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

Lau Skorstengaard  Dominique Devriese
Aarhus University  imec-DistriNet, KU Leuven
lask@cs.au.dk  dominique.devriese@cs.kuleuven.be

Lars Birkedal
Aarhus University
birkedal@cs.au.dk

February 16, 2018

Contents

1 Capability Machine Definition and Operational Semantics 2
  1.1 Domains and Notation .................................................. 2
  1.2 Operational Semantics .................................................. 5

2 Malloc specification .................................................. 9

3 Macros ................................................................. 9
  3.1 Linking and ABI ...................................................... 9
  3.2 Flag table .......................................................... 10
  3.3 Macro definitions .................................................... 10
  3.4 Stack .............................................................. 15
  3.5 Labels ............................................................. 21

4 Examples ............................................................ 25
  4.1 Encapsulation of Local State ........................................ 25
  4.2 Encapsulation of Local State Using Local Capabilities and scall .................................................. 28
  4.3 Well-Bracketedness Using Local Capabilities and scall .... 32
  4.4 Inverted Control and Return From Closure ...................... 38
  4.5 Variant of the “awkward” example ................................. 38

5 Logical Relation ..................................................... 48
  5.1 Worlds ............................................................. 48
  5.2 The logical relation .................................................. 51
  5.3 Useful regions ....................................................... 54
  5.4 Lemmas ............................................................ 54
    5.4.1 Anti-reduction for the observation relation ................ 54
1 Capability Machine Definition and Operational Semantics

1.1 Domains and Notation

\[
\begin{align*}
\text{Addr} & \overset{\text{def}}{=} \mathbb{N} \\
\text{Word} & \overset{\text{def}}{=} \text{Cap} + \mathbb{Z} \\
\text{Reg} & \overset{\text{def}}{=} \text{RegisterName} \rightarrow \text{Word} \\
\text{Mem} & \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word} \\
\text{Perm} & ::= \text{o} \mid \text{ro} \mid \text{rw} \mid \text{rwl} \mid \text{rx} \mid \text{e} \mid \text{rwx} \mid \text{rwlx} \\
\text{ExecConf} & \overset{\text{def}}{=} \text{Reg} \times \text{Mem} \\
\text{Global} & ::= \text{GLOBAL} \mid \text{LOCAL} \\
\text{Cap} & \overset{\text{def}}{=} (\text{Perm} \times \text{Global}) \times \text{Addr} \times (\text{Addr} + \{\infty\}) \times \text{Addr} \\
\text{Conf} & \overset{\text{def}}{=} \text{ExecConf} + \{\text{failed}\} + \{\text{halted}\} \times \text{Mem} \\
\text{MemSegment} & \overset{\text{def}}{=} \text{Addr} \rightarrow \text{Word}
\end{align*}
\]

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 R WL and R WLX that permits writing local capabilities. They are otherwise the same as their non-"permit write local" counterparts.
As we have $\infty$ as a possible address, but our words cannot express $\infty$. We pick $-42$ as a representative for $\infty$ when it is in memory (we could have picked any negative number). Note that $-42$ is not an address, so for address operations $-42$ only represents $\infty$. It is the responsibility 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

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

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.

![Permission hierarchy](image1)

![Locality hierarchy](image2)
<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>No permissions. Grants no permissions</td>
</tr>
<tr>
<td>RO</td>
<td>Read only. Grants read permission</td>
</tr>
<tr>
<td>RW</td>
<td>Read-write. Grants read and write permission. Storage of local capabilities prohibited.</td>
</tr>
<tr>
<td>RWL</td>
<td>Read-write, permit write local. Grants read and write permission. Storage of local capabilities possible.</td>
</tr>
<tr>
<td>RX</td>
<td>Execute permission. Grants execute and read permissions.</td>
</tr>
<tr>
<td>E</td>
<td>Enter permission. This permission grants no access, but when jumped to, it will turn into an RX permission.</td>
</tr>
<tr>
<td>RWX</td>
<td>Read-write-execute permission. Grants read, write, and execute permissions. Storage of local capabilities prohibited.</td>
</tr>
<tr>
<td>RWLX</td>
<td>Read-write-execute, permit write local. Grants read, write, and execute permissions. Storage of local capabilities possible.</td>
</tr>
</tbody>
</table>

Table 1: The permissions in this capability system

Notation:

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

Words and instructions:

\[ lv ::= [r] \]
\[ hv ::= \langle r \rangle_m \]
\[ rv ::= n \mid lv \]
\[ i ::= \text{jmp } lv \mid \text{jnz } lv lv \mid \text{move } lv rv \mid \text{load } lv hv \mid \text{store } hv rv \mid \]
\[ \text{plus } lv rv \mid \text{minus } lv rv \mid \text{lt } lv rv \mid \text{lea } lv rv \mid \text{restrict } lv rv \mid \text{subseg } lv rv rv \mid \]
\[ \text{isptr } lv rv \mid \text{getp } lv lv \mid \text{getl } lv lv \mid \text{getb } lv lv \mid \text{gete } lv lv \mid \text{geta } lv lv \mid \]
\[ \text{fail} \mid \text{halt} \]

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:

\[ \text{decode} : \text{Word} \rightarrow \text{Instructions} \]

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

\[ \text{encodePerm} : \text{Perm} \rightarrow \mathbb{Z} \]
\[ \text{encodeLoc} : \text{Global} \rightarrow \mathbb{Z} \]
\[ \text{encodePermPair} : (\text{Perm} \times \text{Global}) \rightarrow \mathbb{Z} \]

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

\[ \text{decodePermPair} : \mathbb{Z} \rightarrow (\text{Perm} \times \text{Global}) \]

We define the operational semantics as follows:

\[ \Phi \rightarrow [\text{decode}(\Phi.\text{mem}(a))] (\Phi) \quad \text{if } \Phi.\text{reg}(pc) = (((\text{perm}, g), \text{base}, \text{end}, a) \text{ and } \text{base} \leq a \leq \text{end} \text{ and } \text{perm} \in \{\text{RX}, \text{RWX}, \text{RWLX}\}) \]

\[ \Phi \rightarrow \text{failed} \quad \text{otherwise} \]

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

\[ \text{readAllowed}(\text{perm}) = \begin{cases} \text{true} & \text{if } \text{perm} \in \{\text{Rwx}, \text{Rwxl}, \text{Rx}, \text{Rw}, \text{Rwl}, \text{Ro}\} \\ \text{false} & \text{otherwise} \end{cases} \]

\[ \text{writeAllowed}(\text{perm}) = \begin{cases} \text{true} & \text{if } \text{perm} \in \{\text{Rwx}, \text{Rwxl}, \text{Rw}, \text{Rwl}\} \\ \text{false} & \text{otherwise} \end{cases} \]

\[ \text{updatePcPerm}(w) = \begin{cases} (((\text{Rx}, g), \text{base}, \text{end}, a) & \text{if } w = (((\text{E}, g), \text{base}, \text{end}, a) \\ w & \text{otherwise} \end{cases} \]

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

\[ \text{withinBounds}((_, \text{base}, \text{end}, a)) = \begin{cases} \text{true} & \text{if } \text{base} \leq a \leq \text{end} \\ \text{false} & \text{otherwise} \end{cases} \]

\[ \text{updatePc}(\Phi) = \begin{cases} \Phi[\text{reg}.pc \mapsto \text{newPc}] & \text{if } \Phi.\text{reg}(pc) = (((\text{perm}, g), \text{base}, \text{end}, a) \\ \text{and } \text{newPc} = (((\text{perm}, g), \text{base}, \text{end}, a + 1) & \text{failed} \quad \text{otherwise} \end{cases} \]
\[
\begin{align*}
\text{[fail]}(\Phi) &= \text{failed} \\
\text{[halt]}(\Phi) &= (\text{halted}, \Phi.\text{mem}) \\
\text{[jmp \: lv]}(\Phi) &= \Phi[\text{reg.pc} \mapsto \text{updatePcPerm}(\Phi.\text{reg}(\text{lv}))] \\
\text{[jnz \: lv \: rv]}(\Phi) &= \\
& \begin{cases} \\
\Phi[\text{reg.pc} \mapsto \text{updatePcPerm}(\Phi.\text{reg}(\text{lv}))] & \text{if nonZero}(\Phi.\text{reg}(\text{rv})) \\
\text{updatePc}(\Phi) & \text{if not nonZero}(\Phi.\text{reg}(\text{rv})) \\
\text{failed} & \text{otherwise} \end{cases} \\
\text{[load \: [r_1] \: [r_2]]}(\Phi) &= \\
& \begin{cases} \\
\text{updatePc}(\Phi[\text{reg.r_1} \mapsto c]) & \text{if } \Phi.\text{reg}(\text{r_1}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\
\text{and } \text{readAllowed}(\text{perm}) \text{ and } \text{withinBounds}(c) \\
\text{and } w = \Phi.\text{mem}(a) \\
\text{failed} & \text{otherwise} \end{cases} \\
\text{[store \: [r_1] \: [r_2]]}(\Phi) &= \\
& \begin{cases} \\
\text{updatePc}(\Phi[\text{mem.a} \mapsto w]) & \text{if } \Phi.\text{reg}(\text{r_1}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\
\text{and } \text{writeAllowed}(\text{perm}) \text{ and } \text{withinBounds}(c) \\
\text{and } w = \Phi.\text{reg}(\text{r_2}) \\
\text{and if } w = ((\_ , \text{LOCAL}, \_ , \_ , \_ ), \\
\text{then } \text{perm} \in \{\text{RWLX, RWL}\} \\
\text{failed} & \text{otherwise} \end{cases} \\
\text{[move \: [r_1] \: rv]}(\Phi) &= \\
& \begin{cases} \\
\text{updatePc}(\Phi[\text{reg.r_1} \mapsto \text{rv}]) & \text{rv} \in \mathbb{Z} \\
\text{updatePc}(\Phi[\text{reg.r_1} \mapsto \Phi.\text{reg}(\text{rv})]) & \text{otherwise} \end{cases} \\
\text{[lea \: [r_1] \: rv]}(\Phi) &= \\
& \begin{cases} \\
\text{updatePc}(\Phi[\text{reg.r_1} \mapsto c]) & \text{if either } n = \text{rv} \text{ or } \text{rv} = [r_2] \text{ and } n = \Phi.\text{reg}(r_2) \\
\text{and } \Phi.\text{reg}(\text{r_1}) = ((\text{perm}, g), \text{base}, \text{end}, a) \\
\text{and } \text{perm} \neq \text{E} \\
\text{and } c = ((\text{perm}, g), \text{base}, \text{end}, a + n) \\
\text{failed} & \text{otherwise} \end{cases} \\
\text{[restrict \: [r] \: rv]}(\Phi) &= \\
& \begin{cases} \\
\text{updatePc}(\Phi[\text{reg.r} \mapsto c]) & \text{if } \Phi.\text{reg}(r) = (\text{permPair}, \text{base}, \text{end}, a) \\
\text{and either } \text{rv} = n \text{ or } \Phi.\text{reg}(\text{rv}) = n \\
\text{and in either case } n \in \mathbb{Z} \\
\text{and } \text{decodePermPair}(n) \sqsubseteq \text{permPair} \\
\text{and } c = (\text{decodePermPair}(n), \text{base}, \text{end}, a) \\
\text{failed} & \text{otherwise} \end{cases}
\end{align*}
\]
\[
\begin{align*}
\text{[plus | } r_1 | r v_1 r v_2 | \text{]}(\Phi) &= \begin{cases} 
\text{updatePc}(\Phi[\text{reg.} r_1 \mapsto n_1 + n_2]) & \text{if } i \in \{1, 2\} \\
\text{failed} & \text{otherwise}
\end{cases} \\
\text{and in either case } n_1 \in \mathbb{Z} \\
\text{and in either case } n_2 \in \mathbb{Z}
\end{align*}
\]
Define the following macros: \texttt{restrict}, \texttt{subseg}, and \texttt{lea} that does not overwrite the source register. A \texttt{store} that allows integers to be stored directly. \texttt{store} requires a register $r_t$ for storage of temporary values to be available.

\[
\begin{align*}
\text{restrict } r_1 & \quad r_2 \quad r_3 \quad r_4 \quad \overset{def}{=} \text{move } r_1 \quad r_2 \\
\text{restrict } r_1 & \quad r_3 \quad r_4 \\
\text{subseg } r_1 & \quad r_2 \quad r_3 \quad r_4 \quad \overset{def}{=} \text{move } r_1 \quad r_2 \\
\text{subseg } r_1 & \quad r_3 \quad r_4 \\
\text{lea } r_1 & \quad r_2 \quad r_3 \quad r_4 \quad \overset{def}{=} \text{move } r_1 \quad r_2 \\
\text{lea } r_1 & \quad r_3 \\
\text{store } r & \quad n \quad \overset{def}{=} \text{move } r_t \quad n \\
\text{store } r & \quad r_t
\end{align*}
\]

\textbf{Lemma 1} (Determinacy). \textit{If} $\Phi \rightarrow \Phi'$ and $\Phi \rightarrow \Phi''$, \textit{then} $\Phi' = \Phi''$. \textit{If} $\Phi \rightarrow_n \Phi'$ and $\Phi \rightarrow_n \Phi''$, \textit{then} $\Phi' = \Phi''$. \textit{If} $\Phi \rightarrow_n \Phi'$ and $\Phi \rightarrow_n' \text{ (halted, mem'')}$, \textit{then} $n \leq n'$ and $\Phi' \rightarrow_{n'-n} \text{ (halted, mem'')}$. \hfill \blacksquare

\textit{Proof.} By easy inspection of the definition of the operational semantics. \hfill \square
2 Malloc specification

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

\[
\begin{align*}
\forall i. \exists \tau_{ \text{malloc}, 0}. &
(\forall W, i. W(i) = i' \Rightarrow i'.H(i'.s)(\xi^{-1}(W)) = i'.H(i'.s)(\xi^{-1}([i \mapsto W(i)])))
\land \\
\land &
\forall \tau_{ \text{malloc}, 0}. v = \text{perm} \\
\land &
(\forall \Phi \in \text{ExecConf}. \forall ms_{\text{footprint}}, ms_{\text{frame}} \in \text{MemSegment}. \\
\forall i, n, size \in \mathbb{N}. \forall w_{\text{ret}} \in \text{Word}. \\
\land &
\forall \Phi \in \text{ExecConf}. \forall ms_{\text{footprint}}, ms_{\text{alloc}} \in \text{MemSegment}. \\
\land &
\forall j \in \mathbb{N}. j > 0 \land \exists b', e' \in \text{Addr}. \exists \tau'_{\text{malloc}} \in \text{Region}. \\
\land &
\Phi.\text{reg}(r_1) = size \land size \geq 0 \land \Phi.\text{reg}(r_0) = w_{\text{ret}} \\
\land &
\Phi.\text{reg}(pc) = \text{updatePcPerm}(c_{\text{malloc}}) \\
\Rightarrow &
(\forall \Phi \in \text{ExecConf}. (\Phi.\text{reg}(r_1) \notin \mathbb{Z} \lor \Phi.\text{reg}(r_1) < 0) \land \Phi.\text{reg}(pc) = \text{updatePcPerm}(c_{\text{malloc}}) \Rightarrow \exists j \in \mathbb{N}. \Phi \rightarrow_j \text{failed})
\end{align*}
\]

In the specification above \( \tau'_{\text{malloc}} \) 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 \( \text{RegisterName} \subseteq \text{RegisterName} \) available and that \( \text{RegisterName} \) does not contain registers explicitly named in a program nor \( r_0, r_{\text{stk}}, \) 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:

\[ \text{offsetLinkFlag} \overset{\text{def}}{=} 2 \]

### 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 malloc implementation (\( r_{t1} \) and \( r_{t2} \) are registers in RegName_t).

\[
\begin{align*}
\text{move} & \ r \ pc \\
\text{getb} & \ r_{t1} \ r \\
\text{geta} & \ r_{t2} \ r \\
\text{minus} & \ r_{t1} \ r_{t1} \ r_{t2} // \text{Offset to first address, i.e., linking table (b-a)} \\
\text{lea} & \ r \ r_{t1} \\
\text{load} & \ r \ r \\
\text{lea} & \ r \ ... // ... \text{replaced with offset to } f \text{ in the linking table} \\
\text{move} & \ r_{t1} \ 0 \\
\text{move} & \ r_{t2} \ 0 \\
\text{load} & \ r \ r // f \text{ capability loaded to register } r
\end{align*}
\]

**call** \( r(\hat{r}_{args}, \hat{r}_{prev}) \)

\( \hat{r}_{args} \) and \( \hat{r}_{prev} \) 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 $r_{priv}$
  - Code return capability (opc)
  - Activation code
- Store the words in $r_{priv}$ 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 $r_{args}$.

**Jump** Jump to register $r$

**Activation code** The activation code does the following:

- Move the stored “private” words in to their respective $r_{priv}$ registers.
- Load the return capability to pc

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

```plaintext
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
ispdr r_t1 r_1
ispdr r_t2 r_2
minus r_t1 r_t1 r_t2
jnz r_t3 r_t1
  // set up capability for cap case:
moved 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
moved 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
moved r_t4 pc
lea r_t4 ... // .. offset to success

fail:
  // get the flag capability
moved 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
stored 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.

\texttt{mclear \( r \)} Stores 0 to all the memory cells the capability \( r \) governs.\footnote{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.}

Possible implementation:

\begin{verbatim}
move r_t r
getb r_t1 r_t
geta r_t2 r_t
minus r_t2 r_t1 r_t2
\end{verbatim}
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 \check{r}

Moves 0 to all the registers in the list \check{r}.

Possible implementation: Say \check{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
  integrity 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 rwx-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:

\[
\begin{align*}
\text{lea } r_{\text{stk}} & \ 1 \\
\text{store } r_{\text{stk}} & \ r
\end{align*}
\]

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

\[
\begin{align*}
\text{load } r & \ r_{\text{stk}} \\
\text{minus } r_{\text{t1}} & \ 0 \ 1 \\
\text{lea } r_{\text{stk}} & \ r_{\text{t1}}
\end{align*}
\]

scall \( r(\bar{r}_{\text{args}}, \bar{r}_{\text{priv}}) \)
\( \bar{r}_{\text{args}} \) and \( \bar{r}_{\text{priv}} \) are lists of registers. This call assumes \( r_{\text{stk}} \) 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 \( \bar{r}_{\text{priv}} \) 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, r0, rstk, and rargs.

Jump Jump to register r.

Run the on stack restore code Load the stack capability to rstk. 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 rpriv registers.

Possible implementation, say rargs = rargs,1,...,rargs,m and rpriv = rpriv,1,...,rpriv,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 r_t1
// pop the private state into appropriate 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.

\[ \text{crtcls } [(x_1,r_1),\ldots,(x_n,r_n)] \]  
\[ r_{code} \]  
\[ [(x_1,r_1),\ldots,(x_n,r_n)] \] 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 (x, r_v) r_code` where `|(x, r_v)| = n` (i.e. the activation code, is defined later):

```c
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
```

**prestack** \( r \) Tests if register \( r \) contains a capability with permission RWLX. If not fail, otherwise assume \( r \) points to \(((RWLX, g), base, end, a)\) adjust it to \(((RWLX, g), base, end, base - 1)\).

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([rargs,1, ..., rargs,n], [r0, rpriv,1, ..., rpriv,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.
- `reperm` can be used to test whether something can pass as a stack.
- `reglob` 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 \( r_l \not\in \{pc, r_0\} \) is a register.

\[
\begin{align*}
f1: & \text{ malloc } r_l 1 \\
& \text{ store } r_l 1 \\
& \text{ fetch } r_{\text{adv}} \text{ adv} \\
& \text{ call } r_{\text{adv}}(\emptyset,[r_l]) \\
& \text{ assert } r_l 1 \\
\text{ if: } & \text{ halt}
\end{align*}
\]

For \( f1 \) to work, its local state needs to be encapsulated.

**Lemma 2** (Correctness lemma for \( f1 \)).

For all \( n \in \mathbb{N} \) let

\[
\begin{align*}
c_{\text{adv}} & \overset{\text{def}}{=} ((E, \text{GLOBAL}), \text{base}_{\text{adv}}, \text{end}_{\text{adv}}, \text{base}_{\text{adv}} + \text{offsetLinkFlag}) \\
c_{f1} & \overset{\text{def}}{=} ((RWX, \text{GLOBAL}), f1 - \text{offsetLinkFlag}, 1f, 1f) \\
c_{\text{malloc}} & \overset{\text{def}}{=} ((E, \text{GLOBAL}), \text{base}_{\text{malloc}}, \text{end}_{\text{malloc}}, \text{base}_{\text{malloc}} + \text{offsetLinkFlag}) \\
m & \overset{\text{def}}{=} ms_{f1} \uplus ms_{\text{flag}} \uplus ms_{\text{link}} \uplus ms_{\text{adv}} \uplus ms_{\text{malloc}} \uplus ms_{\text{frame}}
\end{align*}
\]

and

- \( c_{\text{malloc}} \) satisfies the specification for malloc and \( t_{\text{malloc}, 0} \) is the region from the specification.

where

\[
\begin{align*}
\text{dom}(ms_{f1}) &= [f1 - \text{offsetLinkFlag}, 1f] \\
\text{dom}(ms_{\text{flag}}) &= [\text{flag}, \text{flag}] \\
\text{dom}(ms_{\text{link}}) &= [\text{link}, \text{link} + 1] \\
\text{dom}(ms_{\text{adv}}) &= [\text{base}_{\text{adv}}, \text{end}_{\text{adv}}] \\
ms_{\text{malloc}} &\cdot_{n} [0 \mapsto t_{\text{malloc}, 0}]
\end{align*}
\]

and

- \( ms_{f1}(f1 - \text{offsetLinkFlag}) = ((RO, \text{GLOBAL}), \text{link}, \text{link} + 1, \text{link}) \), \( ms_{f1}(f1 - \text{offsetLinkFlag} + 1) = ((RW, \text{GLOBAL}), \text{flag}, \text{flag}, \text{flag}) \), the rest of \( ms_{f1} \) contains the code of \( f1 \).

- \( ms_{\text{flag}} = [\text{flag} \mapsto 0] \)
- \( ms_{\text{link}} = [\text{link} \mapsto c_{\text{malloc}}, \text{link} + 1 \mapsto \text{adv}] \)
- \( ms_{\text{adv}} \) contains a global read-only capability for \( ms_{\text{link}} \) on its first address. The remaining cells of the memory segment only contain instructions.

if

\[
(\text{reg}[pc \mapsto c_{f1}], m) \to_{n} (\text{halted}, m'),
\]

then

\[
m'(\text{flag}) = 0
\]

\[\blacksquare\]
Proof of Lemma 2. Let \( n \) be given and assume the premises in the lemma. Consider the following part of the execution:

\[
(reg[pc \mapsto c_{f1}], m) \rightarrow_1 (reg_0[pc \mapsto c_{malloc}][r_0 \mapsto c_{f1}][r_1 \mapsto 1], m)
\]

Where \( c_{f1} \) is the return address. Use the malloc specification with

\[
\begin{align*}
\ell_{malloc} &= \ell_{malloc,0} \\
ms_{footprint} &= ms_{malloc} \\
\Phi_{reg(r_1)} &= size = 1
\end{align*}
\]

to get

\[
(reg_0[pc \mapsto c_{malloc}][r_0 \mapsto c_{f1}][r_1 \mapsto 1], m) \rightarrow_j (reg_0[pc \mapsto c_{f1}][r_0 \mapsto c_{f1}][r_1 \mapsto c_i], m')
\]

for some \( j \) where for some \( \ell'_{malloc} \equiv \text{pub} \ell_{malloc,0} \)

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

Continue the execution to the next malloc hidden in \textbf{call}.

\[
(reg_0[pc \mapsto c_{f1}][r_0 \mapsto c_{f1}][r_1 \mapsto c_i], m') \rightarrow_k (reg_0[pc \mapsto c_{malloc}][r_0 \mapsto c_{f1}][r_1 \mapsto len_{ar}][r_l \mapsto c_i], m'')
\]

where

6. \( m'' = m'[l \mapsto 1] \)

Use the malloc specification notice:

- \( \text{len}_{ar} \) is the needed size for the activation record.
- \[ \] and (downwards closure) gives us the needed memory segment satisfaction.
- \( ms_{footprint} = ms'_{malloc} \)

Get:

\[
(reg_0[pc \mapsto c_{malloc}][r_0 \mapsto c_{f1}'][r_1 \mapsto len_{ar}][r_l \mapsto c_i], m'') \rightarrow_j (reg_0[pc \mapsto c_{f1}][r_0 \mapsto c_{f1}'][r_1 \mapsto c_{ar}][r_l \mapsto c_i], m^{(3)})
\]

for some \( j' \) where for some \( [0 \mapsto \ell'_{malloc}] \equiv \text{pub} [0 \mapsto \ell_{malloc}] \)

7. \( m'' = ms_{f1} \oplus ms_{flag} \oplus ms_{link} \oplus ms_{adv} \oplus ms_l \oplus ms_{ar} \oplus ms''_{malloc} \oplus ms_{frame} \)
8. \( ms''_{malloc} : n_j-j' [0 \mapsto \ell'_{malloc}] \)
9. \( \text{dom}(ms_{ar}) = [b, e], \) and \( e - b = len_{ar} \)
10. \( c_l = ((\text{RWX, GLOBAL}), b, e, b) \)
11. \( \forall a \in [b, c], \text{ms}_{ar}(a) = 0 \)

Continue execution until just after the jump to \( \text{adv} \).

\[
(reg_0[pc\mapsto e_{f1}'][r_0\mapsto e_{j1}'][r_1\mapsto c_{ar}][r_i\mapsto c_{l}], m^{(3)}) \rightarrow_{k'} \left( reg_0[pc\mapsto \text{updatePcPerm}(c_{adv})][r_1\mapsto c_{adv}][r_0\mapsto c_{ar}'], m^{(3)} \right)
\]

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., \( c_{l}, e_{f1}' \) (the return address in \( f1 \)), and activation code.

- \( e_{ar}' = ((E, \text{LOCAL})b, e, b + \text{offset}) \) where \( b + \text{offset} \) is the first address of the activation code.

Define

- \( W = \{0 \mapsto i_{malloc}'[1 \mapsto i_{malloc}'_{\text{base}_a,e_{ps}}][2 \mapsto i_{malloc}'_{\text{perm}, ms_{link}}][3 \mapsto \cdots} \)

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 \( \text{updatePcPerm}(c_{adv}) \) using world \( W \), so show

- \( (n, (\text{base}_a, e_{ps})) \in \text{readCondition}(\text{GLOBAL})(W) \)
  
  - Show: \( i_{malloc}'_{\text{base}_a,e_{ps}} \overset{n}{\sim} i_{malloc}'_{\text{base}_a,e_{ps}} : \text{Follows from Lemma} \text{[22]} \)

Have

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

Let \( n' = n - j - j' - k - k' \) and show

1. \( ms_{n'}, W \)

1.1. Split the memory into the disjoint unions of \( \text{[1]} \) and show:

1.1.1. case: \((n', ms_{malloc}) \in i_{malloc}.H(i_{malloc}.s)(W)\)
  
  1.1.1.1. Use \( ms_{malloc} : i'_{malloc} [0 \mapsto i'_{malloc}] \) with malloc specification context independence property.

1.1.2. case: \((n', ms_{adv}) \in H_{\text{base}_a,e_{ps}}^{\text{malloc}} 1W\)
  
  1.1.2.1. Show \( \forall a \in [\text{base}_a, e_{ps}], ((n' - 1, ms(a)) \in \mathcal{V}(W) \land ms(a) \text{ is non-local}) \)
  
  1.1.2.2. \( a \neq \text{base}_a \): trivial, contains instruction only and they are non-local.

  SFTS \( i_{malloc}'_{\text{perm}, ms_{link}} \overset{n'}{\sim} i_{malloc}'_{\text{link}, link + 1} \) which follows from Lemma \text{[23]}.

1.1.3. \((n', ms_{link}) \in H_{\text{link}, link + 1}^W(1)\):

  This boils down to showing:

  1.1.3.1. \((n' - 1, e_{malloc}) \in \mathcal{V}(W) : \text{Follows from Lemma} \text{[50]} \)
1.1.3.2. \((n' - 1, c_{adv}) \in \mathcal{V}(W)\): for \(n'' < n' - 1\) and \(W' \sqsupseteq_{\text{priv}} W\) show:
\[(n'', \text{updatePcPerm}(c_{adv})) \in \mathcal{E}(W')\].
follows from Lemma 79 together with Lemma 79 and the fact that \(c_{adv}\) is non-local.

1.1.4. The last case follows from Lemma 67.

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

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 \(W' \sqsupseteq_{\text{pub}} W\) be given and show \((n'', \text{updatePcPerm}(c'_{ar})) \in \mathcal{E}(W')\)
\[\text{Let } n^{(3)} \leq n'', m'':_{n^{(3)}} W', \text{ and } (n^{(3)}, reg) \in \mathcal{O}(W')\]
Assume \((reg[pc \mapsto \text{updatePcPerm}(c'_{ar})], ms') \in \mathcal{O}(W')\)
\[\text{Show } (n^{(3)}, (reg[pc \mapsto \text{updatePcPerm}(c'_{ar})], ms') \mapsto_{ms} (\text{halted, } m')) \text{ for some } k'' \leq n^{(3)}, m'\text{ and } ms_{frame}.\]
Due to \(ms' :_{n^{(3)}} W', ms_{f_1}, ms_{flag}, ms'_{ar},\) and \(ms_l\) are unchanged.

The execution loads \(c_l\) to \(r_l\) and jumps to \(c^{(3)}_f\) (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., \(m' = ms' \cup ms_{frame}\) 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
\[\left( n', (reg_0[pc \mapsto \text{updatePcPerm}(c_{adv})][r_1 \mapsto c_{adv}][r_0 \mapsto c'_{ar}], m^{(3)}) \right) \in \mathcal{O}(W)\]

By initial assumption of the lemma, the execution halts. Use \(ms_{frame}, m'\) and the number of steps it takes to halt to get: \(W' \sqsupseteq_{\text{priv}} W, ms_r,\) and \(ms'\) s.t. \(m' = ms_r \cup ms' \cup ms_{frame}\) and \(ms' :_{n} W'\). As \(\iota_{flag}\) is a permanent region, we know it is still in \(W'\), so \(m'(\iota_{flag}) = 0\).

4.2 Encapsulation of Local State Using Local Capabilities and \texttt{scall}

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

\begin{verbatim}
f2: push 1
fetch r1 adv
scall r1([], [])
pop r1
assert r1 1
2f: halt
\end{verbatim}
Lemma 3 (Correctness lemma for f2). Let

\[ c_{adv} \overset{df}{=} ((E, GLOBAL), base_{adv}, end_{adv}, base_{adv} + \text{offsetLinkFlag}) \]
\[ c_{f2} \overset{df}{=} ((RWX, GLOBAL), f2 - \text{offsetLinkFlag}, 2f, f2) \]
\[ c_{malloc} \overset{df}{=} ((E, GLOBAL), base_{malloc}, end_{malloc}, base_{malloc} + \text{offsetLinkFlag}) \]
\[ c_{stk} \overset{df}{=} ((RWLX, LOCAL), base_{stk}, end_{stk}, base_{stk} - 1) \]
\[ c_{link} \overset{df}{=} ((RO, GLOBAL), link, link + 1, link) \]
\[ \text{reg} \in \mathbb{R} \]
\[ m \overset{df}{=} ms_{f2} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame} \]

and

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

where

\[ \text{dom}(ms_{f2}) = [f2 - \text{offsetLinkFlag}, 2f] \]
\[ \text{dom}(ms_{flag}) = [flag, flag] \]
\[ \text{dom}(ms_{link}) = [link, link + 1] \]
\[ \text{dom}(ms_{stk}) = [base_{stk}, end_{stk}] \]
\[ \text{dom}(ms_{adv}) = [base_{adv}, end_{adv}] \]
\[ ms_{malloc} : n[0 \mapsto \iota_{malloc,0}] \quad \text{for all } n \in \mathbb{N} \]

and

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

if

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

then

\[ m'(flag) = 0 \]

\[ \blacksquare \]

Proof of Lemma 3 (using \texttt{scall} lemma). Let \( n \) be given and make the assumptions of the lemma. If we can show

\[ (n, (\text{reg}[pc \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], m \uplus ms_{stk})) \in O(W) \quad (1) \]

for

\[ ms = ms_{f2} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \]

and

\[ W = [0 \mapsto \iota_{malloc,0}][1 \mapsto \iota_{flag}(\text{perm}, ms_{f2} \uplus ms_{flag})][2 \mapsto \iota_{link}(\text{perm}, ms_{link})][3 \mapsto \iota_{base_{adv}, end_{adv}}] \]

29
then we are done as we by assumption has
\[(\text{reg}[\text{pc} \mapsto c_2][r_{\text{stk}} \mapsto c_{\text{stk}}], m) \rightarrow_n (\text{halted}, m')\]
so \([1]\) gives us a \(W' \cong^{\text{priv}} W\) where \(W'\) satisfy part of \(m'\). As \(m_{\text{flag}}\) is governed by a perm region, so it is unchanged. In other words
\[m'(\text{flag}) = 0\]
So it suffices to show \([1]\). To this end use Lemma \([8]\) Let \(m_f\) be given, then
\[(\text{reg}[\text{pc} \mapsto c_2][r_{\text{stk}} \mapsto c_{\text{stk}}], m_f) \rightarrow_k (\text{reg}', m \uplus m_{\text{stk}}' \uplus m_f)\]
where
- \((\text{reg}', m)\) is looking at \(\text{scall} r_{\text{adv}}([], [r_i])\) followed by \(c_{\text{next}}\)
- \(c_{\text{next}}\) is \(c_2\) that points to the instruction after the \(\text{scall}\).
- \(\text{reg}'\) points to stack with \([\text{base}_{\text{stk}} \mapsto 1]\) used and \(m_{\text{unused}}\) unused
  - for some \(m_{\text{unused}}\) where \(m_{\text{stk}}' = [\text{base}_{\text{stk}} \mapsto 1] \uplus m_{\text{unused}}\).
- \(\text{reg}'(r_{\text{adv}}) = c_{\text{adv}}\)

In order to show the observation part necessary for Lemma \([8]\) we use the ”\(\text{scall}\) works”-Lemma (Lemma \([58]\)). Show the following

1. \(m:s_n \rightarrow_k W\)
   Use Lemma \([66]\) with
   1.1. \(m_f \uplus m_{\text{flag}} : s_n \rightarrow l_{\text{sta}}(\text{perm}, m_f \uplus m_{\text{flag}})\)
   Lemma \([67]\)
   1.2. \(m_{\text{adv}} \uplus m_{\text{malloc}} \uplus m_{\text{link}} : s_n \rightarrow_k W_{\text{part}}\)
   where
   \[W_{\text{part}} = [0 \mapsto l_{\text{malloc},0}][2 \mapsto l_{\text{sta},u}(\text{perm}, m_{\text{link}})][3 \mapsto l_{\text{adv},w_{\text{link}}}]\]
   This amounts to
   1.2.1. \((n - k - 1, m_{\text{malloc}}) \in H 1 W_{\text{part}}\) where \(H\) is the interpretation of the \(l_{\text{malloc},0}\) region.
   Follows from the malloc specification.
   1.2.2. \((n - k - 1, m_{\text{adv}}) \in H_{\text{adv}} 1 W_{\text{part}}\)
   Can be shown using Lemma \([25]\)
   1.2.3. \((n - k - 1, m_{\text{link}}) \in H_{\text{link},u}(m_{\text{link}}) 1 W_{\text{part}}\)
   This amounts to showing
   1.2.3.1. \((n - k - 2, c_{\text{malloc}}) \in \mathcal{V}(W_{\text{part}})\) Follows from Lemma \([50]\)
   1.2.3.2. \((n - k - 2, c_{\text{adv}}) \in \mathcal{V}(W_{\text{part}})\)
   Follows from Theorem \([2]\) using Lemma \([22]\)

2. Hyp-Callee
   Assume
   - \(\text{dom}(m_{\text{unused}}) = \text{dom}(m_{\text{act}} \uplus m_{\text{unused}}')\),

30
• $W' = \text{revokeTemp}(W)[\{\text{sta}(\text{temp}, \text{ms}_{\text{stk}} \uplus \text{ms}_{\text{act}}), \text{wul}(\text{dom}(\text{ms}_{\text{unused}}))\}]$
• $\text{ms}''_{n-k-1} W''$
• $\text{reg}'$ points to stack with $\emptyset$ used and $\text{ms}_{\text{unused}}'$ unused
• $\text{reg}' = \text{reg}_0[pc \mapsto \text{updatePcPerm}(c_{\text{adv}}), r_0 \mapsto c_{\text{ret}}, r_{\text{stk}} \mapsto c_{\text{stk}}', r_{\text{adv}} \mapsto c_{\text{adv}}]$
• $(n-k-1, c_{\text{ret}}) \in \mathcal{V}(W')$
• $(n-k-1, c_{\text{stk}}') \in \mathcal{V}(W')$

Show

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

By Theorem 2 we get

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

getting the desired result amounts to

2.1. $(n-k-1, c_{\text{adv}}) \in \mathcal{V}(W)$

To this end let $n' < n-k-1$ and $W'' \supseteq \text{priv} W'$ be given and show

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

Follows from Theorem 2 and Lemma 22.

3. Hyp-Cont

Assume

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

and show

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

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

$$(\text{reg}''', \text{ms}''_f[\text{base}_{\text{stk}} \mapsto 1] \uplus \text{ms}'''_{\text{unused}} \uplus \text{ms}_f) \rightarrow_k (\text{halted, } \text{ms}'' \uplus [\text{base}_{\text{stk}} \mapsto 1] \uplus \text{ms}_{\text{unused}}'')$$

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

ML-like program:

\footnote{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`

```plaintext
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 ∈ ℤ` let

}\[\begin{align*}
    c_{adv} &\overset{\text{def}}{=} ((E, GLOBAL), base_{adv}, end_{adv}, base_{adv} + \text{offsetLinkFlag}) \\
    c_{f3} &\overset{\text{def}}{=} ((RWX, GLOBAL), f3 - \text{offsetLinkFlag}, 3f, f3) \\
    c_{stk} &\overset{\text{def}}{=} ((RWLX, LOCAL), base_{stk}, end_{stk}, base_{stk} - 1) \\
    c_{malloc} &\overset{\text{def}}{=} ((E, GLOBAL), base_{malloc}, end_{malloc}, base_{malloc} + \text{offsetLinkFlag}) \\
    c_{link} &\overset{\text{def}}{=} ((RO, GLOBAL), link, link + 1, link) \\
    reg \in \text{Reg} &\overset{\text{def}}{=} ms_{f3} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{align*}\]

and

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

32
where

\[
\begin{align*}
\text{dom}(ms_{f3}) &= [f3 - offsetLinkFlag, 3f] \\
\text{dom}(ms_{flag}) &= [\text{flag}, \text{flag}] \\
\text{dom}(ms_{link}) &= [\text{link}, \text{link} + 1] \\
\text{dom}(ms_{stk}) &= [\text{base}_{stk}, \text{end}_{stk}] \\
\text{dom}(ms_{adv}) &= [\text{base}_{adv}, \text{end}_{adv}] \\
ms_{malloc} :: n \mapsto \iota_{malloc, 0}
\end{align*}
\]

and

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

if

\[
(reg[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \rightarrow n (\text{halted}, m')
\]
then

\(m'(\text{flag}) = 0\)

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 \(\iota_{nwl,p}\) region

and temporary regions

- a region fixing the private part of the stack
- a \(\iota_{pwl}\) 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

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

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 scalls 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

$$W = \left[ 0 \mapsto \iota_{malloc,0} \right]$$

$$\left[ 1 \mapsto \iota_{\text{sta}}(\text{perm}, ms_{flag} \uplus ms_{f2}) \right]$$

$$\left[ 2 \mapsto \iota_{\text{sta,u}}(\text{perm}, ms_{link}) \right]$$

$$\left[ 3 \mapsto \iota_{\text{base,adv, end,adv}} \right]$$

Further define

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

34
If we can show
\[(n + 1, (\text{reg}[\text{pc} \mapsto c_{f3}][\text{r}_{\text{stk}} \mapsto c_{\text{stk}}], m_{\text{malloc}} \uplus m' \uplus m_{\text{adv}} \uplus m_{\text{stk}})) \in O(W), \tag{3}\]
then using \(m_{\text{frame}}\) as the frame and \(m'\) as the resulting memory, we get that \(m' = m'' \uplus m_r \uplus m_{\text{malloc}} \uplus m'_{\text{stk}}\) for some \(m'\) and \(m_r\) s.t. \(m'' : 1 \ W\). Region 1 guarantees that the assertion flag is unchanged, so we have
\[m'(\text{flag}) = 0\]

So SFTS 3. To do so, we use Lemma 8. Let \(m_f\) be given. The execution proceeds as follows:
\[
(\text{reg}[\text{pc} \mapsto c_{f3}][\text{r}_{\text{stk}} \mapsto c_{\text{stk}}], m' \uplus m_{\text{stk}} \uplus m_f) \rightarrow_i (\text{reg}', m' \uplus [\text{base}_{\text{stk}} \mapsto 1] \uplus m_{\text{stk}} |_{\text{base}_{\text{stk}} + 1, \text{end}_{\text{stk}}} \uplus m_f),
\]
where
\[(\text{reg}', m')\] is looking at \textsc{call} \([[], []]\) followed by \(c_{\text{next}}\)
where \(c_{\text{next}}\) is \(c_{f3}\) adjusted to point to the next instruction, namely \text{pop} \(r_1\). Further we have
- \text{reg}' points to stack with \([\text{base}_{\text{stk}} \mapsto 1]\) used and \(m_{\text{stk}} |_{\text{base}_{\text{stk}} + 1, \text{end}_{\text{stk}}}\) unused
and \(i\) is a suitable number of steps.

To show
\[(n - i, (\text{reg}', m' \uplus [\text{base}_{\text{stk}} \mapsto 1] \uplus m_{\text{stk}} |_{\text{base}_{\text{stk}} + 1, \text{end}_{\text{stk}}})) \in O(W)\]
We use Lemma 58 (we do not use the local frame in the lemma) which requires us to show
1. \(m' : n-i \ W\)
   Partition \(m'\) as follows:
   1.1. \(m_{\text{malloc}}\): governed by \(\iota_{\text{malloc}, 0}\), use malloc specification.
   1.2. \(m_{\text{flag}} \uplus m_{f2}\): governed by region 1, only this memory segment is accepted.
   1.3. \(m_{\text{link}}\): 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. \((n - i, c_{\text{malloc}}) \in \mathcal{V}(W)\): Follows from Lemma 50.
   1.3.2. \((n - i, c_{\text{adv}}) \in \mathcal{V}(W)\):
   We will show
   \[
   \forall W' \sqsupseteq \text{priv} W, (n, c_{\text{adv}}) \in \mathcal{V}(W') \tag{4}
   \]
   which will give us what we need using downwards closure as well as a result for later use.
   Let \(W' \sqsupseteq \text{priv} W\) be given and show
   \[
   (n, (\text{base}_{\text{adv}}, \text{end}_{\text{adv}}, \text{base}_{\text{adv}} + \text{offsetLinkFlag})) \in \text{enterCondition(GLOBAL)}(W')
   \]
to this end let \(W'' \sqsupseteq \text{priv} W'\) and \(n' < n\) be given and show
   \[
   (n', \text{updatePcPerm}(c_{\text{adv}})) \in \mathcal{E}(W'')
   \]
   This follows from the FTLR (Theorem 2) if we can show
   \[
   (n', \text{base}_{\text{adv}}, \text{end}_{\text{adv}}) \in \text{readCondition(GLOBAL)}(W'')
   \]
   \(\iota_{\text{base}_{\text{adv}}, \text{end}_{\text{adv}}}\) governs the adversary, so the result follows from Lemma 22.
1.4. $ms_{adv}$: Follows from Lemma 23.

2. Hyp-Callee

Assume:

- $\text{dom}(ms_{stk}|_{\text{base}_{stk}+1, \text{end}_{stk}}) = \text{dom}(ms_{act} \oplus ms'_{\text{unused}})$
- $W' = \text{revokeTemp}(W)[\text{sta}(\text{temp}, [\text{base}_{stk} \mapsto 1] \oplus ms_{act}), t^\text{pub}(\text{dom}(ms'_{\text{unused}}))]$
- $ms'' : n_i \mapsto W'$
- $\text{reg}''$ points to stack with $\emptyset$ used and $ms'_{\text{unused}}$ unused
- $\text{reg}'' = \text{reg}_0[\text{pc} \mapsto \text{updatePcPerm}(\text{reg}'(r)), r_0 \mapsto c_{ret}, r_{stk} \mapsto c'_{stk}, r \mapsto \text{reg}'(r)]$
- $(n_i - 1, c_{ret}) \in V(W')$
- $(n_i - 1, c'_{stk}) \in V(W')$

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

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

2.1. $ms'' : n_i \mapsto W'$: By the above assumptions

2.2. $(n_i - 1, \text{reg}'') \in V(W')$:

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

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

2.2.3. $(n_i - 1, c_{adv}) \in V(W')$: follows from [23]

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

we get

$$(n_i - 1, (ms'', \text{reg}'')) \in O(W')$$

3. Hyp-Cont

Assume:

- $n' \leq n_i - 2$
- $W'' \equiv \text{pub} \text{revokeTemp}(W)$
- $ms'' : n' \mapsto \text{revokeTemp}(W'')$
- for all $r$, we have that:

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

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

and show

3.1. $(\text{reg}'', ms'' \oplus [\text{base}_{stk} \mapsto 1] \oplus ms''_{\text{unused}}) \in O(\text{revokeTemp}(W''))$

As $W'' \equiv \text{pub} W$, 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:

$$(\text{reg}'', ms'' \oplus [\text{base}_{stk} \mapsto 1] \oplus ms''_{\text{unused}} \oplus ms'_f) \rightarrow \text{E} (\text{reg}'', ms'' \oplus [\text{base}_{stk} \mapsto 2] \oplus ms''_{\text{unused}} \oplus ms'_f)$$

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

By Lemma \[\text{Lemma 58}\], it suffices to show

3.1.1. \((n' - k, (reg^{(3)}, ms'' \uplus [\text{base}_{\text{stk}} \mapsto 2] \uplus ms''_{\text{unused}})) \in \mathcal{O}(\text{revokeTemp}(W''))\)

Show this using Lemma \[\text{Lemma 58}\]. Show:

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

3.1.1.2. Hyp-Call

Assume:

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

This argument is almost identical to the one we just did for the first call:

Using the FTLR, we get \((n - i - 1, updatePcPerm(c_{\text{adv}})) \in \mathcal{E}(W^{(3)})\). Which we use with

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

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

3.1.1.2.2.1. \((n' - k - 1, c_{\text{adv}}) \in \mathcal{V}(W^{(3)})\) by Assumption \[\text{Lemma 41}\].

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

3.1.1.2.2.3. \((n' - k - 1, c''_{\text{stk}})\) 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)} \nexists^{\text{pub}} \text{revokeTemp}(W'')[[\text{sta}(\text{temp}, ms_{\text{stk}})]]_{\text{sta}(\text{temp}, ms^{(3)}_{\text{act}})}\]
• \(ms^{(3)} : n'' \text{ revokeTemp}(W^{(3)})\)
• for all \(r\), we have that:

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

• \(reg'\) points to stack with \([\text{base}_{\text{stk}} \mapsto 2]\) used and \(ms^{(3)}_{\text{act}}\) unused for some \(ms^{(3)}_{\text{act}}\)

and show

\[\left(n'', (reg^{(3)}, ms^{(3)} \uplus [\text{base}_{\text{stk}} \mapsto 2] \uplus ms^{(3)}_{\text{act}})\right) \in \mathcal{O}(\text{revokeTemp}(W^{(3)}))\]

37
To this end let $ms''_f, m^\prime, \text{ and } j \leq n''$ be given and assume
\[
(reg^{(3)}, ms^{(3)} \cup [base_{stk} \mapsto 2] \cup ms^{(3)}_{\text{unused}} \cup ms''_f) \rightarrow j (\text{halted, } m'')
\]
As the execution halts immediately,
\[
m' = ms^{(3)} \cup [base_{stk} \mapsto 2] \cup ms^{(3)}_{\text{unused}} \cup ms''_f
\]
By assumption we had $ms^{(3)} ;_{\text{rev}} \text{revokeTemp}(W^{(3)})$ and the frame is unchanged, so we can split the memory as needed.

\[\Box\]

### 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.

\[g2: \text{move } r_3 \text{ pc}\]
\[\text{lea } r_3 \ldots\]
\[\text{crtcls } [] \mapsto r_3\]
\[\text{rclear RegisterName } \{pc, r_0, r_1\}\]
\[2g: \text{jmp } r_0\]
\[f5: \text{reqglob } r_1\]
\[\text{prefstack } r_{stk}\]
\[\text{scall } r_1([], [r_0, r_{env}])\]
\[mclear r_{stk}\]
\[\text{rclear RegisterName } \{r_0, pc\}\]
\[5f: \text{jmp } r_0\]

### 4.5 Variant of the “awkward” example

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

\[g = \text{fun } _ => \text{let } x = 0 \text{ in}\]
\[\text{fun } f =>\]
\[\text{x := 0;}\]
\[f();\]
\[\text{x := 1;}\]
\[f();\]
\[\text{assert(x == 1)}\]

Our translation of the example:

\[g1: \text{malloc } r_2 1\]
\[\text{store } r_2 0\]
\[\text{move } r_3 \text{ pc}\]
\[\text{lea } r_3 \ldots\]
\[\text{crtcls } [(x, r_2)] \mapsto r_3\]
\[\text{rclear RegisterName } \{pc, r_0, r_1\}\]
1g: jmp r0
f4: regglob r1
  prestack rstk
  store x 0
  scall r1([],[r0, r1, r_env])
  store x 1
  scall r1([],[r0, r_env])
  load r1 x
  assert r1 1
  mclear rstk
  rclear RegisterName \ {r0, pc}
4f: jmp r0

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

\[
c_{\text{adv}} \triangleq ((\text{RWX, GLOBAL}), \text{base}_{\text{adv}}, \text{end}_{\text{adv}}, \text{base}_{\text{adv}} + \text{offsetLinkFlag})
\]

\[
c_{g1} \triangleq ((\text{E, GLOBAL}), g1 - \text{offsetLinkFlag}, 4f, g1)
\]

\[
c_{\text{stk}} \triangleq ((\text{RWLX, LOCAL}), \text{base}_{\text{stk}}, \text{end}_{\text{stk}}, \text{base}_{\text{stk}} - 1)
\]

\[
c_{\text{malloc}} \triangleq ((\text{E, GLOBAL}), \text{base}_{\text{malloc}}, \text{end}_{\text{malloc}}, \text{base}_{\text{malloc}} + \text{offsetLinkFlag})
\]

\[
c_{\text{link}} \triangleq ((\text{RO, GLOBAL}), \text{link}, \text{link}, \text{link})
\]

\[
m \triangleq ms_{g1} \oplus ms_{\text{flag}} \oplus ms_{\text{link}} \oplus ms_{\text{adv}} \oplus ms_{\text{malloc}} \oplus ms_{\text{stk}} \oplus ms_{\text{frame}}
\]

where

- \( c_{\text{malloc}} \) satisfies the specification for \( \text{malloc} \) with \( \iota_{\text{malloc}, 0} \)

\[
\text{dom}(ms_{g1}) = [g1 - \text{offsetLinkFlag}, 4f]
\]

\[
\text{dom}(ms_{\text{flag}}) = [\text{flag}, \text{flag}]
\]

\[
\text{dom}(ms_{\text{link}}) = [\text{link}, \text{link}]
\]

\[
\text{dom}(ms_{\text{stk}}) = [\text{base}_{\text{stk}}, \text{end}_{\text{stk}}]
\]

\[
\text{dom}(ms_{\text{adv}}) = [\text{base}_{\text{adv}}, \text{end}_{\text{adv}}]
\]

\[
ms_{\text{malloc}} : n [0 \mapsto \iota_{\text{malloc}, 0}]
\]

and

- \( ms_{g1}(g1 - \text{offsetLinkFlag}) = ((\text{RO, GLOBAL}), \text{link}, \text{link}, \text{link}), ms_{g1}(g1 - \text{offsetLinkFlag} + 1) = ((\text{RW, GLOBAL}), \text{flag}, \text{flag}, \text{flag}) \), the rest of \( ms_{g1} \) contains the code of \( g1 \) immediately followed by the code of \( f4 \).

- \( ms_{\text{flag}} = [\text{flag} \mapsto 0] \)

- \( ms_{\text{link}} = [\text{link} \mapsto c_{\text{malloc}}] \)

- \( ms_{\text{adv}}(\text{base}_{\text{adv}}) = c_{\text{link}} \) and all other addresses of \( ms_{\text{adv}} \) contain instructions.

- \( \forall a \in \text{dom}(ms_{\text{stk}}), ms_{\text{stk}}(a) = 0 \)
if

\((\text{reg}_0[\text{pc} \mapsto c_{\text{adv}}][r_{\text{stk}} \mapsto c_{\text{stk}}][r_1 \mapsto c_{g_1}], m) \rightarrow_n (\text{halted}, m')\),

then

\(m'(\text{flag}) = 0\)

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

**Definition 2.**

\(t_x = (\text{perm}, 0, \phi_{\text{pub}}, \phi, H_x)\)

\(\phi_{\text{pub}} = \{(0, 1)\}^*\)

\(\phi = (1, 0) \cup \phi_{\text{pub}}\)

\(H_x s W = \{(n, ms) \mid ms(x) = s \land n > 0\} \cup \{(0, ms)\}\)

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

**Proof of Lemma 6.**

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

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

\(W = [0 \mapsto t_{\text{malloc}, 0}]\)

\([1 \mapsto t_{\text{sta}, u}(\text{perm}, ms_{\text{link}})]\)

\([2 \mapsto t_{\text{pwl}, \text{end}}, \text{end}_{\text{adv}}]\)

\([3 \mapsto t_{\text{nwfl}, \text{end}}, \text{end}_{\text{adv}}]\)

\([4 \mapsto t_{\text{sta}}(\text{perm}, ms_{g_1} \cup ms_{\text{flag}})]\)

and

\(ms = ms_{g_1} \cup ms_{\text{flag}} \cup ms_{\text{link}} \cup ms_{\text{adv}} \cup ms_{\text{malloc}}\)

If we can show

\((n, (\text{reg}_0[\text{pc} \mapsto c_{\text{adv}}][r_{\text{stk}} \mapsto c_{\text{stk}}][r_1 \mapsto c_{g_1}], ms \cup ms_{\text{stk}})) \in O(W)\) (5)

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

\(m(\text{flag}) = 0\)

So it suffices to show Eq. 5. To this end use the FTLR to show \((n, c_{\text{adv}}) \in E(W)\), so show
1. \((n, (\text{base}_{adv}, \text{end}_{adv})) \in \text{readCondition}(\text{GLOBAL})(W)\)
   Simple using region 3 in \(W\) and Lemma 22.

2. \((n, (\text{base}_{adv}, \text{end}_{adv})) \in \text{writeCondition}(\text{GLOBAL})(W)\)
   Simple using region 3 in \(W\), using Lemma 15.

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

3. \(ms \sqcup ms_{stk} : n W\)

   3.1. \(ms_{g1} \sqcup ms_{flag} : n [4 \mapsto \iota_{\text{sta}}(\text{perm}, ms_{g1} \sqcup ms_{flag})]\)
   Lemma 67.

   3.2. \(ms_{stk} : n [2 \mapsto \iota_{\text{base}_{stk}, \text{end}_{stk}}]\)
   Lemma 68 and assumption that \(ms_{stk}\) is all 0.

   3.3. \(ms_{malloc} \sqcup ms_{link} \sqcup ms_{adv} : n [0 \mapsto \iota_{ms_{malloc}}][1 \mapsto \iota_{\text{sta}_{u}}(\text{perm}, ms_{link})][3 \mapsto \iota_{\text{base}_{adv}, \text{end}_{adv}}]\)
   For convenience define
   \[W_{\text{mini}} = [0 \mapsto \iota_{ms_{malloc}}][1 \mapsto \iota_{\text{sta}_{u}}(\text{perm}, ms_{link})][3 \mapsto \iota_{\text{base}_{adv}, \text{end}_{adv}}]\]
   Partitioning the memory segment in the components of the disjoint union, the \textit{malloc} part follows from assumption \(ms_{malloc} : n [0 \mapsto \iota_{ms_{malloc}}]\) and the \textit{malloc} specification.
   The linking table part of memory amounts to showing:
   \[(n, ms_{link}) \in H^{\text{sta}_{u}}(ms_{link})(1)(\xi^{-1}(W_{\text{mini}}))\]
   which in turn amounts to showing
   \[n - 1, c_{malloc} \in \mathcal{V}(W_{\text{mini}})\]
   which follows from Lemma 50.

   Showing \(n, ms_{adv} \in H^{\text{sta}}_{\text{base}_{adv}, \text{end}_{adv}}(1)(\xi^{-1}(W_{\text{mini}}))\)
   is a bit more involved. It amounts to
   \[\forall a \in \text{dom}(ms_{adv}). (n - 1, ms_{adv}(a)) \in \mathcal{V}(W_{\text{mini}})\]
   which in turn is trivial for everything but
   \[n - 1, c_{link} \in \mathcal{V}(W_{\text{mini}})\]
   This amounts to showing
   \[(n - 1, (\text{link}, \text{link})) \in \text{readCondition}(\text{GLOBAL})(W_{\text{mini}})\]
   which amounts to
   \[\iota_{\text{sta}_{u}}(\text{perm}, ms_{link}) \nabla_{n - 1} \iota_{\text{link}_{\text{link}}}
   \]
   which follows from Lemma 23.

   Using Lemma 68 repeatedly with 3.1, 3.2, and 3.3 gives the desired memory satisfaction.
4. \((n, \text{reg}_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}]) \in \mathcal{R}(W)\)

This amounts to showing

4.1. \((n, \text{reg}_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}](c_{stk})) \in \mathcal{V}(W)\)

The assumptions on \(c_{stk}\) and \(ms_{stk}\) in the lemma entail

\[ \text{reg}_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}][r_{stk}, m_{stk} \in \text{reg}] \]

and further there is a \(p_{\text{ref}}\) region for \(ms_{stk}\) in \(W\), so the result follows from Lemma 62.

4.2. \((n, c_{g1}) \in \mathcal{V}(W)\)

Let \(n < n\) and \(W_1 \equiv \text{priv} \ W\) and show

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

To this end assume \(n_2 \leq n_1\), \(ms_1 := n_2 W_1\), and \((n_2, \text{reg}_1) \in \mathcal{R}(W_1)\) and show

\((n_2, (\text{reg}_1[pc \mapsto \text{updatePcPerm}(c_{g1})], ms_1)) \in \mathcal{O}(W_1)\)

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

\((n_2, (\text{reg}_2, ms_2 \uplus ms_{\text{malloc}} \uplus ms_{\text{cls}} \uplus ms_x)) \in \mathcal{O}(W_2)\)

where

\[ W_2 = W_1[0 \mapsto i_{\text{malloc}}][i_1 \mapsto \iota_{\text{a}}(\text{perm}, ms_{\text{cls}})][i_2 \mapsto \iota_{x}] \]

where \(i_1, i_2 \not\in \text{dom}(W_1)\) and \(i_1 \neq i_2\) and \(\iota_{x}\) is the region in Definition 2, which is a region by Lemma 6. Also

\[ \iota_{\text{malloc}} \equiv \text{priv} \ i_{\text{malloc}} \]

\[ c_x = ((\text{RWX, GLOBAL}), x, x, x) \]

\[ ms_x = [x \mapsto 0] \]

\[ ms_2 \uplus ms_{\text{malloc}} \uplus ms_{\text{cls}} \uplus ms_x \uplus n_2' W_2 \]

\[ c_{\text{env}} = ((\text{RWX, GLOBAL, env}, env, env, env)) \]

\[ ms_{\text{env}} = [\text{env} \mapsto c_x] \]

\[ c_{f4} = ((\text{RWX, GLOBAL}), g1 - \text{offsetLinkFlag}, f4, f4) \]

\[ ms_{\text{cls}} = ms_{\text{env}} \uplus ms_{\text{act}} \]

Finally assume Hyp-Act:

\[ \forall \text{reg}, ms. \text{reg}(pc) = c_{\text{cls}} \Rightarrow \]

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

Show

\[ (n_2 - i, (\text{reg}_2, ms_2 \uplus ms_{\text{malloc}} \uplus ms_{\text{env}} \uplus ms_x \uplus ms_{\text{cls}})) \in \mathcal{O}(W_2) \]

If \(\text{reg}_1(r_0).\text{perm} \not\in \{E, RX, RWX, RWLX\}\), then the execution fails after the jump and is thus trivially true.
If \( \text{reg}_1(r_0) \), \( \text{perm} \in \{ \text{E, RX, RWX, RWLX} \} \), then either \( \text{executeCondition} \) or \( \text{enterCondition} \) holds for the capability in \( \text{reg}_1(r_0) \). Now use \( W_2 \preceq_{\text{pub}} W_1 \) with the appropriate condition to get

\[
(n_2 - i, \text{updatePcPerm}(\text{reg}_1(r_0))) \in \mathcal{E}(W_2)
\]

which in turn gives us \( n_2 \) if we can show the following

4.2.1. \( ms_2 \uplus ms'_\text{malloc} \uplus ms'_{\text{env}} \uplus ms_x \uplus ms'_{\text{cls}} : \iota_{n_2 - i} W_2 \)

We first show the following:

- \( ms_2 \uplus ms'_{\text{malloc}} : \iota_{n_2 - i} W_1[0 \mapsto \iota_{\text{malloc}}] \): we already know this.
- \( ms_{\text{env}} \uplus ms'_{\text{cls}} : \iota_{n_2 - i} [\iota_1 \mapsto e_{\text{perm}}(\text{perm}, ms_{\text{env}} \uplus ms_{\text{cls}})] \): By Lemma 12.
- \( ms_x : \iota_{n_2 - i} \iota_2 \mapsto \iota_x : ms(x) = 0 \), so okay.

4.2.2. \( (n_2 - i, \text{reg}_2) \in \mathcal{R}(W_2) \)

Amounts to showing

4.2.2.1. \( (n_2 - i, \text{reg}_2(r_0)) \in \mathcal{V}(W_2) \) by assumption \( (n_2, \text{reg}_1) \in \mathcal{R}(W_1) \) and \( \mathcal{V} \) monotonicity wrt. \( \preceq_{\text{pub}} \)

4.2.2.2. \( (n_2 - i, \text{c}_{\text{cls}}) \in \mathcal{V}(W_2) \)

Let \( n_3 < n_2 - i \) and \( W_3 \preceq_{\text{priv}} W_2 \) be given and show

\[
(n_3, \text{updatePcPerm}(c_{\text{cls}})) \in \mathcal{E}(W_3)
\]

To this and let \( n_4 \leq n_3 \), \( ms_3 : \iota_{n_4} W_3 \) and \( (n_4, \text{reg}_3) \in \mathcal{R}(W_3) \) and show

\[
(n_4, (\text{reg}_3, [pc \mapsto \text{updatePcPerm}(c_{\text{cls}})], ms_3)) \in \mathcal{O}(W_3)
\] (8)

Let \( ms'_3 \) and \( ms''_3 \) be memory segments such that \( ms_3 = ms'_3 \uplus ms''_3 \) and \( ms'_3 : \iota_{n_4} \text{revokeTemp}(W_3) \) (using Lemma 64). By \( ms_3 : \iota_{n_4} W_3 \) and \( W_3 \preceq_{\text{priv}} W_2 \), we know \( ms_{\text{cls}} \subseteq ms'_3 \), so using Hyp-Act(11), we get \( j \) such that

\[
\forall ms_f. (\text{reg}_3, [pc \mapsto \text{updatePcPerm}(c_{\text{cls}})], ms'_3 \uplus ms''_3 \uplus ms_f) \rightarrow_j
\]

\[
(\text{reg}_3, [pc \mapsto \text{updatePcPerm}(c_{\text{cls}})][r_{\text{env}} \mapsto c_{\text{env}}], ms'_3 \uplus ms''_3 \uplus ms_f)
\] (9)

Using Lemma 8 it suffices to show

\[
(n_4, (\text{reg}_3, [pc \mapsto \text{updatePcPerm}(c_{\text{cls}})][r_{\text{env}} \mapsto c_{\text{env}}], ms'_3 \uplus ms''_3)) \in \mathcal{O}(W_3)
\]

Use Lemma 8 again. This time let \( ms''_3 \) be given and take \( ms_x \), to be the part of \( ms'_3 \) that \( \text{reg}_3(r_{\text{stk}}) \) does not govern. By the operational semantics, we know\[j\]

\[
(\text{reg}_3, [pc \mapsto \text{updatePcPerm}(c_{\text{cls}})][r_{\text{env}} \mapsto c_{\text{env}}], ms'_3 \uplus ms''_3 \uplus ms_f) \rightarrow_{j'} (\text{reg}_4, ms_4 \uplus ms'_4 \uplus ms''_4)
\]

where

- \( (\text{reg}_4, ms_4) \) is looking at \( \text{scall} \ r_4([c_{\text{env}}], r_0, r_1, r_{\text{env}}) \) followed by \( c_{\text{next}} \) - \( c_{\text{next}} \) is the capability pointing to the next instruction.
- \( \text{reg}_4 \) points to stack with \( \emptyset \) used and \( ms_{\text{unused}} \) unused.
- \( \text{prepstack} \) did not fail, so the stack capability must be RWLX and follow the stack convention.

\[3^{\text{the execution may fail, but then the configuration is trivially in the observation relation.}}\]
- \( \text{reg}_4(r_1) \) is a global capability.
  - \( \text{reg}_{\text{glob}} \) did not fail
- \( ms_4(x) = 0 \)
- \( reg_4(r_{\text{env}}) = c_{\text{env}} \)

region \( i_2 \) (the \( l_x \) 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 \( \text{revokeTemp}(W_3) \) 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:

\[ (n_4 - j - j', (\text{reg}_4, ms_4 \uplus ms_r \uplus \emptyset \uplus ms_{\text{unused}})) \in O(W_4) \]

where \( ms_r \) is the local frame of the \text{scall} lemma.

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

### 4.2.2.2.2. Hyp-Callee

We know \( (n_4, \text{reg}_3(r_1)) \in V(W_3) \). 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 \text{executeCondition} or the \text{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 \( W_5 = \text{revokeTemp}(W_4)[^\text{ste}(\text{temp}, \emptyset \uplus ms_{\text{act}} \uplus ms_{\text{r}}), ^\text{pwt}(\text{dom}(ms'_{\text{unused}})))] \triangleright_{\text{priv}} W_3 \), for some \( ms_{\text{act}} \) and \( ms'_{\text{unused}} \).

By the execute/enter condition, we have

\[ (n_4 - j - j' - 1, \text{updatePcPerm}(\text{reg}_3(r_1))) \in E(W_5) \]

Now it suffices to show

### 4.2.2.2.1. \( ms_5 :_{n_4 - j - j' - 1} W_5 \) for some \( ms_5 \) which is one of the assumptions of Hyp-Callee.

### 4.2.2.2.2. \( (n_4 - j - j' - 1, \text{reg}_5) \in R(W_5) \) where \( \text{reg}_5 \) is as described in the \text{scall} lemma Hyp-callee premise.

Amounts to showing:

1) \( (n_4 - j - j' - 1, \text{reg}_3(r_1)) \in V(W_5) \), use Lemma 59 with \( (n_4 - j - 1, \text{reg}_3(r_1)) \in V(W_3) \), the capability is global, and \( W_5 \triangleright_{\text{priv}} W_3 \). 2) The protected return pointer and the stack capability are in the value relation by Hyp-callee assumptions.

which gives us

\[ (n_4 - j - j' - 1, (\text{reg}_5, ms_5)) \in O(W_5) \]

### 4.2.2.3. Hyp-Cont

Assume

- \( n_5 \leq n_4 - j - j' - 2 \)
- \( W_6 \triangleright_{\text{pub}} \text{revokeTemp}(W_4) \)
- \( ms_6 :_{n_5} \text{revokeTemp}(W_6) \)
- \( \text{reg}_5(pc) = c_{\text{next}}, \text{reg}_6(r_0) = \text{reg}_3(r_0), \text{reg}_6(r_1) = \text{reg}_3(r_1), \text{reg}(r_{\text{env}}) = c_{\text{env}} \)
• \(reg_6\) points to stack with \(\emptyset\) used and \(ms''_{\text{unused}}\) unused

Show
\[(n_5, (reg_6, ms_6 \uplus ms_r \uplus \emptyset \uplus ms''_{\text{unused}})) \in O(W_6)\]

Use the \(O\)-anti-reduction lemma (Lemma 8) followed by the \texttt{scall} lemma (Lemma 58). Given \(ms''_f\), we know by the operational semantics and the fact that the program hasn’t changed that
\[(reg_6, ms_6 \uplus ms_r \uplus ms''_{\text{unused}}) \rightarrow_k (reg_7, ms_6[x \mapsto 1] \uplus ms_r \uplus ms''_{\text{unused}})\]

where
- \((reg_7, ms_6[x \mapsto 1])\) is looking at \texttt{scall} 
  \([[], [r_0, r_{\text{env}}]]\) followed by \(c'_{\text{next}}\)
- \(c'_{\text{next}}\) is the current pc capability but looking at \texttt{load} \(r_1\) \(x\).

In \texttt{revokeTemp}(\(W_6\)), we don’t know which state the \(i_x\) region is in, but state 1 is reachable via a public transition, so let \(W_7\) be \texttt{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 \texttt{scall} lemma, to get

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

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

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

5.3. Hyp-Cont

Assume:
- \(n_6 \leq n_5 - k - 2\)
- \(W_8 \sqsupseteq_{\text{pub}} \texttt{revokeTemp}(W_7)\)
- \(ms_7 : i_6 \texttt{revokeTemp}(W_8)\)
- \(reg_8(r_0, r_{\text{env}}) = reg_7(r_0, r_{\text{env}})\)
- \(reg_8(pc) = c'_{\text{next}}\)
- \(reg_8\) points to stack with \(\emptyset\) used and \(ms^{(6)}_{\text{unused}}\) unused for some \(ms^{(6)}_{\text{unused}}\)

Show:
\[(n_6, (reg_8, ms_7 \uplus ms_r \uplus \emptyset \uplus ms^{(5)}_{\text{unused}})) \in O(W_8)\]

Use Lemma 8. Let \(ms^{(4)}_f\) be given, then
\[(reg_8, ms_7 \uplus ms_r \uplus \emptyset \uplus ms^{(5)}_{\text{unused}} \uplus ms^{(4)}_f) \rightarrow_l (reg_9, ms_7 \uplus ms_r \uplus \emptyset \uplus ms_0 \uplus ms^{(4)}_f)\]

where
- \(reg_9(pc) = \text{updatePcPerm}(reg_3(r_0))\) (note \(reg_8(r_0) = reg_3(r_0)\))
- \(reg_9(r_0) = reg_3(r_0)\)
- For all \(r \notin \{pc, r_0\}\), \(reg_9(r) = 0\).
• \( \text{dom}(ms_0) = \text{dom}(ms_{\text{unused}}^{(5)}) \) and \( \forall a \in \text{dom}(ms_0), ms_0(a) = 0 \)

The execution proceeds as above because \( \iota_x \) 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 \( \text{dom}([W_3]_{\{\text{temp}\}}) \) reinstated. Now we show the following:

5.3.1. \( W_{10} \supseteq \text{pub } W_3 \)

We have

\[ \forall r \in \text{dom}(W_3). W_3(r) = W_{10}(r) \]

if the region was permanent in \( W_3 \), then it is there because \( W_{10} \supseteq \text{priv } W_3 \). 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 \( \iota_x \) is in state 0 or 1. In \( W_{10} \) region \( \iota_x \) is in state 1. State 1 can be reached from 0 and 1 using a public transition, so the \( \iota_x \) in \( W_{10} \) is a public future region of the \( \iota_x \) 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. \( ms_7 \uplus ms_r \uplus \emptyset \uplus ms_0 : n_6 \rightarrow_{l} W_{10} \)

First notice that from

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

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

\[ \text{dom}(ms_{\text{unused}}) \subseteq [stk_a, stk_b]. \]

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

\[ ms_7 : n_6 \text{ revokeTemp}(W_8) \] (10)

and

\[ ms_3 : n_4 W_3 \] (11)

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

\[ P(r) = \begin{cases} 
P_8(r) & r \in \text{dom}([W_8]_{\{\text{perm}\}}) \\
ms_{\text{advstk}} \uplus ms_0 & r = r_{\text{advstk}} \\
P_3(r) & \text{otherwise}
\end{cases} \]

Now let \( r \in \text{fl}W, n_7 < n_6 - l \), 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. \( r \in \text{dom}(\{W_8\}_{\text{perm}}) \)
Use \( [10] \) the fact that \( W_{10} \sqsubseteq^{\text{priv}} \text{revokeTemp}(W_8) \) and that permanent regions respect future private world.

5.3.2.2. \( r = r_{\text{advstk}} \)
In this case we know the region is \( t_{\text{advstk}}^{\text{pwl}} \), so we need to show
\[
(n_7, ms_{\text{advstk}} \uplus ms_0) \in H_{\text{advstk}}^{\text{pwl}}(1)(\xi^{-1}(W_{10}))
\]
which amounts to showing
\[
\forall a \in \text{dom}(ms_0). (n_7 - 1, ms_0(a)) \in \mathcal{V}(W_{10}),
\]
which is trivial, and
\[
\forall a \in \text{dom}(ms_{\text{advstk}}). (n_7 - 1, ms_{\text{advstk}}(a)) \in \mathcal{V}(W_{10})
\]
here we use that \( [11] \) entails
\[
\forall a \in \text{dom}(ms_{\text{advstk}}). (n_4 - 1, ms_{\text{advstk}}(a)) \in \mathcal{V}(W_3)
\]
and the fact that \( \mathcal{V} \) is monotone w.r.t \( \sqsupseteq^{\text{pub}} \), \( W_{10} \sqsupseteq^{\text{pub}} W_3 \), and \( \mathcal{V}(W_{10}) \) is downwards-closed.

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

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 \( \sqsupseteq^{\text{pub}}, W_{10} \sqsupseteq^{\text{pub}} W_3 \).

As we were using Lemma \( [5.3.1.] \) we need to show
\[
(n_6 - l, (reg_3, ms_7 \uplus ms_{7'} \uplus \emptyset \uplus ms_0)) \in \mathcal{O}(W_{10})
\]
To this end the use \( reg_3(r_0) = reg_9(r_0) \) and \( (n_4, reg_3(r_0)) \in \mathcal{V}(W_3) \). Assuming that \( reg_9(r_0), \text{perm} \in \{E, RX, RWX, RLX\} \) (if this is not the case, then it is trivial to show the above as the execution fails), then either the \text{executeCondition} or the \text{enterCondition} hold for appropriate values. Now use that \( n_6 - l < n_4 \) and \( W_{10} \sqsupseteq^{\text{pub}} W_3 \)
\( [5.3.2.] \) to get
\[
(n_6 - l, \text{updatePcPerm}(reg_9(r_0))) \in \mathcal{E}(W_{10})
\]
now using \( [5.3.2.] \) and \( [5.3.3.] \) we get the desired result.

\[\Box\]

\( ^4 \)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.

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

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

$$\xi : \text{Wor} \cong \blacksquare([\{\text{revoked}\} +
\{\text{temp}\} \times \text{State} \times \text{Rel} \times (\text{State} \to (\text{Wor} \xrightarrow{\text{mon, nr}} \upred(\text{MemSegment})) +
\{\text{perm}\} \times \text{State} \times \text{Rel} \times (\text{State} \to (\text{Wor} \xrightarrow{\text{mon, nr}} \upred(\text{MemSegment})))))),$$

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

$$W' \sqsupseteq_{\text{priv}} W \iff \xi(W') \sqsupseteq_{\text{priv}} \xi(W)$$

and

$$W' \sqsupseteq_{\text{pub}} W \iff \xi(W') \sqsupseteq_{\text{pub}} \xi(W).$$

We now define the regions to be

$$\text{Region} = \{\text{revoked}\} \uplus
\{\text{temp}\} \times \text{State} \times \text{Rel} \times (\text{State} \to (\text{Wor} \xrightarrow{\text{mon, nr}} \upred(\text{MemSegment})) \uplus
\{\text{perm}\} \times \text{State} \times \text{Rel} \times (\text{State} \to (\text{Wor} \xrightarrow{\text{mon, nr}} \upred(\text{MemSegment})))).$$

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

$$\text{World} = \text{RegionName} \xrightarrow{\delta_\text{ns}} \text{Region}$$

where $\text{RegionName} = \mathbb{N}$.
The two private future region relations satisfies the following properties:

\[
(s, s') \in \phi \quad (v, \phi_{pub}, \phi, H) = (v', \phi'_{pub}, \phi', H') \quad r \in \text{Region}
\]

\[
(v', s', \phi'_{pub}, \phi', H') \sqsupseteq_{\text{priv}} (v, s, \phi_{pub}, \phi, H)
\]

The two public future region relations satisfies the following properties:

\[
(s, s') \in \phi_{pub} \quad (v, \phi_{pub}, \phi, H) = (v', \phi'_{pub}, \phi', H') \quad (\text{temp, } s, \phi_{pub}, \phi, H) \sqsupseteq_{\text{pub revoked}}
\]

\[
(v', s', \phi'_{pub}, \phi', H') \sqsupseteq_{\text{pub revoked}} (v, s, \phi_{pub}, \phi, H)
\]

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

\[
\text{dom}(W') \supseteq \text{dom}(W) \quad \forall r \in \text{dom}(W). W'(r) \sqsupseteq_{\text{pub}} W(r)
\]

\[
W' \sqsupseteq_{\text{pub revoked}} W
\]

**Proof of Theorem 1.** The theorem follows from a more general solution theorem for the category of 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 \( P \).

\[
F_1((X, \sqsupseteq_{\text{priv}}'), (Y, \sqsupseteq_{\text{pub}}')) =
\]

\[
\upharpoonright\upharpoonleft [\mathbb{N} \xrightarrow{\delta} \{	ext{revoked}\} + \{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((Y, \sqsupseteq_{\text{pub}}') \xrightarrow{\text{mon}, \text{ne}} \text{UPred(MemSegment)})) + \{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((X, \sqsupseteq_{\text{priv}}') \xrightarrow{\text{mon}, \text{ne}} \text{UPred(MemSegment)})))], \sqsupseteq_{\text{priv}}
\]

and

\[
F_2((X, \sqsupseteq_{\text{priv}}'), (Y, \sqsupseteq_{\text{pub}}')) =
\]

\[
\upharpoonright\upharpoonleft [\mathbb{N} \xrightarrow{\delta} \{	ext{revoked}\} + \{\text{temp}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((Y, \sqsupseteq_{\text{pub}}') \xrightarrow{\text{mon}, \text{ne}} \text{UPred(MemSegment)})) + \{\text{perm}\} \times \text{State} \times \text{Rels} \times (\text{State} \to ((X, \sqsupseteq_{\text{priv}}') \xrightarrow{\text{mon}, \text{ne}} \text{UPred(MemSegment)})))], \sqsupseteq_{\text{pub}}
\]

The orderings \( \sqsupseteq_{\text{priv}} \) and \( \sqsupseteq_{\text{pub}} \) 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 \( \sqsupseteq_{\text{priv}} \) and \( \sqsupseteq_{\text{pub}} \) and an isomorphism \( \xi \) 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:

\[ [W]_S \overset{\text{def}}{=} \lambda r. \begin{cases} W(r) & W(r).v \in S \\ \bot & \text{otherwise} \end{cases} \]

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

\[ active : \text{World} \rightarrow 2^{\text{RegionName}} \]
\[ active(W) \overset{\text{def}}{=} \text{dom}([W]_{\{\text{perm, temp}\}}) \]

Memory segment satisfaction:

\[ ms : n \ W \iff \exists P : active(W) \rightarrow \text{MemSegment}. \]
\[ ms : n, P \ W \]

\[ ms : n, P \ W \iff \begin{cases} ms = \biguplus_{r \in active(W)} P(r) \land \\ \forall r \in active(W). \\ \exists H, s. \\ W(r) = (\_, s, \_, H) \land \\ (n, P(r)) \in H(s)(\xi^{-1}(W)) \end{cases} \]

Standard regions for when writing locally is permitted:

\( \iota^{pwl} : \mathcal{P} \rightarrow \text{Region} \)
\[ \iota^{pwl} A \overset{\text{def}}{=} (\text{temp}, 1, =, =, H^{pwl} A) \]

\[ H^{pwl} : \mathcal{P}(\text{Addr}) \rightarrow \text{State} \rightarrow (\text{Wor} \overset{\text{max}}{\rightarrow} \text{Pred}(\text{MemSegment})) \]
\[ H^{pwl} A s W \overset{\text{def}}{=} \left\{ (n, ms) \middle| \begin{array}{l} \text{dom}(ms) = A \land \\ \forall a \in A. (n - 1, ms(a)) \in \mathcal{V}(\xi(W)) \end{array} \right\} \cup \{ (0, ms) \} \]

Revoking all temporary regions:

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

Further define

\[ \iota^{pwl}_{\text{base, end}} \overset{\text{def}}{=} \iota^{pwl}([\text{base, end}]) \]
Standard regions for when write local is not allowed:

\[
\kappa^{\text{null}} : \mathcal{P} (\text{Addr}) \rightarrow \text{Region}
\]

\[
\kappa^{\text{null}} A \overset{\text{def}}{=} \{ \text{temp}, 1, =, =, H^{\text{null}} A \}
\]

\[
\kappa^{\text{null}, p} : \mathcal{P} (\text{Addr}) \rightarrow \text{Region}
\]

\[
\kappa^{\text{null}, p} A \overset{\text{def}}{=} \{ \text{perm}, 1, =, =, H^{\text{null}} A \}
\]

\[
H^{\text{null}} : \mathcal{P} (\text{Addr}) \rightarrow \text{State} \rightarrow (\text{Wor} \overset{\overline{\min, \overline{\min}}}{} \text{UPred(MemSegment)})
\]

\[
H^{\text{null}} A s \hat{W} \overset{\text{def}}{=} \left\{ \begin{array}{l}
\text{dom}(ms) = A \\
\forall a \in A.
ms(a) \text{ is non-local} \\
(n - 1, ms(a)) \in \mathcal{V} (\xi (\hat{W}))
\end{array} \right\} \cup \{(0, ms)\}
\]

Further define

\[
\kappa^{\text{null, end}} \overset{\text{def}}{=} \kappa^{\text{null}} ([\text{base}, \text{end}])
\]

\[
\kappa^{\text{null}, p, end} \overset{\text{def}}{=} \kappa^{\text{null, p}} ([\text{base}, \text{end}])
\]

For convenience define

\[
\text{localityReg}(g, W) \overset{\text{def}}{=} \left\{ \begin{array}{l}
\text{dom}(\lfloor W \rfloor (\text{perm, temp})) \text{ if } g = \text{LOCAL} \\
\text{dom}(\lfloor W \rfloor (\text{perm})) \text{ if } g = \text{GLOBAL}
\end{array} \right\}
\]

\[
\text{localityReg}(\text{LOCAL}, W) \text{ are the regions that local capabilities may govern - that is permanent and temporary regions. localityReg}(\text{GLOBAL}, W) \text{ are the regions that global capabilities may govern - that is permanent regions. Now define the following function}
\]

\[
\text{localityReg}(\text{LOCAL}, W) \text{ are the regions that local capabilities may govern - that is permanent and temporary regions. localityReg}(\text{GLOBAL}, W) \text{ are the regions that global capabilities may govern - that is permanent regions. Now define the following function}
\]

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:

\[
(s, \phi_{\text{pub}}, \phi) = (s', \phi'_{\text{pub}}, \phi') \quad \forall \hat{W}. H s \hat{W} \overset{n}{\subseteq} H' s' \hat{W}
\]

\[
(v, s, \phi_{\text{pub}}, \phi, H) \overset{n}{\subseteq} (v', s', \phi'_{\text{pub}}, \phi', H')
\]

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.
\( t = (v, s, \phi_{\text{pub}}, \phi, H) \) is address-stratified iff

\[ \forall s', \hat{W}, n, ms, ms'. (n, ms), (n, ms') \in H \quad s' \quad \hat{W} \Rightarrow \]

\[ \text{dom}(ms) = \text{dom}(ms') \land \]

\[ \forall a \in \text{dom}(ms). (n, ms[a \mapsto ms'(a)]) \in H \quad s' \quad \hat{W} \]

**writeCondition** : \(((\text{Addr} \times \text{Addr}) \to \text{Region}) \times \text{Global} \to \text{World} \xrightarrow{\text{mon}, \text{ne}} \text{UPred(Addr}^2\text{)})

**writeCondition**\( (\iota, g)(W) = \)

\[ \{(n, (\text{base}, \text{end})) \mid \exists r \in \text{localityReg}(g, W). \]

\[ \exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \]

\[ W(r) \xrightarrow{\text{pub}} i_{\text{base}', \text{end}'} \quad \text{and} \]

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

**readCondition** : \text{Global} \to \text{World} \xrightarrow{\text{mon}, \text{ne}} \text{UPred(Addr}^2\text{)}

**readCondition**\( (g)(W) = \)

\[ \{(n, (\text{base}, \text{end})) \mid \exists r \in \text{localityReg}(g, W). \]

\[ \exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]. \]

\[ W(r) \xrightarrow{\text{pwl}} i_{\text{base}', \text{end}'} \}

**executeCondition**\( (g)(W) = \)

\[ \{(n, (\text{perm}, \text{base}, \text{end})) \mid \forall n' < n. \]

\[ \forall W' \supseteq W. \]

\[ \forall a \in [\text{base}, \text{end}]. \]

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

where \( g = \text{LOCAL} \Rightarrow \exists = \exists_{\text{pub}} \)

and \( g = \text{GLOBAL} \Rightarrow \exists = \exists_{\text{priv}} \)

**enterCondition**\( (g)(W) = \)

\[ \{(n, (\text{base}, \text{end}, a)) \mid \forall n' < n. \]

\[ \forall W' \supseteq W. \]

\[ (n', ((\text{RX}, g), \text{base}, \text{end}, a)) \in \mathcal{E}(W') \}

where \( g = \text{LOCAL} \Rightarrow \exists = \exists_{\text{pub}} \)

and \( g = \text{GLOBAL} \Rightarrow \exists = \exists_{\text{priv}} \)

52
Now define the value relation as follows:

\[ \nu \colon \text{World} \xrightarrow{\text{mem, ms}} \text{UPred(Word)} \]

\[ \nu \triangleq \lambda \ W. \{(n, i) \mid i \in \mathbb{Z} \cup \{\infty\}\} \]

\[ \{(n, ((O, g), base, end, a))\} \]

\[ \{(n, ((RO, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W)\} \]

\[ \{(n, ((RW, g), base, end, a)) \mid (n, (base, end)) \in \text{writeCondition}(\nu^\text{null}, g)(W)\} \]

\[ \{(n, ((RWL, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \wedge (n, (base, end)) \in \text{writeCondition}(\nu^\text{null}, g)(W)\} \]

\[ \{(n, ((RX, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \wedge (n, (RX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ \{(n, ((E, g), base, end, a)) \mid (n, (base, end, a)) \in \text{enterCondition}(g)(W)\} \]

\[ \{(n, ((RWX, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \wedge (n, (base, end)) \in \text{writeCondition}(\nu^\text{null}, g)(W) \wedge (n, (RWX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ \{(n, ((RWLX, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \wedge (n, (base, end)) \in \text{writeCondition}(\nu^\text{null}, g)(W) \wedge (n, (RWLX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ \{(n, (RX, base, end)) \in \text{executeCondition}(g)(W) \wedge (n, (RX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ \{(n, (RWLX, g), base, end, a)) \mid (n, (base, end)) \in \text{readCondition}(g)(W) \wedge (n, (base, end)) \in \text{writeCondition}(\nu^\text{null}, g)(W) \wedge (n, (RWLX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ \{(n, (RX, base, end)) \in \text{executeCondition}(g)(W)\} \]

\[ {\O \colon \text{World} \xrightarrow{\text{priv}} \text{UPred(Reg \times MemSegment)}} \]

\[ {\O \triangleq } \begin{cases} \text{AW}, \{(n, (reg, ms)) \mid \forall ms_f, mem', i \leq n.} \\ \\ \text{(reg, ms } \cup \text{ ms}_f) \rightarrow_i (\text{halted, mem}') \\ \Rightarrow \exists W' \xrightarrow{\text{priv}} W, \exists ms_r, ms'. \\ \text{mem'} = ms' \cup ms_r \cup ms_f \wedge \\ \text{ms'} :_n \equiv W' \end{cases} \]
\[ R : \text{World} \xrightarrow{\text{max} \rightarrow} \text{UPred(Reg)} \]
\[ R \overset{d}{=} \lambda W. \{ (n, \text{reg}) \mid \forall r \in \text{RegisterName} \setminus \{ \text{pc} \}. \]
\[ (n, \text{reg}(r)) \in \mathcal{V}(W) \}

\[ E : \text{World} \xrightarrow{\text{max} \rightarrow} \text{UPred(Word)} \]
\[ E \overset{d}{=} \lambda W. \{ (n, \text{pc}) \mid \forall n' \leq n. \]
\[ \forall (n', \text{reg}) \in \mathcal{R}(W). \]
\[ \forall ms : n' W. \]
\[ (n', (\text{reg}[\text{pc} \mapsto \text{pc}], ms)) \in \mathcal{O}(W) \}

### 5.3 Useful regions

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

\[ t^{\text{sta}}(v, ms) = (v, 1, =, =, H^{\text{sta}} ms) \]
\[ H^{\text{sta}} ms s \hat{W} = \{(n, ms) \mid n > 0 \} \cup \{(0, ms') \mid ms' \in \text{Mem}\} \]

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

\[ t^{\text{sta},u}(v, ms) = (v, 1, =, =, H^{\text{sta},u} ms) \]
\[ H^{\text{sta},u} ms s \hat{W} = \left\{ (n, ms') \mid ms' = ms \land \right. \]
\[ \left. \forall a \in \text{dom}(ms). \right. \]
\[ \left. ms(a) \text{ is non-local} \right. \]
\[ (n - 1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \} \cup \{(0, ms') \mid ms' \in \text{Mem}\} \]

\[ t^{\text{cst}}(v, n) = (v, 1, =, =, H^{\text{cst}} n) \]
\[ H^{\text{cst}} n' s \hat{W} = \{(n, ms) \mid n > 0 \land \forall a \in \text{dom}(ms). ms(a) = n' \} \cup \{(0, ms') \mid ms' \in \text{Mem}\} \]

### 5.4 Lemmas

#### 5.4.1 Anti-reduction for the observation relation

**Lemma 7** (Failing terms are in \( \mathcal{O} \) and \( \mathcal{E} \)). If \( \text{(reg, ms} \not\equiv \text{ms}f) \rightarrow_{*} \text{failed for all ms}_f, \text{ then (n, (reg, ms)} \in \mathcal{O}(W) \text{ for any W.} \]

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

\[
\forall n, n', i, \text{reg}, \text{reg}', \text{ms}, \text{ms}', \text{ms}_r, W, W'.
\]

\[
n' \geq n - i \land W' \supseteq_{\text{priv}} W \land
\]

\[
(\forall ms_f. (\text{reg}, \text{ms} \uplus \text{ms}_r \uplus ms_f) \rightarrow (\text{reg}', \text{ms}' \uplus \text{ms}_r) \uplus ms_f) \land
\]

\[
(n', (\text{reg}', \text{ms}')) \in O(W')
\]

\[
\Rightarrow (n, (\text{reg}, \text{ms} \uplus \text{ms}_r)) \in O(W)
\]

\[\qed\]

Proof of Lemma 8

Assume

1. $n' \geq n - i$

2. $W_2 \supseteq_{\text{priv}} W_1$

3. $\forall ms_f. (\text{reg}, \text{ms} \uplus \text{ms}_r \uplus ms_f) \rightarrow (\text{reg}', \text{ms}' \uplus \text{ms}_r \uplus ms_f)$

4. $(n', (\text{reg}', \text{ms}')) \in O(W_2)$

Show

\[
(n, (\text{reg}, \text{ms} \uplus \text{ms}_r)) \in O(W_1)
\]

To this end let $\text{ms}_{\text{frame}}$, $m'$ and $j$ be given and assume

\[
(\text{reg}, \text{ms} \uplus \text{ms}_r \uplus \text{ms}_{\text{frame}}) \rightarrow (\text{halted}, m') \tag{12}
\]

From[8] instantiated with $\text{ms}_{\text{frame}}$ we know

\[
(\text{reg}, \text{ms} \uplus \text{ms}_r \uplus \text{ms}_{\text{frame}}) \rightarrow (\text{reg}', \text{ms}' \uplus \text{ms}_r \uplus \text{ms}_{\text{frame}}) \tag{13}
\]

Using[12] and[13] we get

\[
(\text{reg}', \text{ms}' \uplus \text{ms}_r \uplus \text{ms}_{\text{frame}}) \rightarrow (\text{halted}, m')
\]

Using this with[4] and $\text{ms}_r \uplus \text{ms}_{\text{frame}}$ as frame, we get $W_3 \supseteq_{\text{priv}} W_2$, $\text{ms}''$ and $\text{ms}_{\text{rev}}$ such that

5. $m' = m'' \uplus \text{ms}_{\text{rev}} \uplus (\text{ms}_r \uplus \text{ms}_{\text{frame}})$

6. $\text{ms}'' : n' - (j - i) W_3$

Now use $\text{ms}_r \uplus \text{ms}_{\text{rev}}$ as the “revoked” memory, $\text{ms}''$ 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, $n$, $\text{ms}$ if

- $\text{ms} : n W$

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

- base $\leq$ end

55
• \( \text{perm} \in \{ \text{rwx, rwx} \} \)

then

\[ \exists r, \text{base}', \text{end}'. [\text{base}, \text{end}] \subseteq [\text{base}', \text{end}'] \land W(r) \supseteq \iota^{\text{pwl}}_{\text{base}', \text{end}'} \]

\[ \blacksquare \]

**Proof of Lemma 9.**

Assume

1. \((n, ((\text{rwx}, g), b, e, a)) \in V(W)\)
2. \( ms :_{\text{n, p}} W \)

From Assumption 1, we get \( r_1, r_2, b_1, b_2, e_1 \) and \( e_2 \) such that

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

From Assumption 2, we get partition \( P \) s.t.

\( ms :_{\text{n, p}} W \)

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 \((n, ms_1) \in H_{b_1, e_1}^{\text{pwl}}, 1 \xi^{-1}(W) \) which means \( \text{dom}(ms_1) = [b_1, e_1] \).

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

Now assume for contradiction \( r_1 \neq r_2 \), then we have a contradiction with \( ms :_{\text{n, p}} W \) 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 \( W(r_1) \supseteq \iota_{b_1, e_1}^{\text{pwl}} \) which by Lemma 24

means \( W(r_1) = \iota_{b_1, e_1}^{\text{pwl}} \).

\( \blacksquare \)

**Lemma 10.** \( H_{\text{base, end}}^{\text{pwl}} s \) is monotone w.r.t \( \supseteq^{\text{pub}} \) for all \( s \in \text{State} \) and \( \text{base and end} \)

**Proof of Lemma 10.** Let \( \hat{W} \supseteq^{\text{pub}} W \) be given and let

\( (n, ms) \in H_{\text{base, end}}^{\text{pwl}} s \hat{W} \)

and show

\( (n, ms) \in H_{\text{base, end}}^{\text{pwl}} s \hat{W}' \)

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

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

now this follows from Lemma 77, \( \hat{W} \supseteq^{\text{pub}} W \), Theorem 1 and Assumption 14

\( \blacksquare \)
Lemma 11. \( pwl_{\text{base,end}} \) is a region for all base and end.

*Proof of Lemma 11.* Follows from Lemma 10.

Lemma 12. \( pwl_{\text{base,end}} \) is address-stratified.

*Proof.* Easy unfolding of definitions.

Lemma 13. \( H_{\text{base,end}} \) is monotone w.r.t \( \sqsupseteq_{\text{priv}} \) for all \( s \in \text{State} \) and base and end.

*Proof of Lemma 13.* Let \( \hat{W}' \sqsupseteq_{\text{priv}} \hat{W} \) be given and let
\[
(n, ms) \in H_{\text{base,end}} s \hat{W}
\]
and show
\[
(n, ms) \in H_{\text{base,end}} s \hat{W}'
\]
From 15 we get \( \text{dom}(ms) = [\text{base, end}] \). Now let \( a \in [\text{base, end}] \) be given and show

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

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

Lemma 14. \( pwl_{\text{base,end}} \) is a region for all base and end.

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

Lemma 15. \( pwl_{\text{base,end}} \) is address-stratified.

*Proof.* Easy unfolding of definitions.

Lemma 16. \( \hat{w}_{\text{base,end}} \) is a region for all base and end.

*Proof of Lemma 16.* Follows from Lemma 13.

Lemma 17. \( \hat{s}_{\text{base,end}}(v, ms) \) is a region for all \( v \in \{\text{perm, temp}\} \) and \( ms \).

*Proof of Lemma 17.* \( \hat{s} \) does not depend on \( \hat{W} \), so it is trivial to show the necessary non-expansive and monotonicity requirements.

Lemma 18. \( \hat{s}_{\text{base,end}}(ms) \) is monotone w.r.t \( \sqsupseteq_{\text{priv}} \) for all \( s \in \text{State} \) and \( ms \).

*Proof of Lemma 18.* Let \( \hat{W}' \sqsupseteq_{\text{priv}} \hat{W} \) be given and let
\[
(n, ms') \in H_{\text{base,end}} s \hat{W}
\]
and show
\[
(n, ms') \in H_{\text{base,end}} s \hat{W}'
\]
From 16 we get \( ms' = ms \). Now let \( a \in \text{dom}(ms) \) be given and show

1. \( ms(a) \) is non-local
2. \( (n - 1, ms(a)) \in V(\xi(\hat{W}')) \)
Lemma 19. \( \iota_{\text{sta},a}(v, ms) \) is a region for all \( v \in \{\text{perm, temp}\} \) and \( ms \).

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

Lemma 20.

\[
H_{\text{nwl base, end}}^\text{nwl} s \hat{W} \subseteq H_{\text{pwl base, end}}^\text{pwl} s \hat{W}
\]

Proof of Lemma 20. Trivial. Let \( (n, ms) \in H_{\text{nwl base, end}}^\text{nwl} s \hat{W} \) and show \( (n, ms) \in H_{\text{pwl base, end}}^\text{pwl} s \hat{W} \).

From the assumption, we get \( \text{dom}(ms) = [\text{base, end}] \). 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.

\[
\forall n \in \mathbb{N}. \forall \text{base, end} \in \text{Addr.}
\]

\[
\iota_{\text{nwl base, end}}^\text{nwl} \subseteq \iota_{\text{pwl base, end}}^\text{pwl}
\]

Proof of Lemma 21. Let \( n, \text{base, end} \) be given and show

\[
\iota_{\text{nwl base, end}}^\text{nwl} \subseteq \iota_{\text{pwl base, end}}^\text{pwl}
\]

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

\[
H_{\text{nwl base, end}}^\text{nwl} 1 \hat{W} \subseteq H_{\text{pwl base, end}}^\text{pwl} 1 \hat{W}
\]

which is true by Lemma 20.

Lemma 22.

\[
\forall n \in \mathbb{N}. \forall \text{base, end} \in \text{Addr.}
\]

\[
\iota_{\text{nwl base, end}}^\text{nwl,p} \subseteq \iota_{\text{pwl base, end}}^\text{pwl}
\]

Proof of Lemma 22. Follows from Lemma 20 (see proof of Lemma 21).
Lemma 23.
\[ \forall n \in \mathbb{N}. \forall \text{base, end} \in \text{Addr}. \forall v \in \{\text{perm, temp}\}. \]
\[ \text{dom}(ms) = [\text{base, end}] \Rightarrow \\
\text{\(^{sta,u}_{\text{base, end}}(v, ms) \overset{n}{\Rightarrow} _{\text{base, end}} \)} \]

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

Lemma 24.
\[ \forall n \in \mathbb{N}. \forall \text{base, end}, b \in \text{Addr}. \forall \iota \in \text{Region} \]
\[ \iota \overset{n}{\simeq} _{\text{base, end}} \land \text{base} \leq \text{end} \Rightarrow \iota \overset{n}{\simeq} _{\text{base, end}} \]

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

To this end let \( b \not\in [\text{base, end}] \) and define the worlds:
\[ \xi(W) = [0 \mapsto _{\text{base, end}}^{\text{pwl}}] \]
\[ [1 \mapsto _{b,b}^{\text{revoked}}] \]
\[ \xi(W') = [0 \mapsto _{\text{base, end}}^{\text{pwl}}] \]
\[ [1 \mapsto \text{revoked}] \]
For these two worlds, we have \( \xi(W') \sqsupseteq_{\text{priv}} \xi(W) \) and from mono. of \( \xi^{-1} \), we have \( W' \sqsupseteq_{\text{priv}} W \). Now define the following memory segment:
\[ ms = [\text{base} \mapsto ((\text{ro, local}), b, b, b), \text{base} + 1 \mapsto 0, \ldots, \text{end} \mapsto 0] \]
It is the case that
\[ (n, ms) \in H 1 W \]
but
\[ (n, ms) \not\in H 1 W' \]
as it is not the case that
\[ (n - 1, ((\text{ro, local}), b, b, b)) \in \forall(\xi(W')). \]
The only other option that remains is \( v = \text{temp} \).

5.4.3 Observation relation

Lemma 25 (Observation relation \((\mathcal{O})\) non-expansive).
\[ W \overset{\mathcal{O}}{\Rightarrow} W' \Rightarrow \mathcal{O}(W) \overset{\mathcal{O}}{=} \mathcal{O}(W') \]

Proof of Lemma 25.
5.4.4 Register-file relation

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

\[ W \models W' \Rightarrow \mathcal{R}(W) \models \mathcal{R}(W') \]

**Proof of Lemma 26.**

5.4.5 Expression relation

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

\[ W \models W' \Rightarrow \mathcal{E}(W) \models \mathcal{E}(W') \]

**Proof of Lemma 28.**

5.4.6 Permission based conditions

**Lemma 29.** If

\[ (n, (\text{base}, \text{end})) \in \text{readCondition}(\text{g})(\text{revokeTemp}(W)) \]

then

\[ (n, (\text{base}, \text{end})) \in \text{readCondition}(\text{g})(W) \]

**Proof of Lemma 29.**

\[ (n, (\text{base}, \text{end})) \in \text{readCondition}(\text{g})(\text{revokeTemp}(W)) \]

Gives \( r \in \text{localityReg}(g, \text{revokeTemp}(W)) \) such that

\[ \forall [\text{base}’, \text{end}’] \subseteq [\text{base}, \text{end}]. \text{revokeTemp}(W)(r) \subseteq \text{pwl}_{[\text{base}’, \text{end}’]} \]

Notice \( \text{revokeTemp}(W)(r) \) is a perm region, so \( \text{revokeTemp}(W)(r) = W(r) \). Using \( r \) as witness, the result is immediate.

**Lemma 30.** If

\[ (n, (\text{base}, \text{end})) \in \text{writeCondition}(i, g)(\text{revokeTemp}(W)) \]

then

\[ (n, (\text{base}, \text{end})) \in \text{writeCondition}(i, g)(W) \]
Proof of Lemma 30.

\((n, (\text{base}, \text{end})) \in \text{writeCondition}(\iota, g)(\text{revokeTemp}(W))\)

Gives \(r \in \text{localityReg}(g, \text{revokeTemp}(W))\) such that

\(\forall [\text{base}', \text{end}'] \subseteq [\text{base}, \text{end}]. \text{revokeTemp}(W)(r) \supseteq \iota_{[\text{base}', \text{end}']}\)

and

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

Notice \(\text{revokeTemp}(W)(r)\) is a perm region, so \(\text{revokeTemp}(W)(r) = W(r)\). Using \(r\) as witness, the result is immediate.

Lemma 31. If

- \((n, (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(g)(\text{revokeTemp}(W))\)

then

\((n, (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(g)(W)\)

Lemma 32. If

- \((n, (\text{a}, \text{base}, \text{end})) \in \text{executeCondition}(g)(\text{revokeTemp}(W))\)

then

\((n, (\text{a}, \text{base}, \text{end})) \in \text{executeCondition}(g)(W)\)

Lemma 33. If

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

then

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

Proof of Lemma 34. Follows from Lemma 22.

Lemma 34 (readCondition monotone w.r.t \(\supseteq^{\text{pub}}\)). If

- \(W' \supseteq^{\text{pub}} W\)

- \((n, (\text{base}, \text{end})) \in \text{readCondition}(g)(W)\)

then

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

Proof of Lemma 34.

61
Lemma 35 (\(\text{readCondition \ global \ monotonicity \ w.r.t} \ \preceq_{\text{priv}}\)). If
- \(W' \preceq_{\text{priv}} W\)
- \((n, (\text{base, end})) \in \text{readCondition} (\text{GLOBAL})(W)\)

then
\[(n, (\text{base, end})) \in \text{readCondition} (\text{GLOBAL})(W')\]

\[
\square
\]

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

Lemma 36 (\(\text{readCondition \ downwards-closed}\)). If
- \(n' \leq n\)
- \((n, (\text{base, end})) \in \text{readCondition} (\text{g})(W)\)

then
\[(n', (\text{base, end})) \in \text{readCondition} (\text{g})(W)\]

\[\square\]

\text{Proof of Lemma 36.}

Lemma 37 (\(\text{writeCondition \ monotone \ w.r.t} \ \preceq_{\text{pub}}\)). If
- \(W' \preceq_{\text{pub}} W\)
- \(\iota \in \{\iota_{\text{pwl}}, \iota_{\text{nwl}}, \iota_{(\text{nwl, p})}\}\)
- \((n, (\text{base, end})) \in \text{writeCondition} (\iota, \text{g})(W)\)

then
\[(n, (\text{base, end})) \in \text{writeCondition} (\iota, \text{g})(W')\]

\[\square\]

\text{Proof of Lemma 37.}

Lemma 38 (\(\text{writeCondition \ global \ monotonicity \ w.r.t} \ \preceq_{\text{priv}}\)). If
- \(W' \preceq_{\text{priv}} W\)
- \(\iota \in \{\iota_{\text{nwl}}, \iota_{(\text{nwl, p})}\}\)
- \((n, (\text{base, end})) \in \text{writeCondition} (\iota, \text{GLOBAL})(W)\)

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

\[\square\]

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

Lemma 39 (\(\text{writeCondition \ downwards-closed}\)). If
• \( n' \leq n \)
• \( t \in \{ \ell^{pwl}, \ell^{nwl}, \ell^{(nwl,p)} \} \)
• \( (n, (\text{base}, \text{end})) \in \text{writeCondition}(t, g)(W) \)

then
\[ (n', (\text{base}, \text{end})) \in \text{writeCondition}(t, g)(W) \]

\[ \blacksquare \]

Proof of Lemma 39.

Lemma 40 (\text{execCondition} monotone w.r.t \( \succeq^{\text{pub}} \)). If

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

then
\[ (n, (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(g)(W') \]

\[ \blacksquare \]

Proof of Lemma 40.

Lemma 41 (\text{execCondition} global monotonicity w.r.t \( \succeq^{\text{priv}} \)). If

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

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

\[ \blacksquare \]

Proof of Lemma 41. Assume \( W_2 \succeq^{\text{priv}} W_1, \text{perm} \in \{ \text{RX}, \text{RWX} \} \) and \( (n, (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(\text{GLOBAL})(W_1) \). Now let \( W_3 \succeq^{\text{priv}} W_2, a \in [\text{base}, \text{end}], \) and \( n' < n, \) and show
\[ (n, ((\text{perm}, \text{GLOBAL}), \text{base}, \text{end}, a)) \in E(W_3) \]
by transitivity we have \( W_3 \succeq^{\text{priv}} W_1, \) so the result follows from \( (n, (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(\text{GLOBAL})(W_1) \)

\[ \blacksquare \]

Lemma 42 (\text{execCondition} downwards-closed). If

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

\[(n', (\text{perm}, \text{base}, \text{end})) \in \text{executeCondition}(g)(W)\]

Proof of Lemma 42.

Lemma 43 (\text{enterCondition} monotone w.r.t \(\supseteq^\text{pub}\)). If

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

then

\[(n, (a, \text{base}, \text{end})) \in \text{enterCondition}(\iota, g)(W')\]

Proof of Lemma 43.

Lemma 44 (\text{enterCondition} global monotonicity w.r.t \(\supseteq^\text{priv}\)). If

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

then

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

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

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

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

Lemma 45 (\text{enterCondition} downwards-closed). If

- \(n' \leq n\)
- \((n, (a, \text{base}, \text{end})) \in \text{enterCondition}(g)(W)\)

then

\[(n', (a, \text{base}, \text{end})) \in \text{enterCondition}(g)(W)\]

Proof of Lemma 45.
5.4.7 LR Sanity lemmas

Lemma 46.

\[ \forall ms, n, W = W'. \quad ms :_n W \land W = W' \Rightarrow ms :_n W' \]

\[ \square \]

Proof of Lemma 46.

Lemma 47 (Heap satisfaction downwards closure).

\[ \forall ms, n', \leq n, W. \quad ms :_n W \Rightarrow ms :_{n'} W \]

\[ \square \]

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

This assumption gives us \( P : active(W) \rightarrow MemSegment \) such that
1. \( ms = \bigcup_{r \in active(W)} P(r) \)
2. \[ \forall r \in active(W). \quad \exists H, s. \quad W(r) = (\omega, s, \omega, H) \land \quad (n', P(r)) \in H(s)(\xi^{-1}(W)) \]

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. \( W(r) = (\omega, s, \omega, H) \)
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
- \( ms :_n W \)
- \( (n, ((perm, g), base, end, a)) \in stdvr(W) \)
- \( base \leq end \)
- \( perm \in \{RWLX, RWL\} \)

then \[ g = \text{LOCAL} \]

\[ \square \]

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 \( (n, \text{updatePcPerm}(w)) \in \mathcal{E}(W) \).

**Proof.**

1. Case \( w = ((\text{perm}, g), \text{base, end, a}) \) and \( \text{base} \leq a \leq \text{end} \) and \( \text{perm} \in \{\text{RX, RWX, RWLX}\} \):
   
   1.1. \( (n + 1, (\text{perm, base, end})) \in \text{executeCondition}(g)(W) \).
       By: definition of \( \mathcal{V}(W) \) using the fact that \( \text{perm} \in \{\text{RX, RWX, RWLX}\} \).
   
   1.2. \( (n, ((\text{perm, g, base, end, a})) \in \mathcal{E}(W) \): By definition of \text{executeCondition} using the fact that \( \text{base} \leq a \leq \text{end} \).

2. Case \( w = ((\text{perm, g, base, end, a}) \) and \( \text{base} \leq a \leq \text{end} \) and \( \text{perm} = \text{E} \):
   
   2.1. \( (n + 1, (\text{base, end, a})) \in \text{enterCondition}(g)(W) \).
       By: definition of \( \mathcal{V}(W) \) using the fact that \( \text{perm} = \text{E} \).
   
   2.2. \( (n, ((\text{RX, g, base, end, a})) \in \mathcal{E}(W) \): By definition of \text{enterCondition} using the fact that \( \text{base} \leq a \leq \text{end} \).
   
   2.3. \( \text{updatePcPerm}(w) = ((\text{RX, g, base, end, a})) \):
       By definition of \text{updatePcPerm}(\cdot)

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

**Lemma 50** (Malloc is safe to pass to adversary). If \( W(r) \equiv_{\text{priv}, \text{malloc}, 0} W' \), then \( (n, \text{c_malloc}) \in \mathcal{V}(W) \) for all \( n \).

**Proof.**

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

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

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

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

5. Assume: \( i < n'', (\text{reg}[pc \mapsto ((\text{RX, GLOBAL}), \text{base, end, a})], ms \equiv ms_{\ell}) \rightarrow_i \) \( (\text{halted}, \text{mem}'') \)
   
   Suffices: \( \exists W'': \equiv_{\text{priv}} W', ms_{\ell}, ms'. \text{mem}' = ms' \equiv ms_{\ell} \equiv ms_{\ell} \) and \( ms' : n'' W'' \)
   
   By: definition of \( \mathcal{O}(W') \)

6. \( W'(r) \equiv_{\text{priv}, \text{malloc}, 0} W' \) and \( W(r) \equiv_{\text{priv}} W' \) using transitivity of \( \equiv_{\text{priv}} \).

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

66
8. Define \( ms_{frame} = \left( \bigcup_{r' \in \text{active}(W'), r' \neq r} P(r') \right) \uplus ms_f \). Then \( ms \uplus ms_f = P(r) \uplus ms_{frame} \) and \( P(r) :_{\omega} W'(r).H \left( W'(r).s \right) \left( \xi^{-1}(W') \right) \). Easy from the previous point.

9. \( P(r) :_{\omega} W'(r).H \left( W'(r).s \right) \left( \xi^{-1}([r \mapsto W'(r)]), \text{i.e.} \right. \) \( P(r) :_{\omega} [r \mapsto W'(r)] \).

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

By: the malloc specification (Specification 1).

10.3. Define \( W'' = W'[r \mapsto \iota'_{\text{malloc}}][i \mapsto \iota'_{\text{malloc}}'] \) for \( i \not\in \text{dom}(W') \). We have that \( W'' \supseteq \text{pub} [r \mapsto \iota'_{\text{malloc}}] \) and \( W'' \supseteq \text{pub} W' \).

By: definition of \( \supseteq \text{pub} \), using the fact that \( \iota'_{\text{malloc}} \supseteq \text{pub} W(r) \).

10.4. \( (n'', (\text{base}', \text{end}')) \in \text{readCondition}(\text{GLOBAL})(W'') \) for all \( n'' \):

By: definition of \( \text{readCondition} \), using the region \( W''(i) \) and Lemma 21.

10.5. \( (n'', (\text{base}', \text{end}')) \in \text{writeCondition}(\text{malloc}, \text{GLOBAL})(W'') \) for all \( n'' \):

By: definition of \( \text{writeCondition} \), using the region \( W''(i) \).

10.6. \( (n'', (\text{base}', \text{end}')) \in \text{executeCondition}(\text{malloc}, \text{GLOBAL})(W'') \) for all \( n'', p \in \{\text{RX, RX}\} \):

By: the definition of \( \text{executeCondition} \), the FTLR (Theorem 2) using Lemmas 38, 35 and the previous two points.

10.7. \( (n'', (\text{RX, GLOBAL}), b', e', b')) \in V(W'') \):

By: definition of \( V(W'') \) and the above three points.

10.8. \( (n'' - j, \Phi.\text{reg}[r_1 \mapsto \{(\text{RX, GLOBAL}), b', e', b'\}] \in \mathbb{R}(W'') \):

By Lemma 76, Lemma 27 using the fact that \( W'' \supseteq \text{pub} W' \) and \( (n'', \Phi.\text{reg}) \in V(W'') \), together with the previous point.

10.9. \( (n'', ms_{\text{alloc}}) \in \text{malloc}^{\text{pub} \cdot H} \text{ alloc}, s \cdot W'' \) for any \( n'' \):

By definition of \( \text{malloc} \), \( H \text{ alloc} \) and \( V(\cdot) \) and the facts that \( \text{dom}(ms_{\text{alloc}}) = [b', e'] \) and \( \forall a \in [b', e'], ms_{\text{alloc}}(a) = 0 \).

10.10. Define \( ms' = \left( \bigcup_{r' \in \text{active}(W'), r' \neq r} P(r') \right) \uplus ms'_{\text{footprint}} \uplus ms_{\text{alloc}} \) Then \( \Phi'.\text{mem} = ms' \uplus ms_f \) and \( ms' :_{\omega}^{-j} W'' \):

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

10.11. \( (n'' - j + 1, w_{\text{ret}}) \in V(W'') \):

By Lemma 74, the fact that \( W'' \supseteq \text{pub} W' \), Lemma 75, and the fact that \( (n'', w_{\text{ret}}) \in V(W') \), which follows from \( w_{\text{ret}} = \Phi.\text{reg}(r_1) \) and \( (n'', \Phi) \in V(W') \).

10.12. \( (n'' - j, \text{updatePcPerm}(w_{\text{ret}})) \in E(W'') \):

By Lemma 49 from the previous point.
10.13. \((n'' - j, (\Phi \cdot \text{reg}[r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')][\text{pc} \mapsto \text{updatePcPerm}(w_n, s)], ms')) \in \mathcal{O}(W'')\):
By: definition of \(\mathcal{E}(W'')\), using the previous point and the facts that \((n'' - j, (\Phi \cdot \text{reg}[r_1 \mapsto ((\text{RWX}, \text{GLOBAL}), b', e', b')][\text{pc} \mapsto \text{updatePcPerm}(w_n, s)], ms')) \in \mathcal{O}(W'')\), \(\mathcal{R}(W'')\), \(ms' : n'' - j \ W''\).

10.14. \(i > j\) and \(\Phi' \rightarrow r_{i-j} (\text{halted, mem'})\).
By combining \((\text{reg}[\text{pc} \mapsto ((\text{RWX}, \text{GLOBAL}), \text{base}, \text{end}, a)], ms \cup ms_f) \rightarrow r_i (\text{halted, mem'})\)
with \((\text{reg}[\text{pc} \mapsto ((\text{RWX}, \text{GLOBAL}), \text{base}, \text{end}, a)], ms \cup ms_f) \rightarrow r_j \Phi'\) using Lemma .

10.15. \(\exists W'' \supseteq \text{pub} W'', ms_r, ms''. \text{mem}' = ms'' \cup ms_r \cup ms_f\) and \(ms'' : n-1 W''\).
By: definition of \(\mathcal{O}(W'')\) from the two previous points.

10.16. \(W'' \supseteq \text{pub} W'\).
By Lemma 72 using the previous point and the fact that \(W'' \supseteq \text{pub} W'\).

11. Case: \(\text{reg}(r_1) \notin \mathbb{Z} \lor \text{reg}(r_1) < 0\)

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

11.2. Contradiction with \((\text{reg}[\text{pc} \mapsto ((\text{RWX}, \text{GLOBAL}), \text{base}, \text{end}, a)], ms \cup ms_f) \rightarrow_i (\text{halted, mem'})\)
\(\square\)

5.4.9 Fundamental theorem of logical relations

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

- \(\Phi \cdot \text{mem} \in W\)
- \(c = ((\text{perm}, g), \text{base}, \text{end}, a)\)
- \((n, c) \in \mathcal{Y}(W)\)
- \(\text{readAllowed}(\text{perm})\)
- \(\text{withinBounds}(c)\)

then \((n - 1, \Phi \cdot \text{mem}(a)) \in \mathcal{Y}(W)\)

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

2. \(\exists r \in \text{localityReg}(g, W), [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}]\). \(W(r) \nsubseteq i^\text{pub}_{\text{base}', \text{end}'}\). By definition of \(\text{readCondition}(g)(W)\).

3. \(\exists P : \text{active}(W) \rightarrow \text{MemSegment}. \Phi \cdot \text{mem} \subseteq P W\). By definition of \(\Phi \cdot \text{mem} \subseteq P W\).

4. \(\Phi \cdot \text{mem} = \bigcup_{r \in \text{active}(W)} P(r)\) and \(\forall r \in \text{active}(W), \exists H, s, W(r) = (\omega, s, \omega, H)\) and \((n, P(r)) \in H(s)(\xi^{-1}(W))\). By definition of \(\Phi \cdot \text{mem} \subseteq P W\).

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

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

7. \((n, P(r)) \in H^\text{pub}_{\text{base}', \text{end}'}(s)(\xi^{-1}(W))\). Follows by combining \((n, P(r)) \in H(s)(\xi^{-1}(W))\) with \(W(r) \nsubseteq i^\text{pub}_{\text{base}', \text{end}'}\) from Step 2.

68
8. \( \text{dom}(P(r)) = [\text{base}', \text{end}'] \) and for all \( a' \in [\text{base}', \text{end}'] \). \((n - 1, P(r)(a')) \in \mathcal{V}(\xi^{-1}(W))\).

By definition of \( H_{\text{base}', \text{end}'}^{\text{pwl}} \).

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

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

\[ \square \]

**Lemma 52** (Conditions for \text{lea} instruction are sufficient). \( \text{If} \)

\begin{itemize}
  \item \((n, ((\text{perm}, g), \text{base}, \text{end}, a)) \in \mathcal{V}(W)\)
  \item \(\text{perm} \notin E\)
\end{itemize}

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

\[ \Box \]

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

\[ \Box \]

**Lemma 53** (pwl writecond implies \text{nw}l). \( \text{If} \ ((n, (\text{base}, \text{end})) \in \text{writeCondition}(\iota_{\text{pwl}}, g)(W) \) then \((n, (\text{base}, \text{end})) \in \text{writeCondition}(\iota_{\text{nw}l}, g)(W)\).

\[ \Box \]

**Proof.**

1. \( \exists r \in \text{localityReg}(g, W). \exists [\text{base}', \text{end}'] \supseteq [\text{base}, \text{end}] \). \( W(r) \nleftarrow \iota_{\text{pwl}}^{\text{nw}l} \) \( \iota_{\text{base}', \text{end}'} \) and \( W(r) \) is address-stratified: by definition of \text{writeCondition}.

2. Sufﬁces: \( W(r) \nleftarrow \iota_{\text{base}', \text{end}'}^{\text{pwl}} \). By definition of \text{writeCondition}

3. \( W(r) \nleftarrow \iota_{\text{base}', \text{end}'}^{\text{pwl}} \Rightarrow \iota_{\text{base}', \text{end}'}^{\text{nw}l} \) follows by Lemma 21

\[ \Box \]

**Lemma 54** (execCond implies \text{entry}Cond). \( \text{If} \ ((n, (\text{rx}, \text{base}, \text{end})) \in \text{execCondition}(g)(W) \) then \((n, (\text{base}, \text{end}, a)) \in \text{executeCondition}(g)(W)\).

\[ \Box \]

**Proof.**

1. Assume: \( n' < n, W' \supseteq W \) where \( g = \text{LOCAL} \Rightarrow \supseteq \supseteq \iota_{\text{pub}} \) and \( g = \text{GLOBAL} \Rightarrow \supseteq \supseteq \).

\[ \supseteq \iota_{\text{pwe}} \]

Sufﬁces: \( (n', ((\text{rx}, g), \text{base}, \text{end}, a)) \in \mathcal{E}(W') \)

2. Case \( a \in [\text{base}, \text{end}] \): Follows from the deﬁnition of \text{executeCondition}.

3. Case \( a \notin [\text{base}, \text{end}] \): Follows by Lemma 7

\[ \Box \]

**Lemma 55** (Conditions for \text{restrict} instruction are sufﬁcient). \( \text{If} \)

\begin{itemize}
  \item \((n, ((\text{perm}, g), \text{base}, \text{end}, a)) \in \mathcal{V}(W)\)
  \item \(\text{perm}' \subseteq (\text{perm}, g)\)
\end{itemize}

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

\[ \Box \]

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

1. If \((n, (\text{base}, \text{end})) \in \text{writeCondition}(\iota_{\text{pwl}}, g)(W) \) then \((n, (\text{base}, \text{end})) \in \text{writeCondition}(\iota_{\text{nw}l}, g)(W)\):

holds by lemma 54

69
2. If \((n, (\text{rx, base, end})) \in \text{executeCondition}(g)(W)\) then \((n, (\text{base, end, a})) \in \text{enterCondition}(g)(W)\). □

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

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

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

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

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

- \(\text{ms} = \text{ms}' \uplus \text{ms}_f\)
- \(\text{ms}' :_n W\)
- \(((\text{perm, g}), \text{base, end, a}) = c\)
- \((n, c) \in \mathcal{V}(W)\)
- \(\text{writeAllowed(perm)}\)
- \(\text{withinBounds}(c)\)
- \((n, w) \in \mathcal{V}(W)\)
- if \(w = ((\_, \text{local}), \_, \_, \_),\) then \(\text{perm} \in \{\text{rwlx, rwl}\}\)

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

**Proof.**

1. \((n, (\text{base, end})) \in \text{writeCondition}(\iota, g)(W)\) where \(\iota = \iota^{\text{pul}}\) or \(\iota = \iota^{\text{nul}}\) and (if \(w = ((\_, \text{local}), \_, \_, \_),\) then \(\iota = \iota^{\text{gul}}\)).

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

2. \(\exists r \in \text{localityReg}(g, W). \exists [\text{base}', \text{end}] \supseteq [\text{base, end}]\). \(W(r)^{n-1}_{\text{base}', \text{end}'}\) and \(W(r)\) is address-stratified. By definition of \(\text{writeCondition}\).

3. \(\exists P : \text{active}(W) \rightarrow \text{MemSegment}. \text{ms}' :_{n, P} W\). By definition of \(\text{ms}' :_{n} W\).

4. \(\text{ms}' = \bigcup_{r \in \text{active}(W)} P(r)\) and \(\forall r \in \text{active}(W). \exists H, s. W(r) = ((\_, s, \_, \_, H)\) and \((n, P(r)) \in H(s)(\xi^{-1}(W))\). By definition of \(\text{ms}' :_{n, P} W\).

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 \(\text{writeCondition}\).
6. \((n, w) \in I.H (i.s) (\xi^{-1}(W))\) by definition of \(i_{\text{pwl}}, i_{\text{nwl}}\) and the fact that \((i = (., \text{LOCAL}), \ldots., \ldots.)\), then \(i = i_{\text{pwl}}\).

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

8. \(\text{dom}(P(r)) = \text{dom}(ms'_n) = [\text{base}', \text{end}'] \ni a\) 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. \(ms'[a \mapsto w] = \bigcup_{r \in \text{active}(W)} P'(r)\) and \(ms'[a \mapsto w] : n, P W\). By definition of \(ms' : n, P W\) and the previous two points.

\[\boxempty\]

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

- \(\text{perm} = \text{RX} \land (n, (\text{base}, \text{end})) \in \text{readCondition}(g)(W)\)
- \(\text{perm} = \text{RX} \land (n, (\text{base}, \text{end})) \in \text{readCondition}(g)(W) \land (n, (\text{base}, \text{end})) \in \text{writeCondition}(i_{\text{nwl}}, g)(W)\)
- \(\text{perm} = \text{RLX} \land (n, (\text{base}, \text{end})) \in \text{readCondition}(g)(W) \land (n, (\text{base}, \text{end})) \in \text{writeCondition}(i_{\text{pwl}}, g)(W)\),

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

\[\blacksquare\]

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

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

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

4. \(i \neq 0\), since \((\text{reg}[\text{pc} \mapsto ((\text{perm}, g), \text{base}, \text{end}, a)], ms \uplus ms_f) \neq (\text{halted}, \Phi')\) for any \(\Phi'\).
   Therefore, assume w.l.o.g. that \(i = 1 + i'\),
   \[\Phi \rightarrow \text{conf'} \rightarrow_{i'} (\text{halted}, \Phi')\]
5. $n \geq n' > 0$, since otherwise $i = 0$ (because $i \leq n' \leq n$) and this is impossible by the previous point.

6. $(n', \Phi.\text{reg}(pc)) \in \mathcal{V}(W)$. Proof:
   6.1. Assume: $\text{perm}' \in \{RX, RWX, RWLX\}$ with $\text{perm}' \subseteq \text{perm}$
   Suffices: $(n', (\text{perm}', \text{base}, \text{end})) \in \text{executeCondition}(g)(W)$
   By: the definition of $\mathcal{V}()$ using the assumptions
   6.2. Assume: $n'' < n', W' \sqsupseteq W, a' \in [\text{base}, \text{end}], g = \text{LOCAL} \Rightarrow \mathbb{Z} = \mathbb{Z}^{\text{pub}}, g = \text{GLOBAL} \Rightarrow \mathbb{Z} = \mathbb{Z}^{\text{priv}}$
   Suffices: $(n'', ((\text{perm}, g), \text{base}, \text{end}, a')) \in \mathcal{E}(W').$ By: definition of $\text{executeCondition}(g)(W)$
   6.3. By induction, using the assumptions and Lemmas 30 and 39.

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

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

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

10. Case $\text{conf}' = (\text{halted, } \Phi)$:
    10.1. Then $i' = 0$ and $\Phi' = \Phi$
    Follows from $(\text{halted, } \Phi) \rightarrow_{\nu} (\text{halted, } \Phi')$
    10.2. For $W' = W, ms_r = \emptyset$ and $ms' = ms$, we have that $\Phi'.\text{mem} = ms' \cup ms_r \cup ms_f$ and $ms' : \rightarrow_{\nu} W'$ (using Lemma 17).

11. Case $\text{conf}' = \Phi'[\text{reg}.pc \rightarrow \text{newPc}]$, and additionally, one of the following holds:
    - $\Phi''.\text{mem} = \Phi.\text{mem}$
    - $\Phi''.\text{mem} = \Phi.\text{mem}[a' \mapsto w]$, with $\Phi.\text{reg}(r_1) = ((\text{perm}', g'), \text{base}', \text{end}', a') = c$ and $\text{writeAllowed}(\text{perm}')$ and $\text{withinBounds}(c)$ and $w = \Phi.\text{reg}(r_2)$ and if $w = ((\text{LOCAL}), \ldots, \ldots)$, then $\text{perm}' \in \{\text{RWLX, RWL}\}$
    and also one of the following holds:
    - $\text{newPc} = \text{updatePcPerm}(\Phi.\text{reg}(lv))$
    - $\text{newPc} = ((\text{perm}', g'), \text{base}', \text{end}', a' + 1)$ and $\Phi''.\text{reg}(pc) = ((\text{perm}', g'), \text{base}', \text{end}', a')$
    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 $\Phi.\text{reg}(r_2) = ((\text{perm}', g'), \text{base}', \text{end}', a') = c$ and $\text{readAllowed}(\text{perm}')$
    and $\text{withinBounds}(c)$ and $w = \Phi.\text{mem}(a')$
    - $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r_1) = ((\text{perm}', g'), \text{base}', \text{end}', a')$ and $\text{perm}' \neq E$ and $c = ((\text{perm}', g'), \text{base}', \text{end}', a' + z)$ for some $z \in \mathbb{Z}$

72
• $\Phi'^\prime\prime.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((\text{perm}'', g'), \text{base}', \text{end}', a')$ and $(\text{perm}''', g''') \subseteq (\text{perm}'', g')$ and $c = ((\text{perm}'''', g''''), \text{base}'', \text{end}'', a'')$

• $\Phi'^\prime\prime.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((\text{perm}', g'), \text{base}', \text{end}', a')$ and $\text{base}' \leq \text{base}'''$ and $\text{end}'' \leq \text{end}'$ and $c = ((\text{perm}', g'), \text{base}''', \text{end}''', a')$ and $\text{perm}' \neq \emptyset$

In this case, we have:

11.1. $\Phi'^\prime\prime.\text{mem} = ms''' \uplus ms_f$ and $ms'' : n_r - 1 \ W$.

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

11.1.2. Case $\Phi'^\prime\prime.\text{mem} = \Phi.\text{mem}[a' \mapsto w]$, with $\Phi.\text{reg}(r_1) = ((\text{perm}', g'), \text{base}', \text{end}', a') = c$ and $\text{writeAllowed}(\text{perm}')$ and $\text{withinBounds}(c)$ and $w = \Phi.\text{reg}(r_2)$ and if $w = (\text{.., local}, \ldots)$, then $\text{perm}' \in \{\text{rwlL, rwL}\}$.

The facts that $\Phi'^\prime\prime.\text{mem} = ms''' \uplus ms_f$ and $ms'' : n_r - 1 \ W$ follow by Lemmas 57 and 47 using the fact that $ms : W$ 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 $r \in \text{RegisterName}$, $(n' - 1, \Phi'^\prime\prime.\text{reg}(r)) \in \mathcal{V}(W)$.

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

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

11.2.3. $\Phi'^\prime\prime.\text{reg}(r) = w$ and $\Phi.\text{reg}(r_2) = ((\text{perm}', g'), \text{base}', \text{end}', a') = c$ and $\text{readAllowed}(\text{perm}')$ and $\text{withinBounds}(c)$ and $w = \Phi.\text{mem}(a')$:

$(n' - 1, \Phi'^\prime\prime.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 51 using the fact that $\Phi.\text{mem} : n_r - 1 \ W$ and $(n', \Phi.\text{reg}(r_2)) \in \mathcal{V}(W)$ which we have from step 7.

11.2.4. $\Phi'^\prime\prime.\text{reg}(r) = c$ and $\Phi.\text{reg}(r_1) = ((\text{perm}', g'), \text{base}', \text{end}', a')$ and $\text{perm}' \neq \emptyset$ and $c = ((\text{perm}', g'), \text{base}', \text{end}', a' + z)$ for some $z \in \mathbb{Z}$:

$(n' - 1, \Phi'^\prime\prime.\text{reg}(r)) \in \mathcal{V}(W)$ 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. $\Phi'^\prime\prime.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((\text{perm}', g'), \text{base}', \text{end}', a')$ and $(\text{perm}''', g''') \subseteq (\text{perm}', g')$ and $c = ((\text{perm}'''', g''''), \text{base}''', \text{end}''', a'')$:

$(n' - 1, \Phi'^\prime\prime.\text{reg}(r)) \in \mathcal{V}(W)$ 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. $\Phi'^\prime\prime.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((\text{perm}', g'), \text{base}', \text{end}', a')$ and $\text{base}' \leq \text{base}'''$ and $\text{end}'' \leq \text{end}'$ and $c = ((\text{perm}', g'), \text{base}''', \text{end}''', a')$ and $\text{perm}' \neq \emptyset$:

$(n' - 1, \Phi'^\prime\prime.\text{reg}(r)) \in \mathcal{V}(W)$ 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. $(n' - 1, \Phi'^\prime\prime.\text{reg}) \in \mathcal{R}(W)$: Follows from the previous point by definition of $\mathcal{R}(W)$.

11.4. $(n' - 1, \text{newPc}) \in \mathcal{E}(W)$:

11.4.1. Case $\text{newPc} = \text{updatePcPerm}(\Phi.\text{reg}(lv))$: We distinguish the following cases:

11.4.1.1. Case $\Phi.\text{reg}(lv) = ((\text{e}, g'), \text{base}', \text{end}', a')$:

11.4.1.1.1. $(n', \Phi.\text{reg}(lv)) \in \mathcal{V}(W)$. Follows from Step 7.

11.4.1.1.2. $(n', (\text{base}', \text{end}', \text{addr})) \in \text{enterCondition}(g')(W)$. By definition of $\mathcal{V}(W)$ from the previous point.

11.4.1.1.3. $(n' - 1, ((\text{nx}, g'), \text{base}', \text{end}', a')) \in \mathcal{E}(W)$: By definition of $\text{enterCondition}(\cdot)$ and taking $n' = n_r - 1$ and $W' = W$.
11.4.1.4. \( \text{updatePcPerm}(\Phi\cdot\text{reg}(lv)) = ((rx, g'), base', end', a') \): by definition of \( \text{updatePcPerm}(\cdot) \).

11.4.1.2. Case \( \Phi\cdot\text{reg}(lv) = ((perm', g'), base', end', a') \) with \( perm' \in \{rx, rwx, rwlx\} \)
and \( \text{withinBounds}(\Phi\cdot\text{reg}(lv)) \):

11.4.1.2.1. \( (n', \Phi\cdot\text{reg}(lv)) \in \mathcal{V}(W) \), follows from Step 7.

11.4.1.2.2. \( (n', (perm', base', end', a')) \in \text{executeCondition}(g')(W) \). By definition of \( \mathcal{V}(W) \) from the previous point.

11.4.1.3. Case not \( (\Phi\cdot\text{reg}(lv) = ((perm', g'), base', end', a')) \) with \( perm' \in \{rx, rwx, rwlx\} \) and \( \text{withinBounds}(\Phi\cdot\text{reg}(lv)) \):

11.4.1.3.1. \( \text{updatePcPerm}(\Phi\cdot\text{reg}(lv)) = \Phi\cdot\text{reg}(lv) \): by definition of \( \text{updatePcPerm}(\cdot) \).

11.4.1.3.2. \( (\text{reg}[pc \mapsto \Phi\cdot\text{reg}(lv)], ms) \rightarrow \text{failed} \) 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 \( \Phi'\cdot\text{reg}(pc) = ((perm', g'), base', end', a') \):

11.4.2.1. Case \( perm' \in \{rx, rwx, rwlx\} \) and \( base' \leq a' + 1 \leq end' \):

11.4.2.1.1. \( (n' - 1, \Phi''\cdot\text{reg}(pc)) \in \mathcal{V}(W) \): by Step 11.2.

11.4.2.1.2. \( (n' - 1, ((perm', g'), base', end', a'+1)) \in \mathcal{V}(W) \): by Lemma 52 from the previous point.

11.4.2.1.3. One of the following holds:

- \( perm' = rx \wedge (n' - 1, (base', end')) \in \text{readCondition}(g)(W) \)

- \( perm' = rwx \wedge (n' - 1, (base', end')) \in \text{readCondition}(g)(W) \wedge (n' - 1, (base', end')) \in \text{writeCondition}(rwx, g)(W) \)

- \( perm' = rwlx \wedge (n' - 1, (base', end')) \in \text{readCondition}(g)(W) \wedge (n' - 1, (base', end')) \in \text{writeCondition}(rwlx, g)(W) \)

This follows from the previous point by definition of \( \mathcal{V}(W) \)

11.4.2.1.4. \( (n' - 1, ((perm', g'), base', end', a'+1)) \in \mathcal{E}(W) \): By the induction hypothesis of this lemma using the previous point.

11.4.2.2. Case not \( (perm'' \in \{rx, rwx, rwlx\} \) and \( base' \leq a' + 1 \leq end' \): The result follows by Lemma 7.

11.5. \( (n' - 1, (\Phi''\cdot\text{reg}[pc \mapsto newPc], ms')) \in \mathcal{O}(W) \): by definition of \( \mathcal{E}(W) \) using the above three points.

11.6. \( \exists W' \cup_{prim} W, ms_r, ms'. \Phi'\cdot\text{mem} = ms' \oplus ms_r \oplus ms_f \) and \( ms' \cup_{\text{halted}} W' \)
By: definition of \( \mathcal{O}(W) \) using the previous step and the evaluation \( con{f'} \rightarrow_{i'} (\text{halted}, \Phi') \) from Step 4.
5.4.10 Scall macro-instruction correctness

Definition 4. We say that \((\text{reg}, \text{ms})\) is looking at \([i_0, \cdots, i_n]\) followed by \(c_{\text{next}}\) iff

- \(\text{reg}(\text{pc}) = ((p, g), b, e, a)\)
- \(p = \text{RWX}, p = \text{RX}, \) or \(p = \text{RWLX}\)
- \(a + n \leq e, b \leq a \leq e\)
- \(\text{ms}(a + 0, \cdots, a + n) = [i_0, \cdots, i_n]\)
- \(c_{\text{next}} = ((p, g), b, e, a + n + 1)\)

Definition 5. We say that \(\text{reg}\) points to stack with \(\text{ms}_{\text{stk}}\) used and \(\text{ms}_{\text{unused}}\) unused iff

- \(\text{reg}(\text{r}_{\text{stk}}) = ((\text{RWLX, LOCAL}), b_{\text{stk}}, e_{\text{stk}}, a_{\text{stk}})\)
- \(\text{dom}(\text{ms}_{\text{unused}}) = [a_{\text{stk}} + 1, \cdots, e_{\text{stk}}]\)
- \(\text{dom}(\text{ms}_{\text{stk}}) = [b_{\text{stk}}, \cdots, a_{\text{stk}}]\)
- \(b_{\text{stk}} - 1 \leq a_{\text{stk}}\)

Lemma 58 (scall works). If

- \(\text{ms} := n \text{ revokeTemp}(W)\)
- \(\text{dom}(\text{ms}_f) \cap (\text{dom}(\text{ms}_{\text{stk}} \uplus \text{ms}_{\text{unused}} \uplus \text{ms})) = \emptyset\)
- \((\text{reg}, \text{ms})\) is looking at \text{scall} \(r(\text{r}_{\text{arg}}, \text{r}_{\text{priv}})\) followed by \(c_{\text{next}}\)
- \(\text{reg}\) points to stack with \(\text{ms}_{\text{stk}}\) used and \(\text{ms}_{\text{unused}}\) unused

\textbf{Hyp-Callee If}

- \(\text{dom}(\text{ms}_{\text{unused}}) = \text{dom}(\text{ms}_{\text{act}} \uplus \text{ms}'_{\text{unused}})\),
- \(W' = \text{revokeTemp}(W)[t^{\text{sta}}(\text{temp}, \text{ms}_{\text{stk}} \uplus \text{ms}_{\text{act}} \uplus \text{ms}_f), t^{\text{pw}}(\text{dom}(\text{ms}'_{\text{unused}}))]\),
- \(\text{ms}'' := n - 1 W'\)
- \(\text{reg}'\) points to stack with \(\emptyset\) used and \(\text{ms}'_{\text{unused}}\) unused
- \(\text{reg}' = \text{reg}_0[\text{pc} \mapsto \text{updatePcPerm}(\text{reg}(r)), \text{r}_{\text{arg}} \mapsto \text{reg}(\text{r}_{\text{arg}}), r_0 \mapsto c_{\text{ret}}, \text{r}_{\text{stk}} \mapsto c_{\text{stk}}, r \mapsto \text{reg}(r)]\)
- \((n - 1, c_{\text{ret}}) \in \mathcal{V}(W')\)
- \((n - 1, c_{\text{stk}}) \in \mathcal{V}(W')\)

then we have that \((n - 1, (\text{reg}', \text{ms}'')) \in \mathcal{O}(W')\)

\textbf{Hyp-Cont If}

- \(n' \leq n - 2\)
- \(W'' \equiv_{\text{pub}} \text{revokeTemp}(W)\)
– \( ms'' : n' \text{ revokeTemp}(W'') \)
– for all \( r \), we have that:

\[
\begin{aligned}
\text{reg}'(r) &= c_{\text{next}} & \text{if } r = \text{pc} \\
\text{reg}'(r) &= \text{reg}(r) & \text{if } r \in \overline{r_{\text{prev}}} \\
\text{reg}'(r) &\in \mathcal{V}(\text{revokeTemp}(W'')) & \text{if reg}'(r) \text{ is a global capability and } r \notin \{\text{pc}, \overline{r_{\text{prev}}}, r_{\text{stk}}\}
\end{aligned}
\]

– \( \text{reg}' \) points to stack with \( ms_{\text{stk}} \) used and \( ms''_{\text{unused}} \) unused for some \( ms''_{\text{unused}} \)

then we have that \( (n', (\text{reg}', ms'' \oplus ms_f \oplus ms_{\text{stk}} \oplus ms''_{\text{unused}})) \in \mathcal{O}(W'') \)

Then

• \( (n, (\text{reg}, ms \oplus ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{unused}})) \in \mathcal{O}(W) \)

\[\square\]

Proof. Assume \( n \) is sufficiently large to execute all the steps up to and including the jump of \( \text{scall} \ r(\overline{\text{arg}}, \overline{r_{\text{priv}}}) \). If this is not the case, then in any given memory frame the execution will not halt successfully fast enough.

Further assume

1. \( ms : n \text{ revokeTemp}(W) \)
2. \( \text{dom}(ms_f) \cap (\text{dom}(ms_{\text{stk}} \oplus ms_{\text{unused}} \oplus ms)) = \emptyset \)
3. \( (\text{reg}, ms) \) is looking at \( \text{scall} \ r(\overline{\text{arg}}, \overline{r_{\text{priv}}}) \) followed by \( c_{\text{next}} \)
4. \( \text{reg} \) points to stack with \( ms_{\text{stk}} \) used and \( ms_{\text{unused}} \) unused
5. Hyp-Callee
6. Hyp-Cont

Now we wish to apply Lemma 8. To this end let \( ms_{\text{frame}} \) be given. Executing the \( \text{scall} \) gives us

\( (\text{reg}, ms \oplus ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{unused}} \oplus ms_{\text{frame}}) \rightarrow_i (\text{reg}_1, ms \oplus ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms''_{\text{unused}} \oplus ms_{\text{frame}}) \)

where

7. \( i \leq n \)
8. \( ms_{\text{act}} \) contains activation record, \( \text{reg}(\overline{r_{\text{prev}}}) \), the code return capability, and the full stack capability (\( \text{reg}(r_{\text{stk}}) \) with the pointer adjusted).
9. \( \forall a \in \text{dom}(ms'_{\text{unused}}). ms'_{\text{unused}}(a) = 0 \)
10. \( \text{dom}(ms_{\text{unused}}) = \text{dom}(ms_{\text{act}} \oplus ms'_{\text{unused}}) \)
11. \( \text{reg}_1(r_0) = c_{\text{ret}} = ((\text{E, LOCAL}), \ldots, \ldots) \) where the range of authority is the same as \( \text{reg}(r_{\text{stk}}) \) and it points to the first instruction of the activation code.
12. \( \text{reg}_1 \) points to stack with \( \emptyset \) used and \( ms'_{\text{unused}} \) unused
13. \( \text{reg}_1(\text{pc}) = \text{updatePcPerm}(\text{reg}(\text{pc})) \)

76
14. \( \text{reg}_1(r) = \text{reg}(r) \)
15. \( \text{reg}_1(\overline{r}_{\text{args}}) = \text{reg}(\overline{r}_{\text{args}}) \)
16. \( \forall r' \in \text{RegisterName} \setminus \{ \text{pc}, r_{\text{stk}}, r, \overline{r}_{\text{args}} \} . \text{reg}_1(r') = 0 \)

In order to use Lemma 8, we now need to show

\( (n_1, (\text{reg}_1, ms_{\oplus} ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms'_{\text{unused}})) \in \mathcal{O}(W_1) \)

where

\[ W_1 = \text{revokeTemp}(W)[v^{\text{sta}}(\text{temp}, ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms_f), v^{\text{pwl}}(\text{dom}(ms'_{\text{unused}}))] \]

to this end use Hyp-Callee (5.). To use this everything is satisfied directly by assumptions but the following:

17. \( ms_{\oplus} ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms'_{\text{unused}} \subset n-1 W_1 \)

Here we apply Lemma 66. By assumption 1., we have \( ms : n \text{ revokeTemp}(W) \). So it suffices to show

\[ ms_f \oplus ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms'_{\text{unused}} \subset n-1 [v^{\text{sta}}(\text{temp}, ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms_f), v^{\text{pwl}}(\text{dom}(ms'_{\text{unused}}))] \]

This turns out to be trivial as \( ms_f, ms_{\text{stk}}, \) and \( ms_{\text{act}} \) match the static region. \( ms'_{\text{unused}} \) is all zeroes, to it trivially satisfies the \( v^{\text{pwl}} \) region.

18. \( (n - 1, \text{reg}(r_{\text{stk}})) \in \mathcal{V}(W_1) \)

Use Lemma 62 with 12., and that \( W_1 \) has region \( v^{\text{pwl}}(\text{dom}(ms'_{\text{unused}})) \).

19. \( (n - 1, c_{\text{ret}}) \in \mathcal{V}(W_1) \)

To this end let

19.1. \( n' < n - 1 \)
19.2. \( W_2 \supseteq v^{\text{pub}} W_1 \)

be given and show

\( (n', \text{updatePcPerm}(c_{\text{ret}})) \in \mathcal{E}(W_2) \)

To this assume

19.3. \( n'' \leq n' \)
19.4. \( (n'', \text{reg}_2) \in \mathcal{R}(W_2) \)
19.5. \( ms' \cdot n'' W_2 \)

be given and show

\( (n'', (\text{reg}_2[pc \mapsto \text{updatePcPerm}(c_{\text{ret}})], ms')) \in \mathcal{O}(W_2) \) (17)

From 19.2. and 19.5., we can deduce that the memory can be split in the following way:

\( ms' = ms''_{\oplus} ms_r \oplus ms_{\text{stk}} \oplus ms_{\text{act}} \oplus ms'_{\text{unused}} \oplus ms_f \)

where \( ms'' \) is the ”permanent” part of memory we get from Lemma 63, \( ms_r \) is the part ”re-voked” of memory from the same lemma that is not otherwise specified, and \( \text{dom}(ms'_{\text{unused}}) = \text{dom}(ms''_{\text{unused}}) \). From Lemma 63 we also get

77
19.6. $m_{s''} : n'' \rightarrow \text{revokeTemp}(W_2)$

Assume $n''$ is large enough to execute the rest of the $\text{scall}$ instructions. If $n''$ is not large enough, then 17 is trivial to show. To show 17 apply Lemma 8 where $m_s$ is the revoked part. Let $m_{s'}_{frame}$ be given, the execution until just after the $\text{scall}$ proceeds as follows:

$$(\text{reg}_2[\text{pc} \mapsto \text{updatePcPerm}(c_{\text{ret}})], m_{s'} \cup m'_{s_{\text{var}}_{frame}}) \rightarrow (\text{reg}_3, m_{s'} \cup m'_{s_{frame}})$$

where

19.7.

$$\text{reg}_3(r) = \begin{cases} 
  c_{\text{next}} & r = \text{pc} \\
  c_{\text{stk}} & r = r_{\text{stk}} \\
  \text{reg}(r) & r \in \{ r_{\text{priv}} \} \\
  \text{reg}_2(r) & \text{otherwise}
\end{cases}$$

19.8. $\text{reg}_3$ points to stack with $m_{s_{\text{stk}}}$ used and $m_{s_{\text{act}}} \cup m'_{s_{\text{unused}}} \text{ unused}$

At this point, we use Hyp-Cont [6] to show the observation predicate condition of Lemma 8

$$(n'', (\text{reg}_3, m'' \cup m'_{s_{\text{stk}}} \cup m_{s_{\text{act}}} \cup m'_{s_{\text{unused}}} \cup m_{s_f})) \in \mathcal{O}(W_2)$$

which

- $n'' \leq n - 2$
  - Follows from 19.1.
- $W_2 \supseteq \text{revokeTemp}(W)$
  - We have $W_1 \supseteq \text{revokeTemp}(W)$ and assumption 19.2, we get this by transitivity of $\supseteq$.
- $m'' : n'' \rightarrow \text{revokeTemp}(W_2)$
  - Exactly 19.6.
- for all $r$, we have that:

  $$\text{reg}_3(r) = \begin{cases} 
  c_{\text{next}} & \text{if } r = \text{pc} \\
  \text{reg}(r) & \text{if } r \in \{ r_{\text{priv}} \} \\
  \mathcal{V}(\text{revokeTemp}(W_2)) & \text{if } \text{reg}_3(r) \text{ is a global capability and } r \not\in \{ \text{pc}, r_{\text{priv}}, r_{\text{stk}} \}
\end{cases}$$

  The two first cases follows from 19.7. The third follow from assumption 19.4. and 79.
- $\text{reg'}$ points to stack with $m_{s_{\text{stk}}}$ used and $m_{s_{\text{act}}} \cup m''_{s_{\text{unused}}} \text{ unused}$
  - Exactly 19.8.
5.4.11 Malloc macro-instruction correctness

**Definition 6.** We say that \((\text{reg}, \text{ms})\) links key as \(j\) to \(c_{\text{malloc}}\) iff

- \(\text{reg}(\text{pc}) = ((\text{perm}, g), \text{base}, \text{end}, a)\)
- \(\text{ms}(\text{base}) = ((\_\_\_), \text{base}_{\text{link}}, \_\_\_)\)
- \(\text{ms}(\text{base}_{\text{link}} + j) = c\)

**Lemma 59 (malloc works).** If

- \((\text{reg}, \text{ms})\) is looking at \(\text{malloc} r k\) followed by \(c_{\text{next}}\)
- \(k \geq 0\)
- \((\text{reg}, \text{ms})\) links \(\text{malloc}\) as \(k\) to \(c_{\text{malloc}}\)
- \(c_{\text{malloc}}\) satisfies the \(\text{malloc}\) specification with \(\iota_{\text{malloc}, 0}\)
- \(W \supseteq \text{priv}[i \mapsto \iota_{\text{malloc}, 0}]\)
- \(\text{ms} ::_n W\)
- \(\text{ms} = \text{ms'} \uplus \text{ms}_{\text{footprint}}\)
- \(\text{ms}_{\text{footprint}} ::_n [i \mapsto W(i)]\)

**Hyp-Cont** If

- \(n' \leq n - 1\)
- \(\iota_{\text{malloc}} \supseteq \text{pub} W(i)\)
- \(\text{ms}'_{\text{footprint}} \uplus \text{ms}' ::_n W[i \mapsto \iota_{\text{malloc}}]\)
- \(\text{ms}'_{\text{footprint}} ::_n [i \mapsto W(i)]\)

\[
\text{reg}'(r') = \begin{cases} 
    c_{\text{next}} & r' = \text{pc} \\
    \text{reg}(r) & r' \neq \text{pc}, r, r_1, \text{RegisterName}, \text{pc}, r_1 \\
\end{cases}
\]

- \(\text{end} - \text{base} = k - 1\)
- \(\text{dom}(\text{ms}_{\text{alloc}}) = [\text{base}, \text{end}]\)
- \(\forall a \in [\text{base}, \text{end}], \text{ms}_{\text{alloc}}(a) = 0\)

Then we have \(\left(n', (\text{reg}', \text{ms'} \uplus \text{ms}'_{\text{footprint}} \uplus \text{ms}_{\text{alloc}})\right) \in \mathcal{O}(W[i_{\text{malloc}}])\)

Then \(\left(n, (\text{reg}, \text{ms})\right) \in \mathcal{O}(W)\)
5.4.12 Create closure macro-instruction correctness

Lemma 60 (crtcls works). If

- \((\text{reg}, \text{ms})\) is looking at \(\text{crtcls}(x, r)\) followed by \(c_{\text{next}}\)
- \((\text{reg}, \text{ms})\) links malloc as \(k\) to \(c_{\text{malloc}}\)
- \(c_{\text{malloc}}\) satisfies the malloc specification with \(t_{\text{malloc}, 0}\)
- \(W \supseteq W[i \mapsto t_{\text{malloc}, 0}]\)
- \(\text{ms} : n W\)
- \(\text{ms} = \text{ms}' \uplus \text{ms}_{\text{footprint}}\)
- \(\text{ms}_{\text{footprint}} : n [i \mapsto W(i)]\)

**Hyp-Cont** If

- \(- n' \leq n\)
- \(- t_{\text{malloc}} \supseteq W(i)\)
- \(- \text{ms}' \uplus \text{ms}'_{\text{footprint}} : n W[i \mapsto t_{\text{malloc}}]\)
- \(- \text{ms}'_{\text{footprint}} : n [i \mapsto t_{\text{malloc}}]\)

\[
\text{reg}'(r') = \begin{cases} 
  c_{\text{next}} & r' = \text{pc} \\
  c_{\text{cls}} = ((\text{E, GLOBAL}), \text{base, end}, \text{base} + 2) & r' = r_1 \\
  \text{reg}(r) & r' \notin \{\text{pc}, r_1\} \cup \text{RegisterName}_t 
\end{cases}
\]

- \(- \text{ms}_{\text{cls}} = \text{ms}_{\text{act}} \uplus \text{ms}_{\text{env}}\)
- \(- c_{\text{cls}} = ((\text{E, GLOBAL}), \ldots)\)
- \(- c_{\text{env}} = ((\text{RW, GLOBAL}), \text{base}_{\text{env}}, \text{end}_{\text{env}}, \text{base}_{\text{env}})\)
- \(- \text{dom}(\text{ms}_{\text{env}}) = [\text{base}_{\text{env}}, \text{end}_{\text{env}}]\)
- \(- \text{ms}_{\text{env}}(\text{base}_{\text{env}}, \ldots, \text{end}_{\text{env}}) = \text{reg}(\overline{r})\)
- **Hyp-act**

**If**

* \(\text{reg}''(\text{pc}) = \text{updatePcPerm}(c_{\text{cls}})\)

Then \(\exists k. \forall \text{ms}_f. (\text{reg}''', \text{ms}'' \uplus \text{ms}_{\text{cls}} \uplus \text{ms}_f) \rightarrow_k (\text{reg}'''', \text{ms}''' \uplus \text{ms}_{\text{cls}} \uplus \text{ms}_f)\) where

\[
\text{reg}'''(r') = \begin{cases} 
  c_{\text{env}} & r' = \text{pc} \\
  \text{updatePcPerm}(\text{reg}(r)) & r' \notin \text{RegisterName}_t 
\end{cases}
\]

Then we have \((n', (\text{reg}', \text{ms}' \uplus \text{ms}_{\text{footprint}} \uplus \text{ms}_{\text{cls}})) \in \mathcal{O}(W[i \mapsto t_{\text{malloc}}])\)

Then

\((n, (\text{reg}, \text{ms})) \in \mathcal{O}(W)\)

\[\blacksquare\]
5.4.13 Helper lemmas about the stack

Lemma 61. If

- \( perm \in \{ rx, rwx, rwlx \} \)
- \( (n, (base, end)) \) readCondition(LOCAL)(W)
- \( (n, (base, end)) \) writeCondition(\( i^{pwl} \), LOCAL)(W)

then

\( (n, perm, base, end) \in executeCondition(LOCAL)(W) \)

\( \blacksquare \)

Proof of Lemma 61. Assume

1. \( perm \in \{ rx, rwx, rwlx \} \)
2. \( (n, (base, end)) \) readCondition(LOCAL)(W)
3. \( (n, (base, end)) \) writeCondition(\( i^{pwl} \), LOCAL)(W)

Let \( W' \supseteq_{\text{pub}} W \), \( a \), and \( n' \leq n \) be given and show

\( (n', ((perm, LOCAL), base, end, a)) \in E(W') \)

Consider each of the three cases for \( perm \):

4. \( perm = rwlx \)
   - In this case \( \iota = i^{pwl} \). If we use the FTLR (Theorem 2), then we are done. It suffices to show:
     4.1. \( (n', (base, end)) \) readCondition(LOCAL)(W')
        Follows from Lemma 34, Lemma 36 and assumption 2.
     4.2. \( (n', (base, end)) \) writeCondition(\( i^{pwl} \), LOCAL)(W')
        Follows from Lemma 37, Lemma 36 and assumption 3.

5. \( perm = rx \)
   - In this case \( \iota = i^{nwl} \). If we use the FTLR (Theorem 2), then we are done. It suffices to show:
     5.1. \( (n', (base, end)) \) readCondition(LOCAL)(W')
        Follows from Lemma 34, Lemma 36 and assumption 2.
     5.2. \( (n', (base, end)) \) writeCondition(\( i^{nwl} \), LOCAL)(W')
        Follows from Lemma 33, Lemma 37, Lemma 36 and assumption 3.

6. \( perm = rwx \)
   - In this case \( \iota = i^{nwl} \). If we use the FTLR (Theorem 2), then we are done. It suffices to show:
     6.1. \( (n', (base, end)) \) readCondition(LOCAL)(W')
        Follows from Lemma 34, Lemma 36 and assumption 2.

\( \square \)
Lemma 62 (Stack capability in value relation). If

- $\text{reg}$ points to stack with $\emptyset$ used and $ms$ unused
- $\exists r. W(r) = v_{pwl}(\text{dom}(ms))$

then

$$(n, \text{reg}(r_{stk})) \in V(W)$$

Proof of Lemma 62. Say

$$\text{reg}(r_{stk}) = c_{stk} = ((\text{rwx}, \text{LOCAL}), \text{base}, \text{end}, \_\_\_)$$

Show

1. $(n, (\text{base}, \text{end})) \in \text{readCondition}((\text{LOCAL})(W))$:
   Amounts to
   $$v_{pwl}(\text{dom}(ms)) \overset{n}{\leq} v_{pwl}(\text{base}, \text{end})$$
   which is true as they are even equal.

2. $(n, (\text{base}, \text{end})) \in \text{writeCondition}(v_{pwl}, (\text{LOCAL})(W))$:
   Using Lemma 12, this amounts to
   $$v_{pwl}(\text{dom}(ms)) \overset{n}{\geq} v_{pwl}(\text{base}, \text{end})$$
   which is true as they are even equal.

3. $(n, (\text{rwlx}, \text{base}_{stk}, \text{end}_{stk})) \in \text{executeCondition}((\text{LOCAL})(W))$
   Using 2 and 1 we can use Lemma 61.

4. $(n, (\text{rwx}, \text{base}_{stk}, \text{end}_{stk})) \in \text{executeCondition}((\text{LOCAL})(W))$
   Using 2 and 1 we can use Lemma 61.

5. $(n, (\text{rx}, \text{base}_{stk}, \text{end}_{stk})) \in \text{executeCondition}((\text{LOCAL})(W))$
   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).

$$\forall ms, n, W, W'$$

$$ms :_n W \Rightarrow \exists ms', ms_r$$

$$ms = ms' \uplus ms_r \land ms' :_n \text{revokeTemp}(W)$$

Proof of Lemma 63.
Lemma 64 (Revoke temporary memory satisfaction 2).
\[
\forall ms, n, W, R : \text{active}(W) \rightarrow \text{MemSegment}.
\]
\[
ms :_{n} p \ W \Rightarrow \\
\exists ms', ms_r.
\]
\[
ms = ms' \uplus ms_r \land \\
ms' :_{n} p_{\text{dom}(\lfloor W \rfloor_{\text{perm}})} \text{revokeTemp}(W) \land \\
ms_r = \biguplus_{r \in \lfloor W \rfloor_{\text{temp}}} P(r) \land \\
ms' = \biguplus_{r \in \lfloor W \rfloor_{\text{perm}}} P(r)
\]

Proof of Lemma 64.

Lemma 65 (Revoke temporary memory with stack).
\[
\forall n, ms, W, \text{reg}, r_{\text{stk}}, g, \text{base}, \text{end}, a.
\]
\[
ms :_{n} W \land (n, \text{reg}) \in R(W) \land \\
\text{reg}(r_{\text{stk}}) = ((\text{RWLX}, g), \text{base}, \text{end}, a) \land b \leq c \\
\exists ms', ms_r,
\]
\[
ms' :_{n} \text{revokeTemp}(W) \land ms = ms' \uplus ms_r
\]

Proof of Lemma 65.

Lemma 66 (Disjoint memory satisfaction).
\[
\forall n, \forall ms, ms', ms''. \forall W, W', W''. 
\]
\[
ms'' = ms \uplus ms' \land W'' = W \uplus W' \land ms :_{n} W \land ms' :_{n} W' \Rightarrow \\
ms'' :_{n} W''
\]

Proof of Lemma 66.

Lemma 67 (Memory satisfaction and static regions).
\[
ms :_{n} [i \mapsto \iota^\text{st}(v, ms)]
\]

Proof of Lemma 67.

Lemma 68 (Data only memory and standard regions). If
\begin{itemize}
  \item $\forall a \in \text{dom}(ms). ms(a) \in \mathbb{N}$
  \item $\iota \in \{\iota^\text{pw}, \iota^\text{nw}, \iota^\text{nul}, \iota^\text{pul}, \iota^\text{null}\}$
\end{itemize}

83
then

\[ ms : n \to (i \mapsto \iota(\text{dom}(ms))) \]

\[ \Box \]

**Proof of Lemma 68.**

5.4.15 Future worlds

**Lemma 69** (World public future world of revoked world).

\[ \forall W. \text{revokeTemp}(W) \supseteq_{\text{pub}} W \]

\[ \Box \]

**Proof of Lemma 69.** For all \( r \) where \( W(r) = (\text{temp}, s, \phi_{\text{pub}}, \phi, H) \), we have \( \text{revokeTemp}(W) = \text{revoked} \). By the public future region relation we have

\[ W(r) = (\text{temp}, s, \phi_{\text{pub}}, \phi, H) \supseteq_{\text{pub}} \text{revokeTemp}(W)(r) = \text{revoked} \]

all other regions remain unchanged, so this follows by reflexivity of the public future region relation.

\[ \Box \]

**Lemma 70** (World private future world of revoked world).

\[ \forall W. \text{revokeTemp}(W) \supseteq_{\text{priv}} W \]

\[ \Box \]

**Proof of Lemma 70.**

**Lemma 71** (Public future world relation included in private future world relation).

\[ W' \supseteq_{\text{pub}} W \Rightarrow W' \supseteq_{\text{priv}} W \]

\[ \Box \]

**Proof of Lemma 71.**

**Lemma 72** (Transitivity properties between private and public future worlds).

\[ W'' \supseteq_{\text{priv}} W' \land W' \supseteq_{\text{pub}} W \Rightarrow W'' \supseteq_{\text{priv}} W \]

and

\[ W'' \supseteq_{\text{pub}} W' \land W' \supseteq_{\text{priv}} W \Rightarrow W'' \supseteq_{\text{priv}} W \]

\[ \Box \]

**Proof of Lemma 72.**

**Lemma 73.**

\[ \forall n, W_1, W_2, W'_1. \quad W_1 \equiv W_2 \land W'_1 \supseteq_{\text{pub}} W_1 \Rightarrow \exists W'_2. W'_2 \equiv W'_1 \land W'_2 \supseteq_{\text{pub}} W_2 \]

\[ \Box \]
Proof of Lemma 73. Construct $W'_2$ as follows:

$$W_2(r) = \begin{cases} (v'_1, s'_1, \phi_{pub2}, \phi_2, H_2) & \text{if } r \in \text{dom}(W_2) \text{ and } W'_1(r) = (v'_1, s'_1, \ldots, \ldots) \\ W'_1(r) & \text{otherwise} \end{cases}$$

Notice $\text{dom}(W'_2) = \text{dom}(W'_1)$.

Lemma 74.

$$\forall n, W_1, W_2, W'_1. \quad W_1 \equiv W_2 \land W'_1 \equiv_{\text{prv}} W_1 \Rightarrow \exists W'_2. W'_2 \equiv W'_1 \land W'_2 \equiv_{\text{prv}} W_2$$

Proof of Lemma 74. Construct $W'_2$ as follows:

$$W_2(r) = \begin{cases} (v'_1, s'_1, \phi_{pub2}, \phi_2, H_2) & \text{if } r \in \text{dom}(W_2) \text{ and } W'_1(r) = (v'_1, s'_1, \ldots, \ldots) \\ W'_1(r) & \text{otherwise} \end{cases}$$

5.4.16 Value relation

Lemma 75 (Value relation downwards closed).

$$n' \leq n \land (n, w) \in V(W) \Rightarrow (n', w) \in V(W)$$

Proof. By definition of $V(W)$ using Lemma 36, 39, 42 and 45.

Lemma 76 (Register relation downwards closed).

$$n' \leq n \land (n, w) \in R(W) \Rightarrow (n', w) \in R(W)$$

Proof. By definition of $R(W)$ using Lemma 75.

Lemma 77 (Value relation monotone wrt $\equiv_{\text{pub}}$).

$$W' \equiv_{\text{pub}} W \land (n, w) \in V(W) \Rightarrow (n, w) \in V(W')$$

Proof of lemma 77. Follows from Lemma 34, Lemma 37, Lemma 40, and Lemma 43.

Lemma 78. If

$$(n, w) \in V(\text{revokeTemp}(W))$$

then

$$(n, w) \in V(W)$$
Proof of Lemma 78. Follows from Lemma 29, Lemma 30, Lemma 31, and Lemma 32.

Lemma 79 (Global capabilities monotone wrt $\sqsupseteq_{\text{priv}}$).

\[ \forall n, \text{perm}, \text{base}, \text{end}, a, W, W'. \]
\[ (n, ((\text{perm}, \text{GLOBAL}), \text{base}, \text{end}, a)) \in \mathcal{V}(W) \land W' \sqsupseteq_{\text{priv}} W \]
\[ \Rightarrow (n, ((\text{perm}, \text{GLOBAL}), \text{base}, \text{end}, a)) \in \mathcal{V}(W') \]

Proof of Lemma 79. Assume

1. $\text{perm} \notin \{\text{rwl, rwlx}\}$
2. $W' \sqsupseteq_{\text{priv}} W$
3. $(n, ((\text{perm}, \text{GLOBAL}), \text{base}, \text{end}, a)) \in \mathcal{V}(W)$

and show

\[ (n, ((\text{perm}, \text{GLOBAL}), \text{base}, \text{end}, a)) \in \mathcal{V}(W') \]

to this end consider the possible cases of $\text{perm}$ and show that each of the necessary conditions hold:

1. $\text{perm} = o$
   Trivial
2. $\text{perm} = \text{ro}$
   Follows from Lemma 35
3. $\text{perm} = \text{rw}$
   Follows from Lemma 35 and Lemma 38
4. $\text{perm} = \text{rx}$
   Follows from Lemma 35 and Lemma 41
5. $\text{perm} = \text{rwx}$
   Follows from Lemma 35, Lemma 38, and Lemma 41
6. $\text{perm} = \text{e}$
   Lemma 44

Lemma 80 (Non local words monotone wrt $\sqsupseteq_{\text{priv}}$).

\[ \forall n, \text{perm, base, end, a, W, W', w.} \]
\[ w \text{ is non-local} \land \]
\[ (n, w) \in \mathcal{V}(W) \land W' \sqsupseteq_{\text{priv}} W \]
\[ \Rightarrow (n, w) \in \mathcal{V}(W') \]

Proof of Lemma 80. If $w = ((\text{perm}, \text{GLOBAL}), \text{base, end, a})$, 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 $W'' \in \text{World}.$
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:

```c
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):

```ml
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):

```plaintext
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. 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):

http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html
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


