```

For `f1` to work, its local state needs to be encapsulated.

Lemma 2 (Correctness lemma for `f1`)**.**
For all $n \in \mathbb{N}$ let

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\mathrm{E}, \mathrm{GLOBAL}), \mathit{base}_{adv}, \mathit{end}_{adv}, \mathit{base}_{adv} + \mathit{offsetLinkFlag}) \\
c_{f1} &\stackrel{def}{=} ((\mathrm{RWX}, \mathrm{GLOBAL}), \mathtt{f1} - \mathit{offsetLinkFlag}, \mathtt{1f}, \mathtt{f1}) \\
c_{malloc} &\stackrel{def}{=} ((\mathrm{E}, \mathrm{GLOBAL}), \mathit{base}_{malloc}, \mathit{end}_{malloc}, \mathit{base}_{malloc} + \mathit{offsetLinkFlag}) \\
m &\stackrel{def}{=} \mathit{ms}_{f1} \uplus \mathit{ms}_{\mathit{flag}} \uplus \mathit{ms}_{\mathit{link}} \uplus \mathit{ms}_{adv} \uplus \mathit{ms}_{malloc} \uplus \mathit{ms}_{\mathit{frame}}
\end{aligned}
$$

and

- *c_{malloc} satisfies the specification for malloc and $\iota_{malloc,0}$ is the region from the specification.*

where

$$
\begin{aligned}
&\mathrm{dom}(\mathit{ms}_{f1}) = [\mathtt{f1} - \mathit{offsetLinkFlag}, \mathtt{1f}] \\
&\mathrm{dom}(\mathit{ms}_{\mathit{flag}}) = [\mathit{flag}, \mathit{flag}] \\
&\mathrm{dom}(\mathit{ms}_{\mathit{link}}) = [\mathit{link}, \mathit{link} + 1] \\
&\mathrm{dom}(\mathit{ms}_{adv}) = [\mathit{base}_{adv}, \mathit{end}_{adv}] \\
&\mathit{ms}_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}
$$

and

- *$\mathit{ms}_{f1}(\mathtt{f1} - \mathit{offsetLinkFlag}) = ((\mathrm{RO}, \mathrm{GLOBAL}), \mathit{link}, \mathit{link}+1, \mathit{link})$, $\mathit{ms}_{f1}(\mathtt{f1} - \mathit{offsetLinkFlag} + 1) = ((\mathrm{RW}, \mathrm{GLOBAL}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, the rest of ms_{f1} contains the code of $f1$.*
- *$\mathit{ms}_{\mathit{flag}} = [\mathit{flag} \mapsto 0]$*
- *$\mathit{ms}_{\mathit{link}} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$*
- *ms_{adv} contains a global read-only capability for $\mathit{ms}_{\mathit{link}}$ on its first address. The remaining cells of the memory segment only contain instructions.*

if

$$(\mathit{reg}[\mathrm{pc} \mapsto c_{f1}], m) \to_n (\mathrm{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

■

Proof of Lemma 2. Let n be given and assume the premises in the lemma. Consider the following part of the execution:

$$(reg[\text{pc} \mapsto c_{f1}], m) \rightarrow_i (reg_0[\text{pc} \mapsto c_{malloc}][r_0 \mapsto c'_{f1}][r_1 \mapsto 1], m)$$

Where c'_{f1} is the return address. Use the malloc specification with

$$\begin{aligned} \iota_{malloc} &= \iota_{malloc,0} \\ ms_{footprint} &= ms_{malloc} \\ \Phi.\text{reg}(r_1) &= size = 1 \end{aligned}$$

to get

$$(reg_0[\text{pc} \mapsto c_{malloc}][r_0 \mapsto c'_{f1}][r_1 \mapsto 1], m) \rightarrow_j (reg_0[\text{pc} \mapsto c'_{f1}][r_0 \mapsto c'_{f1}][r_1 \mapsto c_l], m')$$

for some j where for some $\iota'_{malloc} \sqsupseteq^{pub} \iota_{malloc,0}$

1. $m' = ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_l \uplus ms'_{malloc} \uplus ms_{frame}$
2. $ms'_{malloc} :_{n-j} [0 \mapsto \iota_{malloc}]$
3. $\text{dom}(ms_l) = [l, l]$
4. $c_l = ((\text{RWX}, \text{GLOBAL}), l, l, l)$
5. $ms_l(l) = 0$

Continue the execution to the next malloc hidden in `call`.

$(reg_0[\text{pc} \mapsto c'_{f1}][r_0 \mapsto c'_{f1}][r_1 \mapsto c_l], m') \rightarrow_k (reg_0[\text{pc} \mapsto c_{malloc}][r_0 \mapsto c''_{f1}][r_1 \mapsto len_{ar}][r_l \mapsto c_l], m'')$

where

6. $m'' = m'[l \mapsto 1]$

Use the malloc specification notice:

- len_{ar} is the needed size for the activation record.
- 8. and (downwards closure) gives us the needed memory segment satisfaction.
- $ms_{footprint} = ms'_{malloc}$

Get:

$(reg_0[\text{pc} \mapsto c_{malloc}][r_0 \mapsto c''_{f1}][r_1 \mapsto len_{ar}][r_l \mapsto c_l], m'') \rightarrow_{j'} (reg_0[\text{pc} \mapsto c''_{f1}][r_0 \mapsto c''_{f1}][r_1 \mapsto c_{ar}][r_l \mapsto c_l], m^{(3)})$

for some j' where for some $[0 \mapsto \iota'_{malloc}] \sqsupseteq^{pub} [0 \mapsto \iota_{malloc}]$

7. $m'' = ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_l \uplus ms_{ar} \uplus ms''_{malloc} \uplus ms_{frame}$
8. $ms''_{malloc} :_{n-j-j'} [0 \mapsto \iota'_{malloc}]$
9. $\text{dom}(ms_{ar}) = [b, e]$, and $e - b = len_{ar}$
10. $c_l = ((\text{RWX}, \text{GLOBAL}), b, e, b)$

11. $\forall a \in [b, e].\, ms_{ar}(a) = 0$

Continue execution until just after the jump to *adv*.

$(reg_0[\text{pc} \mapsto c''_{f1}][r_0 \mapsto c''_{f1}][r_1 \mapsto c_{ar}][r_l \mapsto c_l], m^{(3)}) \rightarrow_{k'} (reg_0[\text{pc} \mapsto updatePcPerm(c_{adv})][r_1 \mapsto c_{adv}][r_0 \mapsto c'_{ar}], m^{(3)})$

for some k' where

- $m^{(3)} = ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_l \uplus ms'_{ar} \uplus ms''_{malloc} \uplus ms_{frame}$
- ms'_{ar} contains the activation record, i.e., c_l, $c^{(3)}_{f1}$ (the return address in f1), and activation code.
- $c'_{ar} = ((\text{E}, \text{LOCAL})b, e, b + offset)$ where $b + offset$ is the first address of the activation code.

Define

- $W = [0 \mapsto \iota'_{malloc}][1 \mapsto \iota^{nwl,p}_{base_{adv}, end_{adv}}][2 \mapsto \iota^{sta}(perm, ms_{f1} \uplus ms_{ar} \uplus ms_l \uplus ms_{flag})][3 \mapsto \iota^{sta,u}(perm, ms_{link})]$

define

1. $ms = ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_l \uplus ms'_{ar} \uplus ms''_{malloc}$

Use the FTLR on $updatePcPerm(c_{adv})$ using world W, so show

- $(n, (base_{adv}, end_{adv})) \in readCondition(\text{GLOBAL})(W)$
  - Show: $\iota^{nwl,p}_{base_{adv}, end_{adv}} \overset{n}{\subsetneq} \iota^{pwl}_{base_{adv}, end_{adv}}$: Follows from Lemma 22.

Have

2. $(n, updatePcPerm(c_{adv})) \in \mathcal{E}(W)$

Let $n' = n - j - j' - k - k'$ and show

1. $ms :_{n'} W$

   1.1. Split the memory into the disjoint unions of 1 and show:

   1.1.1. case: $(n', ms_{malloc}) \in \iota'_{malloc}.H(\iota'_{malloc}.s)(W)$

   1.1.1.1. Use $ms_{malloc} :_{n'} [0 \mapsto \iota'_{malloc}]$ with malloc specification context independence property.

   1.1.2. case: $(n', ms_{adv}) \in H^{nwl}_{base_{adv}, end_{adv}} 1 W$

   1.1.2.1. Show $\forall a \in [base_{adv}, end_{adv}].\, ((n' - 1, ms(a)) \in \mathcal{V}(W) \wedge ms(a)$ is non-local)

   1.1.2.1. $a \neq base_{adv}$: trivial, contains instruction only and they are non-local.

   1.1.2.2. $a = base_{adv}$: show $((\text{RO}, \text{GLOBAL}), link, link + 1, link) \in \mathcal{V}(W)$
   GLOBAL capabilities are non-local.
   SFTS $\iota^{sta,u}(perm, ms_{link}) \overset{n'}{\subsetneq} \iota^{pwl}_{link, link+1}$ which follows from Lemma 23.

   1.1.3. $(n', ms_{link}) \in H^{sta,u}(1)(W)$:
   This boils down to showing:

   1.1.3.1. $(n' - 1, c_{malloc}) \in \mathcal{V}(W)$: Follows from Lemma 50.

1.1.3.2. $(n'-1, c_{adv}) \in \mathcal{V}(W)$: for $n'' < n'-1$ and $W' \sqsupseteq^{priv} W$ show: $(n'', updatePcPerm(c_{adv})) \in \mathcal{E}(W')$. Follows from Lemma 49, together with Lemma 79 and the fact that c_{adv} is non-local.

1.1.4. The last case follows from Lemma 67

2. $(n', reg_0[\text{pc} \mapsto updatePcPerm(c_{adv})][r_1 \mapsto c_{adv}][r_0 \mapsto c'_{ar}]) \in \mathcal{R}(W)$

2.1. case: $(n', c_{adv}) \in \mathcal{V}(W)$

2.1.1. Similar to 1.1.3.2.

2.2. case: $(n', c'_{ar}) \in \mathcal{V}(W)$.

2.2.1. Let $n'' < n'$ and $W' \sqsupseteq^{pub} W$ be given and show $(n'', updatePcPerm(c'_{ar})) \in \mathcal{E}(W')$
Let $n^{(3)} \leq n''$, $ms' :_{n^{(3)}} W'$, and $\left(n^{(3)}, reg\right)$ be given
Show: $\left(n^{(3)}, (reg[\text{pc} \mapsto updatePcPerm(c'_{ar})], ms')\right) \in \mathcal{O}(W')$
Assume $(reg[\text{pc} \mapsto updatePcPerm(c'_{ar})], ms' \uplus ms_{frame}) \rightarrow_{k''} (halted, m')$, for some $k'' \leq n^{(3)}$, m' and ms_{frame}. Due to $ms' :_{n^{(3)}} W'$, ms_{f1}, ms_{flag}, ms'_{ar}, and ms_l are unchanged.
The execution loads c_l to r_l and jumps to $c^{(3)}_{f1}$ (the point just before the assertion). As $ms_l = 1$, the assertion is successful and the execution halts. In other words, there were no changes to the memory.
Use W', $ms_r = \emptyset$, and ms' to get the desired result, i.e., $m' = ms' \uplus ms_{frame}$ and $ms' :_{n^{(3)}-k''} W'$ (using downwards closure of memory satisfaction).

2.3. case: $(n', 0) \in \mathcal{V}(W)$ (the contents remaining registers)
Trivial to show.

Get
$$\left(n', (reg_0[\text{pc} \mapsto updatePcPerm(c_{adv})][r_1 \mapsto c_{adv}][r_0 \mapsto c'_{ar}], m^{(3)})\right) \in \mathcal{O}(W)$$

By initial assumption of the lemma, the execution halts. Use ms_{frame}, m' and the number of steps it takes to halt to get: $W' \sqsupseteq^{priv} W$, ms_r and ms' s.t. $m' = ms_r \uplus ms' \uplus ms_{frame}$ and $ms' :_n W'$. As ι_{flag} is a permanent region, we know it is still in W', so $m'(flag) = 0$. □

4.2 Encapsulation of Local State Using Local Capabilities and `scall`

Assembly program using the stack. This program assumes a $r_{stk} \notin \{\text{pc}, r_0\}$ register that contains a stack capability (a local RWLX-capability):

```
f2: push 1
    fetch r1 adv
    scall r1([],[])
    pop r1
    assert r1 1
2f: halt
```

Lemma 3 (Correctness lemma for `f2`)**.** *let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), \mathit{base}_{adv}, \mathit{end}_{adv}, \mathit{base}_{adv} + \mathit{offsetLinkFlag}) \\
c_{f2} &\stackrel{def}{=} ((\text{RWX}, \text{GLOBAL}), \texttt{f2} - \mathit{offsetLinkFlag}, \texttt{2f}, \texttt{f2}) \\
c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), \mathit{base}_{malloc}, \mathit{end}_{malloc}, \mathit{base}_{malloc} + \mathit{offsetLinkFlag}) \\
c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{LOCAL}), \mathit{base}_{stk}, \mathit{end}_{stk}, \mathit{base}_{stk} - 1) \\
c_{link} &\stackrel{def}{=} ((\text{RO}, \text{GLOBAL}), \mathit{link}, \mathit{link} + 1, \mathit{link}) \\
\mathit{reg} &\in \text{Reg} \\
m &\stackrel{def}{=} ms_{f2} \uplus ms_{\mathit{flag}} \uplus ms_{\mathit{link}} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{\mathit{frame}}
\end{aligned}
$$

and

- c_{malloc} *satisfies the specification for malloc and* $\iota_{malloc,0}$ *is the region from the specification.*

where

$$
\begin{aligned}
&\text{dom}(ms_{f2}) = [\texttt{f2} - \mathit{offsetLinkFlag}, \texttt{2f}] \\
&\text{dom}(ms_{\mathit{flag}}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{\mathit{link}}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{stk}) = [\mathit{base}_{stk}, \mathit{end}_{stk}] \\
&\text{dom}(ms_{adv}) = [\mathit{base}_{adv}, \mathit{end}_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}] \qquad \textit{for all } n \in \mathbb{N}
\end{aligned}
$$

and

- $ms_{f2}(\texttt{f2} - \mathit{offsetLinkFlag}) = ((\text{RO}, \text{GLOBAL}), \mathit{link}, \mathit{link}+1, \mathit{link})$, $ms_{f2}(\texttt{f2} - \mathit{offsetLinkFlag} + 1) = ((\text{RW}, \text{GLOBAL}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{f2} *contains the code of* $f2$.
- $ms_{\mathit{flag}} = [\mathit{flag} \mapsto 0]$
- $ms_{\mathit{link}} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- $ms_{adv}(\mathit{base}_{adv}) = c_{link}$ *and* $\forall a \in [\mathit{base}_{adv} + 1, \mathit{end}].\, ms_{adv}(a) \in \mathbb{Z}$

if

$$(\mathit{reg}[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

■

Proof of Lemma 3 (using `scall` *lemma).* Let n be given and make the assumptions of the lemma. If we can show

$$(n, (\mathit{reg}[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], \mathit{ms} \uplus ms_{stk})) \in \mathcal{O}(W) \tag{1}$$

for

$$\mathit{ms} = ms_{f2} \uplus ms_{\mathit{flag}} \uplus ms_{\mathit{link}} \uplus ms_{adv} \uplus ms_{malloc}$$

and

$$W = [0 \mapsto \iota_{malloc,0}][1 \mapsto \iota^{sta}(\text{perm}, ms_{f2} \uplus ms_{\mathit{flag}})][2 \mapsto \iota^{sta,u}(\text{perm}, ms_{\mathit{link}})][3 \mapsto \iota^{nwl,p}_{\mathit{base}_{adv}, \mathit{end}_{adv}}]$$

then we are done as we by assumption has

$$(reg[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], m) \rightarrow_n (halted, m')$$

so 1 gives us a $W' \sqsupseteq^{priv} W$ where W' satisfy part of m'. As ms_{flag} is governed by a perm region, so it is unchanged. In other words

$$m'(flag) = 0$$

So it suffices to show 1. To this end use Lemma 8. Let ms_f be given, then

$$(reg[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], ms \uplus ms_{stk} \uplus ms_f) \rightarrow_k (reg', ms \uplus ms'_{stk} \uplus ms_f)$$

where

- (reg', ms) is looking at `scall` $r_{adv}([], [r_l])$ followed by c_{next}
- c_{next} is c_{f2} that points to the instruction after the `scall`.
- reg' points to stack with $[base_{stk} \mapsto 1]$ used and ms_{unused} unused
  - for some ms_{unused} where $ms'_{stk} = [base_{stk} \mapsto 1] \uplus ms_{unused}$.
- $reg'(r_{adv}) = c_{adv}$

In order to show the observation part necessary for Lemma 8, we use the "`scall` works"-Lemma (Lemma 58). Show the following

1. $ms :_{n-k} W$
   Use Lemma 66 with
   1.1. $ms_{f2} \uplus ms_{flag} :_{n-k} [1 \mapsto \iota^{sta}(\text{perm}, ms_{f2} \uplus ms_{flag})]$
   Lemma 67
   1.2. $ms_{adv} \uplus ms_{malloc} \uplus ms_{link} :_{n-k} W_{part}$
   where

$$W_{part} = [0 \mapsto \iota_{malloc,0}][2 \mapsto \iota^{sta,u}(\text{perm}, ms_{link})][3 \mapsto \iota^{nwl,p}_{base_{adv}, end_{adv}}]$$

   This amounts to
   1.2.1. $(n-k-1, ms_{malloc}) \in H\ 1\ W_{part}$ where H is the interpretaion of the $\iota_{malloc,0}$ region.
   Follows from the malloc specification.
   1.2.2. $(n-k-1, ms_{adv}) \in H^{nwl}\ 1\ W_{part}$
   Can be shown using Lemma 23.
   1.2.3. $(n-k-1, ms_{link}) \in H^{sta,u}(ms_{link})\ 1\ W_{part}$
   This amounts to showing
   1.2.3.1. $(n-k-2, c_{malloc}) \in \mathcal{V}(W_{part})$ Follows from Lemma 50.
   1.2.3.2. $(n-k-2, c_{adv}) \in \mathcal{V}(W_{part})$
   Follows from Theorem 2 using Lemma 22.

2. Hyp-Callee
   Assume
   - $\text{dom}(ms_{unused}) = \text{dom}(ms_{act} \uplus ms'_{unused})$,

- $W' = revokeTemp(W)[\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{act}), \iota^{pwl}(\text{dom}(ms'_{unused}))]$,
- $ms'' :_{n-k-1} W'$
- reg' points to stack with $\emptyset$ used and ms'_{unused} unused
- $reg' = reg_0[\text{pc} \mapsto updatePcPerm(c_{adv}), r_0 \mapsto c_{ret}, r_{stk} \mapsto c'_{stk}, r_{adv} \mapsto c_{adv}]$
- $(n-k-1, c_{ret}) \in \mathcal{V}(W')$
- $(n-k-1, c'_{stk}) \in \mathcal{V}(W')$

Show

$$(n-k-1, (reg', ms'')) \in \mathcal{O}(W')$$

By Theorem 2 we get

$$(n-k-1, updatePcPerm(c_{adv})) \in \mathcal{E}(W')$$

getting the desired result amounts to[2]

2.1. $(n-k-1, c_{adv}) \in \mathcal{V}(W)$
To this end let $n' < n-k-1$ and $W'' \sqsupseteq^{priv} W'$ be given and show

$$(n', updatePcPerm(c_{adv})) \in \mathcal{E}(W'')$$

Follows from Theorem 2 and Lemma 22.

3. Hyp-Cont
Assume

- $n' \leq n-2$
- $W'' \sqsupseteq^{pub} revokeTemp(W)$
- $ms'' :_{n'} revokeTemp(W'')$
- $reg''(\text{pc}) = c_{next}$
- reg'' points to stack with ms_{stk} used and ms''_{unused} unused for some ms''_{unused}

and show

$$(n', (reg'', ms'' \uplus [base_{stk} \mapsto 1] \uplus ms''_{unused})) \in \mathcal{O}(W'')$$

From $ms'' :_{n'} revokeTemp(W'')$, we get that ms_{f2} is unchanged. Given a frame ms'_f and assuming n' is sufficiently large, the execution continues as follows:

$$(reg'', ms''\uplus[base_{stk} \mapsto 1]\uplus ms''_{unused}\uplus ms_f) \to_k (halted, ms''\uplus[base_{stk} \mapsto 1]\uplus ms''_{unused}\uplus ms_f)$$

because 1 is popped of the stack to a register, then it is compared with 1 in the assertion, so the assertion succeeds and halts immediately after.

By assumption we had $ms'' :_{n'} revokeTemp(W'')$ which gives us exactly the memory satisfaction required by $\mathcal{O}(W'')$.

□

ML-like program:

[2] We have memory satisfaction by assumption and the above entails the register-file is in the register-file relation.

```
let f = fun adv =>
          let l = 1 in
          adv(l);
          l := 1;
          adv(0);
          assert(!l == 1)
```

In this example `let l = 1 in` allocates a new local capability `l` with read-write permissions. Assuming `adv` has no access to capabilities with permit write local, they cannot store `l` and thus change its value in the second call.

4.3 Well-Bracketedness Using Local Capabilities and `scall`

```
f3: push 1
    fetch r1 adv
    scall r1([],[])
    pop r1
    assert r1 1
    push 2
    fetch r1 adv
    scall r1([],[])
3f: halt
```

The assertion of $f3$ may seem a bit awkward because it is between two calls. If an adversary could capture the protected return pointer from the first call and save it until the second call, then the adversary could jump to it again. At this point the top of the stack would be 2, so when the execution reaches the assertion, it would fail. However, the produced return pointer is passed as a local capability, so the only place the adversary can store it is on the stack. The adversary loses control of the stack when control is returned to $f3$ where the `scall` makes sure to sanitise the stack and register file before control is passed back to the adversary. In other words, the adversary has no way to capture the continuation which makes the above safe and well-bracketed.

Lemma 4 (Correctness lemma for `f3`)**.** *For all $n \in \mathbb{N}$ let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), base_{adv}, end_{adv}, base_{adv} + offsetLinkFlag) \\
c_{f3} &\stackrel{def}{=} ((\text{RWX}, \text{GLOBAL}), \texttt{f3} - offsetLinkFlag, \texttt{3f}, \texttt{f3}) \\
c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{LOCAL}), base_{stk}, end_{stk}, base_{stk} - 1) \\
c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), base_{malloc}, end_{malloc}, base_{malloc} + offsetLinkFlag) \\
c_{link} &\stackrel{def}{=} ((\text{RO}, \text{GLOBAL}), link, link + 1, link) \\
reg &\in \text{Reg} \\
m &\stackrel{def}{=} ms_{f3} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}
$$

and

- c_{malloc} *satisfies the specification for malloc.*

where

$$\begin{aligned}
&\mathrm{dom}(ms_{f3}) = [\texttt{f3} - \mathit{offsetLinkFlag}, \texttt{3f}]\\
&\mathrm{dom}(ms_{\mathit{flag}}) = [\mathit{flag}, \mathit{flag}]\\
&\mathrm{dom}(ms_{\mathit{link}}) = [\mathit{link}, \mathit{link}+1]\\
&\mathrm{dom}(ms_{\mathit{stk}}) = [\mathit{base}_{\mathit{stk}}, \mathit{end}_{\mathit{stk}}]\\
&\mathrm{dom}(ms_{\mathit{adv}}) = [\mathit{base}_{\mathit{adv}}, \mathit{end}_{\mathit{adv}}]\\
&ms_{\mathit{malloc}} :_n [0 \mapsto \iota_{\mathit{malloc},0}]
\end{aligned}$$

and

- $ms_{f3}(\texttt{f3} - \mathit{offsetLinkFlag}) = ((\text{RO}, \text{GLOBAL}), \mathit{link}, \mathit{link}+1, \mathit{link})$, $ms_{f3}(\texttt{f3} - \mathit{offsetLinkFlag}+1) = ((\text{RW}, \text{GLOBAL}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{f3} *contains the code of* $f3$.
- $ms_{\mathit{flag}} = [\mathit{flag} \mapsto 0]$
- $ms_{\mathit{link}} = [\mathit{link} \mapsto c_{\mathit{malloc}}, \mathit{link}+1 \mapsto c_{\mathit{adv}}]$
- $ms_{\mathit{adv}}(\mathit{base}_{\mathit{adv}}) = c_{\mathit{link}}$ *and all other addresses of* ms_{adv} *contain instructions.*

if

$$(\mathit{reg}[\text{pc} \mapsto c_{f3}][r_{\mathit{stk}} \mapsto c_{\mathit{stk}}], m) \to_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

■

In an attempt to aid the reader, we first provide to high-level descriptions of possible proof of Lemma 4 followed by a more detailed proof.

Proof of Lemma 4 (high-level description). Executing $f2$ until just after the jump in the first scall brings us to a configuration where the stack contains 1 followed by some activation code followed by all zeros. The pc-register contains an executable adversary capability, register r_0 contains a protected return pointer - that is a local enter capability for the execution code, and the r_{stk} contains a capability for the cleared part of the stack.

At this point we can define a world with permanent regions

- fixing the assertion flag, the code of $f2$, and the linking table.
- the initial malloc region
- a $\iota^{\mathit{nwl},p}$ region

and temporary regions

- a region fixing the private part of the stack
- a ι^{pwl} region for the rest of the stack

From the FTLR, we get that in any future world of W, the adversary capability and its executable counter part is in the expression relation and thus safe to execute in suitable configurations. If the configuration we consider right now is suitable, then the execution produces a memory where the permanent invariants of W are kept which means that the flag is 0.

To argue that the configuration is suitable, we need to argue that invoking the continuation produces an admissible result. As the continuation is a LOCAL capability, we take a public future world of W. In this public world, the private part of the stack remains the same as before the jump, so when we reach the assertion it succeeds and execution continues. At the point of the jump in the second scall, the stack contains 2 instead of 1, but otherwise essentially the same. Here we again use that it is safe to execute the adversary and that the continuation in this case halts immediately in a configuration where the assertion flag must be 0. □

Proof of Lemma 4 (high-level description 2). If we can show

$$(reg[\mathrm{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], ms_{malloc} \uplus ms' \uplus ms_{adv} \uplus ms_{stk}) \in \mathcal{O}(W), \tag{2}$$

for a world W where the assertion flag is permanently 0, then it is still 0 in any configuration the execution halts in. W also needs to require the program and the linking table to permanently remain the same, have a region that governs *malloc* and a standard permanent no-write local region for the adversary.

Due to Lemma 58 the `scall` lemma, for each `scall` we have to argue that the adversary and continuation produces results that respect the regions of W. Using Lemma 8 the $\mathcal{O}$ anti reduction lemma, it suffices to argue that each part of $f3$ between `scall`s produces admissible results.

Executing until the first `scall` only pushes 1 to the stack, so the invariants of W are preserved. Due to the `scall` lemma, we need to argue that that the adversary and the continuation produce admissible results.

Using the FTLR, we get that the executable capability for the adversary is in the $\mathcal{E}$-relation. As we provide no arguments to the adversary, most of the conditions are satisfied by assumptions and Lemma 62, which makes sure that the stack capability is in the value relation. Which gives us that the adversary produces an admissible result.

With respect to the continuation, it is passed to the adversary as a local capability, so when we reason about it, we consider public future worlds. The `scall` uses temporary regions for the stack and these persist in public future worlds. This allows us to assume that the private part of the stack still contains 1 after the call. Further, the program, flags, and linking table remain the same in any kind of future world. Therefore, we know that the execution continues by popping 1 from the stack and then asserting that it is indeed 1, which is indeed the case, so 2 is pushed to the stack. At this point we reach another scall. No changes where made to the permanent part of the stack, so the invariants are still satisfied. At this point we use the `scall` lemma one last time. The adversary call code is well-behaved for the same reasons as in the first call. The `scall` lemma lets us assume that the continuation continues in a memory that satisfies the invariants of W. The execution halts immediately in the continuation, so it produces an admissible result. □

Proof of Lemma 4. Assume the premises of the lemma. Now define

$$\begin{aligned} W = {} & [0 \mapsto \iota_{malloc,0}] \\ & [1 \mapsto \iota^{sta}(\mathrm{perm}, ms_{flag} \uplus ms_{f2})] \\ & [2 \mapsto \iota^{sta,u}(\mathrm{perm}, ms_{link})] \\ & [3 \mapsto \iota^{nwl,p}_{base_{adv},end_{adv}}] \end{aligned}$$

Further define

$$ms' = ms_{flag} \uplus ms_{f2} \uplus ms_{link} \uplus ms_{adv}$$

If we can show

$$(n+1, (reg[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], ms_{malloc} \uplus ms' \uplus ms_{adv} \uplus ms_{stk})) \in \mathcal{O}(W), \tag{3}$$

then using ms_{frame} as the frame and m' as the resulting memory, we get that $m' = ms'' \uplus ms_r \uplus ms_{frame}$ for some ms' and ms_r s.t. $ms'' :_1 W$. Region 1 guarantees that the assertion flag is unchanged, so we have

$$m'(\mathit{flag}) = 0$$

So SFTS 3. To do so, we use Lemma 8. Let ms_f be given. The execution proceeds as follows:

$$(reg[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], ms' \uplus ms_{stk} \uplus ms_f) \to_i (reg', ms' \uplus [base_{stk} \mapsto 1] \uplus ms_{stk}|_{base_{stk}+1, end_{stk}} \uplus ms_f),$$

where

$$(reg', ms') \text{ is looking at } \texttt{scall}\ r([],[]) \text{ followed by } c_{next}$$

where c_{next} is c_{f3} adjusted to point to the next instruction, namely `pop r1`. Further we have

- reg' points to stack with $[base_{stk} \mapsto 1]$ used and $ms_{stk}|_{base_{stk}+1, end_{stk}}$ unused

and i is a suitable number of steps.

To show

$$(n-i, (reg', ms' \uplus [base_{stk} \mapsto 1] \uplus ms_{stk}|_{base_{stk}+1, end_{stk}})) \in \mathcal{O}(W)$$

We use Lemma 58 (we do not use the local frame in the lemma) which requires us to show

1. $ms' :_{n-i} W$
   Partition ms' as follows:
   - 1.1. ms_{malloc}: governed by $\iota_{malloc,0}$, use malloc specification.
   - 1.2. $ms_{flag} \uplus ms_{f2}$: governed by region 1, only this memory segment is accepted.
   - 1.3. ms_{link}: governed by region 2, only this memory segment is accepted. We also need to show that the contents is safe, i.e. shoe
     - 1.3.1. $(n-i, c_{malloc}) \in \mathcal{V}(W)$: Follows from Lemma 50.
     - 1.3.2. $(n-i, c_{adv}) \in \mathcal{V}(W)$:
       We will show
       $$\forall W' \sqsupseteq^{priv} W.\, (n, c_{adv}) \in \mathcal{V}(W') \tag{4}$$
       which will give us what we need using downwards closure as well as a result for later use.
       Let $W' \sqsupseteq^{priv} W$ be given and show
       $$(n, (base_{adv}, end_{adv}, base_{adv} + \mathit{offsetLinkFlag})) \in \mathit{enterCondition}(\textsc{global})(W')$$
       to this end let $W'' \sqsupseteq^{priv} W'$ and $n' < n$ be given and show
       $$(n', \mathit{updatePcPerm}(c_{adv})) \in \mathcal{E}(W'')$$
       This follows from the FTLR (Theorem 2) if we can show
       $$(n', base_{adv}, end_{adv}) \in \mathit{readCondition}(\textsc{global})(W'')$$
       $\iota^{nwl,p}_{base_{adv}, end_{adv}}$ governs the adversary, so the result follows from Lemma 22.

1.4. ms_{adv}: Follows from Lemma 23.

2. Hyp-Callee
Assume

- $\mathrm{dom}(ms_{stk}|_{base_{stk}+1,end_{stk}}) = \mathrm{dom}(ms_{act} \uplus ms'_{unused})$
- $W' = revokeTemp(W)[\iota^{sta}(\mathrm{temp}, [base_{stk} \mapsto 1] \uplus ms_{act}), \iota^{pwl}(\mathrm{dom}(ms'_{unused}))]$
- $ms'' :_{n-i-1} W'$
- reg'' points to stack with $\emptyset$ used and ms'_{unused} unused
- $reg'' = reg_0[\mathrm{pc} \mapsto updatePcPerm(reg'(r)), r_0 \mapsto c_{ret}, r_{stk} \mapsto c'_{stk}, r \mapsto reg'(r)]$
- $(n-i-1, c_{ret}) \in \mathcal{V}(W')$
- $(n-i-1, c'_{stk}) \in \mathcal{V}(W')$

for some ms_{act}, ms_{unused}, ms'', reg'', c_{ret}.

Using the FTLR, we get $(n-i-1, updatePcPerm(c_{adv})) \in \mathcal{E}(W')$, from

2.1. $ms'' :_{n-i-1} W'$: By the above assumptions

2.2. $(n-i-1, reg'') \in \mathcal{V}(W')$:
show

2.2.1. $(n-i-1, c_{ret}) \in \mathcal{V}(W')$: by above assumptions.

2.2.2. $(n-i-1, c'_{stk}) \in \mathcal{V}(W')$: by above assumptions.

2.2.3. $(n-i-1, c_{adv}) \in \mathcal{V}(W')$: follows from 4.

2.2.4. The remaining registers we need to consider contain 0 and are thus trivial to show.

we get

$$(n-i-1, (ms'', reg'')) \in \mathcal{O}(W')$$

3. Hyp-Cont
Assume:

- $n' \leq n-i-2$
- $W'' \sqsupseteq^{pub} revokeTemp(W)$
- $ms'' :_{n'} revokeTemp(W'')$
- for all r, we have that:

$$reg''(r) \begin{cases} = c_{next} & \text{if } r = \mathrm{pc} \\ \in \mathcal{V}(W'') & \text{if } reg''(r) \text{ is a global capability and } r \notin \{\mathrm{pc}, r_{stk}\} \end{cases}$$

- reg'' points to stack with $[base_{stk} \mapsto 1]$ used and ms''_{unused} unused for some ms''_{unused}

and show

3.1. $(reg'', ms'' \uplus [base_{stk} \mapsto 1] \uplus ms''_{unused}) \in \mathcal{O}(revokeTemp(W''))$
As $W'' \sqsupseteq^{priv} W$, we know that the program, assertion flag, and linking table remain unchanged in ms''. Given some frame ms'_f, then the execution proceeds by first succeeding the assertion and then pushing 2 to the stack:

$$(reg'', ms'' \uplus [base_{stk} \mapsto 1] \uplus ms''_{unused} \uplus ms'_f) \to_k (reg^{(3)}, ms'' \uplus [base_{stk} \mapsto 2] \uplus ms''_{unused} \uplus ms'_f)$$

where

- $(reg^{(3)}, ms'')$ is looking at `scall` $r([],[])$ followed by c'_{next}
- $reg^{(3)}$ points to stack with $[base \mapsto 2]$ used and ms''_{unused} unused
- $reg^{(3)}(r) = c_{adv}$

By Lemma 8 it suffices to show

3.1.1. $\left(n'-k, (reg^{(3)}, ms'' \uplus [base_{stk} \mapsto 2] \uplus ms''_{unused})\right) \in \mathcal{O}(revokeTemp(W''))$
Show this using Lemma 58 a. Show:

3.1.1.1. $ms'' :_{n'-k} revokeTemp(W'')$ is satisfied by one of the first Hyp-cont assumptions and Lemma 47.

3.1.1.2. Hyp-Callee
Assume:
- $\mathrm{dom}(ms''_{unused}) = \mathrm{dom}(ms'_{act} \uplus ms^{(3)}_{unused})$
- $W^{(3)} = revokeTemp(W'')[\iota^{sta}(\mathrm{temp}, [base_{stk} \mapsto 2] \uplus ms'_{act}), \iota^{pwl}(\mathrm{dom}(ms^{(3)}_{unused}))]$
- $ms^{(3)} :_{n'-k-1} W^{(3)}$
- $reg^{(4)}$ points to stack with $\emptyset$ used and $ms^{(3)}_{unused}$ unused
- $reg^{(4)} = reg_0[\mathrm{pc} \mapsto updatePcPerm(c_{adv}), r_0 \mapsto c'_{ret}, r_{stk} \mapsto c''_{stk}, r \mapsto c_{adv}]$
- $(n'-k-1, c'_{ret}) \in \mathcal{V}(W^{(3)})$
- $(n'-k-1, c''_{stk}) \in \mathcal{V}(W^{(3)})$

This argument is almost identical to the one we just did for the first call: Using the FTLR, we get $(n-i-1, updatePcPerm(c_{adv})) \in \mathcal{E}(W^{(3)})$. Which we use with

3.1.1.2.1. $ms^{(3)} :_{n'-k-1} W^{(3)}$: By assumption.

3.1.1.2.2. $\left(n'-k-1, reg^{(4)}\right) \in \mathcal{R}(W^{(3)})$: Show:

3.1.1.2.2.1. $(n'-k-1, c_{adv}) \in \mathcal{V}(W^{(3)})$ by Assumption 4.

3.1.1.2.2.2. $(n'-k-1, c'_{ret})$ by assumption.

3.1.1.2.2.3. $(n'-k-1, c''_{stk})$ by assumption

to get

$$\left(n'-k-1, (reg^{(4)}, ms^{(3)})\right) \in \mathcal{O}(W^{(3)})$$

3.1.1.3. Hyp-Cont
Assume
- $n'' \leq n'-k-2$
- $W^{(3)} \sqsupseteq^{pub} revokeTemp(W'')[\iota^{sta}(\mathrm{temp}, ms_{stk})][\iota^{sta}(\mathrm{temp}, ms^{(3)}_{unused})]$
- $ms^{(3)} :_{n''} revokeTemp(W^{(3)})$
- for all r, we have that:

$$reg^{(4)}(r) \begin{cases} = c'_{next} & \text{if } r = \mathrm{pc} \\ \in \mathcal{V}(W'') & \text{if } reg^{(4)}(r) \text{ is a global capability and } r \notin \{\mathrm{pc}, r_{stk}\} \end{cases}$$

- reg' points to stack with $[base_{stk} \mapsto 2]$ used and $ms^{(3)}_{unused}$ unused for some $ms^{(3)}_{unused}$

and show

$$(n'', (reg^{(3)}, ms^{(3)} \uplus [base_{stk} \mapsto 2] \uplus ms^{(3)}_{unused})) \in \mathcal{O}(revokeTemp(W^{(3)}))$$

To this end let ms''_f, m'', and $j \leq n''$ be given and assume

$$(reg^{(3)}, ms^{(3)} \uplus [base_{stk} \mapsto 2] \uplus ms^{(3)}_{unused} \uplus ms''_f) \rightarrow_j (halted, m'')$$

As the execution halts immediately,

$$m'' = ms^{(3)} \uplus [base_{stk} \mapsto 2] \uplus ms^{(3)}_{unused} \uplus ms''_f$$

By assumption we had $ms^{(3)} :_{n''} revokeTemp(W^{(3)})$ and the frame is unchanged, so we can split the memory as needed.

□

4.4 Inverted Control and Return From Closure

The following example is constructed to investigate the difficulties of preserving an adversary's local frame. There is no assertion as this is (slightly) beside the point. The lemma we would prove about this should look like Lemma 5, but it is not state and proven here.

```
g2:  move r3 pc
     lea r3 ...
     crtcls [] r3
     rclear RegisterName \ {pc, r0, r1}
2g:  jmp r0
f5:  reqglob r1
     prepstack rstk
     scall r1([],[r0, renv])
     mclear rstk
     rclear RegisterName \ {r0, pc}
5f:  jmp r0
```

4.5 Variant of the "awkward" example

Assembly variant of the "awkward" example from [Dreyer et al., 2010, p. 11] which roughly was:

```
g = fun _ => let x = 0 in
               fun f =>
                 x := 0;
                 f();
                 x := 1;
                 f();
                 assert(x == 1)
```

Our translation of the example:

```
g1:  malloc r2 1
     store r2 0
     move r3 pc
     lea r3 ...
     crtcls [(x, r2)] r3
     rclear RegisterName \ {pc, r0, r1}
```

```
1g:  jmp r_0
f4:  reqglob r_1
     prepstack r_stk
     store x 0
     scall r_1([],[r_0, r_1, r_env])
     store x 1
     scall r_1([],[r_0, r_env])
     load r_1 x
     assert r_1 1
     mclear r_stk
     rclear RegisterName \ {r_0, pc}
4f:  jmp r_0
```

Where the ... is the appropriate offset to make the capability point to `f4`.

Lemma 5 (Correctness of $g1$)**.** *For all $n \in \mathbb{N}$ let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\text{RWX}, \text{GLOBAL}), base_{adv}, end_{adv}, base_{adv} + offsetLinkFlag) \\
c_{g1} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), \texttt{g1} - offsetLinkFlag, \texttt{4f}, \texttt{g1}) \\
c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{LOCAL}), base_{stk}, end_{stk}, base_{stk} - 1) \\
c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{GLOBAL}), base_{malloc}, end_{malloc}, base_{malloc} + offsetLinkFlag) \\
c_{link} &\stackrel{def}{=} ((\text{RO}, \text{GLOBAL}), link, link, link) \\
m &\stackrel{def}{=} ms_{g1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}
$$

where

- *c_{malloc} satisfies the specification for malloc with $\iota_{malloc,0}$*

$$
\begin{aligned}
&\text{dom}(ms_{g1}) = [\texttt{g1} - offsetLinkFlag, \texttt{4f}] \\
&\text{dom}(ms_{flag}) = [flag, flag] \\
&\text{dom}(ms_{link}) = [link, link] \\
&\text{dom}(ms_{stk}) = [base_{stk}, end_{stk}] \\
&\text{dom}(ms_{adv}) = [base_{adv}, end_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}
$$

and

- *$ms_{g1}(\texttt{g1} - offsetLinkFlag) = ((\text{RO}, \text{GLOBAL}), link, link, link)$, $ms_{g1}(\texttt{g1} - offsetLinkFlag + 1) = ((\text{RW}, \text{GLOBAL}), flag, flag, flag)$, the rest of ms_{g1} contains the code of $g1$ immediately followed by the code of $f4$.*
- *$ms_{flag} = [flag \mapsto 0]$*
- *$ms_{link} = [link \mapsto c_{malloc}]$*
- *$ms_{adv}(base_{adv}) = c_{link}$ and all other addresses of ms_{adv} contain instructions.*
- *$\forall a \in \text{dom}(ms_{stk}).\ ms_{stk}(a) = 0$*

if

$$(reg_0[\text{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \to_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

■

In the proof of Lemma 5, we will use the following region

Definition 2.

$$\begin{aligned} \iota_x &= (\mathit{perm}, 0, \phi_{pub}, \phi, H_x) \\ \phi_{pub} &= \{(0,1)\}^* \\ \phi &= (1,0) \cup \phi_{pub} \\ H_x\ s\ \hat{W} &= \{(n, ms) \mid ms(x) = s \wedge n > 0\} \cup \{(0, ms)\} \end{aligned}$$

■

Lemma 6. *Definition 2 defines a region.* ■

Proof of Lemma 6.

- ϕ_{pub} is defined as the reflexive transitive closure, so it is immediately well formed.
- ϕ adds a transition to ϕ_{pub} and is also reflexive and transitive.
- H_x is trivially non-expansive in the state.
- H_x does not depend on the $\hat{W}$, so it also becomes trivially non-expansive and (privately) monotone in $\hat{W}$.

□

Proof of Lemma 5 (using `scall` lemma). Let n be given and make the assumptions of the lemma. Define

$$\begin{aligned} W = &[0 \mapsto \iota_{malloc,0}] \\ &[1 \mapsto \iota^{sta,u}(\text{perm}, ms_{link})] \\ &[2 \mapsto \iota^{pwl}_{base_{stk}, end_{stk}}] \\ &[3 \mapsto \iota^{nwl,p}_{base_{adv}, end_{adv}}] \\ &[4 \mapsto \iota^{sta}(\text{perm}, ms_{g1} \uplus ms_{\mathit{flag}})] \end{aligned}$$

and

$$ms = ms_{g1} \uplus ms_{\mathit{flag}} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc}$$

If we can show

$$(n, (reg_0[\text{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], ms \uplus ms_{stk})) \in \mathcal{O}(W) \tag{5}$$

then the termination assumption gives us that part of m satisfies a private future world of W. Region 4 is permanent, so

$$m(\mathit{flag}) = 0$$

So it suffices to show Eq. 5. To this end use the FTLR to show $(n, c_{adv}) \in \mathcal{E}(W)$, so show

1. $(n, (base_{adv}, end_{adv})) \in readCondition(\text{GLOBAL})(W)$
   Simple using region 3 in W and Lemma 22.

2. $(n, (base_{adv}, end_{adv})) \in writeCondition(\iota^{nwl}, \text{GLOBAL})(W)$
   Simple using region 3 in W, using Lemma 15.

in conclusion $(n, c_{adv}) \in \mathcal{E}(W)$. We get Eq. 5 if we show 3. and 4.:

3. $ms \uplus ms_{stk} :_n W$

   3.1. $ms_{g1} \uplus ms_{flag} :_n [4 \mapsto \iota^{sta}(\text{perm}, ms_{g1} \uplus ms_{flag})]$
   Lemma 67.

   3.2. $ms_{stk} :_n [2 \mapsto \iota^{pwl}_{base_{stk}, end_{stk}}]$
   Lemma 68 and assumption that ms_{stk} is all 0.

   3.3. $ms_{malloc} \uplus ms_{link} \uplus ms_{adv} :_n [0 \mapsto \iota_{malloc,0}][1 \mapsto \iota^{sta,u}(\text{perm}, ms_{link})][3 \mapsto \iota^{nwl,p}_{base_{adv}, end_{adv}}]$

   For convenience define

   $$W_{mini} = [0 \mapsto \iota_{malloc,0}][1 \mapsto \iota^{sta,u}(\text{perm}, ms_{link})][3 \mapsto \iota^{nwl,p}_{base_{adv}, end_{adv}}]$$

   Partitioning the memory segment in the components of the disjoint union, the *malloc* part follows from assumption $ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]$ and the *malloc* specification. The linking table part of memory amounts to showing:

   $$(n, ms_{link}) \in H^{sta,u}(ms_{link})(1)(\xi^{-1}(W_{mini}))$$

   which in turn amounts to showing

   $$(n-1, c_{malloc}) \in \mathcal{V}(W_{mini})$$

   which follows from Lemma 50.
   Showing

   $$(n, ms_{adv}) \in H^{sta}_{base_{adv}, end_{adv}}(1)(\xi^{-1}(W_{mini}))$$

   is a bit more involved. It amounts to

   $$\forall a \in \text{dom}(ms_{adv}).\, (n-1, ms_{adv}(a)) \in \mathcal{V}(W_{mini})$$

   which in turn is trivial for everything but

   $$(n-1, c_{link}) \in \mathcal{V}(W_{mini})$$

   This amounts to showing

   $$(n-1, (link, link)) \in readCondition(\text{GLOBAL})(W_{mini})$$

   which amounts to

   $$\iota^{sta,u}(\text{perm}, ms_{link}) \overset{n-1}{\subseteq} \iota^{pwl}_{link, link}$$

   which follows from Lemma 23.

Using Lemma 66 repeatedly with 3.1., 3.2., and 3.3. gives the desired memory satisfaction.

4. $(n, reg_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}]) \in \mathcal{R}(W)$
   This amounts to showing

   4.1. $(n, c_{stk}) \in \mathcal{V}(W)$
   The assumptions on c_{stk} and ms_{stk} in the lemma entail

   - $reg_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}]$ points to stack with $\emptyset$ used and ms_{stk} unused

   and further there is a ι^{pwl} region for ms_{stk} in W, so the result follows from Lemma 62.

   4.2. $(n, c_{g1}) \in \mathcal{V}(W)$
   Let $n_1 < n$ and $W_1 \sqsupseteq^{priv} W$ and show

$$(n_1, updatePcPerm(c_{g1})) \in \mathcal{E}(W_1)$$

   To this end assume $n_2 \leq n_1$, $ms_1 :_{n_2} W_1$, and $(n_2, reg_1) \in \mathcal{R}(W_1)$ and show

$$(n_2, (reg_1[\text{pc} \mapsto updatePcPerm(c_{g1})], ms_1)) \in \mathcal{O}(W_1)$$

   Using Lemma 59, Lemma 60, Lemma 8 (and some others), it suffices to show

$$(n_2', (reg_2, ms_2 \uplus ms'_{malloc} \uplus ms_{cls} \uplus ms_x)) \in \mathcal{O}(W_2)$$

   where

$$W_2 = W_1[0 \mapsto \iota_{malloc}][i_1 \mapsto \iota^{sta}(perm, ms_{cls})][i_2 \mapsto \iota_x]$$

   where $i_1, i_2 \notin \text{dom}(W_1)$ and $i_1 \neq i_2$ and ι_x is the region in Definition 2 which is a region by Lemma 6. Also

   - $\iota_{malloc} \sqsupseteq^{priv} \iota'_{malloc}$
   - $c_x = ((\text{RWX}, \text{GLOBAL}), x, x, x)$
   - $ms_x = [x \mapsto 0]$
   - $ms_2 \uplus ms'_{malloc} \uplus ms_{cls} \uplus ms_x :_{n_2'} W_2$
   - $c_{env} = ((\text{RWX}, \text{GLOBAL}, env, env, env))$
   - $ms_{env} = [env \mapsto c_x]$
   - $c_{f4} = ((\text{RWX}, \text{GLOBAL}), \texttt{g1} - offsetLinkFlag, \texttt{4f}, \texttt{f4})$
   - $ms_{cls} = ms_{env} \uplus ms_{act}$
   -

$$reg_2(r) = \begin{cases} updatePcPerm(reg_1(r_0)) & r = \text{pc} \\ reg_1(r_0) & r = r_0 \\ c_{cls} & r = r_1 \\ 0 & \text{otherwise} \end{cases}$$

   Finally assume Hyp-Act:

$$\forall reg, ms.\, reg(\text{pc}) = c_{cls} \Rightarrow \\ \exists j.\, \forall ms_f.\, (reg, ms \uplus ms_{cls} \uplus ms_f) \rightarrow_j (reg[\text{pc} \mapsto updatePcPerm(c_{f4})][r_{env} \mapsto c_{env}], ms \uplus ms_{cls} \uplus ms_f) \tag{6}$$

   Show

$$(n_2 - i, (reg_2, ms_2 \uplus ms'_{malloc} \uplus ms_{env} \uplus ms_x \uplus ms_{cls})) \in \mathcal{O}(W_2) \tag{7}$$

   If $reg_1(r_0).perm \notin \{\text{E}, \text{RX}, \text{RWX}, \text{RWLX}\}$, then the execution fails after the jump and is thus trivially true.

If $reg_1(r_0).perm \in \{\text{E}, \text{RX}, \text{RWX}, \text{RWLX}\}$, then either *executeCondition* or *enterCondition* holds for the capability in $reg_1(r_0)$. Now use $W_2 \sqsupseteq^{pub} W_1$ with the appropriate condition to get

$$(n_2 - i, updatePcPerm(reg_1(r_0))) \in \mathcal{E}(W_2)$$

which in turn gives us 4.2. if we can show the following

4.2.1. $ms_2 \uplus ms'_{malloc} \uplus ms_{env} \uplus ms_x \uplus ms_{cls} :_{n_2-i} W_2$
We first show the following:

- $ms_2 \uplus ms'_{malloc} :_{n_2-i} W_1[0 \mapsto \iota_{malloc}]$: we already know this.
- $ms_{env} \uplus ms_{cls} :_{n_2-i} [i_1 \mapsto \iota^{sta}(\text{perm}, ms_{env} \uplus ms_{cls})]$: By Lemma 67.
- $ms_x :_{n_2-i} i_2 \mapsto \iota_x$: $ms(x) = 0$, so okay.

4.2.2. $(n_2 - i, reg_2) \in \mathcal{R}(W_2)$
Amounts to showing

4.2.2.1. $(n_2 - i, reg_2(r_0)) \in \mathcal{V}(W_2)$ by assumption $(n_2, reg_1) \in \mathcal{R}(W_1)$ and $\mathcal{V}$ monotonicity wrt. $\sqsupseteq^{pub}$

4.2.2.2. $(n_2 - i, c_{cls}) \in \mathcal{V}(W_2)$
Let $n_3 < n_2 - i$ and $W_3 \sqsupseteq^{priv} W_2$ be given and show

$$(n_3, updatePcPerm(c_{cls})) \in \mathcal{E}(W_3)$$

To this and let $n_4 \leq n_3$, $ms_3 :_{n_4} W_3$, and $(n_4, reg_3) \in \mathcal{R}(W_3)$ and show

$$(n_4, (reg_3[\text{pc} \mapsto updatePcPerm(c_{cls})], ms_3)) \in \mathcal{O}(W_3) \tag{8}$$

Let ms_3^p and ms_3^t be memory segments such that $ms_3 = ms_3^p \uplus ms_3^t$ and $ms_3^p :_{n_4} revokeTemp(W_3)$ (using Lemma 64). By $ms_3 :_{n_4} W_3$ and $W_3 \sqsupseteq^{priv} W_2$, we know $ms_{cls} \subseteq ms_3^p$, so using Hyp-Act(6), we get j such that

$$\begin{aligned}\forall ms_f.\, &(reg_3[\text{pc} \mapsto updatePcPerm(c_{cls})], ms_3^p \uplus ms_3^t \uplus ms_f) \to_j \\ &(reg_3[\text{pc} \mapsto updatePcPerm(c_{cls})][r_{env} \mapsto c_{env}], ms_3^p \uplus ms_3^t \uplus ms_f)\end{aligned} \tag{9}$$

Using Lemma 8 it suffices to show

$$\big(n_4, (reg_3[\text{pc} \mapsto updatePcPerm(c_{cls})][r_{env} \mapsto c_{env}], ms_3^p \uplus ms_3^t)\big) \in \mathcal{O}(W_3)$$

Use Lemma 8 again. This time let ms''_f be given and take ms_r to be the part of ms_3^t that $reg_3(r_{stk})$ does not govern. By the operational semantics, we know[3]

$$(reg_3[\text{pc} \mapsto updatePcPerm(c_{cls})][r_{env} \mapsto c_{env}], ms_3^p \uplus ms_3^t \uplus ms''_f) \to_{j'} (reg_4, ms_4 \uplus ms_3^t \uplus ms''_f)$$

where

- (reg_4, ms_4) is looking at `scall` $r_1([], [r_0, r_1, r_{env}])$ followed by c_{next}
  - c_{next} is the capability pointing to the next instruction.
- reg_4 points to stack with $\emptyset$ used and ms_{unused} unused
  - `prepstack` did not fail, so the stack capability must be RWLX and follow the stack convention.

[3] the execution may fail, but then the configuration is trivially in the observation relation.

- $reg_4(r_1)$ is a GLOBAL capability.
  - `reqglob` did not fail
- $ms_4(x) = 0$
- $reg_4(r_{env}) = c_{env}$

region i_2 (the ι_x region) can be in either state 0 or 1, so to make sure it is in state 0, we use a private transition. So let W_4 be $revokeTemp(W_3)$ with region i_2 in state 0. We then have

$$ms_4 :_{n_4-j-j'} W_4$$

Now we can use Lemma 58 to show:

$$(n_4 - j - j', (reg_4, ms_4 \uplus ms_r \uplus \emptyset \uplus ms_{unused})) \in \mathcal{O}(W_4)$$

where ms_r is the local frame of the `scall` lemma.

4.2.2.2.1. $ms_4 :_{n_4-j-j'} revokeTemp(W_4)$: follows from $W_4 = revokeTemp(W_4)$

4.2.2.2.2. Hyp-Callee

We know $(n_4, reg_3(r_1)) \in \mathcal{V}(W_3)$. If this is not a capability that becomes executable when jumped to, then the execution fails, so the register memory segment pair is trivially in the observation relation. If it is executable, then either the *executeCondition* or the *enterCondition* holds for appropriate values. We also know that it is a global capability, so we can use it with private future worlds. We have $W_5 = revokeTemp(W_4)[\iota^{sta}(\text{temp}, \emptyset \uplus ms_{act} \uplus ms_r), \iota^{pwl}(\text{dom}(ms'_{unused}))] \sqsupseteq^{priv} W_3$, for some ms_{act} and ms'_{unused}. By the execute/enter condition, we have

$$(n_4 - j - j', updatePcPerm(reg_3(r_1))) \in \mathcal{E}(W_5)$$

Now it suffices to show

4.2.2.2.2.1. $ms_5 :_{n_4-j-j'-1} W_5$ for some ms_5 which is one of the assumptions of Hyp-Callee.

4.2.2.2.2.2. $(n_4 - j - j' - 1, reg_5) \in \mathcal{R}(W_5)$ where reg_5 is as described in the `scall` lemma Hyp-callee premise.

Amounts to showing:
1) $(n_4 - j - j' - 1, reg_3(r_1)) \in \mathcal{V}(W_5)$, use Lemma 79 with $(n_4 - j - 1, reg_3(r_1)) \in \mathcal{V}(W_3)$, the capability is global, and $W_5 \sqsupseteq^{priv} W_3$. 2) The protected return pointer and the stack capability are in the value relation by Hyp-callee assumptions.

which gives us

$$(n_4 - j - j' - 1, (reg_5, ms_5)) \in \mathcal{O}(W_5)$$

4.2.2.2.3. Hyp-Cont

Assume

- $n_5 \leq n_4 - j - j' - 2$
- $W_6 \sqsupseteq^{pub} revokeTemp(W_4)$
- $ms_6 :_{n_5} revokeTemp(W_6)$
- $reg_6(\text{pc}) = c_{next}$, $reg_6(r_0) = reg_3(r_0)$, $reg_6(r_1) = reg_3(r_1)$, $reg(r_{env}) = c_{env}$

• reg_6 points to stack with $\emptyset$ used and ms''_{unused} unused

Show

$$(n_5, (reg_6, ms_6 \uplus ms_r \uplus \emptyset \uplus ms''_{unused})) \in \mathcal{O}(W_6)$$

Use the $\mathcal{O}$-anti-reduction lemma (Lemma 8) followed by the `scall` lemma (Lemma 58). Given ms'''_f, we know by the operational semantics and the fact that the program hasn't changed that

$$(reg_6, ms_6 \uplus ms_r \uplus ms''_{unused} \uplus ms'''_f) \rightarrow_k (reg_7, ms_6[x \mapsto 1] \uplus ms_r \uplus ms''_{unused} \uplus ms'''_f)$$

where

- $(reg_7, ms_6[x \mapsto 1])$ is looking at `scall` $r([], [r_0, r_{env}])$ followed by c'_{next} c'_{next} is the current pc capability but looking at `load` r_1 x.
- $reg_7(r_0, r_1, r_{env}, r_{stk}) = reg_6(r_0, r_1, r_{env}, r_{stk})$

In $revokeTemp(W_6)$, we don't know which state the ι_x region is in, but state 1 is reachable via a public transition, so let W_7 be $revokeTemp(W_6)$ with region i_2 in state 1. It follows easily that

$$ms_6[x \mapsto 1] :_{n_5-k} W_7$$

We continue the proof in item 5.

5. At this point, we apply the `scall` lemma, to get

$$(n_5 - k, (reg_7, ms_6[x \mapsto 1] \uplus ms_r \uplus ms'''_{unused})) \in \mathcal{O}(W_7)$$

show

5.1. $ms_6[x \mapsto 1] :_{n_5-k} revokeTemp(W_7)$, follows from $W_7 = revokeTemp(W_7)$.

5.2. Hyp-Callee: Goes like the first Hyp-Callee (4.2.2.2.2.).

5.3. Hyp-Cont

Assume:

- $n_6 \leq n_5 - k - 2$
- $W_8 \sqsupseteq^{pub} revokeTemp(W_7)$
- $ms_7 :_{n_6} revokeTemp(W_8)$
- $reg_8(r_0, r_{env}) = reg_7(r_0, r_{env})$
- $reg_8(\text{pc}) = c'_{next}$
- reg_8 points to stack with $\emptyset$ used and $ms^{(6)}_{unused}$ unused for some $ms^{(6)}_{unused}$

Show:

$$\left(n_6, (reg_8, ms_7 \uplus ms_r \uplus \emptyset \uplus ms^{(5)}_{unused})\right) \in \mathcal{O}(W_8)$$

Use Lemma 8. Let $ms^{(4)}_f$ be given, then

$$(reg_8, ms_7 \uplus ms_r \uplus \emptyset \uplus ms^{(5)}_{unused} \uplus ms^{(4)}_f) \rightarrow_l (reg_9, ms_7 \uplus ms_r \uplus \emptyset \uplus ms_0 \uplus ms^{(4)}_f)$$

where

- $reg_9(\text{pc}) = updatePcPerm(reg_3(r_0))$ (note $reg_8(r_0) = reg_3(r_0)$)
- $reg_9(r_0) = reg_3(r_0)$
- For all $r \notin \{\text{pc}, r_0\}$, $reg_9(r) = 0$.

- $\mathrm{dom}(ms_0) = \mathrm{dom}(ms^{(5)}_{unused})$ and $\forall a \in \mathrm{dom}(ms_0).\, ms_0(a) = 0$

The execution proceeds as above because ι_x in W_8 is in state 1, so $ms_7(x) = 1$ which causes the assertion to succeed. Subsequently the stack and most of the registers are cleared.

Now take W_{10} to be W_9 with all the regions in $\mathrm{dom}(\lfloor W_3 \rfloor_{\{\mathrm{temp}\}})$ reinstated. Now we show the following:

5.3.1. $W_{10} \sqsupseteq^{pub} W_3$
We have

$$\forall r \in \mathrm{dom}(W_3).\, W_3(r) = W_{10}(r)$$

if the region was permanent in W_3, then it is there because $W_{10} \sqsupseteq^{priv} W_3$. If it was temporary, then it is there because it was just reinstated. If it was revoked in W_3, then it is still there because the only reinstated region were the temporary ones in W_3.
All the future worlds we have been given have been public, so the regions can only have made public transitions. In W_3 region ι_x is in state 0 or 1. In W_{10} region ι_x is in state 1. State 1 can be reached from 0 and 1 using a public transition, so the ι_x in W_{10} is a public future region of the ι_x in W_3.
In other words, all the regions in W_3 have only taken public transitions compared to the corresponding regions in W_{10}.
The relation between the relevant worlds is sketched out in Figure 4.5.

5.3.2. $ms_7 \uplus ms_r \uplus \emptyset \uplus ms_0 :_{n_6 - l} W_{10}$
First notice that from

- $(n_4, reg_3) \in \mathcal{R}(W_3)$
- $ms_3 :_{n_4} W_3$
- $reg(r_{stk}).perm = \mathrm{RWLX}$

using Lemma 9 we get that there exists a region, r_{advstk} such that

$$W_3(r_{advstk}) \overset{n}{=} \iota^{pwl}_{stk_a, stk_b}$$

and $\mathrm{dom}(ms_{unused}) \subseteq [stk_a, stk_b]$. Now take $ms_{advstk} = ms_r|_{[stk_a, stk_b]}$ (notice this not all of $[stk_a, stk_b]$ is in the domain of ms_r). We know

$$ms_7 :_{n_6} revokeTemp(W_8) \tag{10}$$

and

$$ms_3 :_{n_4} W_3 \tag{11}$$

which gives us two partitions say P_8 and P_3 respectively. Now define the partition P as follows:

$$P(r) = \begin{cases} P_8(r) & r \in \mathrm{dom}(\lfloor W_8 \rfloor_{\{\mathrm{perm}\}}) \\ ms_{advstk} \uplus ms_0 & r = r_{advstk} \\ P_3(r) & \text{otherwise} \end{cases}$$

Now let $r \in \mathrm{fl}W$, $n_7 < n_6 - l$, and $W(r) = (\_, s, \_, \_, H)$ and show

$$(n_7, P(r)) \in H(s)(\xi^{-1}(W_{10})).$$

Consider the following cases

5.3.2.1. $r \in \mathrm{dom}(\lfloor W_8 \rfloor_{\{\mathrm{perm}\}})$
Use 10, the fact that $W_{10} \sqsupseteq^{priv} revokeTemp(W_8)$ and that permanent regions respect future private world.

5.3.2.2. $r = r_{advstk}$
In this case we know the region is $\iota^{pwl}_{stk_a,stk_b}$, so we need to show

$$(n_7, ms_{advstk} \uplus ms_0) \in H^{pwl}_{stk_a,stk_b}(1)(\xi^{-1}(W_{10}))$$

which amounts to showing

$$\forall a \in \mathrm{dom}(ms_0).\, (n_7 - 1, ms_0(a)) \in \mathcal{V}(W_{10}),$$

which is trivial, and

$$\forall a \in \mathrm{dom}(ms_{advstk}).\, (n_7 - 1, ms_{advstk}(a)) \in \mathcal{V}(W_{10})$$

here we use that 11 entails

$$\forall a \in \mathrm{dom}(ms_{advstk}).\, (n_4 - 1, ms_{advstk}(a)) \in \mathcal{V}(W_3)$$

and the fact that $\mathcal{V}$ is monotone w.r.t $\sqsupseteq^{pub}$, $W_{10} \sqsupseteq^{pub} W_3$, and $\mathcal{V}(W_{10})$ is downwards-closed.

5.3.2.3. otherwise
Use 11, $W_{10} \sqsupseteq^{pub} W_3$, and the fact that for a temporary region $H(s)$ is monotone w.r.t. $\sqsupseteq^{pub}$.

5.3.3. $(n_6 - l, reg_9) \in \mathcal{R}(W_{10})$
Most registers are cleared. The only interesting register is r_0, so show:

$$(n_6 - l, reg_9(r_0)) \in \mathcal{V}(W_{10})$$

This follows from $reg_9(r_0) = reg_3(r_0)$, $(n_4, reg_3) \in \mathcal{R}(W_3)$, $\mathcal{V}$ monotone w.r.t $\sqsupseteq^{pub}$, $W_{10} \sqsupseteq^{pub} W_3$.

As we were using Lemma 8, we need to show

$$(n_6 - l, (reg_9, ms_7 \uplus ms_r \uplus \emptyset \uplus ms_0)) \in \mathcal{O}(W_{10})$$

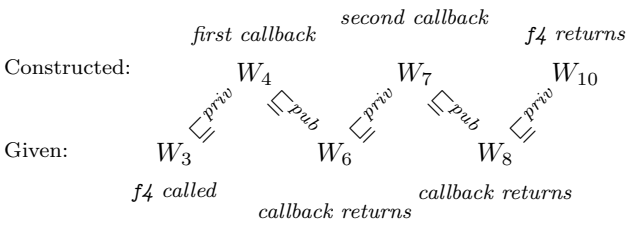
To this end the use $reg_3(r_0) = reg_9(r_0)$ and $(n_4, reg_3(r_0)) \in \mathcal{V}1(W_3)$. Assuming that $reg_9(r_0).perm \in \{\mathrm{E}, \mathrm{RX}, \mathrm{RWX}, \mathrm{RWLX}\}$ (if this is not the case, then it is trivial to show the above as the execution fails), then either the *executeCondition* or the *enterCondition* hold for appropriate values. Now use that $n_6 - l < n_4$ and $W_{10} \sqsupseteq^{pub} W_3$ (5.3.1.)[4] to get

$$(n_6 - l, updatePcPerm(reg_9(r_0))) \in \mathcal{E}(W_{10})$$

now using 5.3.2. and 5.3.3., we get the desired result.

□

[4]We don't know whether the capability is local or global, but it does not matter as we have a public future world relation between the two worlds.



5 Logical Relation

5.1 Worlds

Assume a sufficiently large set of states State that at least contains the states used in this document.

Definition 3.

$$\text{Rels} = \{(\phi_{pub}, \phi) \in \mathcal{P}(\text{State}^2) \times \mathcal{P}(\text{State}^2) \mid \phi_{pub}, \phi \textit{ is reflexive and transitive and } \phi_{pub} \subseteq \phi\}$$

■

Theorem 1. *There exists a c.o.f.e.* Wor *and preorders* $\sqsupseteq^{priv}$ *and* $\sqsupseteq^{pub}$ *such that* $(\text{Wor}, \sqsupseteq^{priv})$ *and* $(\text{Wor}, \sqsupseteq^{pub})$ *are preordered c.o.f.e.'s and there exists an isomorphism* ξ *such that*

$$\begin{aligned}\xi : \text{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} (&\{\text{revoked}\} + \\ &\{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred}(\text{MemSegment}))) + \\ &\{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \text{UPred}(\text{MemSegment})))))\end{aligned}$$

and for $W, W' \in \text{Wor}$

$$W' \sqsupseteq^{priv} W \Leftrightarrow \xi(W') \sqsupseteq^{priv} \xi(W)$$

and

$$W' \sqsupseteq^{pub} W \Leftrightarrow \xi(W') \sqsupseteq^{pub} \xi(W)$$

■

We now define the regions to be

$$\begin{aligned}\text{Region} = &\{\text{revoked}\} \uplus \\ &\{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred}(\text{MemSegment}))) \uplus \\ &\{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \text{UPred}(\text{MemSegment})))\end{aligned}$$

Let $\iota.v$ be the projection of the view of a region.

And the worlds are

$$\text{World} = \text{RegionName} \xrightarrow{fin} \text{Region}$$

where $\text{RegionName} = \mathbb{N}$.

The two *private future* region relations satisfies the following properties:

$$\frac{(s,s') \in \phi \qquad (v, \phi_{pub}, \phi, H) = (v', \phi'_{pub}, \phi', H')}{(v', s', \phi'_{pub}, \phi', H') \sqsupseteq^{priv} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{r \in \text{Region}}{r \sqsupseteq^{priv} (\text{temp}, s, \phi_{pub}, \phi, H)}$$

$$\frac{r \in \text{Region}}{r \sqsupseteq^{priv} \text{revoked}}$$

The two *public future* region relations satisfies the following properties:

$$\frac{(s,s') \in \phi_{pub} \qquad (v, \phi_{pub}, \phi, H) = (v', \phi'_{pub}, \phi', H')}{(v', s', \phi'_{pub}, \phi', H') \sqsupseteq^{pub} (v, s, \phi_{pub}, \phi, H)} \qquad \frac{(\text{temp}, s, \phi_{pub}, \phi, H) \in \text{Region}}{(\text{temp}, s, \phi_{pub}, \phi, H) \sqsupseteq^{pub} \text{revoked}}$$

$$\frac{}{\text{revoked} \sqsupseteq^{pub} \text{revoked}}$$

The two future world relations satisfy the following properties: They allow for any extension of the current world and all existing worlds are allowed to move to an appropriate future region. That is

$$\frac{\text{dom}(W') \supseteq \text{dom}(W) \qquad \forall r \in \text{dom}(W).\, W'(r) \sqsupseteq^{pub} W(r)}{W' \sqsupseteq^{pub} W}$$

$$\frac{\text{dom}(W') \supseteq \text{dom}(W) \qquad \forall r \in \text{dom}(W).\, W'(r) \sqsupseteq^{priv} W(r)}{W' \sqsupseteq^{priv} W}$$

Proof of Theorem 1. The theorem follows from a more general solution theorem for the category of P preordered c.o.f.e.'s, see Birkedal et al. [2010], Birkedal and Bizjak [2014] and Bizjak [2017]. We define two functors F_1 and F_2 from $P^{op} \times P^{op}$ to P.

$$\begin{aligned}
&F_1((X, \sqsupseteq^{priv\prime}), (Y, \sqsupseteq^{pub\prime})) = \\
&\qquad (\blacktriangleright(\mathbb{N} \xrightarrow{fin} (\{\text{revoked}\} + \\
&\qquad\qquad \{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((Y, \sqsupseteq^{pub\prime}) \xrightarrow{mon,\ ne} \text{UPred}(\text{MemSegment}))) + \\
&\qquad\qquad \{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((X, \sqsupseteq^{priv\prime}) \xrightarrow{mon,\ ne} \text{UPred}(\text{MemSegment})))))), \sqsupseteq^{priv})
\end{aligned}$$

and

$$\begin{aligned}
&F_2((X, \sqsupseteq^{priv\prime}), (Y, \sqsupseteq^{pub\prime})) = \\
&\qquad (\blacktriangleright(\mathbb{N} \xrightarrow{fin} (\{\text{revoked}\} + \\
&\qquad\qquad \{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((Y, \sqsupseteq^{pub\prime}) \xrightarrow{mon,\ ne} \text{UPred}(\text{MemSegment}))) + \\
&\qquad\qquad \{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((X, \sqsupseteq^{priv\prime}) \xrightarrow{mon,\ ne} \text{UPred}(\text{MemSegment})))))), \sqsupseteq^{pub})
\end{aligned}$$

The orderings $\sqsupseteq^{priv}$ and $\sqsupseteq^{pub}$ used in the definition of F_1 and F_2 are defined by the properties given above. Note that the image of F_1 and F_2 only differ in the ordering relation, i.e., letting U denote the forgetful functor from the category of preordered c.o.f.e.'s to the category of c.o.f.e.'s, we have $U \circ F_1 = U \circ F_2$. From Bizjak [2017] it then follows that there exists a c.o.f.e. Wor and two preorderings $\sqsupseteq^{priv}$ and $\sqsupseteq^{pub}$ and an isomorphism ξ satisfying the properties claimed in theorem. (Here, in the proof, we have written the ordering explicitly on the c.o.f.e. when using monotone non-expansive functions; in the theorem formulation we have instead annotated the arrow to indicate which ordering is used.) □

Erase all but a set of views:

$$\lfloor W \rfloor_S \stackrel{def}{=} \lambda r. \begin{cases} W(r) & W(r).v \in S \\ \bot & \text{otherwise} \end{cases}$$

Define the function $active(\cdot)$ as follows

$$active : \text{World} \to 2^{\text{RegionName}}$$
$$active(W) \stackrel{def}{=} \text{dom}(\lfloor W \rfloor_{\{\text{perm,temp}\}})$$

Memory segment satisfaction:

$$ms :_n W \text{ iff } \begin{cases} \exists P : active(W) \to \text{MemSegment}. \\ \quad ms :_{n,P} W \end{cases}$$

$$ms :_{n,P} W \text{ iff } \begin{cases} ms = \biguplus_{r \in active(W)} P(r) \wedge \\ \forall r \in active(W). \\ \quad \exists H, s. \\ \qquad W(r) = (\_, s, \_, \_, H) \wedge \\ \qquad (n, P(r)) \in H(s)(\xi^{-1}(W)) \end{cases}$$

Standard regions for when writing locally is permitted:

$$\iota^{pwl} : \mathcal{P} \to \text{Region}$$
$$\iota^{pwl}\ A \stackrel{def}{=} (\text{temp}, 1, =, =, H^{pwl}\ A)$$

$$H^{pwl} : \mathcal{P}(\text{Addr}) \to \text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred}(\text{MemSegment}))$$
$$H^{pwl}\ A\ s\ \hat{W} \stackrel{def}{=} \left\{ (n, ms) \middle| \begin{array}{l} \text{dom}(ms) = A \wedge \\ \forall a \in A.\, (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\}$$

Revoking all temporary regions:

$$revokeTemp : \text{World} \to \text{World}$$
$$revokeTemp(W) \stackrel{def}{=} \lambda r. \begin{cases} \text{revoked} & \text{if } W(r) = (\text{temp}, s, \phi_{pub}, \phi, H) \\ W(r) & \text{otherwise} \end{cases}$$

Further define

$$\iota^{pwl}_{base,end} \stackrel{def}{=} \iota^{pwl}([base, end])$$

Standard regions for when write local is not allowed:

$$\iota^{nwl} : \mathcal{P}(\text{Addr}) \to \text{Region}$$
$$\iota^{nwl}\ A \stackrel{def}{=} (\text{temp}, 1, =, =, H^{nwl}\ A)$$

$$\iota^{nwl,p} : \mathcal{P}(\text{Addr}) \to \text{Region}$$
$$\iota^{nwl,p}\ A \stackrel{def}{=} (\text{perm}, 1, =, =, H^{nwl}\ A)$$

$$H^{nwl} : \mathcal{P}(\text{Addr}) \to \text{State} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \text{UPred}(\text{MemSegment}))$$
$$H^{nwl}\ A\ s\ \hat{W} \stackrel{def}{=} \left\{ (n, ms) \;\middle|\; \begin{array}{l} \text{dom}(ms) = A \wedge \\ \forall a \in A. \\ \quad ms(a) \text{ is non-local} \wedge \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\}$$

Further define

$$\iota^{nwl}_{base,end} \stackrel{def}{=} \iota^{nwl}([base, end])$$
$$\iota^{nwl,p}_{base,end} \stackrel{def}{=} \iota^{nwl,p}([base, end])$$

For convenience define

$$\mathit{localityReg}(g, W) \stackrel{def}{=} \begin{cases} \text{dom}(\lfloor W \rfloor_{\{perm,\text{temp}\}}) & \text{if } g = \textsc{local} \\ \text{dom}(\lfloor W \rfloor_{\{perm\}}) & \text{if } g = \textsc{global} \end{cases}$$

$\mathit{localityReg}(\textsc{local}, W)$ are the regions that local capabilities may govern - that is permanent and temporary regions. $\mathit{localityReg}(\textsc{global}, W)$ are the regions that global capabilities may govern - that is permanent regions. Now define the following function

We need a notion of subset between regions that is almost n-subset, but not quite. The only difference is that the view part of a region is disregarded. Define "semi n-subset" and "semi n-supset" as:

$$\frac{(s, \phi_{pub}, \phi) = (s', \phi'_{pub}, \phi') \qquad \forall \hat{W}.\ H\ s\ \hat{W} \overset{n}{\subseteq} H'\ s'\ \hat{W}}{(v, s, \phi_{pub}, \phi, H) \overset{n}{\lesssim} (v', s', \phi'_{pub}, \phi', H')}$$

5.2 The logical relation

The logical relation is defined by several mutual recursive definitions. In order to handle this mutual recursion and show that this definitions are well-defined, Banach's fixed-point theorem can be used. We have omitted the details of this construction here, but it is done by parameterising all the definitions by the value relation.

$$
\begin{array}{l}
\iota = (v, s, \phi_{pub}, \phi, H) \text{ is address-stratified iff} \\
\quad \forall s', \hat{W}, n, ms, ms'. \\
\qquad\qquad (n, ms), (n, ms') \in H\ s'\ \hat{W} \Rightarrow \\
\qquad\qquad\quad \mathrm{dom}(ms) = \mathrm{dom}(ms') \wedge \\
\qquad\quad \forall a \in \mathrm{dom}(ms).\, (n, ms[a \mapsto ms'(a)]) \in H\ s'\ \hat{W}
\end{array}
$$

$$
\begin{array}{l}
\mathit{writeCondition} : (((\mathrm{Addr} \times \mathrm{Addr}) \to \mathrm{Region}) \times \mathrm{Global}) \to \mathrm{World} \xrightarrow{\mathit{mon},\ \mathit{ne}} \mathrm{UPred}(\mathrm{Addr}^2) \\
\mathit{writeCondition}(\iota, g)(W) = \\
\quad \{(n, (\mathit{base}, \mathit{end})) \mid \exists r \in \mathit{localityReg}(g, W). \\
\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
\qquad\qquad\quad W(r) \overset{n-1}{\gtrsim} \iota_{\mathit{base}', \mathit{end}'} \text{ and} \\
\qquad\qquad\quad W(r) \text{ is address-stratified } \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{readCondition} : \mathrm{Global} \to \mathrm{World} \xrightarrow{\mathit{mon},\ \mathit{ne}} \mathrm{UPred}(\mathrm{Addr}^2) \\
\mathit{readCondition}(g)(W) = \\
\quad \{(n, (\mathit{base}, \mathit{end})) \mid \exists r \in \mathit{localityReg}(g, W). \\
\qquad\qquad\qquad \exists [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}]. \\
\qquad\qquad\qquad\quad W(r) \overset{n}{\lessapprox} \iota^{\mathit{pwl}}_{\mathit{base}', \mathit{end}'} \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{executeCondition}(g)(W) = \\
\quad \{(n, (\mathit{perm}, \mathit{base}, \mathit{end})) \mid \forall n' < n. \\
\qquad\qquad\qquad\qquad \forall W' \sqsupseteq W. \\
\qquad\qquad\qquad\qquad\quad \forall a \in [\mathit{base}', \mathit{end}'] \subseteq [\mathit{base}, \mathit{end}]. \\
\qquad\qquad\qquad\qquad\qquad \big(n', ((\mathit{perm}, g), \mathit{base}', \mathit{end}', a)\big) \in \mathcal{E}(W')\} \\
\quad \text{where } g = \textsc{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub} \\
\quad \text{and } g = \textsc{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}
\end{array}
$$

$$
\begin{array}{l}
\mathit{enterCondition}(g)(W) = \\
\quad \{(n, (\mathit{base}, \mathit{end}, a)) \mid \forall n' < n. \\
\qquad\qquad\qquad\qquad \forall W' \sqsupseteq W. \\
\qquad\qquad\qquad\qquad\quad (n', ((\textsc{rx}, g), \mathit{base}, \mathit{end}, a)) \in \mathcal{E}(W')\} \\
\quad \text{where } g = \textsc{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub} \\
\quad \text{and } g = \textsc{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}
\end{array}
$$

Now define the value relation as follows:

$$
\begin{aligned}
&\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred(Word)}\\
&\mathcal{V} \stackrel{def}{=} \lambda\, W.\, \{(n,i) \mid i \in \mathbb{Z} \cup \{\infty\}\} \cup\\
&\qquad \{(n,((\textsc{o}, g), base, end, a))\} \cup\\
&\qquad \{(n,((\textsc{ro}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W)\} \cup\\
&\qquad \{(n,((\textsc{rw}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W) \wedge\\
&\qquad\quad (n,(base, end)) \in writeCondition(\iota^{nwl}, g)(W)\} \cup\\
&\qquad \{(n,((\textsc{rwl}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W) \wedge\\
&\qquad\quad (n,(base, end)) \in writeCondition(\iota^{pwl}, g)(W)\} \cup\\
&\qquad \{(n,((\textsc{rx}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rx}, base, end)) \in executeCondition(g)(W)\} \cup\\
&\qquad \{(n,((\textsc{e}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end, a)) \in enterCondition(g)(W)\} \cup\\
&\qquad \{(n,((\textsc{rwx}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W) \wedge\\
&\qquad\quad (n,(base, end)) \in writeCondition(\iota^{nwl}, g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rwx}, base, end)) \in executeCondition(g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rx}, base, end)) \in executeCondition(g)(W)\} \cup\\
&\qquad \{(n,((\textsc{rwlx}, g), base, end, a)) \mid\\
&\qquad\quad (n,(base, end)) \in readCondition(g)(W) \wedge\\
&\qquad\quad (n,(base, end)) \in writeCondition(\iota^{pwl}, g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rwlx}, base, end)) \in executeCondition(g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rwx}, base, end)) \in executeCondition(g)(W) \wedge\\
&\qquad\quad (n,(\textsc{rx}, base, end)) \in executeCondition(g)(W)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{O} : \text{World} \xrightarrow{ne} \text{UPred(Reg} \times \text{MemSegment)}\\
&\mathcal{O} \stackrel{def}{=} \lambda W.\, \{(n,(reg, ms)) \mid \forall ms_f, mem', i \le n.\\
&\qquad\qquad (reg, ms \uplus ms_f) \rightarrow_i (\mathit{halted}, mem')\\
&\qquad\qquad\quad \Rightarrow \exists W' \sqsupseteq^{priv} W.\, \exists ms_r, ms'.\\
&\qquad\qquad\qquad mem' = ms' \uplus ms_r \uplus ms_f \wedge\\
&\qquad\qquad\qquad ms' :_{n-i} W'\}
\end{aligned}
$$

$$\mathcal{R} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred(Reg)}$$

$$\mathcal{R} \stackrel{def}{=} \lambda W.\, \{(n, reg) \mid \forall r \in \text{RegisterName} \setminus \{\text{pc}\}.\ (n, reg(r)) \in \mathcal{V}(W)\}$$

$$\mathcal{E} : \text{World} \xrightarrow{ne} \text{UPred(Word)}$$

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

5.3 Useful regions

Static region used for parts of memory that should not change.

$$\iota^{sta}(v, ms) = (v, 1, =, =, H^{sta}\ ms)$$

$$H^{sta}\ ms\ s\ \hat{W} = \{(n, ms) \mid n > 0\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}$$

Static region used for parts of memory that should not change and where you pass control to untrusted code.

$$\iota^{sta,u}(v, ms) = (v, 1, =, =, H^{sta,u}\ ms)$$

$$H^{sta,u}\ ms\ s\ \hat{W} = \left\{ (n, ms') \;\middle|\; \begin{array}{l} ms' = ms \wedge \\ \forall a \in \text{dom}(ms). \\ \quad ms(a) \text{ is non-local} \wedge \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}$$

$$\iota^{cnst}(v, n) = (v, 1, =, =, H^{cnst}\ n)$$

$$H^{cnst}\ n'\ s\ \hat{W} = \{(n, ms) \mid n > 0 \wedge \forall a \in \text{dom}(ms).\ ms(a) = n'\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}$$

5.4 Lemmas

5.4.1 Anti-reduction for the observation relation

Lemma 7 (Failing terms are in $\mathcal{O}$ and $\mathcal{E}$)**.** *If* $(reg, ms \uplus ms_f) \rightarrow_* \textit{failed}$ *for all* ms_f*, then* $(n, (reg, ms)) \in \mathcal{O}(W)$ *for any* W*.*

If $(reg[\text{pc} \mapsto w], ms) \rightarrow_* \textit{failed}$ *for all* reg, ms*, then* $(n, w) \in \mathcal{E}(W)$ *for any* W*.* ■

Proof. Follows from the definitions of $\mathcal{O}(W)$ and $\mathcal{E}(W)$ using an (omitted) determinacy result. □

Lemma 8 (Anti-reduction for $\mathcal{O}$)**.**

$$\begin{array}{l}\forall n, n', i, reg, reg', ms, ms', ms_r, W, W'.\\ \quad n' \geq n - i \wedge W' \sqsupseteq^{priv} W \wedge\\ \quad (\forall ms_f.\, (reg, ms \uplus ms_r \uplus ms_f) \rightarrow_i (reg', ms' \uplus ms_r \uplus ms_f)) \wedge\\ \quad (n', (reg', ms')) \in \mathcal{O}(W')\\ \qquad \Rightarrow (n, (reg, ms \uplus ms_r)) \in \mathcal{O}(W)\end{array}$$

■

Proof of Lemma 8. Assume

1. $n' \geq n - i$
2. $W_2 \sqsupseteq^{priv} W_1$
3. $\forall ms_f.\, (reg, ms \uplus ms_r \uplus ms_f) \rightarrow_i (reg', ms' \uplus ms_r \uplus ms_f)$
4. $(n', (reg', ms')) \in \mathcal{O}(W_2)$

Show

$$(n, (reg, ms \uplus ms_r)) \in \mathcal{O}(W_1)$$

To this end let ms_{frame}, m' and j be given and assume

$$(reg, ms \uplus ms_r \uplus ms_{frame}) \rightarrow_j (halted, m') \tag{12}$$

From 3. instantiated with ms_{frame} we know

$$(reg, ms \uplus ms_r \uplus ms_{frame}) \rightarrow_i (reg', ms' \uplus ms_r \uplus ms_{frame}) \tag{13}$$

Using 12 and 13, we get

$$(reg', ms' \uplus ms_r \uplus ms_{frame}) \rightarrow_{j-i} (halted, m')$$

Using this with 4. and $ms_r \uplus ms_{frame}$ as frame, we get $W_3 \sqsupseteq^{priv} W_2$, ms'' and ms_{rev} such that

5. $m' = ms'' \uplus ms_{rev} \uplus (ms_r \uplus ms_{frame})$
6. $ms'' :_{n'-(j-i)} W_3$

Now use $ms_r \uplus ms_{rev}$ as the "revoked" memory, ms'' as the memory that satisfies some invariants, and W_3 as the desired world, then 5. gives us the split and by downwards closure 6. gives us the desired memory satisfaction. □

5.4.2 Standard regions

Lemma 9. *For all W, base, end, n, ms if*

- $ms :_n W$
- $(n, ((perm, g), base, end, a)) \in \mathcal{V}(W)$
- $base \leq end$

- $perm \in \{\text{RWLX}, \text{RWX}\}$

then

$$\exists r, base', end'. [base, end] \subseteq [base', end'] \wedge W(r) \overset{n}{=} \iota^{pwl}_{base', end'}$$

■

Proof of Lemma 9. Assume

1. $(n, ((\text{RWLX}, g), b, e, a)) \in \mathcal{V}(W)$
2. $ms :_n W$

From Assumption 1., we get r_1, r_2, b_1, b_2, e_1 and e_2 such that

3. $r_1 \in localityReg(g, W)$
4. $r_2 \in localityReg(g, W)$
5. $[b, e] \subseteq [b_1, e_1]$
6. $[b, e] \subseteq [b_2, e_2]$
7. $W(r_1) \overset{n}{\subseteq} \iota^{pwl}_{b_1, e_1}$
8. $W(r_2) \overset{n}{\supseteq} \iota^{pwl}_{b_2, e_2}$
9. $W(r_2)$ is address-stratified.

From Assumption 2., we get partitionen P s.t.

$$ms :_{n,p} W$$

Say $P(r_1) = ms_1$ and $P(r_2) = ms_2$. First from $(n, ms_1) \in W(r_1).H\ W(r_1).s\ \xi^{(-1)}(W)$ using , we get $(n, ms_1) \in H^{pwl}_{b_1, e_1}\ 1\ \xi^{(-1)}(W)$ which means $\text{dom}(ms_1) = [b_1, e_1]$.

Second we know $(n, [b_2 \mapsto 0, \ldots, e_2 \mapsto 0]) \in H^{pwl}_{b_2, e_2}\ 1\ \xi^{(-1)}(W)$ and $(n, ms_2) \in W(r_2).H\ W(r_2).s\ \xi^{(-1)}(W)$ which by Assumption 8. and 9. means $\text{dom}(ms_2) = [b_2, e_2]$.

Now assume for contradition $r_1 \neq r_2$, then we have a contradiction with $ms :_{n,p} W$ because ms_1 and ms_2 are not disjoint (by Assumptions 5. and 6.). So $r_1 = r_2$ which also means $[b_1, e_1] = [b_2, e_2]$, so from Assumption 5.4.2 and 8., we get $W(r_1) \overset{n}{\simeq} \iota^{pwl}_{b_1, e_1}$ which by Lemma 24 means $W(r_1) \overset{n}{=} \iota^{pwl}_{b_1, e_1}$ □

Lemma 10. *$H^{pwl}_{base, end}\ s$ is monotone w.r.t $\sqsupseteq^{pub}$ for all $s \in$ State and base and end* ■

Proof of Lemma 10. Let $\hat{W}' \sqsupseteq^{pub} \hat{W}$ be given and let

$$(n, ms) \in H^{pwl}_{base, end}\ s\ \hat{W} \tag{14}$$

and show

$$(n, ms) \in H^{pwl}_{base, end}\ s\ \hat{W}'$$

From 14, we get $\text{dom}(ms) = [base, end]$. Now let $a \in [base, end]$ be given and show

$$(n - 1, ms(a)) \in \mathcal{V}(\xi(\hat{W}'))$$

now this follows from Lemma 77, $\hat{W}' \sqsupseteq^{pub} \hat{W}$, Theorem 1, and Assumption 14. □

Lemma 11. *$\iota^{pwl}_{base,end}$ is a region for all base and end.* ■

Proof of Lemma 11. Follows from Lemma 10. □

Lemma 12. *$\iota^{pwl}_{base,end}$ is address-stratified.* ■

Proof. Easy unfolding of definitions. □

Lemma 13. *$H^{nwl}_{base,end}\ s$ is monotone w.r.t $\sqsupseteq^{priv}$ for all $s \in$ State and base and end* ■

Proof of Lemma 13. Let $\hat{W}' \sqsupseteq^{priv} \hat{W}$ be given and let

$$(n, ms) \in H^{nwl}_{base,end}\ s\ \hat{W} \tag{15}$$

and show

$$(n, ms) \in H^{nwl}_{base,end}\ s\ \hat{W}'$$

From 15, we get $\mathrm{dom}(ms) = [base, end]$. Now let $a \in [base, end]$ be given and show

1. $ms(a)$ is non-local
2. $(n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W}'))$

1. follows trivially from 15. 1. follows from Assumption 14, 1. (which we just argued), $\hat{W}' \sqsupseteq^{priv} \hat{W}$, Theorem 1, and Lemma 80. □

Lemma 14. *$\iota^{nwl}_{base,end}$ is a region for all base and end.* ■

Proof of Lemma 14. Follows from Lemma 13 and Lemma 71. □

Lemma 15. *$\iota^{nwl}_{base,end}$ is address-stratified.* ■

Proof. Easy unfolding of definitions. □

Lemma 16. *$\iota^{nwl,p}_{base,end}$ is a region for all base and end.* ■

Proof of Lemma 16. Follows from Lemma 13. □

Lemma 17. *$\iota^{sta}(v, ms)$ is a region for all $v \in \{\mathrm{perm}, \mathrm{temp}\}$ and ms.* ■

Proof of Lemma 17. H^{sta} does not depend on $\hat{W}$, so it is trivial to show the necessary non-expansive and monotonicity requirements. □

Lemma 18. *$H^{sta,u}(ms)\ s$ is monotone w.r.t $\sqsupseteq^{priv}$ for all $s \in$ State and ms.* ■

Proof of Lemma 18. Let $\hat{W}' \sqsupseteq^{priv} \hat{W}$ be given and let

$$(n, ms') \in H^{sta,u}(ms)\ s\ \hat{W} \tag{16}$$

and show

$$(n, ms') \in H^{sta,u}(ms)\ s\ \hat{W}'$$

From 16, we get $ms' = ms$. Now let $a \in \mathrm{dom}(ms)$ be given and show

1. $ms(a)$ is non-local
2. $(n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W}'))$

1. follows trivially from 16. 1. follows from Assumption 16, 1. (which we just argued), $\hat{W}' \sqsupseteq^{priv} \hat{W}$, Theorem 1 and Lemma 80. □

Lemma 19. *$\iota^{sta,u}(v, ms)$ is a region for all $v \in \{\text{perm}, \text{temp}\}$ and ms.* ■

Proof of Lemma 19. Follows from Lemma 18 and Lemma 71. □

Lemma 20.

$$H^{nwl}_{base,end}\ s\ \hat{W} \overset{n}{\subseteq} H^{pwl}_{base,end}\ s\ \hat{W}$$

■

Proof of Lemma 20. Trivial. Let

$$(n, ms) \in H^{nwl}_{base,end}\ s\ \hat{W}$$

and show

$$(n, ms) \in H^{pwl}_{base,end}\ s\ \hat{W}$$

From the assumption, we get $\text{dom}(ms) = [base, end]$. We further need to show

$$\forall a \in \text{dom}(ms).\, (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W}))$$

Given a, we know from the assumption that

$$(n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W}))$$

□

Lemma 21.

$$\forall n \in \mathbb{N}.\, \forall base, end \in \text{Addr}.$$
$$\iota^{nwl}_{base,end} \overset{n}{\lessapprox} \iota^{pwl}_{base,end}$$

■

Proof of Lemma 21. Let n, $base$, end be given and show

$$\iota^{nwl}_{base,end} \overset{n}{\lessapprox} \iota^{pwl}_{base,end}$$

They agree on the state and transition systems, so given $\hat{W}$ it suffices to show

$$H^{nwl}_{base,end}\ 1\ \hat{W} \overset{n}{\subseteq} H^{pwl}_{base,end}\ 1\ \hat{W}$$

which is true by Lemma 20. □

Lemma 22.

$$\forall n \in \mathbb{N}.\, \forall base, end \in \text{Addr}.$$
$$\iota^{nwl,p}_{base,end} \overset{n}{\lessapprox} \iota^{pwl}_{base,end}$$

■

Proof of Lemma 22. Follows from Lemma 20 (see proof of Lemma 21). □

Lemma 23.

$$\begin{aligned}&\forall n \in \mathbb{N}.\, \forall base, end \in \text{Addr}.\, \forall v \in \{\text{perm}, \text{temp}\}.\\ &\quad \text{dom}(ms) = [base, end] \Rightarrow\\ &\quad\quad \iota^{sta,u}_{base,end}(v, ms) \overset{n}{\lesssim} \iota^{pwl}_{base,end}\end{aligned}$$

■

Proof of Lemma 23. Essentially the same as the proof of Lemma 21 and Lemma 20. □

Lemma 24.

$$\begin{aligned}&\forall n \in \mathbb{N}.\, \forall base, end, b \in \text{Addr}.\, \forall \iota \in \text{Region}\\ &\iota \overset{n}{\simeq} \iota^{pwl}_{base,end} \wedge base \leq end \Rightarrow \iota \overset{n}{=} \iota^{pwl}_{base,end}\end{aligned}$$

■

Proof of Lemma 24. For $n = 0$ it is trivial, so assume $n > 0$. Say $\iota = (v, s, \phi_{pub}, \phi, H)$, then by $\overset{n}{\simeq}$, we know $s = 1$, $\phi_{pub} \equiv \phi \equiv =$, and $H = H^{pwl}_{base,end}$. It remains to show that $v = \text{temp}$. To do so, we show that it cannot be the case that $v = \text{perm}$. If $v = \text{perm}$, then H must be monotone with respect to $\sqsupseteq^{priv}$. If we can show that this is not the case, then for ι to be a region it must be the case that $v \neq \text{perm}$ and thus $v = \text{temp}$.

To this end let $b \notin [base, end]$ and define the worlds:

$$\begin{aligned}\xi(W) =&[0 \mapsto \iota^{pwl}_{base,end}]\\ &[1 \mapsto \iota^{pwl}_{b,b}]\\ \xi(W') =&[0 \mapsto \iota^{pwl}_{base,end}]\\ &[1 \mapsto \text{revoked}]\end{aligned}$$

For these two worlds, we have $\xi(W') \sqsupseteq^{priv} \xi(W)$ and from mono. of ξ^{-1}, we have $W' \sqsupseteq^{priv} W$. Now define the following memory segment:

$$ms = [base \mapsto ((\text{RO}, \text{LOCAL}), b, b, b), base + 1 \mapsto 0, \ldots, end \mapsto 0]$$

It is the case that

$$(n, ms) \in H\ 1\ W$$

but

$$(n, ms) \notin H\ 1\ W'$$

as it is not the case that

$$(n - 1, ((\text{RO}, \text{LOCAL}), b, b, b)) \in \mathcal{V}(\xi(W')).$$

The only other option that remains is $v = \text{temp}$. □

5.4.3 Observation relation

Lemma 25 (Observation relation ($\mathcal{O}$) non-expansive)**.**

$$W \overset{n}{=} W' \Rightarrow \mathcal{O}(W) \overset{n}{=} \mathcal{O}(W')$$

■

Proof of Lemma 25. □

5.4.4 Register-file relation

Lemma 26 (Register-file relation ($\mathcal{R}$) non-expansive)**.**

$$W \stackrel{n}{=} W' \Rightarrow \mathcal{R}(W) \stackrel{n}{=} \mathcal{R}(W')$$

■

Proof of Lemma 26. □

Lemma 27 (Register-file relation ($\mathcal{R}$) monotone wrt $\sqsupseteq^{pub}$)**.**

$$W' \sqsupseteq^{pub} W \Rightarrow \mathcal{R}(W') \stackrel{n}{\supseteq} \mathcal{R}(W)$$

■

Proof of Lemma 27. □

5.4.5 Expression relation

Lemma 28 (Expression relation ($\mathcal{E}$) non-exapansive)**.**

$$W \stackrel{n}{=} W' \Rightarrow \mathcal{E}(W) \stackrel{n}{=} \mathcal{E}(W')$$

■

Proof of Lemma 28. □

5.4.6 Permission based conditions

Lemma 29. *If*

$$(n, (base, end)) \in readCondition(g)(revokeTemp(W))$$

then

$$(n, (base, end)) \in readCondition(g)(W)$$

■

Proof of Lemma 29.

$$(n, (base, end)) \in readCondition(g)(revokeTemp(W))$$

Gives $r \in localityReg(g, revokeTemp(W))$ such that

$$\forall [base', end'] \subseteq [base, end].\, revokeTemp(W)(r) \stackrel{n}{\subseteq} \iota^{pwl}_{[base', end']}$$

Notice $revokeTemp(W)(r)$ is a perm region, so $revokeTemp(W)(r) = W(r)$. Using r as witness, the result is immediate. □

Lemma 30. *If*

$$(n, (base, end)) \in writeCondition(\iota, g)(revokeTemp(W))$$

then

$$(n, (base, end)) \in writeCondition(\iota, g)(W)$$

■

Proof of Lemma 30.

$$(n, (base, end)) \in writeCondition(\iota, g)(revokeTemp(W))$$

Gives $r \in localityReg(g, revokeTemp(W))$ such that

$$\forall [base', end'] \subseteq [base, end].\, revokeTemp(W)(r) \overset{n-1}{\gtrapprox} \iota_{[base', end']}$$

and

$$revokeTemp(W)(r) \text{ is address-stratified}$$

Notice $revokeTemp(W)(r)$ is a perm region, so $revokeTemp(W)(r) = W(r)$. Using r as witness, the result is immediate. □

Lemma 31. *If*

- $(n, (perm, base, end)) \in executeCondition(g)(revokeTemp(W))$

then

$$(n, (perm, base, end)) \in executeCondition(g)(W)$$

■

Proof of Lemma 31. Use Lemma 69. □

Lemma 32. *If*

- $(n, (a, base, end)) \in executeCondition(g)(revokeTemp(W))$

then

$$(n, (a, base, end)) \in executeCondition(g)(W)$$

■

Proof of Lemma 32. Use Lemma 69. □

Lemma 33. *If*

$$(n, (base, end)) \in writeCondition(\iota^{pwl}, \text{LOCAL})(W)$$

then

$$(n, (base, end)) \in writeCondition(\iota^{nwl}, \text{LOCAL})(W)$$

■

Proof of lemma 33. Follows from Lemma 22. □

Lemma 34 (*readCondition* monotone w.r.t $\sqsupseteq^{pub}$)**.** *If*

- $W' \sqsupseteq^{pub} W$
- $(n, (base, end)) \in readCondition(g)(W)$

then

$$(n, (base, end)) \in readCondition(g)(W')$$

■

Proof of Lemma 34. □

Lemma 35 (*readCondition* global monotonicity w.r.t $\sqsupseteq^{priv}$)**.** *If*

- $W' \sqsupseteq^{priv} W$
- $(n, (base, end)) \in readCondition(\text{GLOBAL})(W)$

then

$$(n, (base, end)) \in readCondition(\text{GLOBAL})(W')$$

■

Proof of Lemma 35. $readCondition(\text{GLOBAL})(W)$ picks a perm region from W. perm regions are persistent over $\sqsupseteq^{priv}$, so we can use the region that the assumption gives us. □

Lemma 36 (*readCondition* downwards-closed)**.** *If*

- $n' \leq n$
- $(n, (base, end)) \in readCondition(g)(W)$

then

$$(n', (base, end)) \in readCondition(g)(W)$$

■

Proof of Lemma 36. □

Lemma 37 (*writeCondition* monotone w.r.t $\sqsupseteq^{pub}$)**.** *If*

- $W' \sqsupseteq^{pub} W$
- $\iota \in \{\iota^{pwl}, \iota^{nwl}, \iota^{(nwl,p)}\}$
- $(n, (base, end)) \in writeCondition(\iota, g)(W)$

then

$$(n, (base, end)) \in writeCondition(\iota, g)(W')$$

■

Proof of Lemma 37. □

Lemma 38 (*writeCondition* global monotonicity w.r.t $\sqsupseteq^{priv}$)**.** *If*

- $W' \sqsupseteq^{priv} W$
- $\iota \in \{\iota^{nwl}, \iota^{(nwl,p)}\}$
- $(n, (base, end)) \in writeCondition(\iota, \text{GLOBAL})(W)$

then

$$(n, (base, end)) \in writeCondition(\iota, \text{GLOBAL})(W')$$

■

Proof of Lemma 38. $writeCondition(\iota, \text{GLOBAL})(W)$ picks a perm region from W. perm regions are persistent over $\sqsupseteq^{priv}$, so we can use the region that the assumption gives us. □

Lemma 39 (*writeCondition* downwards-closed)**.** *If*

- $n' \leq n$
- $\iota \in \{\iota^{pwl}, \iota^{nwl}, \iota^{(nwl,p)}\}$
- $(n, (base, end)) \in writeCondition(\iota, g)(W)$

then

$$(n', (base, end)) \in writeCondition(\iota, g)(W)$$

■

Proof of Lemma 39. □

Lemma 40 (*execCondition* monotone w.r.t $\sqsupseteq^{pub}$)**.** *If*

- $W' \sqsupseteq^{pub} W$
- $perm \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$
- $(n, (perm, base, end)) \in executeCondition(g)(W)$

then

$$(n, (perm, base, end)) \in executeCondition(\iota, g)(W')$$

■

Proof of Lemma 40. □

Lemma 41 (*execCondition* global monotonicity w.r.t $\sqsupseteq^{priv}$)**.** *If*

- $W' \sqsupseteq^{priv} W$
- $perm \in \{\text{RX}, \text{RWX}\}$
- $(n, (perm, base, end)) \in executeCondition(\text{GLOBAL})(W)$

then

$$(n, (perm, base, end)) \in executeCondition(\text{GLOBAL})(W')$$

■

Proof of Lemma 41. Assume $W_2 \sqsupseteq^{priv} W_1$, $perm \in \{\text{RX}, \text{RWX}\}$ and $(n, (perm, base, end)) \in executeCondition(\text{GLOBAL})(W_1)$. Now let $W_3 \sqsupseteq^{priv} W_2$, $a \in [base', end'] \subseteq [base, end]$, and $n' < n$, and show

$$\big(n, ((perm, \text{GLOBAL}), base', end', a)\big) \in \mathcal{E}(W_3)$$

by transitivity we have $W_3 \sqsupseteq^{priv} W_1$, so the result follows from $(n, (perm, base, end)) \in executeCondition(\text{GLOBAL})(W_1)$

□

Lemma 42 (*execCondition* downwards-closed)**.** *If*

- $n' \leq n$
- $perm \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$
- $(n, (perm, base, end)) \in executeCondition(g)(W)$

then

$$(n', (perm, base, end)) \in executeCondition(g)(W)$$

■

Proof of Lemma 42. Follows easily from definition. □

Lemma 43 (*enterCondition* monotone w.r.t $\sqsupseteq^{pub}$)**.** *If*

- $W' \sqsupseteq^{pub} W$
- $(n, (a, base, end)) \in enterCondition(g)(W)$

then

$$(n, (a, base, end)) \in enterCondition(\iota, g)(W')$$

■

Proof of Lemma 43. Follows easily from definition. □

Lemma 44 (*enterCondition* global monotonicity w.r.t $\sqsupseteq^{priv}$)**.** *If*

- $W' \sqsupseteq^{priv} W$
- $(n, (a, base, end)) \in enterCondition(\text{GLOBAL})(W)$

then

$$(n, (a, base, end)) \in enterCondition(\text{GLOBAL})(W')$$

■

Proof of Lemma 44. Assume $W_2 \sqsupseteq^{priv} W_1$ and $(n, (a, base, end)) \in enterCondition(\text{GLOBAL})(W_1)$. Now let $W_3 \sqsupseteq^{priv} W_2$, $n' < n$, and show

$$(n, ((\text{RX}, \text{GLOBAL}), base, end, a)) \in \mathcal{E}(W_3)$$

by transitivity we have $W_3 \sqsupseteq^{priv} W_1$, so the result follows from $(n, (a, base, end)) \in enterCondition(\text{GLOBAL})(W_1)$. □

Lemma 45 (*enterCondition* downwards-closed)**.** *If*

- $n' \leq n$
- $(n, (a, base, end)) \in enterCondition(g)(W)$

then

$$(n', (a, base, end)) \in enterCondition(g)(W)$$

■

Proof of Lemma 45. □

5.4.7 LR Sanity lemmas

Lemma 46.

$$\forall ms, n, W \stackrel{n}{=} W'.$$
$$ms :_n W \wedge W \stackrel{n}{=} W' \Rightarrow ms :_n W'$$

■

Proof of Lemma 46. □

Lemma 47 (Heap satisfaction downwards closure)**.**

$$\forall ms, n' \leq n, W.$$
$$ms :_n W \Rightarrow ms :_{n'} W$$

■

Proof of Lemma 47. Let ms, $n' \leq n$, and W be given and assume

$$ms :_n W$$

This assumption gives us $P : active(W) \to \text{MemSegment}$ such that

1. $ms = \biguplus_{r \in active(W)} P(r)$
2. $$\forall r \in active(W).$$
$$\exists H, s.$$
$$W(r) = (\_, s, \_, \_, H) \wedge$$
$$(n', P(r)) \in H(s)(\xi^{-1}(W))$$

Using P as witness, 1. is the first condition we need. Now let r be given and use 2. to get H and s such that

3. $W(r) = (\_, s, \_, \_, H)$
4. $(n, P(r)) \in H(s)(\xi^{-1}(W))$

We now need to show

$$(n', P(r)) \in H(s)(\xi^{-1}(W))$$

which follows from 4., $n' \leq n$, and $H(s)(\xi^{-1}(W))$ is a UPred(MemSegment). □

Lemma 48. *If*

- $ms :_n W$
- $(n, ((perm, g), base, end, a)) \in \mathcal{V}(W)$
- $base \leq end$
- $perm \in \{\text{RWLX}, \text{RWL}\}$

then

$$g = \text{LOCAL}$$

■

Proof of Lemma 48. It follows as a consequence of Lemma 9. The n-equality forces the region to be temp, so for the region name to be in $localityReg(g, W)$, the locality must be LOCAL. □

5.4.8 Malloc safe to pass to adversary

Lemma 49 (Safe values are safe to invoke.)**.** *If* $(n+1, w) \in \mathcal{V}(W)$*, then* $(n, \mathit{updatePcPerm}(w)) \in \mathcal{E}(W)$*.* ■

Proof. 1. Case $w = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a)$ and $\mathit{base} \leq a \leq \mathit{end}$ and $\mathit{perm} \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$:

1.1. $(n+1, (\mathit{perm}, \mathit{base}, \mathit{end})) \in \mathit{executeCondition}(g)(W)$.
By: definition of $\mathcal{V}(W)$ using the fact that $\mathit{perm} \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$.

1.2. $(n, ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a)) \in \mathcal{E}(W)$: By definition of *executeCondition* using the fact that $\mathit{base} \leq a \leq \mathit{end}$.

2. Case $w = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a)$ and $\mathit{base} \leq a \leq \mathit{end}$ and $\mathit{perm} = \text{E}$:

2.1. $(n+1, (\mathit{base}, \mathit{end}, a)) \in \mathit{enterCondition}(g)(W)$.
By: definition of $\mathcal{V}(W)$ using the fact that $\mathit{perm} = \text{E}$.

2.2. $(n, ((\text{RX}, g), \mathit{base}, \mathit{end}, a)) \in \mathcal{E}(W)$: By definition of *enterCondition* using the fact that $\mathit{base} \leq a \leq \mathit{end}$.

2.3. $\mathit{updatePcPerm}(w) = ((\text{RX}, g), \mathit{base}, \mathit{end}, a)$:
By definition of $\mathit{updatePcPerm}(\cdot)$

3. Otherwise: $(n, \mathit{updatePcPerm}(w)) \in \mathcal{E}(W)$:
By Lemma 7.

□

Lemma 50 (Malloc is safe to pass to adversary)**.** *For* c_{malloc} *that satisfies the specification for malloc with region* $\iota_{malloc,0}$*, if* $W(r) \sqsupseteq^{priv} \iota_{malloc,0}$*, then* $(n, c_{malloc}) \in \mathcal{V}(W)$ *for all* n*.* ■

Proof. 1. $c_{malloc} = ((\text{E}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)$.
By: the malloc specification (Specification 1).

2. Suffices: $(n, (\mathit{base}, \mathit{end}, a)) \in \mathit{enterCondition}(\text{GLOBAL})(W)$.
By definition of $\mathcal{V}(W)$.

3. Assume: $n' < n$, $W' \sqsupseteq^{priv} W$.
Suffices: $(n', ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)) \in \mathcal{E}(W')$.
By: definition of the *enterCondition*

4. Assume: $n'' \leq n'$, $(n'', \mathit{reg}) \in \mathcal{R}(W')$, $\mathit{ms} :_{n''} W'$
Suffices: $(n'', (\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], \mathit{ms})) \in \mathcal{O}(W')$
By: definition of $\mathcal{E}(W')$

5. Assume: $i < n''$, $(\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], \mathit{ms} \uplus \mathit{ms}_f) \rightarrow_i (\mathit{halted}, \mathit{mem}')$
Suffices: $\exists W''. W' \sqsupseteq^{priv} W', \mathit{ms}_r, \mathit{ms}'.\ \mathit{mem}' = \mathit{ms}' \uplus \mathit{ms}_r \uplus \mathit{ms}_f$ and $\mathit{ms}' :_{n''-i} W''$
By: definition of $\mathcal{O}(W')$

6. $W'(r) \sqsupseteq^{priv} \iota_{malloc,0}$
Easy from: $W' \sqsupseteq^{priv} W$ and $W(r) \sqsupseteq^{priv} \iota_{malloc,0}$ using transitivity of $\sqsupseteq^{priv}$.

7. $\exists P : \mathit{active}(W') \rightarrow \text{MemSegment}.\ \mathit{ms} :_{n'',P} W'$, i.e. $\mathit{ms} = \biguplus_{r \in \mathit{active}(W')} P(r)$ and $\forall r \in \mathit{active}(W').\ \exists H, s.\ W'(r) = (\_, s, \_, \_, H)$ and $(n'', P(r)) \in H(s)(\xi^{-1}(W'))$
By: definition of $\mathit{ms} :_{n''} W'$.

8. Define $ms_{frame} = \left(\biguplus_{r' \in active(W'), r' \neq r} P(r')\right) \uplus ms_f$. Then $ms \uplus ms_f = P(r) \uplus ms_{frame}$ and $(n'', P(r)) \in W'(r).H\ (W'(r).s)\ (\xi^{-1}(W'))$. Easy from the previous point.

9. $(n'', P(r)) \in W'(r).H\ (W'(r).s)\ (\xi^{-1}([r \mapsto W'(r)]))$, i.e. $P(r) :_{n''} [r \mapsto W'(r)]$.
By: the malloc specification (Specification 1) from the previous point.

10. Case: $reg(r_1) \in \mathbb{Z}$ and $reg(r_1) \geq 0$

    10.1. Define $size = reg(r_1)$

    10.2. $\exists \Phi' \in \text{ExecConf}, ms'_{footprint}, ms_{alloc} \in \text{MemSegment}, j \in \mathbb{N}, j > 0 \wedge b', e' \in \text{Addr}, \iota'_{malloc} \in \text{Region}.\ (reg[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), base, end, a)], ms \uplus ms_f) \rightarrow_j \Phi'$ and $\Phi'.\text{mem} = ms'_{footprint} \uplus ms_{alloc} \uplus ms_{frame}$ and $\iota'_{malloc} \sqsupseteq^{pub} W'(r)$ and $ms'_{footprint} :_{n''-j} [r \mapsto \iota'_{malloc}]$ and $\text{dom}(ms_{alloc}) = [b', e']$ and $\forall a \in [b', e'].\ ms_{alloc}(a) = 0$ and $\Phi'.\text{reg} = \Phi.\text{reg}[\text{pc} \mapsto updatePcPerm(w_{ret})][r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')]$ and $size - 1 = e' - b')$ with $w_{ret} = \Phi.\text{reg}(r_1)$.
    By: the malloc specification (Specification 1).

    10.3. Define $W'' = W'[r \mapsto \iota'_{malloc}][i \mapsto \iota^{nwl}_{b',e'}]$ for $i \notin \text{dom}(W')$. We have that $W'' \sqsupseteq^{pub} [r \mapsto \iota'_{malloc}]$ and $W'' \sqsupseteq^{pub} W'$.
    By: definition of $\sqsupseteq^{pub}$, using the fact that $\iota'_{malloc} \sqsupseteq^{pub} W(r)$.

    10.4. $\left(n''', (base', end')\right) \in readCondition(\text{GLOBAL})(W'')$ for all n''':
    By: definition of *readCondition*, using the region $W''(i)$ and Lemma 21.

    10.5. $\left(n''', (base', end')\right) \in writeCondition(\iota^{nwl}, \text{GLOBAL})(W'')$ for all n''':
    By: definition of *writeCondition*, using the region $W''(i)$.

    10.6. $\left(n''', (p, base', end')\right) \in executeCondition(\iota^{nwl}, \text{GLOBAL})(W'')$ for all $n''', p \in \{\text{RWX}, \text{RX}\}$:
    By: the definition of *executeCondition*, the FTLR (Theorem 2) using Lemmas 38, 35 and the previous two points.

    10.7. $(n'', ((\text{RWX}, \text{GLOBAL}), b', e', b')) \in \mathcal{V}(W'')$:
    By: definition of $\mathcal{V}(W'')$ and the above three points.

    10.8. $(n'' - j, \Phi.\text{reg}[r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')]) \in \mathcal{R}(W'')$:
    By Lemma 76, Lemma 27 using the fact that $W'' \sqsupseteq^{pub} W'$ and $(n'', \Phi.\text{reg}) \in \mathcal{V}(W')$, together with the previous point.

    10.9. $(n''', ms_{alloc}) \in \iota^{nwl}_{b',e'}.H\ \iota^{nwl}_{b',e'}.s\ W''$ for any n''':
    By definition of ι^{nwl}, H^{nwl} and $\mathcal{V}(\cdot)$ and the facts that $\text{dom}(ms_{alloc}) = [b', e']$ and $\forall a \in [b', e'].\ ms_{alloc}(a) = 0$.

    10.10. Define $ms' = \left(\biguplus_{r' \in active(W'), r' \neq r} P(r')\right) \uplus ms'_{footprint} \uplus ms_{alloc}$. Then $\Phi'.\text{mem} = ms' \uplus ms_f$ and $ms' :_{n''-j} W''$:
    By the facts that $\Phi'.\text{mem} = ms'_{footprint} \uplus ms_{alloc} \uplus ms_{frame}$, $ms_{frame} = \left(\biguplus_{r' \in active(W'), r' \neq r} P(r')\right) \uplus ms_f$, the previous point, the facts that $ms'_{footprint} :_{n''-j} [r \mapsto \iota'_{malloc}]$ and $W'' \sqsupseteq^{pub} [r \mapsto \iota'_{malloc}]$, the facts that $(\forall r \in active(W').\ \exists H, s.\ W'(r) =, (\_, s, \_, \_, H)$ and $(n'', P(r)) \in H(s)(\xi^{-1}(W')))$ and $W'' \sqsupseteq^{pub} W'$ and the public monotonicity and downwards closedness of all regions, and finally the definition of W''.

    10.11. $(n'' - j + 1, w_{ret}) \in \mathcal{V}(W'')$:
    By Lemma 77, the fact that $W'' \sqsupseteq^{pub} W'$, Lemma 75, and the fact that $(n'', w_{ret}) \in \mathcal{V}(W')$, which follows from $w_{ret} = \Phi.\text{reg}(r_1)$ and $(n'', reg) \in \mathcal{R}(W')$.

    10.12. $(n'' - j, updatePcPerm(w_{ret})) \in \mathcal{E}(W'')$:
    By Lemma 49 from the previous point.

10.13. $(n'' - j, (\Phi.\text{reg}[r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')][\text{pc} \mapsto \mathit{updatePcPerm}(w_{ret})], ms')) \in \mathcal{O}(W'')$:
By: definition of $\mathcal{E}(W'')$, using the previous point and the facts that $(n'' - j, \Phi.\text{reg}[r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')]) \in \mathcal{R}(W'')$, $ms' :_{n''-j} W''$

10.14. $i > j$ and $\Phi' \rightarrow_{i-j} (\mathit{halted}, mem')$.
By combining $(\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], ms \uplus ms_f) \rightarrow_i (\mathit{halted}, mem')$ with $(\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], ms \uplus ms_f) \rightarrow_j \Phi'$ using Lemma 1.

10.15. $\exists W'''\ W''', ms_r, ms''.\ mem' = ms'' \uplus ms_r \uplus ms_f$ and $ms'' :_{n-i} W'''$.
By: definition of $\mathcal{O}(W''')$ from the two previous points.

10.16. $W''' \sqsupseteq^{priv} W'$:
By Lemma 72, using the previous point and the fact that $W'' \sqsupseteq^{pub} W'$.

11. Case: $\mathit{reg}(r_1) \notin \mathbb{Z} \vee \mathit{reg}(r_1) < 0$

11.1. $\exists j.\ (\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], ms \uplus ms_f) \rightarrow_j \mathit{failed}$
By: the malloc specification (Specification 1).

11.2. Contradiction with $(\mathit{reg}[\text{pc} \mapsto ((\text{RX}, \text{GLOBAL}), \mathit{base}, \mathit{end}, a)], ms \uplus ms_f) \rightarrow_i (\mathit{halted}, mem')$

□

5.4.9 Fundamental theorem of logical relations

Lemma 51 (Conditions for load instruction are sufficient)**.** *If*

- $\Phi.\text{mem} :_n W$
- $c = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a)$
- $(n, c) \in \mathcal{V}(W)$
- $\mathit{readAllowed}(\mathit{perm})$
- $\mathit{withinBounds}(c)$

then $(n - 1, \Phi.\text{mem}(a)) \in \mathcal{V}(W)$ ■

Proof. 1. $(n, (\mathit{base}, \mathit{end})) \in \mathit{readCondition}(g)(W)$: follows by definition of $\mathcal{V}\cdot$ from $(n, c) \in \mathcal{V}(W)$.

2. $\exists r \in \mathit{localityReg}(g, W),\ [\mathit{base}', \mathit{end}'] \supseteq [\mathit{base}, \mathit{end}].\ W(r) \overset{n}{\lesssim} \iota^{pwl}_{\mathit{base}', \mathit{end}'}$. By definition of $\mathit{readCondition}(g)(W)$.

3. $\exists P : \mathit{active}(W) \rightarrow \text{MemSegment}.\ \Phi.\text{mem} :_{n,P} W$. By definition of $\Phi.\text{mem} :_n W$.

4. $\Phi.\text{mem} = \biguplus_{r \in \mathit{active}(W)} P(r)$ and $\forall r \in \mathit{active}(W), \exists H, s.\ W(r) = (\_, s, \_, \_, H)$ and $(n, P(r)) \in H(s)(\xi^{-1}(W))$. By definition of $\Phi.\text{mem} :_{n,P} W$.

5. $r \in \mathit{localityReg}(g, W) \subseteq \mathit{active}(W)$. By definition of $\mathit{localityReg}(\cdot)$ and $\mathit{active}(\cdot)$.

6. $\exists H, s.\ W(r) = (\_, s, \_, \_, H)$ and $(n, P(r)) \in H(s)(\xi^{-1}(W))$. By specializing the result from Step 4. to the r from Step 2..

7. $(n, P(r)) \in H^{pwl}_{\mathit{base}', \mathit{end}'}(s)(\xi^{-1}(W))$. Follows by combining $(n, P(r)) \in H(s)(\xi^{-1}(W))$ with $W(r) \overset{n}{\lesssim} \iota^{pwl}_{\mathit{base}', \mathit{end}'}$ from Step 2..

8. $\mathrm{dom}(P(r)) = [base', end']$ and for all $a' \in [base', end']$. $(n-1, P(r)(a')) \in \mathcal{V}(\xi(\xi^{-1}(W)))$. By definition of $H^{pwl}_{base',end'}$.

9. $a \in [base, end] \subseteq [base', end']$. By combining *withinBounds*(c) with the fact that $[base', end'] \supseteq [base, end]$. from Step 2..

10. In particular, we get: $\Phi.\mathrm{mem}(a) = P(r)(a)$ and $(n-1, P(r)(a)) \in \mathcal{V}(W)$.

□

Lemma 52 (Conditions for lea instruction are sufficient)**.** *If*

- $(n, ((perm, g), base, end, a)) \in \mathcal{V}(W)$
- $perm \neq \text{E}$

then $(n, ((perm, g), base, end, a')) \in \mathcal{V}(W)$ ■

Proof. Follows by inspection of the cases in the definition of $\mathcal{V}(W)$: a is ignored in all cases except where $perm = \text{E}$. □

Lemma 53 (pwl writecond implies nwl)**.** *If* $(n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W)$ *then* $(n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W)\}$. ■

Proof. 1. $\exists r \in localityReg(g, W).\ \exists [base', end'] \supseteq [base, end].\ W(r) \overset{n-1}{\gtrsim} \iota^{pwl}_{base',end'}$ and $W(r)$ is address-stratified: by definition of *writeCondition*.

2. Suffices: $W(r) \overset{n-1}{\gtrsim} \iota^{nwl}_{base',end'}$. By definition of *writeCondition*

3. $W(r) \overset{n-1}{\gtrsim} \iota^{pwl}_{base',end'} \overset{n-1}{\gtrsim} \iota^{nwl}_{base',end'}$: follows by Lemma 21.

□

Lemma 54 (execCond implies entryCond)**.** *If* $(n, (\text{RX}, base, end)) \in executeCondition(g)(W)$ *then* $(n, (base, end, a)) \in enterCondition(g)(W)$. ■

Proof. 1. Assume: $n' < n$, $W' \sqsupseteq W$ where $g = \text{LOCAL} \Rightarrow \sqsupseteq = \sqsupseteq^{pub}$ and $g = \text{GLOBAL} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$
Suffices: $(n', ((\text{RX}, g), base, end, a)) \in \mathcal{E}(W')$

2. Case $a \in [base, end]$: Follows from the definition of *executeCondition*.

3. Case $a \notin [base, end]$: Follows by Lemma 7.

□

Lemma 55 (Conditions for restrict instruction are sufficient)**.** *If*

- $(n, ((perm, g), base, end, a)) \in \mathcal{V}(W)$
- $(perm', g') \sqsubseteq (perm, g)$

then $(n, ((perm', g'), base, end, a)) \in \mathcal{V}(W)$ ■

Proof. By inspection of the definition of $\mathcal{V}(W)$, everything follows trivially except the following.

1. If $(n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W)$ then $(n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W)$: holds by lemma 53.

2. If $(n, (\text{RX}, base, end)) \in executeCondition(g)(W)$ then $(n, (base, end, a)) \in enterCondition(g)(W)$.

□

Lemma 56 (Conditions for subseg instruction are sufficient)**.** *If*

- $(n, ((perm, g), base, end, a)) \in \mathcal{V}(W)$
- $base \leq base'$
- $end' \leq end$
- $perm \neq \text{E}$

then $\big(n, ((perm, g), base', end', a)\big) \in \mathcal{V}(W)$ ■

Proof. Follows easily from the definitions of $\mathcal{V}(W)$, *readCondition*, *writeCondition*, *executeCondition*.

□

Lemma 57 (Conditions for store instruction are sufficient)**.** *If*

- $ms = ms' \uplus ms_f$
- $ms' :_n W$
- $((perm, g), base, end, a) = c$
- $(n, c) \in \mathcal{V}(W)$
- $writeAllowed(perm)$
- $withinBounds(c)$
- $(n, w) \in \mathcal{V}(W)$
- *if* $w = ((\_, \text{LOCAL}), \_, \_, \_)$*, then* $perm \in \{\text{RWLX}, \text{RWL}\}$

then $a \in \text{dom}(ms')$ *(i.e.* $ms[a \mapsto w] = ms'[a \mapsto w] \uplus ms_f$*) and* $ms'[a \mapsto w] :_n W$ ■

Proof. 1. $(n, (base, end)) \in writeCondition(\iota, g)(W)$ where $\iota = \iota^{pwl}$ or $\iota = \iota^{nwl}$ and (if $w = ((\_, \text{LOCAL}), \_, \_, \_)$, then $\iota = \iota^{pwl}$).

By definition of $\mathcal{V}(W)$ and *writeAllowed*, from $(n, c) \in \mathcal{V}(W)$, $((perm, g), base, end, a) = c$ and $writeAllowed(perm)$ and the fact that (if $w = ((\_, \text{LOCAL}), \_, \_, \_)$, then $perm \in \{\text{RWLX}, \text{RWL}\}$)

2. $\exists r \in localityReg(g, W).\ \exists [base', end'] \supseteq [base, end].\ W(r) \overset{n-1}{\gtrsim} \iota_{base', end'}$ and $W(r)$ is address-stratified. By definition of *writeCondition*.
3. $\exists P : active(W) \to \text{MemSegment}.\ ms' :_{n,P} W$. By definition of $ms' :_n W$.
4. $ms' = \biguplus_{r \in active(W)} P(r)$ and $\forall r \in active(W).\ \exists H, s.\ W(r) = (\_, s, \_, \_, H)$ and $(n, P(r)) \in H(s)(\xi^{-1}(W))$. By definition of $ms' :_{n,P} W$.
5. $\exists H, s.\ W(r) = (\_, s, \_, \_, H)$ and $(n, P(r)) \in H(s)(\xi^{-1}(W))$. By instantiating the previous point to the r from the *writeCondition*.

6. $(n, w) \in \iota.H\ (\iota.s)\ (\xi^{-1}(W))$ by definition of ι^{pwl}, ι^{nwl} and the fact that (if $w = ((\_, \text{LOCAL}), \_, \_, \_)$, then $\iota = \iota^{pwl}$).

7. Define ms'_w such that $\text{dom}(ms'_w) = [base', end']$, $ms'_w(a) = w$ and $ms'_w(a') = 0$ for $a' \neq a$. It's easy to show from the previous point that $(n, ms'_w) \in H(s)(\xi^{-1}(W))$.

8. $\text{dom}(P(r)) = \text{dom}(ms'_w) = [base', end'] \ni a$ and $(n, P(r)[a \mapsto w]) \in H(s)(\xi^{-1}(W))$ by applying the fact that $W(r)$ is address-stratified, combined with the previous point.

9. Define $P'(r) = P(r)[a \mapsto w]$ and $P'(r') = P(r')$ for $r' \neq r$.

10. $ms'[a \mapsto w] = \biguplus_{r \in active(W)} P'(r)$ and $ms'[a \mapsto w] :_{n,P'} W$. By definition of $ms' :_{n,P} W$ and the previous two points.

□

Theorem 2 (Fundamental theorem of logical relations)**.** *For all n, $perm$, $base$, end, a, g, W If one of the following holds:*

- $$perm = \text{RX} \wedge$$
$$(n, (base, end)) \in readCondition(g)(W)$$

- $$perm = \text{RWX} \wedge$$
$$(n, (base, end)) \in readCondition(g)(W) \wedge$$
$$(n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W)$$

- $$perm = \text{RWLX} \wedge$$
$$(n, (base, end)) \in readCondition(g)(W) \wedge$$
$$(n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W),$$

then

$$(n, ((perm, g), base, end, a)) \in \mathcal{E}(W)$$

■

Proof. 1. By induction on n. In other words, assume that the theorem already holds for all $n' < n$.

2. Assume: $n' \leq n$, $(n', reg) \in \mathcal{R}(W)$, $ms :_{n'} W$.
Suffices: $(n', (reg[\text{pc} \mapsto ((perm, g), base, end, a)], ms)) \in \mathcal{O}(W)$.
By: definition of $\mathcal{E}(W)$.

3. Assume: ms_f, mem', $i \leq n'$, $\Phi = (reg[\text{pc} \mapsto ((perm, g), base, end, a)], ms \uplus ms_f)$ and $\Phi \rightarrow_i (halted, mem')$,
Suffices: $\exists W' \sqsupseteq^{priv} W$, ms_r, ms'. $mem' = ms' \uplus ms_r \uplus ms_f$ and $ms' :_{n'-i} W'$
By: definition of $\mathcal{O}(W)$

4. $i \neq 0$, since $(reg[\text{pc} \mapsto ((perm, g), base, end, a)], ms \uplus ms_f) \neq (halted, mem')$ for any mem'. Therefore, assume w.l.o.g. that $i = 1 + i'$,

$$\Phi \rightarrow conf' \rightarrow_{i'} (halted, mem')$$

5. $n \geq n' > 0$, since otherwise $i = 0$ (because $i \leq n' \leq n$) and this is impossible by the previous point.

6. $(n', \Phi.\text{reg}(\text{pc})) \in \mathcal{V}(W)$. Proof:

   6.1. Assume: $perm' \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$ with $perm' \sqsubseteq perm$
   Suffices: $(n', (perm', base, end)) \in executeCondition(g)(W)$
   By: the definition of $\mathcal{V}(\cdot)$ using the assumptions

   6.2. Assume: $n'' < n'$, $W' \sqsupseteq W$, $a' \in [base, end]$, $g = \text{LOCAL} \Rightarrow \sqsupseteq = \sqsupseteq^{pub}$, $g = \text{GLOBAL} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$.
   Suffices: $(n'', ((perm, g), base, end, a')) \in \mathcal{E}(W')$. By: definition of $executeCondition(g)(W)$

   6.3. By induction, using the assumptions and Lemmas 36 and 39.

7. For all $r \in \text{RegisterName}$, $(n', \Phi.\text{reg}(r)) \in \mathcal{V}(W)$.

   7.1. Case $r \neq \text{pc}$: follows from $(n', reg) \in \mathcal{R}(W)$ by definition of $\mathcal{R}(W)$.

   7.2. Case $r = \text{pc}$: by step 6..

8. By inspection of the definitions of $\Phi \rightarrow conf'$ and $[\![decode(\Phi.\text{mem}(a))]\!]$ and $updatePcPerm(\cdot)$ and $updatePc(\cdot)$, it is easy to see that one of the following cases must hold:

9. Case $conf' = failed$: contradiction, since it is not possible that $failed \rightarrow_{i'} (halted, mem')$.

10. Case $conf' = (halted, mem)$:

    10.1. Then $i' = 0$ and $mem' = mem$
    Follows from $(halted, mem) \rightarrow_{i'} (halted, mem)$

    10.2. For $W' = W$, $ms_r = \emptyset$ and $ms' = ms$, we have that $mem = ms' \uplus ms_r \uplus ms_f$ and $ms' :_{n'-1} W'$ (using Lemma 47).

11. Case $conf' = \Phi''[\text{reg.pc} \mapsto newPc]$, and additionally, one of the following holds:

    - $\Phi''.\text{mem} = \Phi.\text{mem}$
    - $\Phi''.\text{mem} = \Phi.\text{mem}[a' \mapsto w]$, with $\Phi.\text{reg}(r_1) = ((perm', g'), base', end', a') = c$ and $writeAllowed(perm')$ and $withinBounds(c)$ and $w = \Phi.\text{reg}(r_2)$ and if $w = ((\_, \text{LOCAL}), \_, \_, \_)$, then $perm' \in \{\text{RWLX}, \text{RWL}\}$

    and also one of the following holds:

    - $newPc = updatePcPerm(\Phi.\text{reg}(lv))$
    - $newPc = ((perm', g'), base', end', a' + 1)$ and $\Phi.\text{reg}(\text{pc}) = ((perm', g'), base', end', a')$

    and finally, for all $r \in \text{RegisterName}$, one of the following holds:

    - $\Phi''.\text{reg}(r) = \Phi.\text{reg}(r)$
    - $\Phi''.\text{reg}(r) = z$ for some $z \in \mathbb{Z}$
    - $\Phi''.\text{reg}(r) = w$ and $\Phi.\text{reg}(r_2) = ((perm', g'), base', end', a') = c$ and $readAllowed(perm')$ and $withinBounds(c)$ and $w = \Phi.\text{mem}(a')$
    - $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r_1) = ((perm', g'), base', end', a')$ and $perm' \neq \text{E}$ and $c = ((perm', g'), base', end', a' + z)$ for some $z \in \mathbb{Z}$

- $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $(perm'', g'') \sqsubseteq (perm', g')$ and $c = ((perm'', g''), base', end', a')$
- $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $base' \leq base''$ and $end'' \leq end'$ and $c = ((perm', g'), base'', end'', a')$ and $perm' \neq \text{E}$

In this case, we have:

11.1. $\Phi''.\text{mem} = ms'' \uplus ms_f$ and $ms'' :_{n'-1} W$.

11.1.1. Case $\Phi''.\text{mem} = \Phi.\text{mem}$: Then $\Phi''.\text{mem} = ms \uplus ms_f$ and $ms :_{n'-1} W$ follows by Lemma 47.

11.1.2. Case $\Phi''.\text{mem} = \Phi.\text{mem}[a' \mapsto w]$, with $\Phi.\text{reg}(r_1) = ((perm', g'), base', end', a') = c$ and $writeAllowed(perm')$ and $withinBounds(c)$ and $w = \Phi.\text{reg}(r_2)$ and if $w = ((\_, \text{LOCAL}), \_, \_, \_)$, then $perm' \in \{\text{RWLX}, \text{RWL}\}$.
The facts that $\Phi''.\text{mem} = ms'' \uplus ms_f$ and $ms'' :_{n'-1} W$ follow by Lemmas 57 and 47 using the fact that $ms :_{n'} W$ and $(n', \Phi.\text{reg}(r_1)) \in \mathcal{V}(W)$ and $(n', \Phi.\text{reg}(r_2)) \in \mathcal{V}(W)$ which follows from Step 7..

11.2. For all $r \in \text{RegisterName}$, $(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$.

11.2.1. Case $\Phi''.\text{reg}(r) = \Phi.\text{reg}(r)$: $(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows from Step 7. using Lemma 75.

11.2.2. $\Phi''.\text{reg}(r) = z$ for some $z \in \mathbb{Z}$. $(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by definition of $\mathcal{V}(\cdot)$

11.2.3. $\Phi''.\text{reg}(r) = w$ and $\Phi.\text{reg}(r_2) = ((perm', g'), base', end', a') = c$ and $readAllowed(perm')$ and $withinBounds(c)$ and $w = \Phi.\text{mem}(a')$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 51 using the fact that $\Phi.\text{mem} :_{n'} W$ and $(n', \Phi.\text{reg}(r_2)) \in \mathcal{V}(W)$ which we have from step 7..

11.2.4. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r_1) = ((perm', g'), base', end', a')$ and $perm' \neq \text{E}$ and $c = ((perm', g'), base', end', a' + z)$ for some $z \in \mathbb{Z}$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 52 and 75 using the fact that $(n', \Phi.\text{reg}(r_1)) \in \mathcal{V}(W)$ which we have from step 7..

11.2.5. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $(perm'', g'') \sqsubseteq (perm', g')$ and $c = ((perm'', g''), base', end', a')$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 55 and 75 using the fact that $(n', \Phi.\text{reg}(r)) \in \mathcal{V}(W)$ which follows from $(n', \Phi.\text{reg}) \in \mathcal{R}(W)$ by definition.

11.2.6. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $base' \leq base''$ and $end'' \leq end'$ and $c = ((perm', g'), base'', end'', a')$ and $perm' \neq \text{E}$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 56 and 75 using the fact that $(n', \Phi.\text{reg}(r)) \in \mathcal{V}(W)$ which follows from $(n', \Phi.\text{reg}) \in \mathcal{R}(W)$ by definition.

11.3. $(n' - 1, \Phi''.\text{reg}) \in \mathcal{R}(W)$: Follows from the previous point by definition of $\mathcal{R}(W)$.

11.4. $(n' - 1, newPc) \in \mathcal{E}(W)$:

11.4.1. Case $newPc = updatePcPerm(\Phi.\text{reg}(lv))$: We distinguish the following cases:

11.4.1.1. Case $\Phi.\text{reg}(lv) = ((\text{E}, g'), base', end', a')$:

11.4.1.1.1. $(n', \Phi.\text{reg}(lv)) \in \mathcal{V}(W)$. Follows from Step 7..

11.4.1.1.2. $\big(n', (base', end', addr')\big) \in enterCondition(g')(W)$. By definition of $\mathcal{V}(W)$ from the previous point.

11.4.1.1.3. $\big(n' - 1, ((\text{RX}, g'), base', end', a')\big) \in \mathcal{E}(W)$: By definition of $enterCondition(\cdot)$ and taking $n' = n' - 1$ and $W' = W$

11.4.1.1.4. $updatePcPerm(\Phi.\text{reg}(lv)) = ((\text{RX}, g'), base', end', a')$: by definition of $updatePcPerm(\cdot)$.

11.4.1.2. Case $\Phi.\text{reg}(lv) = ((perm', g'), base', end', a')$ with $perm' \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$ and $withinBounds(\Phi.\text{reg}(lv))$:

11.4.1.2.1. $(n', \Phi.\text{reg}(lv)) \in \mathcal{V}(W)$. Follows from Step 7..

11.4.1.2.2. $\big(n', (perm', base', end', a')\big) \in executeCondition(g')(W)$. By definition of $\mathcal{V}(W)$ from the previous point.

11.4.1.2.3. $\big(n'-1, ((perm', g'), base', end', a')\big) \in \mathcal{E}(W)$: By definition of $executeCondition(\cdot)$, taking $n' = n'-1$, $W' = W$ and $a = a'$. Note that $a' \in [base', end']$ because we have $withinBounds(\Phi.\text{reg}(lv))$.

11.4.1.2.4. $updatePcPerm(\Phi.\text{reg}(lv)) = ((perm', g'), base', end', a')$: by definition of $updatePcPerm(\cdot)$.

11.4.1.3. Case not $(\Phi.\text{reg}(lv) = ((\text{E}, g'), base', end', a'))$ and not $(\Phi.\text{reg}(lv) = ((perm', g'), base', end', a')$ with $perm' \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$ and $withinBounds(\Phi.\text{reg}(lv)))$:

11.4.1.3.1. $updatePcPerm(\Phi.\text{reg}(lv)) = \Phi.\text{reg}(lv)$: by definition of $updatePcPerm(\cdot)$.

11.4.1.3.2. $(reg[\text{pc} \mapsto \Phi.\text{reg}(lv)], ms) \rightarrow failed$ for any reg,ms: by definition of the evaluation relation.

11.4.1.3.3. $(n'-1, newPc) \in \mathcal{E}(W)$: by Lemma 7 using the previous point.

11.4.2. Case $newPc = ((perm', g'), base', end', a'+1)$ and $\Phi''.\text{reg}(\text{pc}) = ((perm', g'), base', end', a')$:

11.4.2.1. Case $perm' \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$ and $base' \leq a'+1 \leq end'$:

11.4.2.1.1. $(n'-1, \Phi''.\text{reg}(\text{pc})) \in \mathcal{V}(W)$: by Step 11.2..

11.4.2.1.2. $\big(n'-1, ((perm', g'), base', end', a'+1)\big) \in \mathcal{V}(W)$: by Lemma 52 from the previous point.

11.4.2.1.3. One of the following holds:

- $$perm' = \text{RX} \wedge$$
  $$\big(n'-1, (base', end')\big) \in readCondition(g)(W)$$
- $$perm' = \text{RWX} \wedge$$
  $$\big(n'-1, (base', end')\big) \in readCondition(g)(W) \wedge$$
  $$\big(n'-1, (base', end')\big) \in writeCondition(\iota^{nwl}, g)(W)$$
- $$perm' = \text{RWLX} \wedge$$
  $$\big(n'-1, (base', end')\big) \in readCondition(g)(W) \wedge$$
  $$\big(n'-1, (base', end')\big) \in writeCondition(\iota^{pwl}, g)(W),$$

This follows from the previous point by definition of $\mathcal{V}(W)$

11.4.2.1.4. $\big(n'-1, ((perm', g'), base', end', a'+1)\big) \in \mathcal{E}(W)$: By the induction hypothesis of this lemma using the previous point.

11.4.2.2. Case not ($perm' \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$ and $base' \leq a'+1 \leq end'$): The result follows by Lemma 7.

11.5. $(n'-1, (\Phi''.\text{reg}[\text{pc} \mapsto newPc], ms'')) \in \mathcal{O}(W)$: by definition of $\mathcal{E}(W)$ using the above three points.

11.6. $\exists W' \sqsupseteq^{priv} W,\ ms_r,\ ms'.\ mem = ms' \uplus ms_r \uplus ms_f$ and $ms' :_{n'-i} W'$
By: definition of $\mathcal{O}(W)$ using the previous step and the evaluation $conf' \rightarrow_{i'} (halted, mem')$ from Step 4..

□

5.4.10 Scall macro-instruction correctness

Definition 4. *We say that* (reg, ms) *is looking at* $[i_0, \cdots, i_n]$ *followed by* c_{next} *iff*

- $reg(\text{pc}) = ((p, g), b, e, a)$
- $p = \text{RWX}$, $p = \text{RX}$, *or* $p = \text{RWLX}$
- $a + n \leq e$, $b \leq a \leq e$
- $ms(a + 0, \cdots, a + n) = [i_0, \cdots, i_n]$
- $c_{next} = ((p, g), b, e, a + n + 1)$

■

Definition 5. *We say that reg points to stack with* ms_{stk} *used and* ms_{unused} *unused iff*

- $reg(r_{stk}) = ((\text{RWLX}, \text{LOCAL}), b_{stk}, e_{stk}, a_{stk})$
- $\text{dom}(ms_{unused}) = [a_{stk} + 1, \cdots, e_{stk}]$
- $\text{dom}(ms_{stk}) = [b_{stk}, \cdots, a_{stk}]$
- $b_{stk} - 1 \leq a_{stk}$

■

Lemma 58 (`scall` works)**.** *If*

- $ms :_n revokeTemp(W)$
- $\text{dom}(ms_f) \cap (\text{dom}(ms_{stk} \uplus ms_{unused} \uplus ms)) = \emptyset$
- (reg, ms) *is looking at* `scall` $r(\overline{r_{arg}}, \overline{r_{priv}})$ *followed by* c_{next}
- *reg points to stack with* ms_{stk} *used and* ms_{unused} *unused*

Hyp-Callee If

- $\text{dom}(ms_{unused}) = \text{dom}(ms_{act} \uplus ms'_{unused})$,
- $W' = revokeTemp(W)[\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{act} \uplus ms_f), \iota^{pwl}(\text{dom}(ms'_{unused}))]$,
- $ms'' :_{n-1} W'$
- reg' *points to stack with* $\emptyset$ *used and* ms'_{unused} *unused*
- $reg' = reg_0[\text{pc} \mapsto updatePcPerm(reg(r)), \overline{r_{arg}} \mapsto reg(\overline{r_{arg}}), r_0 \mapsto c_{ret}, r_{stk} \mapsto c_{stk}, r \mapsto reg(r)]$
- $(n - 1, c_{ret}) \in \mathcal{V}(W')$
- $(n - 1, c_{stk}) \in \mathcal{V}(W')$

then we have that $(n - 1, (reg', ms'')) \in \mathcal{O}(W')$

Hyp-Cont If

- $n' \leq n - 2$
- $W'' \sqsupseteq^{pub} revokeTemp(W)$

– $ms'' :_{n'} revokeTemp(W'')$

– *for all* r, *we have that:*

$$reg'(r)\begin{cases} = c_{next} & \text{if } r = \mathrm{pc} \\ = reg(r) & \text{if } r \in \overline{r_{priv}} \\ \in \mathcal{V}(revokeTemp(W'')) & \text{if } reg'(r) \text{ is a global capability and } r \notin \{\mathrm{pc}, \overline{r_{priv}}, r_{stk}\} \end{cases}$$

– reg' *points to stack with* ms_{stk} *used and* ms''_{unused} *unused for some* ms''_{unused}

then we have that $(n', (reg', ms'' \uplus ms_f \uplus ms_{stk} \uplus ms''_{unused})) \in \mathcal{O}(W'')$

Then

- $(n, (reg, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{unused})) \in \mathcal{O}(W)$

■

Proof. Assume n is sufficiently large to execute all the steps up to and including the jump of `scall` $r(\overline{r_{arg}}, \overline{r_{priv}})$. If this is not the case, then in any given memory frame the execution will not halt successfully fast enough.

Further assume

1. $ms :_n revokeTemp(W)$
2. $\mathrm{dom}(ms_f) \cap (\mathrm{dom}(ms_{stk} \uplus ms_{unused} \uplus ms)) = \emptyset$
3. (reg, ms) is looking at `scall` $r(\overline{r_{arg}}, \overline{r_{priv}})$ followed by c_{next}
4. reg points to stack with ms_{stk} used and ms_{unused} unused
5. Hyp-Callee
6. Hyp-Cont

Now we wish to apply Lemma 8. To this end let ms_{frame} be given. Executing the `scall` gives us

$$(reg, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{unused} \uplus ms_{frame}) \rightarrow_i (reg_1, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused} \uplus ms_{frame})$$

where

7. $i \leq n$
8. ms_{act} contains activation record, $reg(\overline{r_{priv}})$, the code return capability, and the full stack capability ($reg(r_{stk})$ with the pointer adjusted).
9. $\forall a \in \mathrm{dom}(ms'_{unused}).\, ms'_{unused}(a) = 0$
10. $\mathrm{dom}(ms_{unused}) = \mathrm{dom}(ms_{act} \uplus ms'_{unused})$
11. $reg_1(r_0) = c_{ret} = ((\mathrm{E}, \mathrm{LOCAL}), \_, \_, \_)$ where the range of authority is the same as $reg(r_{stk})$ and it points to the first instruction of the activation code.
12. reg_1 points to stack with $\emptyset$ used and ms'_{unused} unused
13. $reg_1(\mathrm{pc}) = updatePcPerm(reg(\mathrm{pc}))$

14. $reg_1(r) = reg(r)$
15. $reg_1(\overline{r_{args}}) = reg(\overline{r_{args}})$
16. $\forall r' \in \text{RegisterName} \setminus \{\text{pc}, r_{stk}, r, \overline{r_{args}}\}.\ reg_1(r') = 0$

In order to use Lemma 8, we now need to show

$$(n_1, (reg_1, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused})) \in \mathcal{O}(W_1)$$

where

$$W_1 = revokeTemp(W)[\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{act} \uplus ms_f), \iota^{pwl}(\text{dom}(ms'_{unused}))]$$

to this end use Hyp-Callee (5.). To use this everything is satisfied directly by assumptions but the following:

17. $ms \uplus ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused} :_{n-1} W_1$
    Here we apply Lemma 66. By assumption 1. we have $ms :_n revokeTemp(W)$. So it suffices to show

    $$ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused} :_{n-1} [\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{act} \uplus ms_f), \iota^{pwl}(\text{dom}(ms'_{unused}))]$$

    This turns out to be trivial as ms_f, ms_{stk}, and ms_{act} match the static region. ms'_{unused} is all zeroes, to it trivially satisfies the ι^{pwl} region.
18. $(n-1, reg'(r_{stk})) \in \mathcal{V}(W_1)$
    Use Lemma 62 with 12. and that W_1 has region $\iota^{pwl}(\text{dom}(ms'_{unused})$.
19. $(n-1, c_{ret}) \in \mathcal{V}(W_1)$
    To this end let

    19.1. $n' < n - 1$
    19.2. $W_2 \sqsupseteq^{pub} W_1$

    be given and show

    $$(n', updatePcPerm(c_{ret})) \in \mathcal{E}(W_2)$$

    To this assume

    19.3. $n'' \leq n'$
    19.4. $(n'', reg_2) \in \mathcal{R}(W_2)$
    19.5. $ms' :_{n''} W_2$

    be given and show

    $$(n'', (reg_2[\text{pc} \mapsto updatePcPerm(c_{ret})], ms')) \in \mathcal{O}(W_2) \tag{17}$$

    From 19.2. and 19.5., we can deduce that the memory can be split in the following way:

    $$ms' = ms'' \uplus ms_r \uplus ms_{stk} \uplus ms_{act} \uplus ms''_{unused} \uplus ms_f$$

    where ms'' is the "permanent" part of memory we get from Lemma 63, ms_r is the part "revoked" of memory from the same lemma that is not otherwise specified, and $\text{dom}(ms'_{unused}) = \text{dom}(ms''_{unused})$. From Lemma 63 we also get

19.6. $ms'' :_{n''} revokeTemp(W_2)$

Assume n'' is large enough to execute the rest of the `scall` instructions. If n'' is not large enough, then 17 is trivial to show. To show 17 apply Lemma 8 again where ms_r is the revoked part. Let ms'_{frame} be given, the execution until just after the `scall` proceeds as follows:

$$(reg_2[\text{pc} \mapsto updatePcPerm(c_{ret})], ms' \uplus ms'_{varframe}) \rightarrow_j (reg_3, ms' \uplus ms'_{frame})$$

where

19.7.

$$reg_3(r) = \begin{cases} c_{next} & r = \text{pc} \\ c_{stk} & r = r_{stk} \\ reg(r) & r \in \{\overline{r_{priv}}\} \\ reg_2(r) & \text{otherwise} \end{cases}$$

19.8. reg_3 points to stack with ms_{stk} used and $ms_{act} \uplus ms''_{unused}$ unused

At this point, we use Hyp-Cont (6.) to show the observation predicate condition of Lemma 8:

$$(n'', (reg_3, ms'' \uplus ms_{stk} \uplus ms_{act} \uplus ms''_{unused} \uplus ms_f)) \in \mathcal{O}(W_2)$$

which

- $n'' \leq n - 2$
  Follows from (19.1.)
- $W_2 \sqsupseteq^{pub} revokeTemp(W)$
  We have
  $$W_1 \sqsupseteq^{pub} revokeTemp(W)$$
  and assumption 19.2. we get this by transitivity of $\sqsupseteq^{pub}$.
- $ms'' :_{n''} revokeTemp(W_2)$
  Exactly 19.6..
- for all r, we have that:

  $$reg_3(r) \begin{cases} = c_{next} & \text{if } r = \text{pc} \\ = reg(r) & \text{if } r \in \overline{r_{priv}} \\ \in \mathcal{V}(revokeTemp(W_2)) & \text{if } reg_3(r) \text{ is a global capability and } r \notin \{\text{pc}, \overline{r_{priv}}, r_{stk}\} \end{cases}$$

  The two first cases follows from 19.7.. The third follow from assumption 19.4. and 79.
- reg' points to stack with ms_{stk} used and $ms_{act} \uplus ms''_{unused}$ unused
  Exactly 19.8..

□

5.4.11 Malloc macro-instruction correctness

Definition 6. *We say that "(reg, ms) links key as j to c_{malloc}" iff*

- $reg(pc) = ((perm, g), base, end, a)$
- $ms(base) = ((\_, \_), base_{link}, \_, \_)$
- $ms(base_{link} + j) = c$

■

Lemma 59 (`malloc` works)**.** *If*

- (reg, ms) *is looking at* `malloc` $r\ k$ *followed by* c_{next}
- $k \geq 0$
- (reg, ms) *links malloc as* k *to* c_{malloc}
- c_{malloc} *satisfies the malloc specification with* $\iota_{malloc,0}$
- $W \sqsupseteq^{priv} [i \mapsto \iota_{malloc,0}]$
- $ms :_n W$
- $ms = ms' \uplus ms_{footprint}$
- $ms_{footprint} :_n [i \mapsto W(i)]$

Hyp-Cont If

- $n' \leq n - 1$
- $\iota_{malloc} \sqsupseteq^{pub} W(i)$
- $ms'_{footprint} \uplus ms' :_{n'} W[i \mapsto \iota_{malloc}]$
- $ms'_{footprint} :_{n'} [i \mapsto \iota_{malloc}]$

$$reg'(r') = \begin{cases} c_{next} & r' = \mathrm{pc} \\ ((\mathrm{RWX}, \mathrm{GLOBAL}), base, end, a) & r' = r \\ reg(r) & r' \notin \mathrm{RegisterName}_t \cup \{\mathrm{pc}, r, r_1\} \end{cases}$$

- $end - base = k - 1$
- $\mathrm{dom}(ms_{alloc}) = [base, end]$
- $\forall a \in [base, end].\ ms_{alloc}(a) = 0$

Then we have $\left(n', (reg', ms' \uplus ms'_{footprint} \uplus ms_{alloc})\right) \in \mathcal{O}(W[\iota_{malloc}])$

Then

$$(n, (reg, ms)) \in \mathcal{O}(W)$$

■

5.4.12 Create closure macro-instruction correctness

Lemma 60 (`crtcls` works)**.** *If*

- (reg, ms) *is looking at* $\texttt{crtcls}\ \overline{(x, r)}\ r$ *followed by* c_{next}
- (reg, ms) *links malloc as* k *to* c_{malloc}
- c_{malloc} *satisfies the malloc specification with* $\iota_{malloc,0}$
- $W \sqsupseteq^{priv} [i \mapsto \iota_{malloc,0}]$
- $ms :_n W$
- $ms = ms' \uplus ms_{footprint}$
- $ms_{footprint} :_n [i \mapsto W(i)]$

Hyp-Cont If

- $n' \leq n$
- $\iota_{malloc} \sqsupseteq^{pub} W(i)$
- $ms' \uplus ms'_{footprint} :_{n'} W[i \mapsto \iota_{malloc}]$
- $ms'_{footprint} :_n [i \mapsto \iota_{malloc}]$
-

$$reg'(r') = \begin{cases} c_{next} & r' = \text{pc} \\ c_{cls} = ((\text{E}, \text{GLOBAL}), base, end, base + 2) & r' = r_1 \\ reg(r) & r' \notin \{\text{pc}, r_1\} \cup \text{RegisterName}_t \end{cases}$$

- $ms_{cls} = ms_{act} \uplus ms_{env}$
- $c_{cls} = ((\text{E}, \text{GLOBAL}), \dots)$
- $c_{env} = ((\text{RW}, \text{GLOBAL}), base_{env}, end_{env}, base_{env})$
- $\text{dom}(ms_{env}) = [base_{env}, end_{env}]$
- $ms_{env}(base_{env}, \dots, end_{env}) = reg(\overline{r})$
- *Hyp-act*
  If
  - $reg''(\text{pc}) = updatePcPerm(c_{cls})$

  Then $\exists k. \forall ms_f. (reg'', ms'' \uplus ms_{cls} \uplus ms_f) \rightarrow_k (reg''', ms'' \uplus ms_{cls} \uplus ms_f)$ *where*

$$reg'''(r') = \begin{cases} c_{env} & r' = c_{env} \\ updatePcPerm(reg(r)) & r' = \text{pc} \\ reg''(r') & r' \notin \text{RegisterName}_t \end{cases}$$

Then we have $(n', (reg', ms' \uplus ms_{footprint} \uplus ms_{cls})) \in \mathcal{O}(W[i \mapsto \iota_{malloc}])$

Then

$$(n, (reg, ms)) \in \mathcal{O}(W)$$

■

5.4.13 Helper lemmas about the stack

Lemma 61. *If*

- $perm \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$
- $(n, (base, end))$ $readCondition(\text{LOCAL})(W)$
- $(n, (base, end))$ $writeCondition(\iota^{pwl}, \text{LOCAL})(W)$

then

$$(n, perm, base, end) \in executeCondition(\text{LOCAL})(W)$$

■

Proof of Lemma 61. Assume

1. $perm \in \{\text{RX}, \text{RWX}, \text{RWLX}\}$
2. $(n, (base, end))$ $readCondition(\text{LOCAL})(W)$
3. $(n, (base, end))$ $writeCondition(\iota^{pwl}, \text{LOCAL})(W)$

Let $W' \sqsupseteq^{pub} W$, a, and $n' \leq n$ be given and show

$$(n', ((perm, \text{LOCAL}), base, end, a)) \in \mathcal{E}(W')$$

Consider each of the three cases for $perm$:

4. $perm = \text{RWLX}$
   In this case $\iota = \iota^{pwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:
   - 4.1. $(n', (base, end)) \in readCondition(\text{LOCAL})(W')$
     Follows from Lemma 34, Lemma 36, and assumption 2..
   - 4.2. $(n', (base, end)) \in writeCondition(\iota^{pwl}, \text{LOCAL})(W')$
     Follows from Lemma 37, Lemma 36, and assumption 3..

5. $perm = \text{RX}$
   In this case $\iota = \iota^{nwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:
   - 5.1. $(n', (base, end)) \in readCondition(\text{LOCAL})(W')$
     Follows from Lemma 34, Lemma 36, and assumption 2..
   - 5.2. $(n', (base, end)) \in writeCondition(\iota^{nwl}, \text{LOCAL})(W')$
     Follows from Lemma 33, Lemma 37, Lemma 36, and assumption 3..

6. $perm = \text{RWX}$
   In this case $\iota = \iota^{nwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:
   - 6.1. $(n', (base, end)) \in readCondition(\text{LOCAL})(W')$
     Follows from Lemma 34, Lemma 36, and assumption 2..

□

Lemma 62 (Stack capability in value relation)**.** *If*

- *reg points to stack with* $\emptyset$ *used and ms unused*
- $\exists r.\, W(r) = \iota^{pwl}(\mathrm{dom}(ms))$

then

$$(n, reg(r_{stk})) \in \mathcal{V}(W)$$

■

Proof of Lemma 62. Say

$$reg(r_{stk}) = c_{stk} = ((\textsc{rwlx}, \textsc{local}), base, end, \_)$$

Show

1. $(n, (base, end)) \in readCondition(\textsc{local})(W)$:
   Amounts to
   $$\iota^{pwl}(\mathrm{dom}(ms)) \overset{n}{\subsetneq} \iota^{pwl}_{base,end}$$
   which is true as they are even equal.
2. $(n, (base, end)) \in writeCondition(\iota^{pwl}, \textsc{local})(W)$:
   Using Lemma 12, this amounts to
   $$\iota^{pwl}(\mathrm{dom}(ms)) \overset{n}{\supsetneq} \iota^{pwl}_{base,end}$$
   which is true as they are even equal.
3. $(n, (\textsc{rwlx}, base_{stk}, end_{stk})) \in executeCondition(\textsc{local})(W)$
   Using 2. and 1., we can use Lemma 61.
4. $(n, (\textsc{rwx}, base_{stk}, end_{stk})) \in executeCondition(\textsc{local})(W)$
   Using 2. and 1., we can use Lemma 61.
5. $(n, (\textsc{rx}, base_{stk}, end_{stk})) \in executeCondition(\textsc{local})(W)$
   Using 1. and 2., we can use Lemma 61.

□

5.4.14 Memory Segment Satisfaction

We expect the following lemmas to hold true:

Lemma 63 (Revoke temporary memory satisfaction)**.**

$$\begin{aligned}
&\forall ms, n, W, W'.\\
&\quad ms :_n W \Rightarrow\\
&\qquad \exists ms', ms_r.\\
&\qquad\quad ms = ms' \uplus ms_r \wedge ms' :_n revokeTemp(W)
\end{aligned}$$

■

Proof of Lemma 63. □

Lemma 64 (Revoke temporary memory satisfaction 2)**.**

$$\begin{aligned}
&\forall ms, n, W, R : \mathit{active}(W) \to \text{MemSegment}.\\
&\quad ms :_{n,P} W \Rightarrow\\
&\qquad \exists ms', ms_r.\\
&\qquad\quad ms = ms' \uplus ms_r \wedge\\
&\qquad\quad ms' :_{n,P|_{\mathrm{dom}(\lfloor W \rfloor_{\{\mathrm{perm}\}})}} \mathit{revokeTemp}(W) \wedge\\
&\qquad\quad ms_r = \biguplus_{r \in \lfloor W \rfloor_{\{\mathrm{temp}\}}} P(r) \wedge\\
&\qquad\quad ms' = \biguplus_{r \in \lfloor W \rfloor_{\{\mathrm{perm}\}}} P(r)
\end{aligned}$$

■

Proof of Lemma 64. □

Lemma 65 (Revoke temporary memory with stack)**.**

$$\begin{aligned}
&\forall n, ms, W, reg, r_{stk}, g, base, end, a.\\
&\quad ms :_n W \wedge (n, reg) \in \mathcal{R}(W) \wedge\\
&\quad reg(r_{stk}) = ((\textsc{rwlx}, g), base, end, a) \wedge b \leq e\\
&\qquad \exists ms', ms_r.\\
&\qquad\quad ms' :_n \mathit{revokeTemp}(W) \wedge ms = ms' \uplus ms_r
\end{aligned}$$

■

Proof of Lemma 65. □

Lemma 66 (Disjoint memory satisfaction)**.**

$$\begin{aligned}
&\forall n. \forall ms, ms', ms''. \forall W, W', W''.\\
&\quad ms'' = ms \uplus ms' \wedge W'' = W \uplus W' \wedge ms :_n W \wedge ms' :_n W' \Rightarrow\\
&\qquad ms'' :_n W''
\end{aligned}$$

■

Proof of Lemma 66. □

Lemma 67 (Memory satisfaction and static regions)**.**

$$ms :_n [i \mapsto \iota^{sta}(v, ms)]$$

■

Proof of Lemma 67. □

Lemma 68 (Data only memory and standard regions)**.** *If*

- $\forall a \in \mathrm{dom}(ms).\, ms(a) \in \mathbb{N}$
- $\iota \in \{\iota^{pwl}, \iota^{nwl}, \iota^{nwl,p}\}$

then

$$ms :_n [i \mapsto \iota(\mathrm{dom}(ms))]$$

■

Proof of Lemma 68. □

5.4.15 Future worlds

Lemma 69 (World public future world of revoked world)**.**

$$\forall W.\, \mathit{revokeTemp}(W) \sqsupseteq^{pub} W$$

■

Proof of Lemma 69. For all r where $W(r) = (\mathrm{temp}, s, \phi_{pub}, \phi, H)$, we have $\mathit{revokeTemp}(W) = \mathrm{revoked}$. By the public future region relation we have

$$W(r) = (\mathrm{temp}, s, \phi_{pub}, \phi, H) \sqsupseteq^{pub} \mathit{revokeTemp}(W)(r) = \mathrm{revoked}$$

all other regions remain unchanged, so this follows by reflexivity of the public future region relation. □

Lemma 70 (World private future world of revoked world)**.**

$$\forall W.\, \mathit{revokeTemp}(W) \sqsupseteq^{priv} W$$

■

Proof of Lemma 70. □

Lemma 71 (Public future world relation included in private future world relation)**.**

$$W' \sqsupseteq^{pub} W \Rightarrow W' \sqsupseteq^{priv} W$$

■

Proof of Lemma 71. □

Lemma 72 (Transitivity proberties between private and public future worlds)**.**

$$W'' \sqsupseteq^{priv} W' \wedge W' \sqsupseteq^{pub} W \Rightarrow W'' \sqsupseteq^{priv} W$$

and

$$W'' \sqsupseteq^{pub} W' \wedge W' \sqsupseteq^{priv} W \Rightarrow W'' \sqsupseteq^{priv} W$$

■

Proof of Lemma 72. □

Lemma 73.

$$\forall n, W_1, W_2, W_1'. \qquad W_1 \stackrel{n}{=} W_2 \wedge W_1' \sqsupseteq^{pub} W_1 \Rightarrow \exists W_2'.\, W_2' \stackrel{n}{=} W_1' \wedge W_2' \sqsupseteq^{pub} W_2$$

■

Proof of Lemma 73. Construct W_2' as follows:

$$W_2(r) = \begin{cases} (v_1', s_1', \phi_{pub2}, \phi_2, H_2) & \text{if } r \in \mathrm{dom}(W_2) \text{ and } W_1'(r) = (v_1', s_1', \_, \_, \_) \\ & \quad \text{and } W_2(r) = (\_, \_, \phi_{pub2}, \phi_2, H_2) \\ W_1'(r) & \text{otherwise} \end{cases}$$

Notice $\mathrm{dom}(W_2') = \mathrm{dom}(W_1')$. □

Lemma 74.

$$\forall n, W_1, W_2, W_1'. \qquad W_1 \stackrel{n}{=} W_2 \wedge W_1' \sqsupseteq^{priv} W_1 \Rightarrow \exists W_2'. W_2' \stackrel{n}{=} W_1' \wedge W' \sqsupseteq^{priv} W_2$$

■

Proof of Lemma 74. Construct W_2' as follows:

$$W_2(r) = \begin{cases} (v_1', s_1', \phi_{pub2}, \phi_2, H_2) & \text{if } r \in \mathrm{dom}(W_2) \text{ and } W_1'(r) = (v_1', s_1', \_, \_, \_) \\ & \quad \text{and } W_2(r) = (\_, \_, \phi_{pub2}, \phi_2, H_2) \\ W_1'(r) & \text{otherwise} \end{cases}$$

□

5.4.16 Value relation

Lemma 75 (Value relation downwards closed)**.**

$$n' \leq n \wedge (n, w) \in \mathcal{V}(W) \Rightarrow (n', w) \in \mathcal{V}(W)$$

■

Proof. By definition of $\mathcal{V}(W)$ using Lemma 36, 39, 42 and 45. □

Lemma 76 (Register relation downwards closed)**.**

$$n' \leq n \wedge (n, w) \in \mathcal{R}(W) \Rightarrow (n', w) \in \mathcal{R}(W)$$

■

Proof. By definition of $\mathcal{R}(W)$ using Lemma 75. □

Lemma 77 (Value relation monotone wrt $\sqsupseteq^{pub}$)**.**

$$W' \sqsupseteq^{pub} W \wedge (n, w) \in \mathcal{V}(W) \Rightarrow (n, w) \in \mathcal{V}(W')$$

■

Proof of lemma 77. Follows from Lemma 34, Lemma 37, Lemma 40, and Lemma 43. □

Lemma 78. *If*

$$(n, w) \in \mathcal{V}(\mathit{revokeTemp}(W))$$

then

$$(n, w) \in \mathcal{V}(W)$$

■

Proof of Lemma 78. Follows from Lemma 29, Lemma 30, Lemma 31, and Lemma 32. □

Lemma 79 (Global capabilities monotone wrt $\sqsupseteq^{priv}$)**.**

$$\begin{aligned}&\forall n, perm, base, end, a, W, W'.\\ &\quad (n, ((perm, \textsc{global}), base, end, a)) \in \mathcal{V}(W) \wedge W' \sqsupseteq^{priv} W\\ &\qquad \Rightarrow (n, ((perm, \textsc{global}), base, end, a)) \in \mathcal{V}(W')\end{aligned}$$

■

Proof of Lemma 79. Assume

1. $perm \notin \{\textsc{rwl}, \textsc{rwlx}\}$
2. $W' \sqsupseteq^{priv} W$
3. $(n, ((perm, \textsc{global}), base, end, a)) \in \mathcal{V}(W)$

and show

$$(n, ((perm, \textsc{global}), base, end, a)) \in \mathcal{V}(W')$$

to this end consider the possible cases of *perm* and show that each of the necessary conditions hold:

1. $perm = \textsc{o}$
   Trivial
2. $perm = \textsc{ro}$
   Follows from Lemma 35.
3. $perm = \textsc{rw}$
   Follows from Lemma 35 and Lemma 38.
4. $perm = \textsc{rx}$
   Follows from Lemma 35 and Lemma 41.
5. $perm = \textsc{rwx}$
   Follows from Lemma 35, Lemma 38, and Lemma 41.
6. $perm = \textsc{e}$
   Lemma 44

□

Lemma 80 (Non local words monotone wrt $\sqsupseteq^{priv}$)**.**

$$\begin{aligned}&\forall n, perm, base, end, a, W, W', w.\\ &\quad w \textit{ is non-local} \wedge\\ &\quad (n, w) \in \mathcal{V}(W) \wedge W' \sqsupseteq^{priv} W\\ &\qquad \Rightarrow (n, w) \in \mathcal{V}(W')\end{aligned}$$

■

Proof of Lemma 80. If $w = ((perm, \textsc{global}), base, end, a)$, then let follows from Lemma 79.
If $w \in \mathbb{Z}$, then it follows from the fact that $i \in \mathcal{V}(W'')$ for all $i \in \mathbb{Z}$ and $W'' \in$ World. □

6 Other examples and applications

This section contains some ideas about other examples and applications than the ticket dispenser example.

6.1 Stack and return pointer handling without OS involvement using local capabilities

The idea of this example would be to work out and prove a calling convention that enforces well-bracketed control flow and encapsulation of local variables using CHERI's local capabilities.

When one function invokes another function, the essential idea is that:

- Stack pointer is passed as a local and store-local capability.
- Return pointer is passed as a local capability.

Since local pointers cannot leave the registers except into regions for which a store-local capability is available, this basic idea seems to enforce a number of useful properties: well-bracketedness of control flow and encapsulation of private state stored on the stack. On the other hand, it also seems to validate the standard C treatment of the stack: the stack can be reused after a function returns, even between distrusting parties. However, safety/security of this design is very non-trivial and seems to rely on some non-trivial reasoning:

Only stack is store-local? A critical assumption is that adversary code has no way to *store* local capabilities except on the stack. The reason that it is fine to store local capabilities on the stack is that the adversary only has a *local* capability to the stack and cannot usefully store that capability anywhere. However, this means that we need to rely on the runtime system of our programming language to be careful when handing out store-local capabilities: only the libc startup code should initialise the stack as store-local and malloc should *not* produce them. This basically means that the libc initialisation code (or whatever component produces the initial stack pointer) is part of our TCB.

Requirement for clearing the stack Imagine the following trusted C function:

```
void myfunction(){
  advfunction1();
  advfunction2();
}
```

where advfunction1() and advfunction2() are adversary functions. In the standard C treatment of the stack, advfunction2() would get the same stack pointer as advfunction1(). This is supposed to be safe since advfunction1() cannot have kept capabilities for the stack after its execution. But what if we require that the two functions have no way of communicating with each other? Concretely, advfunction1() has access to some secrets that must not be leaked to advfunction2(). How can we prevent advfunction1() from storing the secret somewhere on the stack and relying on advfunction2() from receiving the same stack pointer where it can read the secret? The most obvious solution seems to be that we should fully clear the stack (overwrite it with zeros) after the return of any adversary function, but this could cause an important overhead. Perhaps the processor should accommodate this with a special instruction that can zero the entire array that a capability points to?

What do return pointers look like? An important question is what return pointers look like? Since we want to protect the caller from the callee, it's important that the return pointer is opaque, i.e. an entry pointer. The entry pointer will point to a closure that contains the next instruction to execute, as well as the previous stack pointer. But since stack pointers are local, this means that the return pointer closure should be stored in a region of memory for which we have store-local permission, i.e. on the stack. This means we need the following in our calling convention: before invoking a function, we push the stack pointer and the instruction pointer after invocation on the stack, we construct a return pointer by copying the stack pointer, limiting it to these two entries and making it an entry pointer. Then we shrink the stack pointer to the unused part of the stack and jump.

Only one-way protection in higher-order settings? Another important point is that, in a sense, local capabilities provide only one-way protection: the caller is protected from the callee but not vice-versa. Concretely: when invoking a function with some arguments marked as local, the caller is guaranteed that the callee will not have been able to store the capabilities anywhere (except perhaps on the stack, see above). However, the callee seems to have more limited guarantees: Particularly, the caller may have kept its own stack capability and this stack capability may (and typically will) also cover the part of the stack that is "owned" by the callee. In this sense, the guarantees are more limited than in a linear language.

So what does this mean? In a first-order language, this is all fine, but what if we are in a higher-order language. Imagine the following (in some ML-like language):

```
let f = fun callback =>
          let ... in
          let ret = callback() in
          ...
//adversary top function
let advtop = f( (fun y => ...) )
```

Our trusted function f is invoked by the adversary (from function advtop()) and wants to invoke an untrusted callback received from the adversary. When invoking the closure, we don't want it to be able to access f's local variables which it has stored on the stack. To achieve this, we only give it a stack pointer that covers the part of the stack that is unused by f. However, the callback may be implemented as an entry pointer that carries capabilities, particularly the capability to advtop's stack pointer, which includes the part of the stack that is now used by f and contains f's local variables.

So how do we deal with this? Perhaps we should use the fact that this is only possible when f's callback argument is allocated to some part of the memory to which advtop has store-local permissions (since the callback contains a reference to the stack to which advtop only has a local capability). I see basically three ways to do this, all based on the idea of enforcing that the callback should be constructed in a part of memory for which no store-local permissions are available:

- One way to exclude the scenario is to require that callbacks are provided as non-local capabilities. The downside of this is that local callbacks can be useful for the caller to prevent the callee from storing them.
- Another way to exclude the scenario is to require that the stack is allocated in a fixed part of the address space and to check that callbacks point outside of this region before invoking them.

- Perhaps we should require that store-local permissions cannot be removed from a capability and simply require that callback pointers do not have store-local set. Perhaps we can allow store-local permissions to be given up, but only if the corresponding part of memory is fully zeroed in the process (or at least all local capabilities stored in the region).

6.2 A result to prove...

The simplest thing that comes to mind as a formal result for all of the above is to look at a concrete program that clearly relies on properties like well-bracketed control flow and encapsulation of local variables and prove it correct. As a concrete example: we might show an assembly program that corresponds to the following (a higher-order program that crosses trust boundaries and relies on local variable encapsulation and well-bracketed control flow):

```
let trustedCode = fun adversary =>
                    let x = ref 0 in
                    let callback = fun adv2 =>
                      x := !x + 1;
                      let y = ref (!x) in
                      adv2 unit;
                      assert (!x == !y);
                      x := !x - 1)
                    let _ = adversary callback
                    assert (!x == 0)
```

7 Related reading

This is a list of related work that might be interesting to read in the context of this project.

7.1 Capability machines

7.1.1 M-Machine

More than 20 years ago, Carter et al. [1994] have described the use of capabilities in the M-Machine. They do seem to have a reference for the instruction set after all [Dally et al., 1995]; it seems like the server was just temporarily down when we were looking for this the first time...

7.1.2 CHERI

The CHERI processor is a much more recent capability machine, described by Woodruff et al. [2014], Watson et al. [2015].

Another result of this project is also CheriBSD: an adaptation of FreeBSD to the CHERI processor.[5] It is not separately described in a published paper, but mentioned in the papers cited above and in some tech reports (see url). This work includes a pure-capability ABI that could provide some interesting examples.

The CHERI team also has a webpage with all of their CHERI-related publications (including TRs and such)[6].

[5] `http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html`
[6] `http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`

7.2 Logical Relations

Some papers on logical relations that are relevant for this work are the following:

Hur and Dreyer [2011] describe a logical relation between ML and a (standard) assembly language for expressing compiler correctness. Relevant because they target an assembly language, and they use biorthogonality.

Dreyer et al. [2010] describe a logical relation for a ML-like language and use public/private transitions to reason about well-bracketed control flow. Relevant because we are considering to cover an example of enforcing well-bracketed control flow in a capability machine.

Devriese et al. [2016] describe a logical relation for a JavaScript-like language with object capabilities. Relevant because it treats object capabilities, albeit in a JavaScript-like lambda calculus. It also deals with an untyped language, using a semantic unitype.

References

Lars Birkedal and Aleš Bizjak. A Taste of Categorical Logic — tutorial notes. `http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf`, 2014.

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102 – 4122, 2010. ISSN 0304-3975.

A. Bizjak. Some theorems about mutually recursive domain equations in the category of preordered COFEs. Unpublished note. Available at `http://cs.au.dk/~abizjak/documents/notes/mutually-recursive-domain-eq.pdf`, 2017.

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. ACM. ISBN 0-89791-660-3. doi: 10.1145/195473.195579. URL `http://doi.acm.org/10.1145/195473.195579`.

William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The m-machine instruction set reference manual v1.55. Technical Report Memo 59, CVA, Stanford, 1995. URL `http://cva.stanford.edu/publications/1997/isa-1.55.ps.Z`.

Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities using logical relations and effect parametricity. In *IEEE European Symposium on Security and Privacy*. IEEE, 2016.

Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *International Conference on Functional Programming*, pages 143–156. ACM, 2010. doi: 10.1145/1863543.1863566.

Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ml and assembly. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146. ACM, 2011. doi: 10.1145/1926385.1926402.

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015. doi: 10.1109/SP.2015.9.

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *International Symposium on Computer Architecuture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.