Proofs of a couple of Lemmas:

Lemma 24,	p. 49
Lemma 25, 26, 27	p. 50
Lemma 33	p. 51
Lemma 35 – 38	p. 52
Lemma 39, 41	p. 53
Lemma 42, 44	p. 54
Lemma 58 – 59	p. 70
Lemma 60 – 63	p. 71
Lemma 65 – 67	p. 72

Some lemmas have been checked but nothing has been written.

~~Proof~~ of Lemma 24

Induction n: $n = 0$ trivial (no step, won't halt).

Assume $W \overset{n}{=} W'$

let $(n', (reg, ms)) \in \mathcal{O}(W)^{(I)}$ for $n' < n$ show

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

let ms_f, mem', isn' be given and assume

$(reg, ms \uplus ms_f) \rightarrow_i (halted, mem')$

by (I) get

$W_1 \overset{priv}{\sqsupseteq} W, ms_r, ms'$ s.t.

$mem' = ms' \uplus ms_r \uplus ms_f$

$ms' :_{n-i} W_1$

by Lemma 73. get $W_2 \overset{n}{=} W_1$ and $W_2 \overset{priv}{\sqsupseteq} W'$.

~~As $i \geq 0$ (took atleast one step to halt)~~, by lemma and mem-sat DL

$ms' :_{n-i} W_2$.

Lemma (mem sat n.e.)

$ms :_n W \wedge W \overset{n}{=} W' \Rightarrow ms :_n W'$

Proof

Follows from interpretation function n.e. and $\mathcal{E}^{-1}$ preserves n.e.

Proof of lemma 58

From $ms :_n W$ get partition P and let

$$ms' = \biguplus_{r \in \lfloor W \rfloor_{\{perm\}}} P(r) \qquad ms_r = \biguplus_{r \in \lfloor W \rfloor_{\{temp\}}} P(r)$$

Use $P' = P|$ ~~[illegible]~~ $\text{dom}(\lfloor W \rfloor_{\{perm\}})$ for the second part.

As $\text{revokeTemp}(W) \sqsupseteq^{priv} W$ all the interpretation functions can "handle" this change. i.e. for

$r \in \text{dom}(P')$ show

$$(n, \underset{P(r)}{\underset{\shortparallel}{P'(r)}}) \in P'(r).H \; P'(r).s \; \text{revokeTemp}(W)$$

we know from $ms :_n W$ that

$$(n, P(r)) \in \underset{\text{mono } \sqsupseteq^{priv}}{P(r).H} \; P'(r).s \; W$$

Proof of Lemma 59

Goes like the proof of lemma 58.

Proof of Lemma 60

Follows from Lemma 58

Proof of Lemma 61

Use the combination of the partitions from $ms :_n W$ and $ms' :_n W'$

By assumption the two memory segments are disjoint, so o.k. partitioning

Further

$$W'' \sqsupseteq^{pub} W \quad \text{and} \quad W'' \sqsupseteq^{pub} W'$$

gives us the necessary satisfaction because all interpretation functions are at least mono w.r.t. $\sqsupseteq^{pub}$.

Reasoning about a Capability Machine with Local Capabilities
Provably Safe Stack and Return Pointer Management (without OS Support)
Technical Appendix Including Proofs and Details

February 17, 2017

Contents

- $w \in \mathbb{Z}$

$\blacksquare$

$$\begin{tikzcd}
\text{global} \arrow[d, dash] \\
\text{local}
\end{tikzcd}$$

Figure 1: Permission hierarchy

Things to note:

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

o	No permissions. Grants no permissions
ro	Read only. Grants read permission
rw	Read-write. Grants read and write permission. Storage of local capabilities prohibited.
rwl	Read-write, permit write local. Grants read and write permission. Storage of local capabilities possible.
rx	Execute permission. Grants execute and read permissions.
e	Enter permission. This permission grants no access, but when jumped to, it will turn into an rx permission.
rwx	Read-write-execute permission. Grants read, write, and execute permissions. Storage of local capabilities prohibited.
rwlx	Read-write-execute, permit write local. Grants read, write, and execute permissions. Storage of local capabilities possible.

Table 1: The permissions in this capability system

Further, assume an inverse function, *decodePermPair*, that decodes permissions

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

We define the operational semantics as follows:

$$\begin{array}{ll} \Phi \to [\![decode(\Phi.\text{mem}(a))]\!](\Phi) & \text{if } \Phi.\text{reg}(\text{pc}) = ((perm, g), base, end, a) \\ & \quad \text{and } base \leq a \leq end \\ & \quad \text{and } perm \in \{\text{rx}, \text{rwx}, \text{rwlx}\} \\ \Phi \to failed & \text{otherwise} \end{array}$$

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

$$readAllowed(perm) = \begin{cases} true & \text{if } perm \in \{\text{rwx}, \text{rwlx}, \text{rx}, \text{rw}, \text{rwl}, \text{ro}\} \\ false & \text{otherwise} \end{cases}$$

$$writeAllowed(perm) = \begin{cases} true & \text{if } perm \in \{\text{rwx}, \text{rwlx}, \text{rw}, \text{rwl}\} \\ false & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

$$\llbracket \texttt{subseg}\ \lfloor r \rfloor\ rv_1\ rv_2 \rrbracket = \begin{cases} \mathit{updatePc}(\Phi[\mathrm{reg}.r \mapsto c]) & \text{if } \Phi.\mathrm{reg}(r) = ((\mathit{perm}, g), \mathit{base}, \mathit{end}, a) \\ & \text{and for } i \in \{1,2\} \\ & n_i = rv_i \text{ or } n_i = \Phi.\mathrm{reg}(rv_i) \\ & \text{and in either case } n_i \in \mathbb{Z} \\ & \text{and } \mathit{base} \le n_1 \text{ and } n_2 \le \mathit{end} \\ & \text{and } \mathit{perm} \neq \mathrm{e} \\ & \text{and } c = ((\mathit{perm}, g), n_1, n_2, a) \\ \mathit{failed} & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

After adding local capabilities, most of the definitions have changed slightly to take into account the new form the capabilities take. The only two instructions that have undergone enough change to deserve to be mentioned are **`store`** and **`restrict`**.

`store` has the same conditions as before, but if what we try to write is a local capability, then it needs to be a write access that permits write local.

`restrict` looks like before, but now it uses a "permission pair" which is the access permission and the locality. Whether a **`restrict`** succeeds depends on the ordering, so the real change lies in the new ordering, which is described further above (it is the point-wise ordering of the pair).

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

■

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

3 Macros

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

3.1 Linking and ABI

In order to make capabilities to trusted code (and possibly untrusted code) available, we assume that some sort of linker has made these available. This is done in the following way: For every function, the first memory cell the capability for that function governs contains a capability for the linking table. Each function name in a program corresponds to an offset in the table, e.g., `malloc` could be at offset 0. When a name is used in a program, it indicates what entry from the linking table to pick. The table should always be accessible by taking a copy of the capability in the pc-register and adjusting it to point to the first cell it governs.

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

3.2 Flag table

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

The flag table should never be shared between distrusting parties.

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

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

3.3 Macro definitions

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

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

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

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

A more detailed description of each of the above steps:

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.

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

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

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

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

A more detailed description of the above steps:

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

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

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

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

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

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

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.

`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.

`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.

`reqglob` r Tests if register r contains a global capability. If not fail, otherwise continue execution.

`reqperm` r n Tests if register r contains a capability with permission $\mathit{decodePerm}(n)$. If not fail, otherwise continue execution.

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

`mclear` r Clears all cells r has authority over.

Note:

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

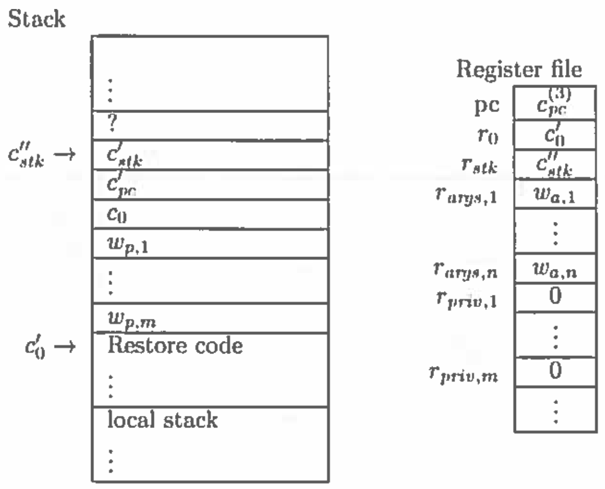

Figure 5: Stack and register-file after the c'_{stk} has been limited to only give authority over the empty part of the stack (the new capability is c''_{stk}). The empty part of the stack has been cleared. c'_0 is made from c'_{stk} by setting it to point to the restore code and restricting it to a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

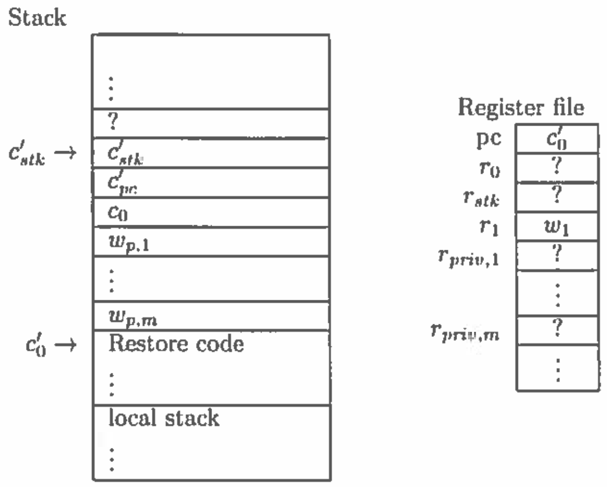

Figure 6: Stack and register-file upon return from f. At this point we have no idea what is in the register-file apart from the pc which we know points to the restore code. The contents of the stack we released access to is also unknown. (Notice that we have changed the order of the registers as we are 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.)

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.

4 Examples

4.1 Encapsulation of Local State

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

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

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

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

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\mathrm{e}, \mathrm{global}), base_{adv}, end_{adv}, base_{adv} + \mathit{offsetLinkFlag}) \\
c_{f1} &\overset{def}{=} ((\mathrm{rwx}, \mathrm{global}), \mathtt{f1} - \mathit{offsetLinkFlag}, \mathtt{1f}, \mathtt{f1}) \\
c_{malloc} &\overset{def}{=} ((\mathrm{e}, \mathrm{global}), base_{malloc}, end_{malloc}, base_{malloc} + \mathit{offsetLinkFlag}) \\
m &\overset{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{frame}
\end{aligned}$$

and

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

where

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

and

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

7. $m'' = ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{l} \uplus ms_{ar} \uplus ms''_{malloc} \uplus ms_{frame}$
8. $ms''_{malloc} :_{n-j-j'} [0 \mapsto \iota'_{malloc}]$
9. $\mathrm{dom}(ms_{ar}) = [b, e]$, and $e - b = len_{ar}$
10. $c_l = ((\mathrm{rwx}, \mathrm{global}), b, e, b)$
11. $\forall a \in [b, e].\, ms_{ar}(a) = 0$

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

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

for some k' where

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

Define

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

define

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

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

- $(n, (base_{adv}, end_{adv})) \in readCondition(\mathrm{global})(W)$
  - Show: $\iota^{nwl,p}_{base_{adv}, end_{adv}} \overset{n}{\precsim} \iota^{pwl}_{base_{adv}, end_{adv}}$: Follows from Lemma 21.

Have

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

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

1. $ms :_{n'} W$
   1.1. Split the memory into the disjoint unions of 1 and show:
      1.1.1. case: $(n', ms_{malloc}) \in \iota'_{malloc}.H(\iota'_{malloc}.s)(W)$
         1.1.1.1. Use $ms_{malloc} :_{n'} [0 \mapsto \iota'_{malloc}]$ with malloc specification context independence property.
      1.1.2. case: $(n', ms_{adv}) \in H^{nwl}_{base_{adv}, end_{adv}} 1 W$
         1.1.2.1. Show $\forall a \in [base_{adv}, end_{adv}].((n' - 1, ms(a)) \in \mathcal{V}(W) \wedge ms(a)$ is non-local)
         1.1.2.1. $a \neq base_{adv}$: trivial, contains instruction only and they are non-local.
         1.1.2.2. $a = base_{adv}$: show $((\mathrm{ro}, \mathrm{global}), link, link + 1, link) \in \mathcal{V}(W)$
         global capabilities are non-local.
         SFTS $\iota^{sta,u}(perm, ms_{link}) \overset{n'}{\precsim} \iota^{pwl}_{link, link+1}$ which follows from Lemma 22.

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

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\mathrm{e}, \mathrm{global}), base_{adv}, end_{adv}, base_{adv} + \mathit{offsetLinkFlag}) \\
c_{f2} &\overset{def}{=} ((\mathrm{rwx}, \mathrm{global}), \mathtt{f2} - \mathit{offsetLinkFlag}, \mathtt{2f}, \mathtt{f2}) \\
c_{malloc} &\overset{def}{=} ((\mathrm{e}, \mathrm{global}), base_{malloc}, end_{malloc}, base_{malloc} + \mathit{offsetLinkFlag}) \\
c_{stk} &\overset{def}{=} ((\mathrm{rwlx}, \mathrm{local}), base_{stk}, end_{stk}, base_{stk} - 1) \\
c_{link} &\overset{def}{=} ((\mathrm{ro}, \mathrm{global}), link, link + 1, link) \\
reg &\in \mathrm{Reg} \\
m &\overset{def}{=} ms_{f2} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}$$

and

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

where

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

and

- $ms_{f2}(\mathtt{f2} - \mathit{offsetLinkFlag}) = ((\mathrm{ro}, \mathrm{global}), link, link + 1, link)$, $ms_{f2}(\mathtt{f2} - \mathit{offsetLinkFlag} + 1) = ((\mathrm{rw}, \mathrm{global}), flag, flag, flag)$, *the rest of* ms_{f2} *contains the code of* $f2$.
- $ms_{flag} = [flag \mapsto 0]$
- $ms_{link} = [link \mapsto c_{malloc}, link + 1 \mapsto c_{adv}]$
- $ms_{adv}(base_{adv}) = c_{link}$ *and* $\forall a \in [base_{adv} + 1, end]. \; ms_{adv}(a) \in \mathbb{Z}$

if

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

then

$$m'(flag) = 0$$

■

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

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

for

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

and

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

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

Show

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

By Theorem 2 we get

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

getting the desired result amounts to[2]

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

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

Follows from Theorem 2 and Lemma 21.

3. Hyp-Cont
Assume

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

and show

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

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

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

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

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

□

ML-like program:

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

where

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

and

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

if

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

then

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

■

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

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

At this point we can define a world with permanent regions

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

and temporary regions

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

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

If we can show

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

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

$$m'(flag) = 0$$

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

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

where

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

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

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

and i is a suitable number of steps.

To show

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

We use Lemma 55 (we do not use the local frame in the lemma) which requires us to show

1. $ms' :_{n-i} W$
   Partition ms' as follows:
   1.1. ms_{malloc}: governed by $\iota_{malloc,0}$, use malloc specification.
   1.2. $ms_{flag} \uplus ms_{f2}$: governed by region 1, only this memory segment is accepted.
   1.3. ms_{link}: governed by region 2, only this memory segment is accepted. We also need to show that the contents is safe, i.e. shoe
      1.3.1. $(n-i, c_{malloc}) \in \mathcal{V}(W)$: Follows from Lemma 47.
      1.3.2. $(n-i, c_{adv}) \in \mathcal{V}(W)$: Follows from Theorem 2 and Lemma 21.
   1.4. ms_{adv}: Follows from Lemma 22.

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

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

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

- $reg^{(4)}$ points to stack with $\emptyset$ used and $ms^{(3)}_{unused}$ unused
- $reg^{(4)} = reg_0[\mathrm{pc} \mapsto updatePcPerm(c_{adv}), r_0 \mapsto c'_{ret}, r_{stk} \mapsto c''_{stk}, r \mapsto c_{adv}]$
- $(n' - k - 1, c'_{ret}) \in \mathcal{V}(W^{(3)})$
- $(n' - k - 1, c''_{stk}) \in \mathcal{V}(W^{(3)})$

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

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

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

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

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

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

to get

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

3.1.1.3. Hyp-Cont

Assume

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

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

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

and show

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

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

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

As the execution halts immediately,

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

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

$\square$

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

$$\begin{aligned}
c_{adv} &\overset{def}{=} ((\text{rwx}, \text{global}), base_{adv}, end_{adv}, base_{adv} + \mathit{offsetLinkFlag}) \\
c_{g1} &\overset{def}{=} ((\text{e}, \text{global}), \mathtt{g1} - \mathit{offsetLinkFlag}, \mathtt{4f}, \mathtt{g1}) \\
c_{stk} &\overset{def}{=} ((\text{rwlx}, \text{local}), base_{stk}, end_{stk}, base_{stk} - 1) \\
c_{malloc} &\overset{def}{=} ((\text{e}, \text{global}), base_{malloc}, end_{malloc}, base_{malloc} + \mathit{offsetLinkFlag}) \\
c_{link} &\overset{def}{=} ((\text{ro}, \text{global}), link, link, link) \\
m &\overset{def}{=} ms_{g1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}$$

where

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

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

and

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

if

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

then

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

■

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

Definition 2.

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

■

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

For convenience define

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

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

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

which in turn amounts to showing

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

which follows from Lemma 47.
Showing

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

is a bit more involved. It amounts to

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

which in turn is trivial for everything but

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

This amounts to showing

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

which amounts to

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

which follows from Lemma 22.

Using Lemma 61 repeatedly with 3.1., 3.2., and 3.3. gives the desired memory satisfaction.

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

4.1. $(n, reg_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}](c_{stk})) \in \mathcal{V}(W)$
The assumptions on c_{stk} and ms_{stk} in the lemma entail
- $reg_0[r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}]$ points to stack with $\emptyset$ used and ms_{stk} unused

and further there is a ι^{pwl} region for ms_{stk} in W, so the result follows from Lemma 57.

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

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

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

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

Use Lemma 8, so let ms'_f be given, then

$$(reg_1[\text{pc} \mapsto updatePcPerm(c_{g1})], ms_1 \uplus ms'_f) \to_i (reg_2, ms_2 \uplus ms'_{malloc} \uplus ms_{env} \uplus ms_x \uplus ms_{cls} \uplus ms'_f)$$

for some i. Where

Use Lemma 8, let ms''_f be given and take ms_r to be the part of ms^t_3 that $reg_3(r_{stk})$ does not govern. By the operational semantics, we know[3]

$$(reg_3[\mathrm{pc} \mapsto updatePcPerm(c_{cls})], ms^p_3 \uplus ms^t_3 \uplus ms''_f) \to_j (reg_4, ms_4 \uplus ms^t_3 \uplus ms''_f)$$

where

- (reg_4, ms_4) is looking at `scall` $r_1([], [r_0, r_1, r_{env}])$ followed by c_{next}
  - c_{next} is the capability pointing to the next instruction.
- reg_4 points to stack with $\emptyset$ used and ms_{unused} unused
  - `prepstack` did not fail, so the capability must be rwlx and follow the stack convention.
- $reg_3(r_1)$ is a global capability.
  - `reqglob` did not fail
- $ms_4(x) = 0$
- $reg_3(r_{env}) = c_{env}$

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

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

Now we can use Lemma 55 to show:

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

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

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

4.2.2.2.2. Hyp-Callee

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

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

Now it suffices to show

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

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

Amounts to showing:

1) $(n_4 - j - 1, reg_3(r_1)) \in \mathcal{V}(W_5)$, use Lemma 74 with $(n_4 - j - 1, reg_3(r_1)) \in \mathcal{V}(W_3)$, the capability is global, and $W_5 \sqsupseteq^{priv} W_3$. 2) The protected return pointer and the stack capability are in the value relation by Hyp-callee assumptions.

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

Show:

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

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

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

where

- $reg_9(\mathrm{pc}) = updatePcPerm(reg_3(r_0))$ (note $reg_8(r_0) = reg_3(r_0)$)
- $reg_9(r_0) = reg_3(r_0)$
- For all $r \not\in \{\mathrm{pc}, r_0\}$, $reg_9(r) = 0$.
- $\mathrm{dom}(ms_0) = \mathrm{dom}(ms^{(5)}_{unused})$ and $\forall a \in \mathrm{dom}(ms_0).\ ms_0(a) = 0$

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

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

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

We have

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

if the region was permanent in W_3, then it is there because $W_{10} \sqsupseteq^{priv} W_3$. If it was temporary, then it is there because it was just reinstated. If it was revoked in W_3, then it is still there because the only reinstated region were the temporary ones in W_3.

All the future worlds we have been given have been public, so the regions can only have made public transitions. In W_3 region ι_x is in state 0 or 1. In W_{10} region ι_x is in state 1. State 1 can be reached from 0 and 1 using a public transition, so the ι_x in W_{10} is a public future region of the ι_x in W_3.

In other words, all the regions in W_3 have only taken public transitions compared to the corresponding regions in W_{10}.

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

First notice that from

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

we get that there exists a region, r_{advstk} in W_3 such that

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

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

$$ms_7 :_{n_6} revokeTemp(W_8) \tag{7}$$

and

$$ms_3 :_{n_4} W_3 \tag{8}$$

hold for appropriate values. Now use that $n_6 - l < n_4$ and $W_{10} \sqsupseteq^{pub} W_3$ (5.3.1.)[4] to get

$$(n_6 - l, \mathit{updatePcPerm}(\mathit{reg}_0(r_0))) \in \mathcal{E}(W_{10})$$

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

□

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.

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

■

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

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

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

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

and

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

■

We now define the regions to be

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

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

And the worlds are

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

where RegionName = $\mathbb{N}$.

[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.

Erase all but a set of views:

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

Define the function *active*(·) as follows

$$\begin{aligned} &\mathit{active} : \text{World} \rightarrow 2^{\text{RegionName}} \\ &\mathit{active}(W) \stackrel{def}{=} \text{dom}(\lfloor W \rfloor_{\{\text{perm},\text{temp}\}}) \end{aligned}$$

Memory segment satisfaction:

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

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

Standard regions for when writing locally is permitted:

$$\begin{aligned} &\iota^{pwl}_{base,end} : \text{Region} \\ &\iota^{pwl}_{base,end} \stackrel{def}{=} (\text{temp}, 1, =, =, H^{pwl}_{base,end}) \end{aligned}$$

$$H^{pwl} : \text{Addr}^2 \rightarrow \text{State} \rightarrow (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{MemSegment}))$$

$$H^{pwl}_{base,end}\ s\ \hat{W} \stackrel{def}{=} \left\{ (n, ms) \middle| \begin{array}{l} \text{dom}(ms) = [base, end] \wedge \\ \forall a \in [base, end].\, (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\}$$

Revoking all temporary regions:

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

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

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

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

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

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

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

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

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

5.3 Useful regions

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

$$\iota^{sta}(v, ms) = (v, 1, =, =, H^{sta}\ ms)$$
$$H^{sta}\ ms\ s\ \hat{W} = \{(n, ms) \mid n > 0\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}$$

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

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

$$\iota^{cnst}(v, n) = (v, 1, =, =, H^{cnst}\ n)$$
$$H^{cnst}\ n\ s\ \hat{W} = \{(n, ms) \mid n > 0 \wedge \forall a \in \text{dom}(ms).\ ms(a) = n\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}$$

5.4 Lemmas

5.4.1 Anti-reduction for the observation relation

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

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

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

Lemma 10. $\iota^{pwl}_{base,end}$ *is a region for all base and end.* ■

Proof of Lemma 10. Follows from Lemma 9. □

Lemma 11. $\iota^{pwl}_{base,end}$ *is address-stratified.* ■

Proof. Easy unfolding of definitions. □

Lemma 12. $H^{nwl}_{base,end}\ s$ *is monotone w.r.t* $\sqsupseteq^{priv}$ *for all* $s \in \text{State}$ *and base and end* ■

Proof of Lemma 12. Let $\hat{W}' \sqsupseteq^{priv} \hat{W}$ be given and let

$$(n, ms) \in H^{nwl}_{base,end}\ s\ \hat{W} \tag{12}$$

and show

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

From 12, we get $\text{dom}(ms) = [base, end]$. Now let $a \in [base, end]$ be given and show

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

1. follows trivially from 12. 1. follows from Assumption 11, 1. (which we just argued), $\hat{W}' \sqsupseteq^{priv} \hat{W}$, Theorem 1, and Lemma 75. □

Lemma 13. $\iota^{nwl}_{base,end}$ *is a region for all base and end.* ■

Proof of Lemma 13. Follows from Lemma 12 and Lemma 66. □

Lemma 14. $\iota^{nwl}_{base,end}$ *is address-stratified.* ■

Proof. Easy unfolding of definitions. □

Lemma 15. $\iota^{nwl,p}_{base,end}$ *is a region for all base and end.* ■

Proof of Lemma 15. Follows from Lemma 12. □

Lemma 16. $\iota^{sta}(v, ms)$ *is a region for all* $v \in \{\text{perm}, \text{temp}\}$ *and* ms. ■

Proof of Lemma 16. H^{sta} does not depend on $\hat{W}$, so it is trivial to show the necessary non-expansive and monotonicity requirements. □

Lemma 17. $H^{sta,u}(ms)\ s$ *is monotone w.r.t* $\sqsupseteq^{priv}$ *for all* $s \in \text{State}$ *and* ms. ■

Proof of Lemma 17. Let $\hat{W}' \sqsupseteq^{priv} \hat{W}$ be given and let

$$(n, ms') \in H^{sta,u}(ms)\ s\ \hat{W} \tag{13}$$

and show

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

From 13, we get $ms' = ms$. Now let $a \in \text{dom}(ms)$ be given and show

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

Lemma 22.

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

■

Proof of Lemma 22. Essentially the same as the proof of Lemma 20 and Lemma 19. □

Lemma 23.

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

■

Proof of Lemma 23. For $n = 0$ it is trivial, so assume $n > 0$. Say $\iota = (v, s, \phi_{pub}, \phi, H)$, then by $\overset{n}{\simeq}$, we know $s = 1$, $\phi_{pub} \equiv \phi \equiv =$, and $H = H^{pwl}_{base,end}$. It remains to show that $v = \mathrm{temp}$. To do so, we show that it cannot be the case that $v = \mathrm{perm}$. If $v = \mathrm{perm}$, then H must be monotone with respect to $\sqsupseteq^{priv}$. If we can show that this is not the case, then for ι to be a region it must be the case that $v \neq \mathrm{perm}$ and thus $v = \mathrm{temp}$.

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

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

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

$$ms = [\mathit{base} \mapsto ((\mathrm{ro}, \mathrm{local}), b, b, b), \mathit{base} + 1 \mapsto 0, \ldots, \mathit{end} \mapsto 0]$$

It is the case that

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

but

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

as it is not the case that

$$(n - 1, ((\mathrm{ro}, \mathrm{local}), b, b, b)) \in \mathcal{V}(\xi(W')).$$

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

5.4.3 Observation relation

✓ **Lemma 24** (Observation relation ($\mathcal{O}$) non-expansive).

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

■

Proof of Lemma 24. □

Proof of Lemma 29.

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

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

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

and

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

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

Lemma 30. *If*

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

then

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

■

Proof of Lemma 30. Use Lemma 64. □

Lemma 31. *If*

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

then

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

■

Proof of Lemma 31. Use Lemma 64. □

Lemma 32. *If*

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

then

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

■

Proof of lemma 32. Follows from Lemma 21. □

V **Lemma 33** (*readCondition* monotone w.r.t $\sqsupseteq^{pub}$). *If*

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

then

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

■

Proof of Lemma 33. □

~~Direct from def.~~

Directly from def.

Addr-stat dc because $H_\sigma W$ is.
$+ \underset{\sim}{\sigma}^{n-1}$ dc.

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

then

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

■

Proof of Lemma 38. □

Ⓥ **Lemma 39** (*execCondition* monotone w.r.t $\sqsupseteq^{pub}$). *If*

- $W' \sqsupseteq^{pub} W$
- $perm \in \{\text{rx}, \text{rwx}, \text{rwlx}\}$
- $(n, (perm, base, end)) \in executeCondition(g)(W)$

then

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

■

Proof of Lemma 39. transitivity of $\sqsupseteq^{pub}$. □

Lemma 40 (*execCondition* global monotonicity w.r.t $\sqsupseteq^{priv}$). *If*

- $W' \sqsupseteq^{priv} W$
- $perm \in \{\text{rx}, \text{rwx}\}$
- $(n, (perm, base, end)) \in executeCondition(\text{global})(W)$

then

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

■

Proof of Lemma 40. Assume $W_2 \sqsupseteq^{priv} W_1$, $perm \in \{\text{rx}, \text{rwx}\}$ and $(n, (perm, base, end)) \in executeCondition(\text{global})(W$
Now let $W_3 \sqsupseteq^{priv} W_2$, $a \in [base, end]$, and $n' < n$, and show

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

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

Ⓥ **Lemma 41** (*execCondition* downwards-closed). *If*

- $n' \leq n$
- $perm \in \{\text{rx}, \text{rwx}, \text{rwlx}\}$
- $(n, (perm, base, end)) \in executeCondition(g)(W)$

5.4.7 LR Sanity lemmas

Lemma 45 (Heap satisfaction downwards closure).

$$\begin{aligned}&\forall ms, n' \leq n, W.\\&\quad ms :_n W \Rightarrow ms :_{n'} W\end{aligned}$$

■

Proof of Lemma 45. Let ms, $n' \leq n$, and W be given and assume

$$ms :_n W$$

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

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

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

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

We now need to show

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

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

5.4.8 Malloc safe to pass to adversary

Lemma 46 (Safe values are safe to invoke.). *If* $(n+1, w) \in \mathcal{V}(W)$, *then* $(n, updatePcPerm(w)) \in \mathcal{E}(W)$. ■

Proof. 1. Case $w = ((perm, g), base, end, a)$ and $base \leq a \leq end$ and $perm \in \{\text{rx}, \text{rwx}, \text{rwlx}\}$:

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

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

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

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

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

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

10.4. $(n''', (\mathit{base}', \mathit{end}')) \in \mathit{readCondition}(\mathrm{global})(W'')$ for all n''':
By: definition of *readCondition*, using the region $W''(i)$ and Lemma 20.

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

10.6. $(n''', (p, \mathit{base}', \mathit{end}')) \in \mathit{executeCondition}(\iota^{nwl}, \mathrm{global})(W'')$ for all n''', $p \in \{\mathrm{rwx}, \mathrm{rx}\}$:
By: the definition of *executeCondition*, the FTLR (Theorem 2) using Lemmas 37, 34 and the previous two points.

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

10.8. $(n'' - j, \Phi.\mathrm{reg}[r_1 \mapsto ((\mathrm{rwx}, \mathrm{global}), b', e', b')]) \in \mathcal{R}(W'')$:
By Lemma 71, Lemma 26 using the fact that $W'' \sqsupseteq^{pub} W'$ and $(n'', \Phi.\mathrm{reg}) \in \mathcal{V}(W')$, together with the previous point.

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

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

10.11. $(n'' - j + 1, w_{ret}) \in \mathcal{V}(W'')$:
By Lemma 72, the fact that $W'' \sqsupseteq^{pub} W'$, Lemma 70, and teh fact that $(n'', w_{ret}) \in \mathcal{V}(W')$, which follows from $w_{ret} = \Phi.\mathrm{reg}(r_1)$ and $(n'', \mathit{reg}) \in \mathcal{R}(W')$.

10.12. $(n'' - j, \mathit{updatePcPerm}(w_{ret})) \in \mathcal{E}(W'')$:
By Lemma 46 from the previous point.

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

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

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

10.16. $W''' \sqsupseteq^{priv} W'$:
By Lemma 67, using the previous point and the fact that $W'' \sqsupseteq^{pub} W'$.

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

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

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

Lemma 50 (pwl writecond implies nwl). *If* $(n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W)$ *then* $(n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W)\}$. ■

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

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

3. $W(r) \overset{n-1}{\underset{\sim}{\sqsupset}} \iota^{pwl}_{base',end'} \overset{n-1}{\underset{\sim}{\sqsupset}} \iota^{nwl}_{base',end'}$: follows by Lemma 20.

□

Lemma 51 (execCond implies entryCond). *If* $(n, (\mathrm{rx}, base, end)) \in executeCondition(g)(W)$ *then* $(n, (base, end, a)) \in enterCondition(g)(W)$. ■

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

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

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

□

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

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

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

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

1. If $(n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W)$ then $(n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W)$: holds by lemma 50.

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

□

Lemma 53 (Conditions for subseg instruction are sufficient). *If*

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

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

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

- $$\begin{gathered} perm = \mathrm{rx} \wedge \\ (n, (base, end)) \in readCondition(g)(W) \end{gathered}$$

- $$\begin{gathered} perm = \mathrm{rwx} \wedge \\ (n, (base, end)) \in readCondition(g)(W) \wedge \\ (n, (base, end)) \in writeCondition(\iota^{nwl}, g)(W) \end{gathered}$$

- $$\begin{gathered} perm = \mathrm{rwlx} \wedge \\ (n, (base, end)) \in readCondition(g)(W) \wedge \\ (n, (base, end)) \in writeCondition(\iota^{pwl}, g)(W), \end{gathered}$$

then

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

■

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

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

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

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

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

6. $(n', \Phi.\mathrm{reg}(\mathrm{pc})) \in \mathcal{V}(W)$. Proof:
   6.1. Assume: $perm' \in \{\mathrm{rx}, \mathrm{rwx}, \mathrm{rwlx}\}$ with $perm' \sqsubseteq perm$
   Suffices: $(n', (perm', base, end)) \in executeCondition(g)(W)$
   By: the definition of $\mathcal{V}(\cdot)$ using the assumptions
   6.2. Assume: $n'' < n'$, $W' \sqsupseteq W$, $a' \in [base, end]$, $g = \mathrm{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub}$, $g = \mathrm{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$.
   Suffices: $(n'', ((perm, g), base, end, a')) \in \mathcal{E}(W')$. By: definition of $executeCondition(g)(W)$
   6.3. By induction, using the assumptions and Lemmas 35 and 38.

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

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

11.2.1. Case $\Phi''.\text{reg}(r) = \Phi.\text{reg}(r)$: $(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows from Step 7. using Lemma 70.

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

11.2.3. $\Phi''.\text{reg}(r) = w$ and $\Phi.\text{reg}(r_2) = ((perm', g'), base', end', a') = c$ and $readAllowed(perm')$ and $withinBounds(c)$ and $w = \Phi.\text{mem}(a')$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 48 using the fact that $\Phi.\text{mem} :_{n'} W$ and $(n', \Phi.\text{reg}(r_2)) \in \mathcal{V}(W)$ which we have from step 7..

11.2.4. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r_1) = ((perm', g'), base', end', a')$ and $perm' \neq \text{e}$ and $c = ((perm', g'), base', end', a' + z)$ for some $z \in \mathbb{Z}$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 49 and 70 using the fact that $(n', \Phi.\text{reg}(r_1)) \in \mathcal{V}(W)$ which we have from step 7..

11.2.5. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $(perm'', g'') \sqsubseteq (perm', g')$ and $c = ((perm'', g''), base', end', a')$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 52 and 70 using the fact that $(n', \Phi.\text{reg}(r)) \in \mathcal{V}(W)$ which follows from $(n', \Phi.\text{reg}) \in \mathcal{R}(W)$ by definition.

11.2.6. $\Phi''.\text{reg}(r) = c$ and $\Phi.\text{reg}(r) = ((perm', g'), base', end', a')$ and $base' \leq base''$ and $end'' \leq end'$ and $c = ((perm', g'), base'', end'', a')$ and $perm' \neq \text{e}$:
$(n' - 1, \Phi''.\text{reg}(r)) \in \mathcal{V}(W)$ follows by Lemmas 53 and 70 using the fact that $(n', \Phi.\text{reg}(r)) \in \mathcal{V}(W)$ which follows from $(n', \Phi.\text{reg}) \in \mathcal{R}(W)$ by definition.

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

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

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

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

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

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

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

11.4.1.1.4. $updatePcPerm(\Phi.\text{reg}(lv)) = ((\text{rx}, g'), base', end', a')$: by definition of $updatePcPerm(\cdot)$.

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

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

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

11.4.1.2.3. $(n' - 1, ((perm', g'), base', end', a')) \in \mathcal{E}(W)$: By definition of $executeCondition(\cdot)$, taking $n' = n' - 1$, $W' = W$ and $a = a'$. Note that $a' \in [base', end']$ because we have $withinBounds(\Phi.\text{reg}(lv))$.

11.4.1.2.4. $updatePcPerm(\Phi.\text{reg}(lv)) = ((perm', g'), base', end', a')$: by definition of $updatePcPerm(\cdot)$.

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

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

11.4.1.3.2. $(reg[\text{pc} \mapsto \Phi.\text{reg}(lv)], ms) \to failed$ for any reg, ms: by definition of the evaluation relation.

- $reg(r_{stk}) = ((\text{rwlx}, \text{local}), b_{stk}, e_{stk}, a_{stk})$
- $\text{dom}(ms_{unused}) = [a_{stk} + 1, \cdots, e_{stk}]$
- $\text{dom}(ms_{stk}) = [b_{stk}, \cdots, a_{stk}]$
- $b_{stk} - 1 \leq a_{stk}$

■

Lemma 55 (**scall** works). *If*

- $ms :_n revokeTemp(W)$
- $\text{dom}(ms_f) \cap (\text{dom}(ms_{stk} \uplus ms_{unused} \uplus ms)) = \emptyset$
- *(reg, ms) is looking at* **scall** $r(\overline{r_{arg}}, \overline{r_{priv}})$ *followed by c_{next}*
- *reg points to stack with ms_{stk} used and ms_{unused} unused*

Hyp-Callee If

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

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

Hyp-Cont If

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

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

- *reg' points to stack with ms_{stk} used and ms''_{unused} unused for some ms''_{unused}*

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

Then

- $(n, (reg, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{unused})) \in \mathcal{O}(W)$

■

17. $ms \uplus ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused} :_{n-1} W_1$
Here we apply Lemma 61. By assumption 1. we have $ms :_n \mathit{revokeTemp}(W)$. So it suffices to show

$ms_f \uplus ms_{stk} \uplus ms_{act} \uplus ms'_{unused} :_{n-1} [\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{act} \uplus ms_f), \iota^{pwl}(\text{dom}(ms'_{unused}))]$

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

18. $(n-1, reg'(r_{stk})) \in \mathcal{V}(W_1)$
Use Lemma 57 with 12. and that W_1 has region $\iota^{pwl}(\text{dom}(ms'_{unused})$.

19. $(n-1, c_{ret}) \in \mathcal{V}(W_1)$
To this end let

19.1. $n' < n-1$

19.2. $W_2 \sqsupseteq^{pub} W_1$

be given and show

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

To this assume

19.3. $n'' \leq n'$

19.4. $(n'', reg_2) \in \mathcal{R}(W_2)$

19.5. $ms' :_{n''} W_2$

be given and show

$$(n'', (reg_2[\text{pc} \mapsto \mathit{updatePcPerm}(c_{ret})], ms')) \in \mathcal{O}(W_2) \tag{14}$$

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

$$ms' = ms'' \uplus ms_r \uplus ms_{stk} \uplus ms_{act} \uplus ms''_{unused} \uplus ms_f$$

where ms'' is the "permanent" part of memory we get from Lemma 58, ms_r is the part "revoked" of memory from the same lemma that is not otherwise specified, and $\text{dom}(ms'_{unused}) = \text{dom}(ms''_{unused})$. From Lemma 58 we also get

19.6. $ms'' :_{n''} \mathit{revokeTemp}(W_2)$

Assume n'' is large enough to execute the rest of the **scall** instructions. If n'' is not large enough, then 14 is trivial to show. To show 14 apply Lemma 8 again where ms_r is the revoked part. Let ms'_{frame} be given, the execution until just after the **scall** proceeds as follows:

$$(reg_2[\text{pc} \mapsto \mathit{updatePcPerm}(c_{ret})], ms' \uplus ms'_{varframe}) \to_j (reg_3, ms' \uplus ms'_{frame})$$

where

19.7.

$$reg_3(r) = \begin{cases} c_{next} & r = \text{pc} \\ c_{stk} & r = r_{stk} \\ reg(r) & r \in \{\overline{r_{priv}}\} \\ reg_2(r) & \text{otherwise} \end{cases}$$

Let $W' \sqsupseteq^{pub} W$, a, and $n' \leq n$ be given and show

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

Consider each of the three cases for *perm*:

4. $perm = \text{rwlx}$
   In this case $\iota = \iota^{pwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:

   4.1. $(n', (base, end)) \in readCondition(\text{local})(W')$
   Follows from Lemma 33, Lemma 35, and assumption 2..

   4.2. $(n', (base, end)) \in writeCondition(\iota^{pwl}, \text{local})(W')$
   Follows from Lemma 36, Lemma 35, and assumption 3..

5. $perm = \text{rx}$
   In this case $\iota = \iota^{nwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:

   5.1. $(n', (base, end)) \in readCondition(\text{local})(W')$
   Follows from Lemma 33, Lemma 35, and assumption 2..

   5.2. $(n', (base, end)) \in writeCondition(\iota^{nwl}, \text{local})(W')$
   Follows from Lemma 32, Lemma 36, Lemma 35, and assumption 3..

6. $perm = \text{rwx}$
   In this case $\iota = \iota^{nwl}$. If we use the FTLR (Theorem 2), then we are done. It suffices to show:

   6.1. $(n', (base, end)) \in readCondition(\text{local})(W')$
   Follows from Lemma 33, Lemma 35, and assumption 2..

□

Lemma 57 (Stack capability in value relation). *If*

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

then

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

■

Proof of Lemma 57. Say

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

Show

1. $(n, (base, end)) \in readCondition(\text{local})(W)$:
   Amounts to
   $$\iota^{pwl}(\text{dom}(ms)) \overset{n}{\subsetneq} \iota^{pwl}_{base,end}$$
   which is true as they are even equal.

✓ **Lemma 60** (Revoke temporary memory with stack).

$$\begin{aligned}&\forall n, ms, W, reg, r_{stk}, g, base, end, a.\\ &\quad ms :_n W \land (n, reg) \in \mathcal{R}(W) \land\\ &\quad reg(r_{stk}) = ((\mathrm{rwlx}, g), base, end, a) \land b \leq e\\ &\qquad \exists ms', ms_r.\\ &\qquad\quad ms :_n revokeTemp(W) \land ms = ms' \uplus ms_r\end{aligned}$$

■

lem:revoke-temp-stack. □

✓ **Lemma 61** (Disjoint memory satisfaction).

$$\begin{aligned}&\forall n. \forall ms, ms', ms''. \forall W, W', W''.\\ &\quad ms'' = ms \uplus ms' \land W'' = W \uplus W' \land ms :_n W \land ms' :_n W' \Rightarrow\\ &\quad \underset{ms''}{\cancel{ms \uplus ms'}} :_n \underset{W''}{\cancel{W \uplus W'}}\end{aligned}$$

■

Proof of Lemma 61. □

✓ **Lemma 62** (Memory satisfaction and static regions).

$$ms :_n [i \mapsto \iota^{sta}(v, ms)]$$

■

Proof of Lemma 62. Follows from def. □

✓ **Lemma 63** (Data only memory and standard regions). *If*

- $\forall a \in \mathrm{dom}(ms).\, ms(a) \in \mathbb{N}$
- $\iota \in \{\iota^{pwl}, \iota^{nwl}, \iota^{nwl,p}\}$

then

$$ms :_n [i \mapsto \iota^{[illegible]}(\mathrm{dom}(ms))]$$

■

Proof of Lemma 63. Trivial, as data always in $\mathcal{V}$. □

5.4.13 Future worlds

Lemma 64 (World public future world of revoked world).

$$\forall W.\, revokeTemp(W) \sqsupseteq^{pub} W$$

■

Proof of Lemma 64. For all r where $W(r) = (\mathrm{temp}, s, \phi_{pub}, \phi, H)$, we have $revokeTemp(W) =$ revoked. By the public future region relation we have

$$W(r) = (\mathrm{temp}, s, \phi_{pub}, \phi, H) \sqsupseteq^{pub} revokeTemp(W)(r) = \mathrm{revoked}$$

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

5.4.14 Value relation

Lemma 70 (Value relation downwards closed).

$$n' \leq n \wedge (n, w) \in \mathcal{V}(W) \Rightarrow (n', w) \in \mathcal{V}(W)$$

■

Proof. By definition of $\mathcal{V}(W)$ using Lemma 35, 38, 41 and 44. □

Lemma 71 (Register relation downwards closed).

$$n' \leq n \wedge (n, w) \in \mathcal{R}(W) \Rightarrow (n', w) \in \mathcal{R}(W)$$

■

Proof. By definition of $\mathcal{R}(W)$ using Lemma ??. □

Lemma 72 (Value relation monotone wrt $\sqsupseteq^{pub}$).

$$W' \sqsupseteq^{pub} W \wedge (n, w) \in \mathcal{V}(W) \Rightarrow (n, w) \in \mathcal{V}(W')$$

■

Proof of lemma 72. Follows from Lemma 33, Lemma 36, Lemma 39, and Lemma 42. □

Lemma 73. *If*

$$(n, w) \in \mathcal{V}(\mathit{revokeTemp}(W))$$

then

$$(n, w) \in \mathcal{V}(W)$$

■

Proof of Lemma 73. Follows from Lemma 28, Lemma 29, Lemma 30, and Lemma 31. □

Lemma 74 (Global capabilities monotone wrt $\sqsupseteq^{priv}$).

$$\begin{aligned}&\forall n, perm, base, end, a, W, W'.\\ &\quad (n, ((perm, \text{global}), base, end, a)) \in \mathcal{V}(W) \wedge W' \sqsupseteq^{priv} W\\ &\qquad \Rightarrow (n, ((perm, \text{global}), base, end, a)) \in \mathcal{V}(W')\end{aligned}$$

■

Proof of Lemma 74. Assume

1. $perm \notin \{\text{rwl}, \text{rwlx}\}$
2. $W' \sqsupseteq^{priv} W$
3. $(n, ((perm, \text{global}), base, end, a)) \in \mathcal{V}(W)$

and show

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

to this end consider the possible cases of *perm* and show that each of the necessary conditions hold:

Only stack is store-local? A critical assumption is that adversary code has no way to *store* local capabilities except on the stack. The reason that it is fine to store local capabilities on the stack is that the adversary only has a *local* capability to the stack and cannot usefully store that capability anywhere. However, this means that we need to rely on the runtime system of our programming language to be careful when handing out store-local capabilities: only the libc startup code should initialise the stack as store-local and malloc should *not* produce them. This basically means that the libc initialisation code (or whatever component produces the initial stack pointer) is part of our TCB.

Requirement for clearing the stack Imagine the following trusted C function:

```
void myfunction(){
  advfunction1();
  advfunction2();
}
```

where advfunction1() and advfunction2() are adversary functions. In the standard C treatment of the stack, advfunction2() would get the same stack pointer as advfunction1(). This is supposed to be safe since advfunction1() cannot have kept capabilities for the stack after its execution. But what if we require that the two functions have no way of communicating with each other? Concretely, advfunction1() has access to some secrets that must not be leaked to advfunction2(). How can we prevent advfunction1() from storing the secret somewhere on the stack and relying on advfunction2() from receiving the same stack pointer where it can read the secret? The most obvious solution seems to be that we should fully clear the stack (overwrite it with zeros) after the return of any adversary function, but this could cause an important overhead. Perhaps the processor should accommodate this with a special instruction that can zero the entire array that a capability points to?

What do return pointers look like? An important question is what return pointers look like? Since we want to protect the caller from the callee, it's important that the return pointer is opaque, i.e. an entry pointer. The entry pointer will point to a closure that contains the next instruction to execute, as well as the previous stack pointer. But since stack pointers are local, this means that the return pointer closure should be stored in a region of memory for which we have store-local permission, i.e. on the stack. This means we need the following in our calling convention: before invoking a function, we push the stack pointer and the instruction pointer after invocation on the stack, we construct a return pointer by copying the stack pointer, limiting it to these two entries and making it an entry pointer. Then we shrink the stack pointer to the unused part of the stack and jump.

Only one-way protection in higher-order settings? Another important point is that, in a sense, local capabilities provide only one-way protection: the caller is protected from the callee but not vice-versa. Concretely: when invoking a function with some arguments marked as local, the caller is guaranteed that the callee will not have been able to store the capabilities anywhere (except perhaps on the stack, see above). However, the callee seems to have more limited guarantees: Particularly, the caller may have kept its own stack capability and this stack capability may (and typically will) also cover the part of the stack that is "owned" by the callee. In this sense, the guarantees are more limited than in a linear language.

So what does this mean? In a first-order language, this is all fine, but what if we are in a higher-order language. Imagine the following (in some ML-like language):

7 Related reading

This is a list of related work that might be interesting to read in the context of this project.

7.1 Capability machines

7.1.1 M-Machine

More than 20 years ago, Carter et al. [1994] have described the use of capabilities in the M-Machine. They do seem to have a reference for the instruction set after all [Dally et al., 1995]; it seems like the server was just temporarily down when we were looking for this the first time...

7.1.2 CHERI

The CHERI processor is a much more recent capability machine, described by Woodruff et al. [2014], Watson et al. [2015].

Another result of this project is also CheriBSD: an adaptation of FreeBSD to the CHERI processor.[5] It is not separately described in a published paper, but mentioned in the papers cited above and in some tech reports (see url). This work includes a pure-capability ABI that could provide some interesting examples.

The CHERI team also has a webpage with all of their CHERI-related publications (including TRs and such)[6].

7.2 Logical Relations

Some papers on logical relations that are relevant for this work are the following:

Hur and Dreyer [2011] describe a logical relation between ML and a (standard) assembly language for expressing compiler correctness. Relevant because they target an assembly language, and they use biorthogonality.

Dreyer et al. [2010] describe a logical relation for a ML-like language and use public/private transitions to reason about well-bracketed control flow. Relevant because we are considering to cover an example of enforcing well-bracketed control flow in a capability machine.

Devriese et al. [2016] describe a logical relation for a JavaScript-like language with object capabilities. Relevant because it treats object capabilities, albeit in a JavaScript-like lambda calculus. It also deals with an untyped language, using a semantic unitype.

References

Lars Birkedal and Aleš Bizjak. A Taste of Categorical Logic — tutorial notes. `http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf`, 2014.

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102 – 4122, 2010. ISSN 0304-3975.

A. Bizjak. Some theorems about mutually recursive domain equations in the category of preordered COFEs. Unpublished note. Available at `http://cs.au.dk/~abizjak/documents/notes/mutually-recursive-domain-eq.pdf`, 2017.

[5] `http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html`

[6] `http://www.cl.cam.ac.uk/research/security/ctsrd/cheri/`