# STK Tokens: Enforcing Well-bracketed Control Flow and Stack Encapsulation using Linear Capabilities

Technical Report with Proves and Details

July 18, 2018

## Contents

1. **The two capability machines**
   - 1.1 Domains .......................................................... 2
   - 1.1.1 Useful definitions ........................................... 2
   - 1.2 Syntax ............................................................ 3
   - 1.3 Permissions ..................................................... 3
   - 1.4 Operational Semantics ....................................... 4
     - 1.4.1 Notes ......................................................... 4
     - 1.4.2 Helpful functions, sets, and conventions .......... 4
     - 1.4.3 Step relations ............................................ 5
     - 1.4.4 Instruction Interpretation ............................. 8
   - 1.5 Components .................................................... 14
   - 1.6 Linking .......................................................... 16
   - 1.7 Programs, contexts, initial execution configuration .. 16

2. **Compiler** .......................................................... 16

3. **Logical Relation** ................................................ 17
   - 3.1 Worlds ........................................................... 17
   - 3.2 Future world ................................................... 19
   - 3.3 Memory satisfaction ........................................... 20
   - 3.4 Relation ........................................................ 21
   - 3.5 Permission based conditions ............................... 22
   - 3.6 Standard regions .............................................. 23
   - 3.7 Reasonable Components ...................................... 24
   - 3.8 Fundamental Theorem of Logical Relations ........... 29
   - 3.9 Related components ......................................... 30
   - 3.10 FTLR for components ....................................... 32
   - 3.11 Related execution configurations ........................ 40

4. **Full Abstraction** ................................................. 43

5. **Lemmas** ............................................................ 45

6. **Proofs** .............................................................. 46
   - 6.1 Lemmas .......................................................... 46
   - 6.2 FTLR proof ...................................................... 57

7. **Notes** ............................................................... 82
   - 7.1 Notes on linear capabilities ................................. 82
   - 7.2 Calling convention design decisions ...................... 82
     - 7.2.1 Returning the full stack ............................... 82
     - 7.2.2 Restriction on stack allocation ....................... 84
1 The two capability machines

1.1 Domains

\[
\begin{align*}
\text{a, base} & \in \text{Addr} \iff N \\
\sigma_{\text{base}}, \sigma & \in \text{Seal} \iff N \\
\text{w} & \in \text{Word} \iff \mathbb{Z} \cup \text{Cap} \\
\text{perm} & \in \text{Perm} ::= \ldots \\
\text{l} & ::= \text{linear} | \text{normal} \\
\text{end} & \in \text{Addr} \cup \{\infty\} \\
\sigma_{\text{end}} & \in \text{Seal} \cup \{\infty\} \\
\text{sc} & \in \text{SealableCap} ::= ((\text{perm}, \text{l}), \text{base}, \text{end}, \text{a}) \\
\text{c} & \in \text{Cap} ::= \text{sc} | \text{sealed}(\sigma, \text{sc}) \\
\text{r} & \in \text{RegisterName} ::= \text{pc} | \text{r}_{\text{retdata}} | \text{r}_{\text{retcode}} | \text{r}_{\text{stk}} | r_{\text{data}} | r_{11} | r_{12} | \ldots \\
\text{RegisterFile} & \iff \text{RegisterName} \rightarrow \text{Word} \\
\text{Memory} & \iff \text{Addr} \rightarrow \text{Word} \\
\text{MemorySegment} & \iff \text{Addr} \rightarrow \text{Word} \\
\text{frame} & \in \text{StackFrame} \iff \text{Addr} \times \text{MemorySegment} \\
\text{stk} & \in \text{Stack} \iff \text{StackFrame}^* \\
\Phi & \in \text{ExecConf} \iff \text{Memory} \times \text{RegisterFile} \times \text{Stack} \times \text{MemorySegment} \\
\text{Conf} & \iff \text{ExecConf} \cup \{\text{failed}\} \cup \{\text{halted}\}
\end{align*}
\]

The target language domains are all the non blue parts in the above. The source language domains are the black and blue parts in the above. Further:

- l defines domain Linear
- sc defines domain SealableCap
- c defines domain Cap
- r defines the finite set RegisterName.
- Perm is defined as the set of permissions in Figure [1]

In the source language, Stack is a call stack that contains the data for each call. The call stack consists of a number of StackFrame’s that contains 1) the old pc and 2) caller’s private stack.

In both languages, the base address of the stack is known (the stack grows downwards in memory, so the base address marks the end of the stack). In the target language, this will be the base address of some linear capability. The base address can be any address on the machine, but it will have to remain the same during all of execution. We write this constant as 

\[
\text{stk}_\text{base}
\]

1.1.1 Useful definitions

Definition 1. For a capability \(c = (\_, \_, b, e, \_\) we say it has range \([b, e]\) and we define

\[
\text{range}(c) = [b, e]
\]

Similarly for seals and stack pointers:

\[
\text{range}(\text{seal}(\sigma_{\text{b}}, \sigma_{\text{e}}, \_)) = [\sigma_{\text{b}}, \sigma_{\text{e}}]
\]

and

\[
\text{range}(\text{stack-ptr}(\_, b, e, \_)) = [b, e]
\]
1.2 Syntax

The target machine is a simple capability machine with memory capabilities and sealed capabilities[1](inspired by CHERI). The syntax of the instructions of the target machine is defined as follows:

\[
\begin{align*}
\mathit{n} & \in \mathbb{Z} \\
\mathit{r} & \in \text{RegisterName} \\
\mathit{rn} & ::= \mathit{r} \mid \mathit{n} \\
\mathit{i} & ::= \mathit{fail} \mid \mathit{halt} \mid \mathit{jmp} \mathit{r} \mid \mathit{jnz} \mathit{r} \mathit{rn} \mid \mathit{gettype} \mathit{r} \mathit{rn} \mid \mathit{geta} \mathit{r} \mathit{r} \mathit{r} \mathit{n} \mid \mathit{getb} \mathit{r} \mathit{r} \mathit{n} \mid \mathit{getl} \mathit{r} \mathit{rn} \mid \mathit{getp} \mathit{r} \mathit{r} \mathit{n} \mid \\
& \quad \mathit{load} \mathit{r} \mathit{rr} \mid \mathit{cca} \mathit{r} \mathit{rn} \mid \mathit{restrict} \mathit{r} \mathit{rn} \mid \mathit{lt} \mathit{r} \mathit{rn} \mathit{rn} \mid \\
& \quad \mathit{plus} \mathit{r} \mathit{rn} \mathit{rn} \mid \mathit{minus} \mathit{r} \mathit{rn} \mathit{rn} \mid \mathit{seta} \mathit{2b} \mathit{r} \mathit{rn} \mid \mathit{xjmp} \mathit{r} \mathit{rr} \mid \mathit{cseal} \mathit{r} \mathit{r} \mathit{n} \mid \\
& \quad \mathit{split} \mathit{r} \mathit{rr} \mathit{rn} \mid \mathit{splice} \mathit{rr} \mathit{r} \mathit{rr}
\end{align*}
\]

\(\mathit{i}\) defines the set \(\mathit{Instr}\).

The source machine is also a capability machine with memory capabilities and sealed capabilities. Unlike the target machine, the source machine has a built in stack along with special stack and return tokens used in place of the actual capabilities. The syntax of the source machine language is as follows:

\[
\mathit{off}_{\mathit{pc}}, \mathit{off}_{\mathit{\sigma}} \in \mathbb{N} \\
\mathit{i} ::= \mathit{i} \mid \mathit{call}^{\mathit{off}_{\mathit{pc}}, \mathit{off}_{\mathit{\sigma}} \mathit{r} \mathit{r}}
\]

There is one syntactic difference between the source language and the target language, namely the target language has an extra instruction in the \(\mathit{call}^{\mathit{off}_{\mathit{pc}}, \mathit{off}_{\mathit{\sigma}}}\) instruction (the \(o\) refers to the seal it uses, namely the offset in the seals made available by linking).

The source machine allows for variable length instructions which we utilise for \(\mathit{call}^{\mathit{off}_{\mathit{pc}}, \mathit{off}_{\mathit{\sigma}}}\). In other words, when \(\mathit{call}^{\mathit{off}_{\mathit{pc}}, \mathit{off}_{\mathit{\sigma}}}\) is decoded, a series of addresses in memory must contain what corresponds to the call instruction (and be within the range of the \(\mathit{pc}\) capability). This is described in more detail when we present the decoding function.

1.3 Permissions

We assume functions \(\mathit{decodePerm}\) and \(\mathit{encodePerm}\).

1.4 Operational Semantics

The source machine is parameterized with a set of trusted addresses \(T_A\). \(T_A\) are the only addresses from which the \(\mathit{call}\) will be interpreted. The source machine represents a virtual intermediate machine which we use to argue well-bracketedness and local state encapsulation. It is not meant to be the machine that the actual code is executed on. Further, it is the well-bracketedness and local state encapsulation of the compiled code, not the context we are concerned with. On the other hand, the target machine is not parameterized with \(T_A\) as all the instructions on the target machine is available to the adversary.

1.4.1 Notes

Generally:

- Linear capabilities are cleared when they move around in memory.

Source language:

- Variable length instructions that match the length of the compiled instructions

[1] In previous work, we used enter capabilities but for this the complexity introduced by mixing writable and executable memory is difficult to handle.
This is needed for correctness.

It is only used for the call instruction.

Target language:

1.4.2 Helpful functions, sets, and conventions

\[ \pi_b(c) = \begin{cases} b & \text{if } sc = ((\_\_), b, \_\_) \\ b & \text{if } sc = \text{sealed}(\_\_, b) \\ b & \text{if } sc = \text{ret-ptr-data}(b, \_\_) \\ b & \text{if } sc = \text{ret-ptr-code}(b, \_\_) \end{cases} \]

\[ \pi_e(c) = \begin{cases} e & \text{if } sc = ((\_\_), \_\_, e, \_\_) \\ e & \text{if } sc = \text{sealed}(\_\_, e) \\ e & \text{if } sc = \text{ret-ptr-data}(\_\_, e) \\ e & \text{if } sc = \text{ret-ptr-code}(\_\_, e, \_\_) \end{cases} \]

\[ \pi_l(c) = \begin{cases} l & \text{if } sc = ((\_\_), \_\_, e, \_\_) \\ l & \text{if } sc = \text{sealed}(\_\_, e, \_\_) \\ l & \text{if } sc = \text{stack-ptr}(\_\_, \_\_, e) \\ l & \text{if } sc = \text{ret-ptr-data}(\_\_, e) \\ l & \text{if } sc = \text{ret-ptr-code}(\_\_, \_\_, e) \end{cases} \]

\[ \Phi[pc \mapsto ((\text{perm}, l), b, e, a + 1)] \]

We now define

\[ \text{readAllowed} \overset{\Delta}{=} \{\text{RWX, RW, RX, R}\} \]

\[ \text{writeAllowed} \overset{\Delta}{=} \{\text{RWX, RW}\} \]

\[ \text{isLinear}(c) \overset{\Delta}{=} \begin{cases} \top & \text{if } c = ((\_\_\_, \_\_, \_\_, \_\_)) \text{ or } \text{sealed}(\_\_, c) \text{ and } \text{isLinear}(c') \\ \top & \text{if } c = \text{stack-ptr}(\_\_, \_\_, \_\_) \\ \top & \text{if } c = \text{ret-ptr-data}(\_\_, \_\_) \\ \bot & \text{otherwise} \end{cases} \]

\[ \text{nonLinear}(c) \overset{\Delta}{=} \neg \text{isLinear}(c) \]

\[ \text{linearityConstraint}(w) \overset{\Delta}{=} \begin{cases} 0 & \text{if } \text{isLinear}(w) \\ w & \text{otherwise} \end{cases} \]

\[ \text{linearityConstraintPerm}(perm, w) \overset{\Delta}{=} \begin{cases} \text{perm} \in \text{writeAllowed} & \text{if } \text{isLinear}(w) \\ \text{true} & \text{otherwise} \end{cases} \]

\[ \text{executable}(sc) \overset{\Delta}{=} \begin{cases} sc = ((\_\_, \_\_, \_\_, \_\_)) \text{ and } \text{perm} \in \{\text{RWX, RX}\} \\ \text{true} \end{cases} \]

\[ \text{nonExecutable}(sc) \overset{\Delta}{=} \neg \text{executable}(sc) \]

\[ \text{withinBounds}(sc) \overset{\Delta}{=} \begin{cases} b \leq a \leq e & \text{if } sc = ((\_\_, b, e, a)) \text{ or } \text{sealed}(\_\_, b) \text{ and } \text{ret-ptr-data}(\_\_, b) \\ \sigma_b \leq \sigma_a \leq \sigma_e & \text{if } sc = \text{sealed}(\_\_, \sigma, \sigma) \\ \bot & \text{otherwise} \end{cases} \]
nonZero\( (w) \triangleq \begin{cases} \\ \\ w \in \mathbb{Z} \text{ and } w = 0 \\ \top \quad \text{otherwise} \\ \end{cases} \)

For convenience, we introduce the following notation:

\[ \Phi(r) \triangleq \Phi.\text{reg}(r) \]

where \( r \in \text{RegisterName} \).

For \( rn \) which can be a register or an integer, we take

\[ \Phi(rn) = n \]

to mean

either \( n = rn \) or \( n = \Phi.\text{reg}(rn) \) and in either case \( n \in \mathbb{Z} \)

1.4.3 Step relations

Decode and encode functions We assume functions \( \text{decodeInstruction} : \text{Word} \rightarrow \text{Instr} \) and \( \text{encodeInstruction} : \text{Instr} \rightarrow \mathbb{Z} \) where \( \text{Instr} \) is the set of target level instructions.

The \( \text{decodeInstruction} \) function must be surjective and injective for all non-fail instructions. For any \( c \in \text{Cap} \), we have that \( \text{decodeInstruction}(c) = \text{fail} \). The \( \text{encodeInstruction} \) function must be injective. Further \( \text{decodeInstruction} \) must be the left inverse of \( \text{encodeInstruction} \) that is for all \( i \in \text{Instr} \)

\[ \text{decodeInstruction}(\text{encodeInstruction}(i)) = i \]

These functions are used for both the target and source level machine. When we write instructions in places where words are required, we will assume that \( \text{encodeInstruction} \) is implicit.

Step relation The instruction \( \text{call}^{\text{off}}_{pc, \text{off}} \) has length \( \text{call}_\text{len} \) which means that it does not fit in one memory address. In fact, \( \text{call}^{\text{off}}_{pc, \text{off}} \) should be seen as a different way of interpreting a series of instruction rather than an instruction on its own. We therefore introduce \( \text{call}^{\text{off}}_{pc, \text{off}} \) \( r_1 \) \( r_2 \), \ldots, \( \text{call}^{\text{off}}_{pc, \text{off}} \) \( r_1 \) \( r_2 \) as aliases for the instructions that constitute \( \text{call}^{\text{off}}_{pc, \text{off}} r_1 \) \( r_2 \) (see Paragraph 1.4.3 for details). We define the following condition that indicates that \( a \) is the first address of a \( \text{call}^{\text{off}}_{pc, \text{off}} r_1 \) \( r_2 \) instruction in the configuration \( \Phi \):

\[ \text{callCondition}(\Phi, r_1, r_2, \text{off}_{pc}, \text{off}_{\sigma}, a) = \begin{cases} \\ \Phi.\text{mem}(a) = \text{call}^{\text{off}}_{pc, \text{off}} r_1 r_2 \text{ and} \\ : \\ \Phi.\text{mem}(a + \text{call}_\text{len} - 1) = \text{call}^{\text{off}}_{pc, \text{off}} r_1 r_2 \\ \end{cases} \]

We use the following step relation:

\[ \Phi \rightarrow_{T_A, \text{stk} \text{base}} \left[ \text{call}^{\text{off}}_{pc, \text{off}} r_1 r_2 \right] (\Phi) \quad \text{if } \Phi(\text{pc}) = (\text{perm}, b, c, a) \text{ and} \]

\[ \text{callCondition}(\Phi, r_1, r_2, \text{off}_{pc}, \text{off}_{\sigma}, a) \text{ and} \]

\[ [a, a + \text{call}_\text{len} - 1] \subseteq T_A \text{ and} \]

\[ [a, a + \text{call}_\text{len} - 1] \subseteq [b, e] \text{ and} \]

\[ \text{executable}(\Phi(\text{pc})) \]

\[ \Phi \rightarrow_{T_A, \text{stk} \text{base}} \left[ \text{decodeInstruction}(\Phi.\text{mem}(a)) \right] (\Phi) \quad \text{if } \Phi(\text{pc}) = (\text{perm}, b, c, a) \text{ and} \]

\[ \text{!callCondition}(\Phi, r_1, r_2, \text{off}_{pc}, \text{off}_{\sigma}, a) \text{ or} \]

\[ e < a + \text{call}_\text{len} - 1 \text{ and} \]

\[ \text{withinBounds}(\Phi(\text{pc})) \text{ and} \]

\[ \text{executable}(\Phi(\text{pc})) \]

\[ \Phi \rightarrow_{T_A, \text{stk} \text{base}} \text{failed} \quad \text{otherwise} \]

On the source machine, the instruction interpretation also takes \( gc \), the global constants. It does, however, just pass it around and never makes changes to it, so we will leave it implicit.

Terminating computations We use the following notation to write that a configuration \( \Phi \) successfully terminates:

\[ \Phi \uparrow_{T_A, \text{stk} \text{base}} \triangleq \Phi \rightarrow_{T_A, \text{stk} \text{base}} \text{halted} \]

if we just know \( \Phi \) terminates in some number of steps, then we write

\[ \Phi \uparrow_{T_A, \text{stk} \text{base}} \triangleq \exists \iota, \Phi \rightarrow_{T_A, \text{stk} \text{base}} \]

5
Call implementation  This paragraph contains the implementation of \( \text{call}^{\text{off}_{\text{pc}} \text{off}_{\sigma}} \). That is each of the instructions in the implementation corresponds to \( \text{call}^{\text{off}_{\text{pc}} \text{off}_{\sigma}} r_1 r_2 \ldots \text{call}^{\text{off}_{\text{pc}} \text{off}_{\sigma}} r_{\text{len}-1} r_1 r_2 \), respectively.

// push 42 on the stack (so it is non-empty).
move r1 {42}
store rstk r1
cca rstk (-1)
// split the stack at its current address - rstk done.
geta r1 rstk
split rstk r1 retdata rstk r1
// load the seal for the return pointer through the pc capability.
move r1 pc
cca r1 (off pc - 5) // off pc is the offset from pc to the location of the seal for this code.
load r1 r1
cca r1 off_σ // off_σ is the offset for the seal used by this call.
// seal the used stack frame as the data part of the return pointer pair.
cseal r1 retdata r1
// obtain the code part of the return pointer pair and seal it too.
move retcode pc
cca retcode 5 // magic number is offset to return code
cseal retcode r1
// now clear temporary register and jump to the adversary.
move r1 {0}
xjmp r1 r2
// the following is the return code
// check that the stack pointer is the same we handed out.
getb r1 rstk
minus r1 r1 stk_base // stk_base is the stack base constant
move r2 pc
cca r2 {5} // magic number is the offset to fail
jnz r2 r1
cca r1 1 // magic number is the offset to after fail
jmp r2
fail
// join our stored private stack frame with the rest of the stack (this also finishes the stack pointer check).
splice rstk rstk rdata
// pop the magic number 42
cca rstk 1
// clear temporary registers used
move r12 {0}
// continue program after invocation.

The call code does the following:

- Store 42 (it could be anything) to the stack (this ensures the stack is non-empty), and decrement the pointer according to convention.
- Get the current address of the stack pointer and split according to it.
- Retrieve the seal of the program.
- Cca the seal, so the seal to be used is active
- Seal our private part of the stack capability.
- Move the pc out of the pc register, adjust it to point to the first address of the return code, and seal it.
- Clear the temporary register.
- Cross jump to the two specified registers.
- Upon return:
  - get the base of the stack and check that it matches up with the global base of the stack. If not, fail.
  - splice the returned stack pointer and the stack pointer for our private stack.
adjust stack pointer to first empty address (the address with the end address is considered free)

clear the temporary registers. (Note that \( r_1 \) is not cleared with 0 as it already contains 0 after the stk_base check. If it didn’t contain 0, then the execution would have failed.)

For convenience, we will add the convention that the memory update \([\text{mem}.a \mapsto \text{call}_1 \text{of}, \text{off}, r_1 r_2]\) corresponds to

\[
\begin{align*}
\text{mem}.a & \mapsto \text{call}_0 \text{of}, \text{off}, r_1 r_2 \\
\text{mem}.a + 1 & \mapsto \text{call}_1 \text{of}, \text{off}, r_1 r_2 \\
\cdots & \mapsto \text{call}_{\text{call}_\text{len}-1} \text{of}, \text{off}, r_1 r_2
\end{align*}
\]

### 1.4.4 Instruction Interpretation

We have unified the two languages in the below definitions. Everything written in black is common for both source and target language. Everything written in blue is specific to the source language.

**fail and halt**

\[
\begin{align*}
\text{fail} & (\Phi) = \text{failed} \\
\text{halt} & (\Phi) = \text{halted}
\end{align*}
\]

**jmp and jnz**

\[
\begin{align*}
\text{jmp } r & (\Phi) = \begin{cases}
\Phi[\text{reg}.r \mapsto w][\text{reg}.\text{pc} \mapsto \Phi(r)] & w = \text{linearityConstraint}(\Phi(r)) \\
\Phi[\text{reg}.r \mapsto w][\text{reg}.\text{pc} \mapsto \Phi(r)] & \text{ otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{jnz } r \text{ rn} & (\Phi) = \begin{cases}
\Phi[\text{reg}.r \mapsto w][\text{reg}.\text{pc} \mapsto \Phi(r)] & w = \text{linearityConstraint}(\Phi(r)) \text{ and } \text{nonZero}(\Phi(rn)) \\
\Phi[\text{reg}.r \mapsto w][\text{reg}.\text{pc} \mapsto \Phi(r)] & \text{ otherwise}
\end{cases}
\end{align*}
\]

**getype**

In the definitions of the semantics below, we use a function \( \text{encodeType} : \text{Word} \rightarrow Z. \) This is an encoding function for which the specific implementation does not matter. As the words of the two machines differ, we really need two functions which we call \( \text{encodeType}_{\text{src}} \) and \( \text{encodeType}_{\text{trg}}. \) These two functions need to be related in the following way:

- \( \text{encodeType}_{\text{src}}((\cdot, \cdot, \cdot, \cdot, \cdot)), \text{encodeType}_{\text{src}}(\text{seal}(\cdot, \cdot, \cdot)), \text{encodeType}_{\text{src}}(\text{sealed}(\cdot, \cdot, \cdot)), \) and \( \text{encodeType}_{\text{src}}(i) \) where \( i \in Z \) are all distinct.

- For all \( w \in \text{SealableCap} \) (only the words on the target machine), \( \text{encodeType}_{\text{trg}}(w) = \text{encodeType}_{\text{src}}(w). \)

- Finally,

\[
\text{encodeType}_{\text{src}}(\text{stack-ptr}(\cdot, \cdot, \cdot, \cdot, \cdot)) = \text{encodeType}_{\text{trg}}((\cdot, \cdot, \cdot, \cdot, \cdot))
\]

and

\[
\text{encodeType}_{\text{src}}(\text{ret-ptr-data}(\cdot, \cdot, \cdot)) = \text{encodeType}_{\text{trg}}(\text{ret-ptr-code}(\cdot, \cdot, \cdot)) = \text{encodeType}_{\text{trg}}((\cdot, \cdot, \cdot))
\]

In English this means that each type of word is represented by a distinct value and that the tokens on the source machine has the type of the capability they represent on the target machine.

\[
\begin{align*}
\text{getype } r_1 r_2 & (\Phi) = \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto \text{encodeType}_{\text{trg}}(\Phi(r_2))])
\end{align*}
\]

**geta, getb, gete, getp, and getl**

We assume functions to encode and decode permissions as well as a function to encode linearity. The functions are used implicitly when a permission or linearity is used in a place where they need to be a word.

Specifically for the permission function, \( \text{encocePerm} : \text{Perm} \rightarrow Z \) and \( \text{decodePerm} : Z \rightarrow \text{Perm} \) encodes and decodes permissions, respectively. Where \( \text{decodePerm} \) is the left inverse of \( \text{encocePerm} \), \( \text{encocePerm} \) is injective and for all \( \text{perm} \in \text{Perm} \) \( \text{encocePerm}(\text{perm}) \neq -1 \) (as this is used as an error value). \( \text{decodePerm} \) is surjective.
For the linearity encoding, we make similar assumptions: $\text{encoceLin} : \text{Linear} \rightarrow \mathbb{Z}$ encodes linearity. $\text{encoceLin}$ is injective and for all $l \in \text{Linear}$ $\text{encocePerm}(l) \neq -1$ (as this is used as an error value). As a capabilities linearity stays the same, we do not need a decoding function for linearity.

$\text{geta } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto a]) & \Phi(r_2) = (\varepsilon, \varepsilon, a) \\
& \text{or } \Phi(r_2) = \text{seal}(\varepsilon, a) \\
& \text{or } \Phi(r_2) = \text{stack-ptr}(\varepsilon, \varepsilon, a) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto -1]) & \text{otherwise} \end{array} \right.$

$\text{getb } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto b]) & \Phi(r_2) = (\varepsilon, \varepsilon, b) \\
& \text{or } \Phi(r_2) = \text{seal}(b, \varepsilon) \\
& \text{or } \Phi(r_2) = \text{stack-ptr}(b, \varepsilon, \varepsilon) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto -1]) & \text{otherwise} \end{array} \right.$

$\text{gete } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto e]) & \Phi(r_2) = (\varepsilon, \varepsilon, e) \\
& \text{or } \Phi(r_2) = \text{seal}(e, \varepsilon) \\
& \text{or } \Phi(r_2) = \text{stack-ptr}(e, \varepsilon, \varepsilon) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto -1]) & \text{otherwise} \end{array} \right.$

$\text{getp } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto \text{perm}]) & \Phi(r_2) = ((\text{perm}, \varepsilon), \varepsilon, \varepsilon) \\
& \text{or } \Phi(r_2) = \text{stack-ptr}(\text{perm}, \varepsilon, \varepsilon) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto -1]) & \text{otherwise} \end{array} \right.$

$\text{getl } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r_1 \mapsto \text{linear}]) & \text{isLinear}(\Phi(r_2)) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto \text{normal}]) & \text{nonLinear}(\Phi(r_2)) \end{array} \right.$

$\text{move } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc}(\Phi[\text{reg}.r \mapsto r n]) & r \neq \text{pc} \land r n \in \mathbb{Z} \\
\text{updatePc}(\Phi[\text{reg}.r \mapsto w][\text{reg}.r \mapsto \Phi(r n)]) & r \neq \text{pc} \land w = \text{linearityConstraint}(\Phi(r n)) \\
\text{failed} & \text{otherwise} \end{array} \right.$

(Notice that in the case where we are moving a linear capability and $r = r n$ the order of the updates matter.)

$\text{store } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc} \left( \Phi[\text{reg}.r_2 \mapsto \text{w}][\text{mem}.a \mapsto \Phi(r_2)] \right) & \Phi(r_1) = ((\text{perm}, l), b, e, a) \text{ and} \\
& \text{perm} \in \text{writeAllowed} \text{ and} \\
& \text{withinBounds}(\Phi(r_1)) \text{ and} \\
& \text{w} = \text{linearityConstraint}(\Phi(r_2)) \text{ and } r_2 \neq \text{pc} \\
& a \in \text{dom}(\Phi.\text{mem}) \end{array} \right.$

$\text{store } r_1 r_2 \mathcal{K}(\Phi) = \left\{ \begin{array}{ll} \text{updatePc} \left( \Phi[\text{reg}.r_2 \mapsto \text{w}][\text{ms}.a \mapsto \Phi(r_2)] \right) & \Phi(r_1) = \text{stack-ptr}(\text{perm}, b, e, a) \text{ and} \\
& \text{perm} \in \text{writeAllowed} \text{ and} \\
& \text{withinBounds}(\Phi(r_1)) \text{ and} \\
& a \in \text{dom}(\Phi.\text{ms}) \text{ and} \\
& \text{w} = \text{linearityConstraint}(\Phi(r_2)) \text{ and } r_2 \neq \text{pc} \text{ otherwise} \end{array} \right.$
load

\[
\begin{cases}
\text{updatePc} \left( \Phi[\text{mem} \cdot a \rightarrow w_2] \right) & \Phi(r_2) = ((\text{perm}, l), b, e, a) \text{ and } \\
& \text{perm} \in \text{readAllowed and } \\
& \text{withinBounds}(\Phi(r_2)) \text{ and } \\
& w = \Phi.\text{mem}(a) \text{ and } r_1 \neq \text{pc} \text{ and } \\
& w_2 = \text{linearityConstraint}(w) \text{ and } \text{linearityConstraintPerm}(\text{perm}, w) \\
\end{cases}
\]

\[
\begin{cases}
\text{updatePc} \left( \Phi[\text{ms}_{\text{stk}} \cdot a \rightarrow w_2] \right) & \Phi(r_2) = \text{stack-ptr}(\text{perm}, b, e, a) \text{ and } \\
& \text{perm} \in \text{readAllowed and } \\
& \text{withinBounds}(\Phi(r_2)) \text{ and } \\
& a \in \text{dom}(\Phi.\text{ms}_{\text{stk}}) \text{ and } \\
& w = \Phi.\text{ms}_{\text{stk}}(a) \text{ and } r_1 \neq \text{pc} \text{ and } \\
& w_2 = \text{linearityConstraint}(w) \text{ and } \text{linearityConstraintPerm}(\text{perm}, w) \\
\end{cases}
\]

failed

**CCA**

*Change Current Address*

\[
\begin{cases}
\text{updatePc}(\Phi[\text{reg} \cdot r \rightarrow c]) & \Phi(\text{rn}) = n \text{ and } \\
& \Phi(r) = ((\text{perm}, l), b, e, a) \text{ and } \\
& c = ((\text{perm}, l), b, e, a + n) \text{ and } \\
& r \neq \text{pc} \\
\end{cases}
\]

\[
\begin{cases}
\text{updatePc}(\Phi[\text{reg} \cdot r \rightarrow s]) & \Phi(\text{rn}) = n \text{ and } \\
& \Phi(r) = \text{seal}(\sigma_b, \sigma_e, \sigma) \text{ and } \\
& s = \text{seal}(\sigma_b, \sigma_e, \sigma + n) \\
\end{cases}
\]

\[
\begin{cases}
\text{updatePc}(\Phi[\text{reg} \cdot r \rightarrow c]) & \Phi(\text{rn}) = n \text{ and } \\
& \Phi(r) = \text{stack-ptr}(\text{perm}, b, e, a) \text{ and } \\
& c = \text{stack-ptr}(\text{perm}, b, e, a + n) \\
\end{cases}
\]

failed

**Restrict**

This instruction uses the *decodePerm* function.

\[
\begin{cases}
\text{updatePc} \left( \Phi[\text{reg} \cdot r_1 \rightarrow c] \right) & \Phi(r_1) = ((\text{perm}, l), b, e, a) \text{ and } \\
& \Phi(\text{rn}) = n \text{ and } \\
& \text{decodePerm}(n) \subseteq \text{perm and } \\
& c = ((\text{decodePerm}(n), l), b, e, a) \text{ and } \\
& r_1 \neq \text{pc} \\
\end{cases}
\]

\[
\begin{cases}
\text{updatePc} \left( \Phi[\text{reg} \cdot r_1 \rightarrow c] \right) & \Phi(r_1) = \text{stack-ptr}(\text{perm}, b, e, a) \text{ and } \\
& \Phi(\text{rn}) = n \text{ and } \\
& \text{decodePerm}(n) \subseteq \text{perm and } \\
& c = \text{stack-ptr}(\text{decodePerm}(n), b, e, a) \\
\end{cases}
\]

failed

**Lt**

\[
\begin{cases}
\text{updatePc}(\Phi[\text{reg} \cdot r_0 \rightarrow 1]) & \text{if } i \in \{1, 2\} \text{ and } \\
& \Phi(\text{rn}_1) = n_i \text{ and } \\
& n_1 < n_2 \\
\end{cases}
\]

\[
\begin{cases}
\text{updatePc}(\Phi[\text{reg} \cdot r_0 \rightarrow 0]) & \text{if } i \in \{1, 2\} \text{ and } \\
& \Phi(\text{rn}_1) = n_i \text{ and } \\
& n_1 < n_2 \\
\end{cases}
\]

failed

otherwise
plus and minus

\[ [\text{plus } r_0 \; r_{n_1} \; r_{n_2}] (\Phi) = \begin{cases} 
\text{updatePc}(\Phi[\text{reg}.r_0 \mapsto n_1 + n_2]) & \text{if } i \in \{1, 2\} \\
\Phi(r_{n_i}) = n_i & \text{otherwise}
\end{cases} \]

\[ [\text{minus } r_0 \; r_{n_1} \; r_{n_2}] (\Phi) = \begin{cases} 
\text{updatePc}(\Phi[\text{reg}.r_0 \mapsto n_1 - n_2]) & \text{if } i \in \{1, 2\} \\
\Phi(r_{n_i}) = n_i & \text{otherwise}
\end{cases} \]

\text{seta2b}

\[ [\text{seta2b } r_1] (\Phi) = \begin{cases} 
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto c]) & \text{if } r_1 \neq \text{pc} \\
\Phi(r_1) = \text{seal}(\sigma_b, \sigma_c, \ldots) & \text{and} \\
c = \text{seal}(\sigma_b, \sigma_c, \sigma_b) \\
\text{updatePc}(\Phi[\text{reg}.r_1 \mapsto c]) & \text{if } r_1 = \text{stack-ptr}(\text{perm}, b, e, \ldots) \\
c = \text{stack-ptr}(\text{perm}, b, e, b) \\
\text{failed} & \text{otherwise}
\end{cases} \]

\text{xjmp}

\[ [\text{xjmp } r_1 \; r_2] (\Phi) = \begin{cases} 
\Phi' & \Phi(r_1) = \text{sealed}(\sigma_1, c_1) \text{ and } \Phi(r_2) = \text{sealed}(\sigma_2, c_2) \text{ and} \\
\sigma_1 = \sigma_2 & \text{and} \\
w_1 = \text{linearityConstraint}(c_1) \text{ and } \\
w_2 = \text{linearityConstraint}(c_2) \text{ and} \\
\Phi' = \Phi[\text{reg}.r_1 \mapsto w_1][\text{reg}.r_2 \mapsto w_2] \text{ and} \\
\Phi'' = \text{xjumpResult}(c_1, c_2, \Phi') \\
\text{failed} & \text{otherwise}
\end{cases} \]

\text{xjumpResult}(c_1, c_2, \Phi) = \begin{cases} 
\Phi[\text{reg}.pc \mapsto c_1] & c_1 \neq \text{ret-ptr-code}(\ldots) \text{ and} \\
\Phi[\text{reg}.r_{\text{data}} \mapsto c_2] & c_2 \neq \text{ret-ptr-data}(\ldots) \text{ and} \\
&& \text{nonExecutable}(c_2) \\
\Phi'[\text{reg}.pc \mapsto c_{\text{opc}}] & c_1 = \text{ret-ptr-code}(b, e, a) \text{ and} \\
&& \text{ret-ptr-data}(a_{\text{stk}}, e_{\text{stk}, \text{priv}}) \text{ and} \\
\Phi'[\text{reg}.r_{\text{data}} \mapsto 0] & c_2 = \text{ret-ptr-data}(a_{\text{stk}}, e_{\text{stk}, \text{priv}}) \text{ and} \\
\Phi'[\text{reg}.r_{\text{stk}} \mapsto c_{\text{stk}}] & \Phi(r_{\text{stk}}) = \text{stack-ptr}(\text{rw}, \text{stk}_{\text{base}}, e_{\text{stk}, \ldots}) \text{ and} \\
\Phi'[\text{reg}.r_{\text{fr}} \mapsto 0] & \text{stk}_{\text{base}} \leq e_{\text{stk}} \text{ and} \\
\Phi'[\text{reg}.r_{\text{fr}} \mapsto 0] & \Phi = (\text{mem}, \text{reg}, \text{stk}_{\text{frame}} :: \text{stk}_{\text{stk}_{\text{base}}}) \text{ and} \\
&& \text{stk}_{\text{frame}} = (\text{opc}, m_{\text{stk}_{\text{priv}}} \text{ and} \\
&& \text{opc} = a \text{ and} \\
&& c_{\text{opc}} = (\text{RW}, \text{normal}, b, e, \text{opc}) \text{ and} \\
&& \text{dom}(m_{\text{stk}_{\text{priv}}}) = [e_{\text{stk}} + 1, e_{\text{stk}_{\text{priv}}}] \text{ and} \\
&& e_{\text{stk}} + 1 = a_{\text{stk}} \text{ and} \\
&& e_{\text{stk}} = \text{stack-ptr}(\text{rw}, \text{stk}_{\text{base}}, e_{\text{stk}_{\text{priv}}}, a_{\text{stk}}) \text{ and} \\
&& \Phi' = (\text{mem}, \text{reg}, \text{stk}_{\text{stk}_{\text{priv}}}, m_{\text{stk}_{\text{stk}_{\text{base}}}}) \text{ and} \\
\text{failed} & \text{otherwise}
\end{cases} \]

cseal
\[
[c\text{seal } r_1 \ r_2](\Phi) = \begin{cases} 
\text{updatePc} \left( \Phi[\text{reg } r_1 \mapsto sc] \right) & \Phi(r_1) \in \text{SealableCap and} \\
\phi = \text{seal}(\sigma_b, \sigma_e, \sigma) & \sigma_b \leq \sigma \leq \sigma_e \\
sc = \text{sealed}(\sigma, \Phi(r_1)) & \text{otherwise} \\
\text{failed} & \end{cases}
\]

**split and splice**

We would like splice and split to have following properties:

1. No authority amplification - splitting or splicing capabilities should give you no more authority than you already had.

2. Split should be dual to splice in the sense that a split on a capability followed by a splice of the two resulting capabilities should yield the same capability.

3. Take the addresses governed by a linear capability to be a multiset. If this capability is split, then the union of the two multisets of addresses governed by the resulting capabilities should be the same as the first multiset. In other words, splice and split should not break linearity.

Split cannot create “empty capabilities” (a capability that governs no segment of the memory, i.e. a capability where the base address is greater than the end address). We partly do not allow this out of convenience as it makes the implementation of call simpler. We do not need empty capabilities as they have no semantic value in the sense that they allow you to do essentially the same as a piece of data.

\[
[s\text{plit } r_1 \ r_2 \ r_3 \ r_4](\Phi) = \begin{cases} 
\text{updatePc} \left( \Phi[\text{reg } r_3 \mapsto w] \right) & \Phi(r_3) = ((perm, l), b, e, a) \text{ and} \\
\phi = n & \Phi(rn_4) = n \text{ and} \\
b \leq n \text{ and } n < e \text{ and} \\
c_1 = ((perm, l), b, n, a) \text{ and} \\
c_2 = ((perm, l), n + 1, e, a) \text{ and} \\
w = \text{linearityConstraint}(\Phi(r_3)) \text{ and} \\
r_1, r_2, r_3 \neq pc \\
\text{failed} & \end{cases}
\]

Two important points about **splice** related to the calling convention: (1) Splice fails if two capabilities are not adjacent. This means that if a caller tries to use a return pointer with a stack that is not immediately adjacent to the private stack, then it fails. (2) Splice prohibit splicing with an empty capability! This means that a callee cannot return an empty stack (this also means that it is impossible to make a call when all of the stack is used - this may indeed be undesirable, but without this restriction we need to handle other things). Note: because **splice** does not allow empty stacks, it is not “left inverse” to **split** (because of the empty case). Intuitively, it
is weird that a split followed by a splice does not yield the same capability.

\[
\text{updatePc } (\Phi) = \left\{ \begin{array}{l}
\text{updatePc } (\Phi[\text{reg.r}2 \mapsto w_2]) \\
\text{updatePc } (\Phi[\text{reg.r}3 \mapsto w_3]) \\
\text{updatePc } (\Phi[\text{reg.r}1 \mapsto c]) \\
\end{array} \right.
\]

\[
\text{splice } r_1, r_2, r_3}(\Phi) = \left\{ \begin{array}{l}
\text{updatePc } (\Phi[\text{reg.r}2 \mapsto w_2]) \\
\text{updatePc } (\Phi[\text{reg.r}3 \mapsto w_3]) \\
\text{updatePc } (\Phi[\text{reg.r}1 \mapsto c]) \\
\end{array} \right.
\]

\[
\Phi(r_2) = ((\text{perm}, l), b_2, e_2, \_ ) \text{ and } \\
\Phi(r_3) = ((\text{perm}, l), b_3, e_3, a_3) \text{ and } \\
e_2 + 1 = b_3 \text{ and } b_2 \leq e_2 \text{ and } b_3 \leq e_3 \text{ and } \\
c = ((\text{perm}, l), b_2, e_3, a_3) \text{ and } \\
w_2 = \text{linearityConstraint}(\Phi(r_2)) \text{ and } \\
w_3 = \text{linearityConstraint}(\Phi(r_3)) \text{ and } \\
r_1, r_2, r_3 \neq \text{pc}
\]

\[
\Phi(r_2) = \text{seal}(\sigma_{b,2}, \sigma_{e,2}, \_) \text{ and } \\
\Phi(r_3) = \text{seal}(\sigma_{b,3}, \sigma_{e,3}, \sigma) \text{ and } \\
\sigma_{e,2} + 1 = \sigma_{b,3} \text{ and } \sigma_{b,2} \leq \sigma_{e,2} \text{ and } \\
\sigma_{b,3} \leq \sigma_{e,3} \text{ and } \\
c = \text{seal}(\sigma_{b,2}, \sigma_{e,3}, \sigma)
\]

\[
\Phi(r_2) = \text{stack-ptr}(\text{perm}, b_2, e_2, \_) \text{ and } \\
\Phi(r_3) = \text{stack-ptr}(\text{perm}, b_3, e_3, a_3) \text{ and } \\
e_2 + 1 = b_3 \text{ and } b_2 \leq e_2 \text{ and } b_3 \leq e_3 \text{ and } \\
c = \text{stack-ptr}(\text{perm}, b_2, e_3, a_3)
\]

\[
\text{failed}
\]

call

\[
\text{call}^{\text{off_p}, \text{off_s}} r_1, r_2}(\Phi) = \left\{ \begin{array}{l}
\Phi'[\text{reg.r}1 \mapsto w_1] \\
\Phi'[\text{reg.r}2 \mapsto w_2] \\
\Phi'[\text{reg.r}3 \mapsto c_{stk}] \\
\Phi'[\text{regởitecode} \mapsto \text{sealed}(\sigma_{\gamma}, c_{\text{opc}})] \\
\Phi'[\text{reg.retorno} \mapsto \text{sealed}(\sigma_{\gamma}, c_{\text{priv.data}})] \\
\Phi'[\text{reg.r}1 \mapsto 0] \\
\end{array} \right.
\]

\[
\text{failed}
\]

\[
\text{xjumpResult } c_1, c_2, \left\{ \begin{array}{l}
\Phi'[\text{reg.r}1 \mapsto w_1] \\
\Phi'[\text{reg.r}2 \mapsto w_2] \\
\Phi'[\text{reg.r}3 \mapsto c_{stk}] \\
\text{failed}
\end{array} \right.
\]

\[
r_1 \neq r_1 \text{ and } r_2 \neq r_1 \text{ and } \\
\Phi(r_1) = \text{sealed}(\sigma_1, c_1) \text{ and } \\
\Phi(r_2) = \text{sealed}(\sigma_2, c_2) \text{ and } \\
\sigma_1 = \sigma_2 \text{ and } \\
\text{nonExecutable}(c_2) \text{ and } \\
\Phi = (\text{mem}, \text{reg}, \text{stk}, ms_{\text{stk}}) \text{ and } \\
\Phi(r_{\text{stk}}) = \text{stack-ptr}(\text{rw}, b_{\text{stk}}, e_{\text{stk}}, a_{\text{stk}}) \text{ and } \\
b_{\text{stk}} < a_{\text{stk}} \leq e_{\text{stk}} \text{ and } \\
ms_{\text{stk}}, \text{priv} = ms_{\text{stk}} | [a_{\text{stk}}, e_{\text{stk}}] [a_{\text{stk}} \mapsto 42] \text{ and } \\
ms_{\text{stk}}, \text{rest} = ms_{\text{stk}} - ms_{\text{stk}} | [a_{\text{stk}}, e_{\text{stk}}] \text{ and } \\
c_{\text{stk}} = \text{stack-ptr}(\text{rw}, b_{\text{stk}}, a_{\text{stk}} - 1, a_{\text{stk}} - 1) \text{ and } \\
c_{\text{priv.data}} = \text{ret-ptr-data}(a_{\text{stk}}, e_{\text{stk}}) \text{ and } \\
\Phi(\text{pc}) = ((\_ , \_), b, e, a) \text{ and } \\
\text{opc} = a + \text{call.len} \text{ and } \\
c_{\text{opc}} = \text{ret-ptr-code}(b, e, a + \text{call.len}) \text{ and } \\
\text{stk}' = (\text{opc}, ms_{\text{stk}}, \text{priv}) : : \text{stk} \text{ and } \\
\Phi' = (\text{mem}, \text{reg}, \text{stk}' , ms_{\text{stk}}, \text{rest}) \text{ and } \\
\text{mem}(a + \text{off}_p) = \text{sealed}(\sigma_b, \sigma_e, \sigma_a) \text{ and } \\
b < a + \text{off}_p + e \text{ and } \\
\sigma_{\gamma} = \sigma_a + \text{off}_p \text{ and } \\
\sigma_b \leq \sigma_{\gamma} \leq \sigma_e \text{ and } \\
w_1 = \text{linearityConstraint}(\Phi(r_1)) \text{ and } \\
w_2 = \text{linearityConstraint}(\Phi(r_2)) \text{ and } \\
\text{otherwise}
\]

Note: the caller may have split part of the stack pointer off and even pass the fragments split off to the callee in registers. This behavior is in principle fine. Source semantics will define that only the non-split-off part will be encapsulated. The parts that were split off and passed to the adversary are not protected, as expected.

1.5 Components

A component can be either a component with a main (i.e. a program that still needs to be linked with library implementations) or one without (i.e. a library implementation that will be used by other components). It contains code memory, data memory, a list of imported symbols, a list of exported symbols, a list of seals used for producing return capability pairs and a list of seals used for producing closures.

We define a component as follows:

\[
\begin{align*}
s &\in \text{Symbol} \\
\text{import} &::= a \leftarrow s \\
\text{export} &::= s \rightarrow w \\
\text{comp}_0 &::= (m_{\text{code}}, m_{\text{data}}, \overline{\text{import}}, \overline{\text{export}}, \sigma_\text{ret}, \sigma_\text{cloes}, A_{\text{linear}}) \\
\text{comp} &::= \text{comp}_0 \\
&\mid (\text{comp}_0, c_{\text{main}}, c_{\text{main}}, d)
\end{align*}
\]
We define inductively when a component is valid \((T_A \vdash \text{comp})\) by the below inference rules:

\[
\begin{align*}
ms_{\text{code}}(a) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) & \quad [\sigma_b, \sigma_e] = (\sigma_{\text{ret}} \cup \sigma_{\text{clos}}) \\
\sigma_{\text{ret}}, \sigma_{\text{ret,owned}}, \sigma_{\text{clos}}, T_A \vdash \text{comp-code } ms_{\text{code}}, a \quad & (a \cdots a + \text{call_len} - 1) \subseteq T_A \land ms_{\text{code}}([a \cdots a + \text{call_len} - 1]) = \text{call}^{\text{off} \cdot \text{off}_s}_i r_1 r_2 \Rightarrow \\
ms_{\text{code}}(a + \text{off}_s) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \land \sigma_b + \text{off}_s \in \sigma_{\text{ret,owned}}
\end{align*}
\]

\[
\begin{align*}
ms_{\text{code}}(a) \in \mathbb{Z} & \quad ([a \cdots a + \text{call_len} - 1] \subseteq T_A \land ms_{\text{code}}([a \cdots a + \text{call_len} - 1]) = \text{call}^{\text{off}_s \cdot \text{off}}_i r_1 r_2) \Rightarrow \\
\sigma_{\text{ret}}, \sigma_{\text{ret,owned}}, \sigma_{\text{clos}}, T_A \vdash \text{comp-code } ms_{\text{code}}, a
\end{align*}
\]

\[
\begin{align*}
ms_{\text{code}} \text{ has no hidden calls} & \quad \forall \sigma_{\text{ret}} \# \sigma_{\text{clos}} \\
\exists \sigma_{\text{ret}} : \text{dom}(ms_{\text{code}}) \rightarrow P(\text{Seal}), \sigma_{\text{ret}} = \bigcup_{a \in \text{dom}(ms_{\text{code}})} d_\sigma(a) \text{ and } \\
\forall a \in \text{dom}(ms_{\text{code}}), \sigma_{\text{ret}}, d_\sigma(a), \sigma_{\text{clos}}, T_A \vdash \text{comp-code } ms_{\text{code}}, a \\
\exists \sigma, ms_{\text{code}}(a) = \text{seal}(\sigma_b, \sigma_e, \ldots) \land [\sigma_b, \sigma_e] \neq \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{perm} \subseteq \text{rw} & \quad l = \text{linear} \Rightarrow [b, e] \subseteq A_{\text{own}} \quad l = \text{normal} \Rightarrow [b, e] \subseteq A_{\text{non-linear}} \\
A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp-value} ((\text{perm}, l), [b, e], a) \\
A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp-value} \text{ sc}
\end{align*}
\]

\[
\begin{align*}
A_{\text{code}}, A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp-value } \text{sealed}(\sigma, \text{sc}) & \quad \exists \sigma \in \sigma_{\text{clos}} \\
A_{\text{code}}, [b, e] \subseteq A_{\text{code}}, \sigma \in \sigma_{\text{clos}} \\
A_{\text{code}}, A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp-export } \text{s} \mapsto \text{sealed}(\sigma, ((\text{RX, normal}), [b, e], a))
\end{align*}
\]

\[
\begin{align*}
\text{dom}(ms_{\text{code}}) = [b, e] & \quad [b - 1, e + 1] \# \text{dom}(ms_{\text{data}}) \\
\sigma_{\text{pad}} = [b - 1 \mapsto 0] \cup [e + 1 \mapsto 0] \quad \sigma_{\text{clos}, T_A \vdash \text{comp-code } ms_{\text{code}}}
\end{align*}
\]

\[
\begin{align*}
A_{\text{own}} : \text{dom}(ms_{\text{data}}) & \rightarrow P(\text{dom}(ms_{\text{data}})) \\
\text{dom}(ms_{\text{data}}) = A_{\text{non-linear}} \uplus A_{\text{linear}} \\
A_{\text{linear}} = \bigcup_{a \in \text{dom}(ms_{\text{data}})} A_{\text{own}}(a)
\end{align*}
\]

\[
\begin{align*}
\text{export} = \text{import} & \quad \exists \text{import} \subseteq \text{dom}(ms_{\text{data}}) \\
\sigma_{\text{import}} \neq \sigma_{\text{export}} & \quad \forall \text{import} \subseteq \text{dom}(ms_{\text{data}})
\end{align*}
\]

\[
\begin{align*}
A_{\text{code}} \vdash \text{comp} & \quad \text{ms}_{\text{code}} \neq \text{ms}_{\text{data}} \vdash \text{comp}, \text{import}, \text{export}, \sigma_{\text{ret}}, \sigma_{\text{clos}}, A_{\text{linear}}
\end{align*}
\]

\[
\begin{align*}
\text{comp} & \quad \text{ms}_{\text{code}} \neq \text{ms}_{\text{data}} \vdash \text{comp}, \text{import}, \text{export}, \sigma_{\text{ret}}, \sigma_{\text{clos}}, A_{\text{linear}}
\end{align*}
\]

\[
\begin{align*}
T_A \vdash \text{comp} & \quad (\text{ms}_{\text{code}} \neq \text{ms}_{\text{data}}, \text{import}, \text{export}, \sigma_{\text{ret}}, \sigma_{\text{clos}}, A_{\text{linear}})
\end{align*}
\]

where the following definition is used

**Definition 2** (No hidden calls). We say that a memory segment \(ms_{\text{code}}\) has no hidden calls iff

\[
\forall a \in \text{dom}(ms_{\text{code}}), \\
\forall i \in [0, \text{call_len} - 1] \\
ms_{\text{code}}(a + i) = \text{call}^{\text{off}_s \cdot \text{off}_s}_i r_1 r_2 \Rightarrow \\
(\text{dom}(ms_{\text{code}}) \supseteq [a - i, a + \text{call_len} - i - 1] \land ms_{\text{code}}([a - i, a + \text{call_len} - i - 1]) = \text{call}^{\text{off}_s \cdot \text{off}}_i r_1 r_2) \lor \\
\exists j \in [a - i, a + \text{call_len} - i - 1] \cap \text{dom}(ms_{\text{code}}), ms_{\text{code}}(j) \neq \text{call}^{\text{off}_s \cdot \text{off}_s}_j r_1 r_2
\]

\[
\square
\]
1.6 Linking

\[ \begin{align*}
\text{comp}_1 &= (m_{\text{code},1}, m_{\text{data},1}, \text{import}_{1}, \text{export}_{1}, \sigma_{\text{ret},1}, \sigma_{\text{close},1}, A_{\text{linear},1}) \\
\text{comp}_2 &= (m_{\text{code},2}, m_{\text{data},2}, \text{import}_{2}, \text{export}_{2}, \sigma_{\text{ret},2}, \sigma_{\text{close},2}, A_{\text{linear},2}) \\
\text{comp}_3 &= (m_{\text{code},3}, m_{\text{data},3}, \text{import}_{3}, \text{export}_{3}, \sigma_{\text{ret},3}, \sigma_{\text{close},3}, A_{\text{linear},3})
\end{align*} \]

\[ m_{\text{code},3} = m_{\text{code},1} \sqcup m_{\text{code},2} \]
\[ \text{export}_3 = \text{export}_1 \cup \text{export}_2 \]
\[ \sigma_{\text{ret},3} = \sigma_{\text{ret},1} \cup \sigma_{\text{ret},2} \]
\[ \sigma_{\text{close},3} = \sigma_{\text{close},1} \cup \sigma_{\text{close},2} \]
\[ A_{\text{linear},3} = A_{\text{linear},1} \cup A_{\text{linear},2} \]
\[ \text{dom}(m_{\text{code},3}) \# \text{dom}(m_{\text{data},3}) \]

\[ \text{comp}_3 = \text{comp}_1 \bowtie \text{comp}_2 \]

\[ \text{comp}'' = \text{comp}_0 \bowtie \text{comp}' \]

\[ \text{comp}_0'' = \text{comp}_0 \bowtie \text{comp}_0' \]

\[ \text{comp}_0'' = \text{comp}_0 \bowtie \text{comp}_0' \]

1.7 Programs, contexts, initial execution configuration

A program is intuitively a component that is ready to be executed, i.e. it must have an empty import list and a pair of capabilities to be used as main. A context for a given component is any other component that can be linked with it to produce a program.

**Definition 3** (Programs and Contexts). We define a program to be a component \((\text{comp}_0, e_{\text{main},c}, e_{\text{main},d})\) with an empty import list.

A context for a component \(\text{comp}\) is another component \(\text{comp}'\) such that \(\text{comp} \bowtie \text{comp}'\) is a program.

**Definition 4** (Initial execution configuration).

\[ \begin{align*}
&c_{\text{main},c} = \text{sealed}(\sigma_1, c_{\text{main},c}) \\
&c_{\text{main},d} = \text{sealed}(\sigma_2, c_{\text{main},d}) \\
&\sigma_1 = \sigma_2 \\
&\text{reg}(\text{pc}) = c'_{\text{main},c} \\
&\text{reg}(\text{rdata}) = c'_{\text{main},d} \\
&\text{reg}((\text{rstk})) = (\text{bstk}, \text{cstk}) \\
&\text{range}(\text{msstk}) = \{0\} \\
&\text{dom}(\text{msstk}) = [\text{bstk}, \text{cstk}] = \text{dom}(\text{msstk}) \\
&[\text{bstk} - 1, \text{cstk} + 1] \# (\text{dom}(\text{mscode}) \cup \text{dom}(\text{msdata})) \\
&\text{import} = \emptyset
\end{align*} \]

**Definition 5** (Plugging a component into a context). When \(\text{comp}'\) is a context for component \(\text{comp}\) and \(\text{comp} \bowtie \text{comp}'\), then we write \(\text{comp}'[\text{comp}]\) for the execution configuration \(\Phi\).

**Lemma 1.** For components \(\mathcal{C}\) and \(\text{comp}\), if

- \(\emptyset \vdash \mathcal{C}\)
- \(\text{dom}(\text{comp}.m_{\text{code}}) \vdash \text{comp}\)
- \(\mathcal{C}[\text{comp}]\) is defined

Then

\[ \text{dom}(\text{comp}.m_{\text{code}}) \vdash \mathcal{C}\]

**Proof.** Follows by definition.

2 Compiler

The compiler is the identity.

3 Logical Relation

In the following definitions, blue is used to indicate values related to the source machine. This is unlike previous definitions, where blue was used to indicate source language specific parts of definitions.
3.1 Worlds

Theorem 1. There exists a c.o.f.e. Wor and preorder $\sqsupseteq$ such that $(\text{Wor, } \sqsupseteq)$ and there exists an isomorphism $\xi$ such that

$$\xi : \text{Wor} \cong \langle \text{World}_{\text{heap}} \times \text{World}_{\text{private stack}} \times \text{World}_{\text{free stack}} \rangle$$

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

$$\hat{W}' \sqsupseteq \hat{W} \iff \xi(\hat{W}') \sqsupseteq \xi(\hat{W})$$

Where $\text{World}_{\text{private stack}}$, $\text{World}_{\text{heap}}$, and $\text{World}_{\text{free stack}}$ are defined as follows

$$\text{World}_{\text{heap}} = \text{RegionName} \rightarrow (\text{Region}_{\text{spatial}} + \text{Region}_{\text{shared}})$$

and

$$\text{World}_{\text{private stack}} = \text{RegionName} \rightarrow (\text{Region}_{\text{spatial}} \times \text{Addr})$$

and

$$\text{World}_{\text{free stack}} = \text{RegionName} \rightarrow \text{Region}_{\text{spatial}}$$

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

$$\text{Region}_{\text{shared}} = \{\text{pure}\} \times (\text{Wor} \xrightarrow{\text{min, ne}} \text{URel}((\text{MemorySegment})^2)) \times$$

$$\text{Seal} \rightarrow \text{Wor} \xrightarrow{\text{min, ne}} \text{URel}((\text{SealableCap}) \times (\text{SealableCap}))$$

and

$$\text{Region}_{\text{spatial}} = \begin{cases} 
\{\text{spatial}\} \times (\text{Wor} \xrightarrow{\text{min, ne}} \text{URel}((\text{MemorySegment})^2))\| 
\{\text{spatial}_{\text{owned}}\} \times (\text{Wor} \xrightarrow{\text{min, ne}} \text{URel}((\text{MemorySegment})^2))\| 
\{\text{revoked}\} 
\end{cases}$$

where spatial and spatial$_{\text{owned}}$ are regions governing segments of memory addressed by linear capabilities. spatial$_{\text{owned}}$ signifies that this region is addressable. spatial signifies that the region is not owned and can thus not be addressed. At the same time it signifies that if something else addresses it, it is a linear capability. Finally, pure signifies that the region is only addressed by non-linear capabilities. Notice that no region allows for both linear and non-linear capabilities to address it. Notice also that pure regions have an additional component that allows them to claim ownership of part of the address space of seals and impose a relational invariant on everything signed with those seals.

We introduce a bit of notation for projecting out each part of the world:

$$W.\text{heap} = \pi_1(W)$$
$$W.\text{priv} = \pi_2(W)$$
$$W.\text{free} = \pi_3(W)$$

as well as projections for the regions:

$$\langle v, s, \phi_{\text{pub}}, \phi, H \rangle.v = v$$
$$\text{revoked}.v = \text{revoked}$$

We define erasure for worlds as follows:

$$\lfloor (W_{\text{heap}}, W_{\text{priv}}, W_{\text{free}}) \rfloor_S = (\lfloor W_{\text{heap}} \rfloor_S, \lfloor W_{\text{priv}} \rfloor_S, \lfloor W_{\text{free}} \rfloor_S)$$

where erasure for each part of a world is defined as follows:

$$\lfloor W_{\text{heap}} \rfloor_S = \lambda r. \begin{cases} W_{\text{heap}}(r).v \in S \\
\perp \end{cases}$$
$$\lfloor W_{\text{priv}} \rfloor_S = \lambda r. \begin{cases} W_{\text{priv}}(r).\text{region}.v \in S \\
\perp \end{cases}$$
$$\lfloor W_{\text{free}} \rfloor_S = \lambda r. \begin{cases} W_{\text{free}}(r).v \in S \\
\perp \end{cases}$$

The active function takes a world and filters away all the revoked regions, so

$$\text{active}(W) = \lfloor W \rfloor_{\text{spatial,spatial}_{\text{owned}},\text{pure}}$$
Disjoint union of worlds. Joins together two alike worlds with strictly disjoint ownership over spatial_owned-regions. Two worlds can be joined together if all of their three parts agree on the region names and each of their regions can be joined together.

\[ W_1 \oplus W_2 = W \text{ iff } \ \text{dom}(W.\text{heap}) = \text{dom}(W_1.\text{heap}) = \text{dom}(W_2.\text{heap}) \text{ and } \ \text{dom}(W.\text{free}) = \text{dom}(W_1.\text{free}) = \text{dom}(W_2.\text{free}) \text{ and } \ \text{dom}(W.\text{priv}) = \text{dom}(W_1.\text{priv}) = \text{dom}(W_2.\text{priv}) \text{ and } \ \forall r \in \text{dom}(W.\text{heap}), W.\text{heap}(r) = W_1.\text{heap}(r) \oplus W_2.\text{heap}(r) \text{ and } \ \forall r \in \text{dom}(W.\text{free}), W.\text{free}(r) = W_1.\text{free}(r) \oplus W_2.\text{free}(r) \text{ and } \ \forall r \in \text{dom}(W.\text{priv}). \pi_1(W.\text{priv}(r)) = \pi_1(W_1.\text{priv}(r)) \oplus \pi_1(W_2.\text{priv}(r)) \]

\( \oplus \) on regions is defined as follows

\[(\text{pure}, H, H_\sigma) \oplus (\text{pure}, H, H_\sigma) = (\text{pure}, H, H_\sigma) \]

\[(\text{spatial}, H) \oplus (\text{spatial}, H) = (\text{spatial}, H) \]

\[(\text{spatial}_\text{owned}, H) \oplus (\text{spatial}, H) = (\text{spatial}, H) \oplus (\text{spatial}_\text{owned}, H) = (\text{spatial}_\text{owned}, H) \]

and for all other cases \( \oplus \) is undefined. Specifically, \( \oplus \) is not defined when both sides are a spatial_owned-region. It is further not defined if the two sides do not agree on region type or heap or sealed value relations.

**Lemma 2.** \( \oplus \) is associative and commutative. Also, left-hand sides in the commutativity and associativity laws are defined whenever the right-hand sides are defined and vice versa.

**Proof.** Follows easily from the definitions.

**Lemma 3** (\( \oplus \) and future worlds). If \( W'' \sqsupseteq W_1 \oplus W_2 \), then there exist \( W'_1, W'_2 \) such that \( W' = W'_1 \oplus W'_2 \) and \( W'_1 \sqsubseteq W_1 \) and \( W'_2 \sqsubseteq W_2 \).

**Proof.** We define \( W'_1 \) and \( W'_2 \) to have the same regions as \( W' \) with a possibly different visibility. For regions that are present in \( W_1 \) and \( W_2 \), we give them the same visibility in \( W'_1 \) and \( W'_2 \) respectively. For regions that are new in \( W' \), we make them pure or spatial in both \( W'_1 \) and \( W'_2 \) if they are in \( W' \) and we make them spatial_owned in \( W'_1 \) but spatial in \( W'_2 \) if they are spatial in \( W' \). It is then easy to check that the required equations hold.

We also define a second disjoint union operator of worlds:

\[ W_1 \uplus W_2 = W \text{ iff } \ \text{dom}(W.\text{heap}) = \uplus \text{dom}(W_1.\text{heap}) \text{ and } \ \text{dom}(W.\text{free}) = \uplus \text{dom}(W_1.\text{free}) \text{ and } \ \text{dom}(W.\text{priv}) = \uplus \text{dom}(W_1.\text{priv}) \]

The two operators \( \uplus \) and \( \oplus \) are quite different. The difference is most clear in the treatment of pure regions: \( \uplus \) allows both worlds to have the same pure region, while \( \oplus \) forbids this. To understand this different treatment (\( W_1 \uplus W_2 \) and \( W_1 \oplus W_2 \)), you should understand that the two are intended for different usages of worlds. The \( W_1 \oplus W_2 \) operator treats the worlds as specifications of authority: taking the disjoint union of worlds specifying non-exclusive ownership of a block of memory is allowed and produces a new world that also specifies non-exclusive ownership of world. The \( W_1 \uplus W_2 \) operator treats worlds as specifications of memory contents: taking the disjoint union of worlds specifying the presence of the same memory range is not allowed. The latter operator is used in the logical relation for components which specifies (among other things) that the world should specify the presence of the component’s data memory. Linking two components then produces a new component with both components’ data memory. The linked component is then valid in a world that has the combined memory presence specifications, not the combined authority. In other words, \( \uplus \) specifies disjoint authority distribution, while \( \oplus \) specifies disjoint memory allocation.

Note also that this picture is further complicated by our usage of non-authority-carrying spatial regions. They are really only there in a world \( W \) as a shadow copy of a spatial_owned region in another world \( W' \) that \( W \) will be combined with. The shadow copy is used for specifying when a memory satisfies a world: the memory should contain all memory ranges that anyone has authority over, not just the ones whose authority belongs to the memory itself. For example, if a register contains a linear pointer to a range of memory, then the register file will be valid in a world where the corresponding region is spatial_owned, while the memory will be valid in a world with the corresponding region only spatial. However, for the memory to satisfy the world, the block of memory needs to be there, i.e. the memory should contain blocks of memory satisfying every region that is spatial_owned, pure, but also just spatial (because it may be spatial_owned in, for example, the register file’s world).

**Lemma 4.** \( \uplus \) is associative and commutative.

**Proof.** Follows easily from the definitions.
Lemma 5 (Odd distributivity of ⊕ and ⊔).

\[(W_1 ⊕ W_2) ⊔ (W_3 ⊕ W_4) = (W_1 ⊔ W_3) ⊕ (W_2 ⊔ W_4)\]

Also, the left expression is defined iff the right expression is.

Proof. Follows by definition-chasing.

3.2 Future world

The future world relation becomes:

\[\exists m : \text{RegionName} \rightarrow \text{RegionName}, \text{injective.} \text{dom}(W'.i) \supseteq \text{dom}(m_i(W.i)) \land \forall r \in \text{dom}(W.i). W'.i(m_i(r)) \supseteq W.i(r)\]

Future regions allow spatial regions to become revoked. Also: the future region relation allows spatial regions to become spatial_owned, which expresses that our system is affine, rather than linear.

<table>
<thead>
<tr>
<th>revoked ⊒ (spatial, _)</th>
<th>revoked ⊒ revoked</th>
<th>(spatial_owned, H) ⊒ (spatial, H)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(spatial_owned, H) ⊒ (spatial_owned, H)</td>
<td>(spatial, H) ⊒ (spatial, H)</td>
<td></td>
</tr>
</tbody>
</table>

Definition 6 (The pure part of a world). For any world W, we define

\[\text{purePart}(W) \overset{df}{=} (\text{purePart}(W.\text{heap}), \text{purePart}(W.\text{priv}), \text{purePart}(W.\text{free}))\]

<table>
<thead>
<tr>
<th>purePart(W_{heap})</th>
<th>purePart(W_{priv})</th>
<th>purePart(W_{free})</th>
</tr>
</thead>
<tbody>
<tr>
<td>[W_{\text{heap}}(r)] if [W_{\text{heap}}(r) = (\text{pure, sm})]</td>
<td>[((\text{spatial, sm}), \text{opc})] if [W_{\text{priv}}(r) = ((\text{spatial, sm}), \text{opc})]</td>
<td>[W_{\text{free}}(r)] if [W_{\text{free}}(r) = (\text{pure, sm})]</td>
</tr>
<tr>
<td>(spatial, sm) if [W_{\text{heap}}(r) = (\text{spatial, sm})]</td>
<td>(spatial, sm) if [W_{\text{priv}}(r) = ((\text{spatial, sm}), \text{opc})]</td>
<td>(spatial, sm) if [W_{\text{free}}(r) = (\text{spatial, sm})]</td>
</tr>
<tr>
<td>revoked if [W_{\text{heap}}(r) = \text{revoked}]</td>
<td>revoked, opc if [W_{\text{priv}}(r) = (\text{revoked, opc})]</td>
<td>revoked if [W_{\text{free}}(r) = \text{revoked}]</td>
</tr>
</tbody>
</table>

Lemma 6 (purePart is duplicable). For all W, we have that \[W = W ⊕ \text{purePart}(W)\].

Proof. Follows from the definition of purePart and ⊕.

Lemma 7 (purePart is idempotent). For all W, we have that \[\text{purePart}(W) = \text{purePart}(\text{purePart}(W))\] and \[\text{purePart}(W) = \text{purePart}(W) ⊕ \text{purePart}(W)\].

Proof. The first part follows easily from the definition. The second statement follows from the first, together with Lemma 6.

Lemma 8 (purePart respects ⊕). For all \[W_1, W_2\], we have that \[\text{purePart}(W_1 ⊕ W_2) = \text{purePart}(W_1) ⊕ \text{purePart}(W_2) = \text{purePart}(W_1) = \text{purePart}(W_2)\]. Also, all worlds in the equations above are defined when \[W_1 ⊕ W_2\] is defined (but not necessarily vice versa).

Proof. Follows from the definition of purePart and ⊕.

Lemma 9 (purePart is monotone). For all \[W' \supseteq W\], we have that \[\text{purePart}(W') \supseteq \text{purePart}(W)\].

Proof. Follows easily from the definition of purePart and ⊇.

Lemma 10 (Increasing authority is the future). For all \[W_1, W_2\], we have that: \[W_1 ⊕ W_2 \supseteq W_1\]

Proof. Follows easily from the definitions.
Lemma 11 (Adding memory is the future). For all $W_1, W_2$, we have that: $W_1 \uplus W_2 \supseteq W_1$

Proof. Follows easily from the definitions.

Lemma 12 (Purity is a thing of the past). For all $W$, we have that $W \sqsupseteq \text{purePart}(W)$.


Lemma 13 (Partial authority is better than nothing). If $W = W_1 \uplus W_2$, then $W_1 \sqsupseteq \text{purePart}(W)$.

Proof. Follows easily from the definitions.

3.3 Memory satisfaction

Memory satisfaction for new worlds:

\[
\begin{align*}
\text{Lemma 13} & \quad \text{Lemma 12} \\
\text{Lemma 11} & \quad \text{Lemma 10}
\end{align*}
\]

- For all $W_1, W_2$, we have that: $W_1 \uplus W_2 \supseteq W_1$
- For all $W$, we have that $W \sqsupseteq \text{purePart}(W)$.
- If $W = W_1 \uplus W_2$, then $W_1 \sqsupseteq \text{purePart}(W)$.

\[
\begin{align*}
\text{Proof.} & \quad \text{Follows easily from the definitions.}
\end{align*}
\]

\[
\begin{align*}
\text{Lemma 11} \quad & \text{Lemma 12} \\
\text{Lemma 13} \quad & \text{Lemma 10}
\end{align*}
\]

\[
\begin{align*}
\text{Lemma 11} & \quad \text{Lemma 12} \\
\text{Lemma 13} & \quad \text{Lemma 10}
\end{align*}
\]

\[
\begin{align*}
\text{Proof.} & \quad \text{Follows easily from the definitions.}
\end{align*}
\]
Lemma 14 (Combined independent heap memory satisfies disjoint world). If \((n, (σ_1, ms_{T,1})) ∈ H(W_1, heap)(W)\) and \((n, (σ_2, ms_{T,2})) ∈ H(W_2, heap)(W)\), then \((n, (σ_1 ⊕ σ_2, ms_{T,1} ⊕ ms_{T,2})) ∈ H(W_1 ⊕ W_2, heap)(W)\).

Proof. Unfolding the definitions, it’s easy to construct the memory and seal partitions \(R_{ms,3}\) and \(R_{sealed,3}\) and \(R_{W,3}\) from the corresponding partitions of the separate memories, seals and worlds.

3.4 Relation

Two expression relations: one for sealed code-data pairs being jumped to and one for capabilities being jumped to in the regular way. The argument for having one relation relate pairs of pairs of capabilities and the other relate pairs of capabilities is that that is how xjump and regular jumps work: xjump takes pairs while regular jumps take single capabilities.
We write $R_{\text{untrusted}}^{\square,gc}(W)$ to mean $R_{\text{untrusted}}^{\square,gc}(\emptyset)(W)$. That is, if we do not need to exclude extra registers, then we simply omit that argument.

$$\mathcal{V}_{\text{untrusted}}^{\square,gc}(W) = \{(n, (i, i)) \mid i \in \mathbb{Z}\} \cup$$

$$\begin{cases} \{( \text{sealed}(\sigma, sc_{\text{S}}), \text{sealed}(\sigma, sc_{\text{T}})) \} & \text{ if isLinear}(sc_{\text{S}}) \land \exists r \in \text{dom}(W.\text{heap}), \sigma, \sigma_{\text{ret}}, \sigma_{\text{clos}} \in \text{ms} \land \text{readcondition}^{\square,gc}(W, r) \land \\
\exists r \in \text{dom}(W.\text{heap}), \sigma, \sigma_{\text{ret}}, \sigma_{\text{clos}} \in \text{ms} \land \text{readcondition}^{\square,gc}(W, r) \land \\
\end{cases}$$

$$\mathcal{V}_{\text{trusted}}^{\square,gc}(W) = \mathcal{V}_{\text{untrusted}}^{\square,gc}(W) \cup$$

$$\begin{cases} \{( \text{sealed}(\sigma_{b}, \sigma), \text{sealed}(\sigma_{b}, \sigma_{e})) \} & \text{ if notLinear}(sc_{\text{S}}) \Rightarrow \\
\{( \text{sealed}(\sigma_{b}, \sigma), \text{sealed}(\sigma_{b}, \sigma_{e})) \} & \text{ if notLinear}(sc_{\text{S}}) \Rightarrow \\
\{( \text{stack-ptr}(\text{perm}, b, e, a), (\text{perm}, \text{linear}, b, e, a)) \} & \text{ if stackWriteCondition}^{\square,gc}(W) \land \\
\{( \text{stack-ptr}(\text{perm}, b, e, a), (\text{perm}, \text{linear}, b, e, a)) \} & \text{ if stackWriteCondition}^{\square,gc}(W) \land \\
\end{cases}$$

$$\begin{cases} \{( \text{stack-ptr}(\text{perm}, b, e, a), (\text{perm}, \text{linear}, b, e, a)) \} & \text{ if stackWriteCondition}^{\square,gc}(W) \land \\
\end{cases}$$

**Lemma 15** (Untrusted is trusted).

- $R_{\text{trusted}}^{\square,gc}(W) \supseteq R_{\text{untrusted}}^{\square,gc}(W)$

**Proof.** Follows easily by definition.

Note: the case for sub-RX capabilities in the trusted value relation allows for pointers to trusted code blocks. Such pointers will not satisfy the read condition, which requires the standard region, which is defined in terms of the untrusted value relation. Trusted code blocks contain trusted seal capabilities which do not satisfy the untrusted code relation. An alternative might be to introduce a trust parameter to the read condition and standard region and merge the case with the regular case for rx capabilities.

### 3.5 Permission based conditions

addressable(l, W) = \begin{align*}
& \{ r \mid W(r) = (\text{pure}, .) \} & \text{ if } l = \text{normal} \\
& \{ r \mid W(r) = (\text{spatial, owned}, .) \} & \text{ otherwise (i.e. } l = \text{linear})
\end{align*}

readCondition^{\square,gc}(l, W) = \begin{align*}
& \{ n, A \mid \exists S \subseteq \text{addressable}(l, W) \land \\
& \exists R : S \to P(N) \land \\
& (l = \text{linear} \Rightarrow \forall r, |R(r)| = 1) \land \\
& \forall r \in S. \text{W.heap}(r).H \subseteq \text{std, p}^{\square,gc} H \}
\end{align*}
We say that a region \( \iota \) is address stratified iff

\[
\forall n, ms_S, ms_T, ms'_S, ms'_T, s, W.
\]

\[
(n, (ms_S, ms_T)), (n, (ms'_S, ms'_T)) \in H W \land
\]

\[
\text{dom}(ms_S) = \text{dom}(ms_T) = \text{dom}(ms'_S) = \text{dom}(ms'_T) \Rightarrow
\]

\[
\forall a \in \text{dom}(ms_S). (n, (ms_S[a \mapsto ms'_S(a)], ms_T[a \mapsto ms'_T(a)])) \in H W
\]

\[\blacksquare\]

where \( \iota_A \) is a standard region defined in Section 3.6.

\[
\text{stackReadCondition}^{\sqcap,a}(W) = \left\{ (n, A) \mid \exists S \subseteq \text{addressable}(l, W.\text{heap}).
\right. \]

\[
\exists R : S \rightarrow P(N).
\]

\[
\forall r \in S. |R(r)| = 1 \land
\]

\[
\forall r \in S. W.\text{heap}(r).H^n \subseteq R(r).gc.H
\]

where \( \iota_A \) is a standard region defined in Section 3.6.

\[
\text{readXCondition}^{\sqcap,a}(W) = \left\{ (n, A) \mid \exists r \in \text{addressable}(normal, W.\text{heap}).
\right. \]

\[
W.\text{heap}(r) \subseteq l_{code,\|} \land
\]

\[
\text{dom}(code) \supseteq A
\]

\[\blacksquare\]

3.6 Standard regions

Standard region:

\[
\iota_{A,gc}^{\text{std},v} \overset{\text{def}}{=} (v, H_A^{\text{std}} gc), v \in \{s, so\}
\]

for readability, we use so short for spatial\_owned, s short for spatial, and p as short for pure.

\[
\iota_{A,gc}^{\text{std},p} \overset{\text{def}}{=} (p, H_A^{\text{std}} gc, \lambda_\_ \emptyset)
\]

where \( H_A^{\text{std}} \) is defined as follows:

\[
H_A^{\text{std}} gc W \overset{\text{def}}{=} \left\{ (n, ms_S, ms_T) \mid \exists S : A \rightarrow \text{World}. \xi(W) = \oplus a \in A S(a) \land
\right.

\[
\forall a \in A. (n, (ms_S(a), ms_T(a))) \in V^{untrusted}(S(a))
\]

\[\blacksquare\]
\( H_{\text{sta}_{\mathcal{T}_s}, \text{ms}_{\mathcal{T}_s}}(\text{gc}) = \{ (n, (\text{ms}_s, \text{ms}_T)) \mid \exists S : \text{dom}(\text{ms}) \to \text{World}. \xi(\mathcal{W}) = \bigoplus_{a \in \text{dom}(\text{ms})} S(a) \wedge \forall a \in \text{dom}(\text{ms}). (n, (\text{ms}_s(a), \text{ms}_T(a))) \in V_{\text{untrusted}}. S(a) \} \)

\[
\sigma_{\text{code}}, \sigma_{\text{ret}}, \sigma_{\text{clo}}, \sigma_{\text{gc}} \triangleq (\text{pure}, H_{\text{code}}, \sigma_{\text{ret}}, \sigma_{\text{clo}}, \sigma_{\text{code}}, \sigma_{\text{gc}})
\]

\[
H_{\text{code}} \sigma_{\text{ret}}, \sigma_{\text{clo}}, \text{code} \ (T_A, \sigma_{\text{glob}, \text{ret}}, \sigma_{\text{glob}, \text{clo}}) \ W =
\[
\begin{cases}
\text{(n, (code \lor ms_{pad}), code \lor ms_{pad}))} \\
\text{dom(code) = [b, e] \land} \\
\{(b - 1, e + 1) \subseteq T_A \land \sigma_{\text{ret}} \subseteq \sigma_{\text{glob}, \text{ret}} \land \sigma_{\text{clo}} \subseteq \sigma_{\text{glob}, \text{clo}} \land \tau = \text{trusted} \} \\
\text{ms_{pad} = [b - 1 \lor 0] \lor [e + 1 \lor 0] \land} \\
\sigma_{\text{ret}}, \sigma_{\text{clo}}, T_A \vdash \text{comp-code code} \land} \\
\forall a \in \text{dom(code)}.
\end{cases}
\]

\[
H_{\sigma} \sigma_{\text{ret}}, \sigma_{\text{clo}}, \text{code} \ (T_A, \text{stk}, \text{base}) \ W =
\[
\begin{cases}
\text{(n, \text{ret-ptr-code}(b, e, a', + \text{call}_\text{len}),)} \\
\text{((\text{rw}, \text{normal}), b, e, a)} \\
\sigma_{\text{ret}} \subseteq \sigma_{\text{glob}, \text{ret}} \text{ and} \\
\text{dom(code) \subseteq T_A} \text{ and} \\
\text{decodeInstruction(code([a', a' + \text{call}_\text{len} - 1])) = call_{\text{off}}.off_{\sigma} r_1 r_2 \text{ and} } \\
\text{a = a' + ret_{ptr}.offset and} \\
\text{code(a' + off_{pc}) = seal(σ_b, σ_c, σ_b) and σ = σ_b + off_σ ∈ σ_{\text{ret}}} \text{ and} \\
\text{[a', a' + \text{call}_\text{len} - 1] \subseteq [b, e]} \text{ and} \\
\text{(n, \text{ret-ptr-data}(b, e),)} \\
\text{((\text{rw}, \text{linear}), b, e, b - 1))} \\
\sigma_{\text{ret}} \subseteq \sigma_{\text{glob}, \text{ret}} \text{ and} \\
\text{dom(code) \subseteq T_A} \text{ and} \\
\exists r \in \text{addressable(linear, ξ(W), priv). ξ(W), priv} (r). H \triangleq \left( \text{sta}_{\mathcal{T}_s}, \text{so}_{\mathcal{T}_s}, (T_A, \text{stk}, \text{base}), a' + \text{call}_\text{len} \right) \text{ and} \\
\text{dom(ms}_s) = \text{dom(ms}_s) = [b, e] \text{ and} \\
\text{decodeInstruction(code([a', a' + \text{call}_\text{len} - 1])) = call_{\text{off}}.off_{\sigma} r_1 r_2 \text{ and} } \\
\text{code(a' + off_{pc}) = seal(σ_b, σ_c, σ_b) and σ = σ_b + off_σ ∈ σ_{\text{ret}}} \text{ and} \\
\left( \text{if σ ∈ σ_{\text{ret}}} \right)
\end{cases}
\]

and

\[
H_{\sigma} \sigma_{\text{ret}}, \sigma_{\text{clo}}, \text{code} \ (T_A, \text{stk}, \text{base}) \ W =
\[
\begin{cases}
\text{(n, (sc, sc'))} \\
\text{(dom(code) \# T_A and (n, (sc, sc')) ∈ V_{\text{untrusted}}. ξ(\mathcal{W})) or} \\
\text{(dom(code) \subseteq T_A and σ_{\text{clo}} \subseteq σ_{\text{glob}, \text{clo}} and σ_{\text{ret}} \subseteq σ_{\text{glob}, \text{ret}} and} \\
\text{executeable(sc) \land (n, (sc, sc')) \in V_{\text{trusted}}. ξ(\mathcal{W}) \lor} \\
\text{nonExecuteable(sc) \land (n, (sc, sc')) \in V_{\text{untrusted}}. ξ(\mathcal{W}))} \\
\text{if σ ∈ σ_{\text{clo}}}
\end{cases}
\]

\[\text{if σ ∈ σ_{\text{clo}}}\]

3.7 Reasonable Components

Take a set of trusted addresses \( T_A \) and sets of return pointer and closure seals \( σ_{\text{glob}, \text{ret}} \text{ and } σ_{\text{glob}, \text{clo}} \). We define that a word \( w \) is reasonable up to \( n \) steps in memory \( ms \) and free stack \( ms_{\text{stk}} \) if \( n = 0 \) or the following implications hold.

**Definition 8 (Reasonable word).**

- If \( w = \text{seal}(σ_b, σ_c, ...) \), then \([σ_b, σ_c] \# (σ_{\text{glob}, \text{ret}} \cup σ_{\text{glob}, \text{clo}})\)
- If \( w = \text{sealed}(σ, sc) \text{ and } σ \notin (σ_{\text{glob}, \text{ret}} \cup σ_{\text{glob}, \text{clo}}) \) then \( sc \) is reasonable up to \( n - 1 \) steps.
- If \( w = ((\text{perm}, ...), b, e, ...) \text{ and perm ∈ readAllowed and } n > 0 \), then \( ms(a) \) is reasonable up to \( n - 1 \) steps for all \( a ∈ ([b, e] \setminus T_A) \)
• If \( w = \text{stack-ptr}(\text{perm}, b, e, \ldots) \) and \( \text{perm} \in \text{readAllowed} \) and \( n > 0 \), then \( ms_{stk}(a) \) is reasonable up to \( n - 1 \) steps for all \( a \in [b, e] \)

\[\square\]

**Definition 9** (Reasonable configuration). We say that an execution configuration \( \Phi \) is reasonable up to \( n \) steps with \( (T_A, stk\_base, \sigma_{glob,ret}, \sigma_{glob, clos}) \) iff for \( n' \leq n \):

• Guarantee stack base address before call If
  
  \(- \Phi \) points to \( \text{call}^{off_{pc}, off_{\cdot}} r_1 r_2 \) in \( T_A \) for some \( r_1 \) and \( r_2 \)

  Then all of the following hold:
  
  \(- \Phi(r_{stk}) = \text{stack-ptr}(\cdot, stk\_base, \ldots) \)
  
  \(- r_1 \neq r_{11} \)
  
  \(- n' = 0 \) or \( \Phi(pc) + \text{call}\_len \) behaves reasonably up to \( n' - 1 \) steps

• Use return seals only for calls, use closure seals appropriately If
  
  \(- \Phi \) points to \( \text{cseal} r_1 r_2 \) in \( T_A \) and \( \Phi(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma) \)

  Then one of the following holds:
  
  \(- \Phi \) is inside \( \text{call}^{off_{pc}, off_{\cdot}} r_1' r_2' \) and \( \sigma \in \sigma_{glob, ret} \)
  
  \(- \sigma \in \sigma_{glob, clos} \) and one of the following holds:
  
  \(* \) executable(\( \Phi(r_1) \)) and \( n' = 0 \) or \( \Phi(r_1) \) behaves reasonably up to \( n' - 1 \) steps.
  
  \(* \) nonExecutable(\( \Phi(r_1) \)) and \( n' = 0 \) or \( \Phi(r_1) \) is reasonable up to \( n' - 1 \) steps in memory \( \Phi.ms \) and free stack \( \Phi.ms_{stk} \)

• Don’t store private stuff... If
  
  \(- \Phi \) points to \( \text{store} r_1 r_2 \) in \( T_A \), then

  Then \( n' = 0 \) or \( \Phi.reg(r_2) \) is reasonable in memory \( \Phi.mem \) up to \( n' - 1 \) steps.

• Don’t leak private stuff... If
  
  \(- \Phi \rightarrow_{T_A, stk\_base} \Phi' \)

  Then one of the following holds:
  
  \(- \) all of the following hold:
  
  \(* \) \( \Phi'.reg(pc) = ((\text{perm}, l), b, e, a') \) and \( \Phi.reg(pc) = ((\text{perm}, l), b, e, a) \)
  
  \(* \) \( \Phi \) does not point to \( \text{jmp} r_1 r_2 \) for some \( r_1 \) and \( r_2 \)
  
  \(* \) \( \Phi \) does not point to \( \text{call}^{off_{pc}, off_{\cdot}} r_1 r_2 \) for some \( r_1 \) and \( r_2 \), \( off_{pc}, off_{\cdot} \)
  
  \(* \) \( n' = 0 \) or \( \Phi' \) is reasonable up to \( n' - 1 \) steps
  
  \(- \) \( \Phi \) points to \( \text{call}^{off_{pc}, off_{\cdot}} r_1 r_2 \) for some \( r_1 \) and \( r_2 \)
  
  \(* \) \( n' = 0 \) or \( \Phi.reg(r) \) is reasonable in memory \( \Phi.mem \) and free stack \( \Phi.ms_{stk} \) up to \( n' - 1 \) steps for all \( r \neq pc \)
  
  \(- \) \( \Phi \) points to \( \text{jmp} r_1 r_2 \) for some \( r_1 \) and \( r_2 \)
  
  \(* \) \( n' = 0 \) or \( \Phi.reg(r) \) is reasonable in memory \( \Phi.mem \) and free stack \( \Phi.ms_{stk} \) up to \( n' - 1 \) steps for all \( r \neq pc \)

\[\square\]

**Lemma 16.** For all \( n' \leq n \) if

\( \bullet \) \( \Phi \) is reasonable up to \( n \) steps

Then

\( \bullet \) \( \Phi \) is reasonable up to \( n' \) steps

\[\square\]

**Proof.** Follows from the definition.
Definition 10 (Reasonable, pc). We say that an executable capability \( c = (\text{perm}, \text{normal}), b, e, a \) behaves reasonably up to \( n \) steps if for any \( \Phi \) such that

- \( \Phi.\text{reg}(pc) = c \)
- \( \Phi.\text{reg}(r) \) is reasonable up to \( n \) steps in memory \( \Phi.\text{mem} \) and free stack \( \Phi.\text{ms}_{\text{stk}} \) for all \( r \neq pc \)
- \( \Phi.\text{mem}, \Phi.\text{ms}_{\text{stk}} \) and \( \Phi.\text{stk} \) are all disjoint

We have that \( \Phi \) is reasonable up to \( n \) steps.

Definition 11 (Reasonable component). We say that a component \( (\text{ms}_{\text{code}}, \text{ms}_{\text{data}}, \text{import}, \text{export}, \text{ret}, \text{clos}, A_{\text{linear}}) \) is reasonable if the following hold: For all \( (s \mapsto \text{sealed}(\sigma, sc)) \in c_{\text{export}} \), with executable\((sc)\), we have that \( sc \) behaves reasonably up to any number of steps \( n \).

We say that a component \( (\text{comp}_0, c_{\text{main}}, c, c_{\text{main,d}}) \) is reasonable if \( \text{comp}_0 \) is reasonable.

Lemma 17. If \( W.\text{heap}(\text{r}_{\text{code}}) \# = \text{code}_{\text{reg}} \oplus \text{code}_{\text{gc}} \) and \( \text{ms}_s, \text{stk}, \text{ms}_{\text{stk}}, \text{ms}_{\text{tr}} : \text{gc} W \), then there exists \( W'' \) and \( W' \) such that \( W = W' \oplus W'' \) and \( (n', (\text{code}, \text{code})) \in H.\text{code}_{\text{ret}} \text{code}_{\text{gc}} \xi^{-1}(W) \) for all \( n' < n \).

Proof. By definition of \( \text{ms}_s, \text{stk}, \text{ms}_{\text{stk}}, \text{ms}_{\text{tr}} : \text{gc} W \), we get that \( W = W_{\text{stack}} \oplus W_{\text{free, stack}} \oplus W_{\text{heap}} \) and \( (n, (\sigma, \text{ms}_s, \text{ms}_{\text{tr}} \text{heap})) \in H.\text{W.heap}(\text{W.heap}) \) for some \( \text{ms}_s \subseteq \text{ms}_s \) and \( \text{ms}_{\text{tr}} \text{heap} \subseteq \text{ms}_{\text{tr}} \).

By definition of \( (n, (\sigma, \text{ms}_s, \text{ms}_{\text{tr}} \text{heap})) \in H.\text{W.heap}(\text{W.heap}) \), we get an \( R_{\text{ms}} : \text{dom(\text{active}(\text{W.heap})))} \rightarrow \text{World such that} \ W' = \oplus_{r \in \text{dom(\text{active}(\text{W.heap}))}} R_W(r) \) and \( (n', R_{\text{ms}}(\text{r}_{\text{code}})) \in \text{W.heap}(\text{r}_{\text{code}}) H.\xi^{-1}(W_{\text{r}_{\text{code}}}) \) for all \( n' < n \).

Since \( W.\text{heap}(\text{r}_{\text{code}}) \# = \text{code}_{\text{reg}} \oplus \text{code}_{\text{gc}} \), this implies that also \( (n', R_{\text{ms}}(\text{r}_{\text{code}})) \in \text{code}_{\text{reg}} \oplus \text{code}_{\text{gc}} H.\xi^{-1}(W_{\text{r}_{\text{code}}}) \).

i.e. \( (n', R_{\text{ms}}(\text{r}_{\text{code}})) \in \text{code}_{\text{reg}} \oplus \text{code}_{\text{gc}} H.\xi^{-1}(W_{\text{r}_{\text{code}}}) \).

Lemma 18 (Untrusted source values are reasonable). If

- \( gc = (T_A, \text{stk}_{\text{base}}, \text{global}_{\text{ret}}, \text{global}_{\text{clos}}) \)
- \( (n, (\text{w, } \xi)) \in V_{\text{untrusted}}(W_w) \)
- \( \text{ms}_s, \text{stk}, \text{ms}_{\text{stk}}, \text{ms}_{\text{tr}} : \text{gc} W_M \)
- \( \text{purePart}(W_w) \oplus \text{purePart}(W_M) \) is defined

then, with respect to \( T_A, \text{global}_{\text{ret}}, \text{global}_{\text{clos}}, w \) is reasonable up to \( n \) steps in memory \( \text{ms}_s \) and free stack \( \text{ms}_{\text{stk}} \).

Proof. Induction over \( n \). For \( n = 0 \) all words are reasonable. For \( n > 0 \), we need to prove four implications:

- If \( w = \text{sealed}(\sigma_b, \sigma_e, \xi) \), then \( [\sigma_b, \sigma_e] \# ([\text{global}_{\text{ret}} \cup \text{global}_{\text{clos}}) \)

This follows directly from \( (n, (\text{w, } \xi)) \in V_{\text{untrusted}}(W_w) \) by definition of \( V_{\text{untrusted}} \).

- If \( w = ((\text{perm}_{\text{m}}, \zeta), b, e, \xi) \), then \( [b, e] \# T_A \)

This follows directly from \( (n, (\text{w, } \xi)) \in V_{\text{untrusted}}(W_w) \) by definition of \( V_{\text{untrusted}} \).

- If \( w = \text{sealed}(\sigma, sc) \) and \( \sigma \notin ([\text{global}_{\text{ret}} \cup \text{global}_{\text{clos}}) \) then \( sc \) is reasonable for \( n - 1 \) steps:

By \( (n, (\text{w, } \xi)) \in V_{\text{untrusted}}(W_w) \), we get some region \( r \in \text{dom(W.heap)} \) and a set of return seals, a set of closure seals and a code memory: \( \text{sigma}, \text{code}_{\text{ms}}, \text{ms}_{\text{code}} \) such that \( W.\text{heap}(\text{r}) = (\text{purePart, } H_L) \) and \( H_L \# H_{\text{sigma}} \oplus \text{code}_{\text{ms}} \oplus \text{code}_{\text{gc}} \) and \( (n', \text{sealed}(\sigma, sc, \xi)) \in H_L \sigma \xi^{-1}(W) \) for all \( n' < n \), so in particular for \( n' = n - 1 \). It follows easily from the above that also \( (n - 1, (sc, \xi)) \in H_{\text{sigma}} \oplus \text{code}_{\text{ms}} \oplus \text{code}_{\text{gc}} \) and \( \xi^{-1}(W) \).

This gives us three cases for \( sc \). The two first cases \( sc = \text{ret-ptr-data(\_ \_ \_)} \) and \( sc = \text{ret-ptr-code(\_ \_ \_)} \) are easily discharged as the reasonability definition puts no requirements on return pointers. For the final case, \( \sigma \in \text{global}_{\text{clos}} \) and either

- \( \text{dom(m}_{\text{code}}) \neq T_A \) and \( (n - 1, (sc, \xi)) \in V^\text{untrusted}(W) \); or

- \( \text{dom(m}_{\text{code}}) \subseteq T_A, \text{global}_{\text{clos}} \subseteq \text{global}_{\text{clos}} \), and \( (n, (sc, \xi)) \in V^\text{untrusted}(W) \) for \( \text{tst} = \text{untrusted iff nonExecutable(sc)} \).

In the first case, the result follows from the induction hypothesis. In the second case, we have a contradiction \( \sigma \notin ([\text{global}_{\text{ret}} \cup \text{global}_{\text{clos}}) \).

- If \( w = ((\text{perm}_{\text{m}}, \zeta), b, e, \xi) \) and \( \text{perm} \in \text{readAllowed} \) and \( n > 0 \), then \( \text{ms}_s(a) \) is reasonable up to \( n - 1 \) steps for all \( a \in ((b, e) \setminus T_A) \):

From \( (n, (\text{w, } \xi)) \in V^\text{untrusted}(W_w) \), we get \( (n, [b, e]) \in \text{readCondition} \oplus \text{gc}(\text{perm}, l) \).

From \( (n, [b, e]) \in \text{readCondition} \oplus \text{gc}(\text{perm}, l) \) we get \( S \subseteq \text{addressable}(l, \text{W.heap}) \) and \( R : S \rightarrow \mathcal{P}(N) \) such that
− ∪_{r ∈ S} R(r) ⊇ [b, c]
− (l = linear ⇒ ∀r, |R(r)| = 1)
− ∀r ∈ S,W.heap(r).H ⊆ n \overset{\mathrm{std,p,□}}{ R(r),gc}.H

given a ∈ [b, c] we know by the above that there exists r such that a ∈ R(r).

By ms, stk, ms stk, ∈ M, we get R ms : dom(active(W.heap)) → MemorySegment × MemorySegment such that ms = ∪_{r ∈ dom(active(W.heap))} π1(R ms(r)).

Further, we get R W : dom(active(W.heap)) → World such that W ′ = ∩_{r ∈ dom(active(W.heap))} R W(r) and ∀r ∈ dom(active(W.heap)). (n, R ms(r)) ∈ W.heap(r).H ξ⁻¹(R W(r)). For W M = W ′ ⊕ W ″ for some W ″.

In particular, we have (n, R ms(r)) ∈ W.heap(r).H ξ⁻¹(R W(r)).

We know W.heap(r).H ⊆ n \overset{\mathrm{std,p,□}}{ R(r),gc}.H, so (n′, R ms(r)) ∈ \overset{\mathrm{std,p,□}}{ R(r),gc}.H(ξ⁻¹(R W(r))) for n′ < n. This gives us (n′, (π1(R ms(r))(a), b)) ∈ V_{untrusted}^{\mathrm{gc}}(R W(a)) where R W : [b, e] → World such that ∪_{a ∈ dom(π1(R ms(r)))} R W(a) = R W(r).

At this point we apply the induction hypothesis which is possible as we have the following:

− (n−1, (π1(R ms(r))(a), b)) ∈ V_{untrusted}^{\mathrm{gc}}(R W(a))
− ms, stk, ms stk, ∈ M
− purePart(R W(a)) ⊃ purePart(W M)

Which follows by definition of purePart and the fact that R W(a) is W M with part of its ownership, but purePart strips away the ownership making the two compatible.

which gives us that, with respect to T_A, σ_{glob, get}, σ_{glob, close}, π1(R ms(r))(a) is reasonable up to n − 1 steps in memory ms and free stack ms stk. Which is what we wanted as π1(R ms(r))(a) = ms(a).

• If w = stack-ptr(perm, b, e, c) and perm ∈ readAllowed and n > 0, then ms stk(a) is reasonable up to n − 1 steps for all a ∈ [b, c]:

This case is proven in the same way the previous case was. The main difference is that the free part of the world is used instead of the heap part:

From (n, (w, c)) ∈ V_{untrusted}^{\mathrm{gc}}(W w) and perm ∈ readAllowed, we get (n, [b, c]) ∈ stackReadCondition □,gc(perm).

From (n, [b, c]) ∈ stackReadCondition □,gc(perm) we get S ⊆ addressable(linear, W.free) and R : S → P(\mathbb{N}) such that

− ∪_{r ∈ S} R(r) ⊇ [b, c]
− ∀r, |R(r)| = 1
− ∀r ∈ S,W.heap(r).H ⊆ n \overset{\mathrm{std,p,□}}{ R(r),gc}.H

given a ∈ [b, c] we know by the above that there exists r such that a ∈ R(r).

By ms, stk, ms stk, ∈ M, we get R ms : dom(active(W.free)) → MemorySegment × MemorySegment such that ms stk = ∪_{r ∈ dom(active(W.free))} π1(R ms(r)).

Further, we get R W : dom(active(W.free)) → World such that W ′ = ∩_{r ∈ dom(active(W.free))} R W(r) and ∀r ∈ dom(active(W.free)). (n, R ms(r)) ∈ W.free(r).H ξ⁻¹(R W(r)). For W M = W ′ ⊕ W ″ for some W ″.

In particular, we have (n, R ms(r)) ∈ W.free(r).H ξ⁻¹(R W(r)).

We know W.free(r).H ⊆ n \overset{\mathrm{std,p,□}}{ R(r),gc}.H, so (n′, R ms(r)) ∈ \overset{\mathrm{std,p,□}}{ R(r),gc}.H(ξ⁻¹(R W(r))) for n′ < n. This gives us (n′, (π1(R ms(r))(a), b)) ∈ V_{untrusted}^{\mathrm{gc}}(R W(a)) where R W : [b, e] → World such that ∪_{a ∈ dom(π1(R ms(r)))} R W(a) = R W(r).

At this point we apply the induction hypothesis which is possible as we have the following:

− (n−1, (π1(R ms(r))(a), b)) ∈ V_{untrusted}^{\mathrm{gc}}(R W(a))
− We get this by the above.
− ms, stk, ms stk, ∈ M
− This follows by assumption and Lemma

26
Lemma 19 (Untrusted register files are reasonable). If

- \(\text{purePart}(R_W'(a)) \oplus \text{purePart}(W_M)\)

Which follows by definition of \(\text{purePart}\) and the fact that \(R_W'(a)\) is \(W_M\) with part of its ownership, but \(\text{purePart}\) strips away the ownership making the two compatible.

which gives us that, with respect to \(T_A, \sigma_{\text{glob,ret}}, \sigma_{\text{glob, clos}}\), \(\pi_1(R_{\text{ms}}(r))(a)\) is reasonable up to \(n - 1\) steps in memory \(ms_S\) and free stack \(ms_{\text{stk}}\). Which is what we wanted as \(\pi_1(R_{\text{ms}}(r))(a) = ms_{\text{stk}}(a)\).

\[\square\]

Lemma 20 (Reasonable things don’t need to be trusted). If

- \((n, (w, w')) \in Y_{\text{trusted}}^{\text{gc}}(W_w)\)
- \(w\) is reasonable up to \(n\) steps in memory \(ms_S\) and free stack \(ms_{\text{stk}}\), with respect to code, \(T_A, \sigma_{\text{glob,ret}}, \sigma_{\text{glob, clos}}\)
- \(n > 0\)
- \ thoerem 2 holds up to \(n\) steps.

Then

- \((n, (w, w')) \in Y_{\text{untrusted}}^{\text{gc}}(W_w)\)

\[\square\]

Proof. Assume that \((n, (w, w_T)) \in Y_{\text{trusted}}^{\text{gc}}(W_w)\)

By definition of \(Y_{\text{trusted}}^{\text{gc}}\), it is clear that also \((n, (w, w_T)) \in Y_{\text{untrusted}}^{\text{gc}}(W_w)\), except in the following two cases:

- \(w = \text{seal}(\sigma, \sigma_b, \sigma_e)\)
- \(w_T = \text{seal}(\sigma, \sigma_b, \sigma_e)\)
- \(r \in \text{dom}(W_{\text{heap}})\)
- \(\text{W.heap}(r) = \text{code, } T_{\text{code, ret, code, } T_{A, \text{stk, base, } \sigma_{\text{glob,ret}}, \sigma_{\text{glob, clos}}}}\)
- \(\text{dom}(\text{code}) \subseteq T_A\)
- \([\sigma_b, \sigma_e] \subseteq (\sigma_{\text{ret}} \cup \sigma_{\text{clos}})\)
- \(\sigma_{\text{ret}} \subseteq \sigma_{\text{glob,ret}}\)
Then (Untrusted environments produce safe closures)

Lemma 21  

code blocks which may require seals for trusted seals which are not in the untrusted value relation.

If perm = 0, then (n, (w, wT)) \in \mathcal{V}_{\text{untrusted}}^{\sqsubseteq, gc}(W_w) follows directly.

If perm \neq 0, then by definition of reasonability, we get immediately that \{b, e\} \notin T_A.

By definition of \mathcal{V}_{\text{untrusted}}^{\sqsubseteq, gc}, it suffices to show that (n, [b, e]) \in \mathsf{readCondition}_{\sqsubseteq, gc}(W_{\text{normal}}) and (n, [b, e]) \in \mathsf{executeCondition}_{\sqsubseteq, gc}(W_{\text{normal}}). The former follows directly from Lemma 27. The latter follows by Theorem 2, Lemmas 30 and 47 and definition of \mathsf{executeCondition}_{\sqsubseteq, gc}.

\[ \square \]

3.8 Fundamental Theorem of Logical Relations

Theorem 2 (FTLR).  

For all n, W, l, b, e, a, If

\[ \{b, e\} \subseteq \mathsf{readCondition}_{\sqsubseteq, gc}(W) \]

and one of the following sets of requirements holds:

- \[ \{b, e\} \subseteq T_A \]
- \((\{\text{rx, normal}\}, b, e, a)\) behaves reasonably up to n steps.
- \[ \{b, e\} \notin T_A \]

Then

\[ \left( n, \left( \{\text{rx, normal}\}, b, e, a \right) \right) \in \mathcal{E}_{\sqsubseteq, gc}(W) \]

Note: we don’t require the readcondition in the trusted case because trusted code pointers point to trusted code blocks which may require seals for trusted seals which are not in the untrusted value relation.

Lemma 21  (Untrusted environments produce safe closures).  

If

\[ \{n, (w_1, w'_1)\} \in \mathcal{E}_{\sqsubseteq, gc}(W_1) \text{ or } \text{nonExecutable}(w_1) \text{ and nonExecutable}(w'_1) \]

\[ w_1 \neq \text{ret-ptr-code}(\_). \]

\[ \{n, (w_2, w'_2)\} \in \mathcal{V}_{\text{untrusted}}^{\sqsubseteq, gc}(W_2) \text{ or } \text{executeable}(w_2) \text{ and executeable}(w'_2) \]

Then

\[ \{n, ((w_1, w_2), (w'_1, w'_2))\} \in \mathcal{E}_{\text{jump}}(W_1 \oplus W_2) \]

Proof. Take \( n' \leq n \), \( \text{reg}_S, \text{reg}_T, \text{ms}_S, \text{ms}_T, \text{ms}_{\text{stk}}, \text{stk}, W_\mathcal{R}, W_\mathcal{M} \) and assume

\[ \{n', (\text{reg}_S, \text{reg}_T)\} \in \mathcal{V}_{\text{untrusted}}^{\sqsubseteq, gc}(\{\text{data}\})(W_\mathcal{R}) \]

\[ (\text{ms}_S, \text{stk}, \text{ms}_{\text{stk}}, \text{ms}_T)_{\sqsubseteq, gc} W_\mathcal{M} \]

\[ \Phi_S = (\text{ms}_S, \text{reg}_S, \text{stk}, \text{ms}_{\text{stk}}) \]

\[ \Phi_T = (\text{ms}_T, \text{reg}_T) \]

\( W_1 \oplus W_2 \oplus W_\mathcal{R} \oplus W_\mathcal{M} \) is defined

Then we need to prove that for \( \Phi'_S, \Phi'_T \)

\[ \Phi'_S = \text{jumpResult}(w_1, w_2, \Phi_S) \]
Using the fact that \( w_1 \neq \text{ret-.ptr-code}(\_), \) we know by definition of \( \text{xjumpResult}(w_1, w_2, \Phi_S) \) that we must be in one of the following two cases:

- \( w_1 \neq \text{ret-ptr-code}(\_), w_2 \neq \text{ret-ptr-data}(\_), \) nonExecutable\( (w_2) \) and \( \Phi'_S = \Phi_S[\text{reg.pc} \mapsto w_1][\text{reg.r.data} \mapsto w_2]. \)

  From nonExecutable\( (w_2) \) and \( (n, (w_2, w'_2)) \in V^{\square.gc}_{\text{untrusted}}(W_2) \) or executable\( (w_2) \), it follows that also nonExecutable\( (w_2') \) and \( \Phi_T = \Phi_T[\text{reg.pc} \mapsto w'_1][\text{reg.r.data} \mapsto w'_2]. \)

  We can now combine \((n', (\text{reg.S}, \text{reg.T})) \in \mathcal{R}^{\square.gc}_{\text{untrusted}}(\{\text{r.data}\})(W_R) \) and \((n, (w_2, w'_2)) \in V^{\square.gc}_{\text{untrusted}}(W_2) \) into \((n', (\text{reg.S}[\text{r.data} \mapsto w_2], \text{reg.T}[\text{r.data} \mapsto w'_2])) \in \mathcal{R}^{\square.gc}_{\text{untrusted}}(W_R \oplus W_2) \), using Lemmas 43, 40, 12, and 44.

  With our other assumptions above, \((n, (w_1, w'_1)) \in E^{\square.gc}(W_1) \) then gives us that \((n', (\Phi'_S, \Phi'_T)) \in O^{\square.gc}, \) as required. On the other hand, if nonExecutable\( (w_1) \) and nonExecutable\( (w'_1) \), then \( \Phi'_S \mapsto gc \) failed and \( \Phi'_T \mapsto gc \) failed and the result follows by definition of \( O^{\square.gc}_{(T_A, \text{stk.base}, \text{glob.r.data}, \text{gc}, \text{global.exe})} \) and \( O^{\square.gc}_{(T_A, \text{stk.base}, \text{glob.r.data}, \text{gc}, \text{global.exe})}. \)

- Otherwise: \( \Phi'_S = \text{failed}. \)

  From \( w_1 \neq \text{ret-ptr-code}(\_) \) and \((n, (w_2, w'_2)) \in V^{\square.gc}_{\text{untrusted}}(W_2) \) or executable\( (w_2) \) and executable\( (w'_2) \),

  the only way we can get to this case is that executable\( (w_2) \) and executable\( (w'_2) \).

  It follows that \( \Phi_T = \text{failed}. \)

  The result follows by definition of \( O^{\square.gc}_{(T_A, \text{stk.base}, \text{glob.r.data}, \text{gc}, \text{global.exe})} \) and \( O^{\square.gc}_{(T_A, \text{stk.base}, \text{glob.r.data}, \text{gc}, \text{global.exe})}. \)

\[ \Box \]

**Lemma 22** (Safe values are safe to execute). If

- \((n, (w_1, w'_1)) \in V^{\square.gc}_{\text{untrusted}}(W_1)\)
- \((n', (w_2, w'_2)) \in V^{\square.gc}_{\text{untrusted}}(W_2) \) or executable\( (w_2) \) and executable\( (w'_2) \)
- \(n' < n\)

Then

- \((n', ((w_1, w_2), (w'_1, w'_2))) \in E_{\text{xjump}}(W_1 \oplus W_2)\)

\[ \Box \]

**Proof.** By Lemma 21, it suffices to prove the following:

- \((n', (w_1, w'_1)) \in E^{\square.gc}(W_1) \) or (nonExecutable\( (w_1) \) and nonExecutable\( (w'_1) \))
- \(w_1 \neq \text{ret-ptr-code}(\_). \)

  The latter follows immediately from \((n, (w_1, w'_1)) \in V^{\square.gc}_{\text{untrusted}}(W_1) \) by definition. It also follows that either (nonExecutable\( (w_1) \) and nonExecutable\( (w'_1) \)) or \(w_1 = w'_1 = (\{\text{perm}, \text{normal}, b, c, a\} \) and \((n, [b, c]) \in \text{executeCondition}^{\square.gc}(W_1). \)

  In the latter case, it follows by definition of \( \text{executeCondition}^{\square.gc} \) that \((n', (w_1, w'_1)) \in E^{\square.gc}(W_1) \) as required. \[ \Box \]

### 3.9 Related components

\[
C^{\square.gc}(W) = \left\{ (n, \text{comp}, \text{comp}) \mid \begin{array}{l}
\text{comp} = (m_{\text{code}}, m_{\text{data}}, \mathcal{A}_{\text{import}} \leq \mathcal{A}_{\text{import}}, \mathcal{S}_{\text{export}} \mapsto \mathcal{W}_{\text{export}}, \mathcal{S}_{\text{ret}} \mapsto \mathcal{S}_{\text{clo}s}) \\
\text{For all } W' \not\subseteq W.
\end{array} \right.
\]

If \((n', (w_{\text{import}}, w_{\text{import}})) \in V^{\square.gc}_{\text{untrusted}}(\text{purePart}(W')) \) for all \( n' < n \)

and \( m_{\text{data}} = m_{\text{data}}(\mathcal{A}_{\text{import}} \mapsto \mathcal{W}_{\text{import}}) \)

then \((n, (\mathcal{S}_{\text{ret}} \cup \mathcal{S}_{\text{clo}s}, m_{\text{code}} \uplus m_{\text{data}}', m_{\text{code}} \uplus m_{\text{data}})) \in H(\text{W.heap})(W') \) and

\[
(n, (w_{\text{export}}, w_{\text{export}})) \in V^{\square.gc}_{\text{untrusted}}(\text{purePart}(W'))
\]

\[ \cup \left\{ (n, (\text{comp}_0, c_{\text{main}}, c_{\text{main}}), (\text{comp}_0, c_{\text{main}}, c_{\text{main}})) \mid (n, (\text{comp}_0, \text{comp}_0)) \in C^{\square.gc}(W) \text{ and } \{(_{\mapsto c_{\text{main}}}, c_{\text{main}}) \subseteq \mathcal{W}_{\text{export}} \right\} \]

\[ 29 \]
Lemma 23 (Compatibility lemma for linking). If

- \((n, (\text{comp}_1, \text{comp}_1)) \in C^{\square, gc}(W_1)\)
- \((n, (\text{comp}_2, \text{comp}_2)) \in C^{\square, gc}(W_2)\)

then

- \((n, (\text{comp}_1 \bowtie \text{comp}_2, \text{comp}_1 \bowtie \text{comp}_2)) \in C^{\square, gc}(W_1 \uplus W_2).\)

Proof. First consider the case where one of the two pairs of components have a pair of “main” capabilities. By the definition of \(C^{\square, gc}\), these capabilities have to be part of that components’ export list. It is then easy to see that the other component cannot have “main” capabilities (otherwise linking is not defined) and it is sufficient to prove the result for the underlying components (without the “main” capabilities). Therefore, we can restrict ourselves to the case where both pairs of components are of the form \(\text{comp}_0\), i.e. no “main” capabilities.

By definition of \(\bowtie\), we have that the following hold:

- \(\text{comp}_1 = (ms_{\text{code},1}, ms_{\text{data},1}, \overline{\text{import}}_1, \text{export}_1, \sigma_{\text{ret},1}, \sigma_{\text{clos},1}, A_{\text{linear},1})\)
- \(\text{comp}_2 = (ms_{\text{code},2}, ms_{\text{data},2}, \overline{\text{import}}_2, \text{export}_2, \sigma_{\text{ret},2}, \sigma_{\text{clos},2}, A_{\text{linear},2})\)
- \(\text{comp}_3 = (ms_{\text{code},3}, ms_{\text{data},3}, \overline{\text{import}}_3, \text{export}_3, \sigma_{\text{ret},3}, \sigma_{\text{clos},3}, A_{\text{linear},3})\)
- \(ms_{\text{code},3} = ms_{\text{code},1} \uplus ms_{\text{code},2}\)
- \(ms_{\text{data},3} = (ms_{\text{data},1} \uplus ms_{\text{data},2})[a \mapsto w | (a \leftrightarrow s) \in \overline{\text{import}}_1 \cup \overline{\text{import}}_2, (s \rightarrow w) \in \text{export}_3]\)
- \(\text{export}_3 = \text{export}_1 \cup \text{export}_2\)
- \(\overline{\text{import}}_3 = \{a \leftrightarrow s \in (\overline{\text{import}}_1 \cup \overline{\text{import}}_2) | s \mapsto w \notin \text{export}_3\}\)
- \(\sigma_{\text{ret},3} = \sigma_{\text{ret},1} \cup \sigma_{\text{ret},2}\)
- \(\sigma_{\text{clos},3} = \sigma_{\text{clos},1} \cup \sigma_{\text{clos},2}\)
- \(A_{\text{linear},3} = A_{\text{linear},1} \uplus A_{\text{linear},2}\)
- \(\text{dom}(ms_{\text{code},3}) \neq \text{dom}(ms_{\text{data},3})\)
- \(\sigma_{\text{ret},3} \neq \sigma_{\text{clos},3}\)

Now take \(W' \supset (W_1 \cup W_2)\) and assume that \((n', (w_{\overline{\text{import}},3}, w_{\text{import},3})) \in \nu^{\square, gc}_{\text{untrusted}}(\text{purePart}(W'))\) for all \(n' < n\). Take \(\overline{\text{import}}_3 = \overline{\text{import}}_1 \leftrightarrow s_{\overline{\text{import}},3}\). Take \(ms'_{\text{data},3} = ms_{\text{data},3}[\overline{\text{import}}_3 \mapsto w_{\text{import}}]\). Then it remains to show that

\[ (n, (\sigma_{\text{ret},3} \cup \sigma_{\text{clos},3}, ms_{\text{code},3} \uplus ms'_{\text{data},3}, ms_{\text{code},3} \uplus ms'_{\text{data},3})) \in H((W_1 \cup W_2).\text{heap})(W') \]

and

\[ (n, (c_{\text{export},3}, c_{\text{export},3})) \in \nu^{\square, gc}_{\text{untrusted}}(\text{purePart}(W')) \]

for all \(n' < n\).

First, note that

\[ ms'_{\text{data},3} = ms_{\text{data},3}[\overline{\text{import}}_3 \mapsto w_{\text{import}}] \]

\[ = (ms_{\text{data},1} \uplus ms_{\text{data},2})[a \mapsto w | (a \leftrightarrow s) \in (\overline{\text{import}}_1 \cup \overline{\text{import}}_2), (s \rightarrow w) \in \text{export}_3] \]

\[ = ms_{\text{data},1}[a \mapsto w | (a \leftrightarrow s) \in \overline{\text{import}}_1, (s \rightarrow w) \in \text{export}_2][w_{\text{import}} \mapsto w_{\text{import}}][\overline{\text{import}}_3 \mapsto w_{\text{import}}, \overline{\text{import}}_3 \mapsto \overline{\text{import}}, \overline{\text{import}}_3 \mapsto a_{\overline{\text{import}},3}] \]

\[ = ms_{\text{data},1} \uplus ms_{\text{data},2}[a \mapsto w | (a \leftrightarrow s) \in \overline{\text{import}}_2, (s \rightarrow w) \in \text{export}_1] \]

First, we prove that for all substituted values \(w\) in the equations above, we have that

\[ (n', (w, w)) \in \nu^{\square, gc}_{\text{untrusted}}(\text{purePart}(W')) \]

for all \(n' < n\). We know by assumption that this is true for the \(w_{\text{import}}\) such that \(a_{\overline{\text{import}},3} \mapsto \in \overline{\text{import}}_1\) or \(a_{\overline{\text{import}},3} \mapsto \in \overline{\text{import}}_2\). On the other hand, we know by induction that this is true for the \(w\) such that \((s \rightarrow w) \in \text{export}_1\) or \((s \rightarrow w) \in \text{export}_2\).
Then, it follows from our assumptions \((n, (\text{comp}_1, \text{comp}_1)) \in \mathcal{C}^{\square,gc}(W_1)\) and \((n, (\text{comp}_2, \text{comp}_2)) \in \mathcal{C}^{\square,gc}(W_2)\), and using Lemma 11 that
\[
(n, (\sigma_{\text{ret},1} \uplus \sigma_{\text{clo}_{1}}, ms_{\text{code},1} \uplus ms'_{\text{data},1}, ms_{\text{code},1} \uplus ms'_{\text{data},1})) \in \mathcal{H}(W_1, \text{heap})(W')
\]
\[
(n, (\sigma_{\text{ret},2} \uplus \sigma_{\text{clo}_{2}}, ms_{\text{code},2} \uplus ms'_{\text{data},2}, ms_{\text{code},2} \uplus ms'_{\text{data},2})) \in \mathcal{H}(W_2, \text{heap})(W')
\]
It follows by Lemma 13 that
\[
(n, (\sigma_{\text{ret},3} \uplus \sigma_{\text{clo}_{3}}, ms_{\text{code},3} \uplus ms'_{\text{data},3}, ms_{\text{code},3} \uplus ms'_{\text{data},3})) \in \mathcal{H}((W_1 \uplus W_2), \text{heap})(W')
\]
Finally, for each
\[
(n, (c_{\text{export},1}, c_{\text{export}})) \in [n, (c_{\text{export},2}, c_{\text{export},3})]
\]
we have that \((n, (c_{\text{export},1}, c_{\text{export}}))\) is either in \((n, (c_{\text{export},1}, c_{\text{export},1}))\) or \((n, (c_{\text{export},2}, c_{\text{export},2}))\). By unfolding \((n, (\text{comp}_1, \text{comp}_1)) \in \mathcal{C}^{\square,gc}(W_1)\) and \((n, (\text{comp}_2, \text{comp}_2)) \in \mathcal{C}^{\square,gc}(W_2)\) and using Lemma 11 this follows from the results above.

3.10 FTLR for components

Lemma 24 (Untrusted code regions’ seals specify value safety). If
- \(\text{dom}(\text{code}) \neq T_A\)
- \(\sigma \in \sigma_{\text{clo}}\)

Then
\[
H^\text{code,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} \text{ code } (T_A, \text{stk}, \text{base}) \sigma = n_{\square,gc}^{\text{untrusted}}
\]

Proof. Follows easily by definition.

Lemma 25 (Code region for untrusted components stronger than standard safe memory region). If
- \(\text{dom}(ms_{\text{code}}) \neq T_A\)

then \(\text{dom}(ms_{\text{code}}, gc) \subseteq \text{dom}(ms_{\text{code}})\), \(H^{\text{code,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} \text{ code } gc}_{\text{std}, p, □} W \n_{\text{dom}(ms_{\text{code}}), gc}^n W\)

Proof. Take \(\hat{W}\), then we need to prove that
\[
H^{\text{code,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} ms_{\text{code}} gc}_{\text{std}, p, □} W \n_{\text{dom}(ms_{\text{code}}), gc}^n W \subseteq H^{\text{std,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} ms_{\text{code}} gc}_{\text{std}, p, □} W
\]
So take \((n, ms_S, ms_T) \in H^{\text{code,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} ms_{\text{code}} gc}_{\text{std}, p, □} W\), then we know that
- \(ms_S = ms_T = ms_{\text{code}} \uplus ms_{\text{pad}}\)
- \(\text{dom}(ms_{\text{code}}) = [b, e]\)
- \([b - 1, e + 1] \subseteq T_A \lor ([b - 1, e + 1] \neq T_A \land \sigma_{\text{ret}} = \emptyset)\), but we know that \(\text{dom}(ms_{\text{code}}) \neq T_A\), so we get that \(\sigma_{\text{ret}} = \emptyset\).
- For all \(a \in \text{dom}(\text{code})\), \((n, (\text{code}(a), \text{code}(a))) \in V^{\square,gc}_{\text{untrusted}}(\text{purePart}(\xi(\hat{W})))\)

By definition of \(H^{\text{std,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} ms_{\text{code}} gc}_{\text{std}, p, □} W\), we need to prove that
- \(\text{dom}(ms_{\text{code}}) = \text{dom}(ms_{\text{code}}) = \text{dom}(ms_{\text{code}})\)
- there exists a \(S : \text{dom}(ms_{\text{code}}) \rightarrow \text{World} with_web(\xi(\hat{W})) = \oplus_{a \in \text{dom}(ms_{\text{code}})} S(a)\) and for all \(a \in \text{dom}(ms_{\text{code}})\), we have that \((n, (ms_S(a), ms_T(a))) \in V^{\square,gc}_{\text{untrusted}}(S(a))\)

We take \(S\) to map every address to \(\text{purePart}(\hat{W})\), except for a single one that we map to \(\hat{W}\). The result then follows by Lemmas 3, 10 and 47 and the above fact that for all \(a \in \text{dom}(\text{code})\), \((n, (\text{code}(a), \text{code}(a))) \in V^{\square,gc}_{\text{untrusted}}(\text{purePart}(\xi(\hat{W})))\).

Lemma 26 (Code memory for untrusted components safely readable). If
- \(\text{dom}(ms_{\text{code}}) \neq T_A\)
- \(W.\text{heap}(r_{\text{code}}) = n_{\text{code,□}_{\sigma} \sigma_{\text{ret}} \sigma_{\text{clo}} ms_{\text{code}}, gc}^n W\)
• $A \subseteq \text{dom}(ms_{\text{code}})$,
we have that $(n, A) \in \text{readCondition}^{\square, gc}(\text{normal}, W)$.

Proof. Follows by definition of $\text{readCondition}^{\square, gc}$ from Lemma 25.

Lemma 27 (Code memory for untrusted components safely readable). If

• $A \neq T_A$
• $(n, A) \in \text{readXCondition}^{\square, gc}(W)$

we have that $(n, A) \in \text{readCondition}^{\square, gc}(\text{normal}, W)$.

Proof. Follows by definition of $\text{readCondition}^{\square, gc}$ and $\text{readXCondition}^{\square, gc}$ from Lemma 25.

Lemma 28 (Code region scaling invariant for untrusted components implies untrusted safety). If

• $\emptyset \neq \text{dom}(ms_{\text{code}}) \neq T_A$
• $\sigma \in \sigma_{\text{close}}$

then $\text{exec}_{\text{code}, \square}^{\square, gc}(ms_{\text{code}}, gc, H_\sigma \sigma \leq \text{untrusted for all } n}$

Proof. The result follows easily by definition of $\text{exec}_{\text{code}, \square}$ and $H_{\text{code}, \square}$.

Lemma 29 (non-linear words are pure). If $(n, A) \in \text{readCondition}^{\square, gc}(\text{normal}, W)$, then $(n, A) \in \text{readCondition}^{\square, gc}(\text{normal}, W)$.

• If $(n, A) \in \text{writeCondition}^{\square, gc}(\text{normal}, W)$, then $(n, A) \in \text{writeCondition}^{\square, gc}(\text{normal}, \text{purePart}(W))$
• If $(n, A) \in \text{executeCondition}^{\square, gc}(W)$, then $(n, A) \in \text{executeCondition}^{\square, gc}(\text{purePart}(W))$
• If $(n, A) \in \text{readXCondition}^{\square, gc}(W)$, then $(n, A) \in \text{readXCondition}^{\square, gc}(\text{purePart}(W))$
• If $(n, (w_1, w_2)) \in \text{V}_{\text{untrusted}}(W)$ and $(\text{nonLinear}(w_1) \text{ or } \text{nonLinear}(w_2))$, then $(n, (w_1, w_2)) \in \text{V}_{\text{untrusted}}(\text{purePart}(W))$.

Proof. Follows easily by inspecting the definitions of $\text{exec}_{\text{code}, \square}$, $\text{readCondition}$, $\text{addressable}$, $\text{writeCondition}$, $\text{executeCondition}$ and $\text{readXCondition}$ and using Lemma 7.

Lemma 30 (permission-based conditions shrinkable). If $A = [b_1, e_1] \cup [b_2, e_2]$, then

• If $(n, A) \in \text{readCondition}^{\square, gc}(l, W)$ and $\emptyset \neq A' \subseteq A$, then $(n, A') \in \text{readCondition}^{\square, gc}(l, W)$.
• If $(n, A) \in \text{writeCondition}^{\square, gc}(l, W)$ and $\emptyset \neq A' \subseteq A$, then $(n, A') \in \text{writeCondition}^{\square, gc}(l, W)$.
• If $(n, A) \in \text{readXCondition}^{\square, gc}(W)$ and $A' \subseteq A$, then $(n, A') \in \text{readXCondition}^{\square, gc}(W)$.
• If $(n, A) \in \text{executeCondition}^{\square, gc}(W)$ and $\emptyset \neq A' \subseteq A$, then $(n, A') \in \text{executeCondition}^{\square, gc}(W)$.

Proof. Follows easily from the definitions. In the case for executeCondition$^{\square, gc}$ where $l = \text{linear}$, we have a partition of the world into $W = \oplus_{a \in A} W_a(a)$ that we need to convert into a partition $W = \oplus_{a \in A'} W'_a(a)$. It suffices construct new $W'_a$ by taking $W_a$ but adding $\oplus_{a \in (A \setminus A')} a$ to an $W_a(a)$ for an arbitrary $a \in A'$ to make this work.

Lemma 31 (permission-based conditions splitable). If $A = [b_1, e_1] \cup [b_2, e_2]$, then

• If $(n, A) \in \text{readCondition}^{\square, gc}(l, W)$, then there exists $W_1, W_2$ such that $W = W_1 \oplus W_2$ such that $(n, [b_1, e_1]) \in \text{readCondition}^{\square, gc}(l, W_1)$ and $(n, [b_2, e_2]) \in \text{readCondition}^{\square, gc}(l, W_2)$.
• If $(n, A) \in \text{writeCondition}^{\square, gc}(l, W)$, then there exists $W_1, W_2$ such that $W = W_1 \oplus W_2$ such that $(n, [b_1, e_1]) \in \text{writeCondition}^{\square, gc}(l, W_1)$ and $(n, [b_2, e_2]) \in \text{writeCondition}^{\square, gc}(l, W_2)$.
• If $(n, A) \in \text{executeCondition}^{\square, gc}(W)$, then there exists $W_1, W_2$ such that $W = W_1 \oplus W_2$ such that $(n, [b_1, e_1]) \in \text{executeCondition}^{\square, gc}(W_1)$ and $(n, [b_2, e_2]) \in \text{executeCondition}^{\square, gc}(W_2)$.
Proof. If \( l = \text{normal} \), then Lemma \([29]\) tells us that it suffices to consider pure worlds \( W = \text{purePart}(W) \) and Lemma \([8]\) tells us that then \( W = W \oplus W \). The results then follow from the previous Lemma \([30]\).

As such, we can limit ourselves to the case where \( l = \text{linear} \).

The read- and write-conditions then require separate islands for each individual address in \( A \) and we can distribute ownership of those islands according to whether those addresses are in \([b_1, e_1]\) or \([b_2, e_2]\), i.e. make the island \( W.\text{heap}(\cdot) \) with \( R(\cdot) = \{a\} \) spatial-owned in \( W_1 \) and spatial in \( W_2 \) if \( a \in [b_1, e_1] \) and vice versa. It is then easy to check that our results hold.

For the execute condition, we get a partition of the world into \( W = \bigoplus_{a \in A} W_a(a) \) and we can define \( W_1 = \bigoplus_{a \in [b_1, e_1]} W_a(a) \) and likewise for \( W_2 \). The results then follow easily from the definitions. \(\square\)

**Lemma 32** (permission-based conditions splicable). If

- \([b, e] = [b_1, e_1] \uplus [b_2, e_2]\)
- \([b, e] \not\in T_A\)
- \(W_1 \oplus W_2\) is defined.

then

- \(\text{If } (n, [b_1, e_1]) \in \text{readCondition}^\Box^{\text{gc}}(l, W_1) \text{ and } (n, [b_2, e_2]) \in \text{readCondition}^\Box^{\text{gc}}(l, W_2), \text{ then } (n, [b, e]) \in \text{readCondition}^\Box^{\text{gc}}(l, W_1 \oplus W_2)\)
- \(\text{If } (n, [b_1, e_1]) \in \text{writeCondition}^\Box^{\text{gc}}(l, W_1) \text{ and } (n, [b_2, e_2]) \in \text{writeCondition}^\Box^{\text{gc}}(l, W_2), \text{ then } (n, [b, e]) \in \text{writeCondition}^\Box^{\text{gc}}(l, W_1 \oplus W_2)\)
- \(\text{If } (n, [b_1, e_1]) \in \text{executeCondition}^\Box^{\text{gc}}(W_1) \text{ and } (n, [b_2, e_2]) \in \text{executeCondition}^\Box^{\text{gc}}(W_2), \text{ then } (n, [b, e]) \in \text{executeCondition}^\Box^{\text{gc}}(W_1 \oplus W_2)\)

If

- \([b, e] = [b_1, e_1] \uplus [b_2, e_2]\)
- \(W_1 \oplus W_2 \oplus W_M\) is defined.
- \(\text{msg}_s, \text{stk}, \text{msg\_stk}, \text{msg\_T} ;^\Box^{\text{gc}} W_M\)

then

- \(\text{If } (n, [b_1, e_1]) \in \text{readXCondition}^\Box^{\text{gc}}(l, W_1) \text{ and } (n, [b_2, e_2]) \in \text{readXCondition}^\Box^{\text{gc}}(l, W_2), \text{ then } (n, [b, e]) \in \text{readXCondition}^\Box^{\text{gc}}(l, W_1 \oplus W_2)\)
- \(\text{If } [b, e] = [b_1, e_1] \uplus [b_2, e_2], \text{ then }\)
- \(\text{If } (n, [b_1, e_1]) \in \text{stackReadCondition}^\Box^{\text{gc}}(W_1) \text{ and } (n, [b_2, e_2]) \in \text{stackReadCondition}^\Box^{\text{gc}}(W_2), \text{ then } (n, [b, e]) \in \text{stackReadCondition}^\Box^{\text{gc}}(W_1 \oplus W_2)\)
- \(\text{If } (n, [b_1, e_1]) \in \text{stackWriteCondition}^\Box^{\text{gc}}(W_1) \text{ and } (n, [b_2, e_2]) \in \text{stackWriteCondition}^\Box^{\text{gc}}(W_2), \text{ then } (n, [b, e]) \in \text{stackWriteCondition}^\Box^{\text{gc}}(W_1 \oplus W_2)\).

\(\square\)

Proof. The results for \(\text{readCondition}^\Box^{\text{gc}}, \text{writeCondition}^\Box^{\text{gc}}, \text{stackReadCondition}^\Box^{\text{gc}}\) and \(\text{stackWriteCondition}^\Box^{\text{gc}}\) follow by taking the union of the two sets \( S \), and the union of \( R \) for every \( r \).

The result for \(\text{executeCondition}^\Box^{\text{gc}}\) follows easily by definition and using Lemmas \([8]\) and \([10]\).

For \(\text{readXCondition}^\Box^{\text{gc}}\), we get \( r_i \in \text{addressable}(\text{normal}, W_i) \) such that \( W_i.\text{heap}(r_i) \equiv \text{code}_{i,b}^{\text{code}, \Box^{\text{gc}}} \) such that \( \text{dom}(\text{code}_{i}) \supseteq [b_i, e_i] \). From \( \text{msg}_s, \text{stk}, \text{msg\_stk}, \text{msg\_T} ;^\Box^{\text{gc}} W_M \), it follows that \( \text{code}_{i} = [b_1', e_1'] \) and if \( r_1 \neq r_2 \), then \( [b_1' - 1, e_1' + 1] \not\subseteq [b_2' - 1, e_2' + 1] \). Because \([b, e] = [b_1, e_1] \uplus [b_2, e_2] \) and \([b_1', e_1'] = \text{dom}(\text{code}_{i}) \supseteq [b_i, e_i] \), we can derive that \( r_1 = r_2 \). The result then follows by definition of \(\text{readXCondition}^\Box^{\text{gc}}\). \(\square\)

**Lemma 33** (FTLR for component code capabilities). If

- \((n, \text{dom}(\text{msg\_code})) \in \text{readCondition}^\Box^{\text{gc}}(W)\)
- \((\text{tst} = \text{untrusted \ and } \text{dom}(\text{msg\_code} \not\in T_A)) \text{ or } (\text{tst} = \text{trusted \ and } \text{dom}(\text{msg\_code}) \subseteq T_A)\)
- \(w = ((\text{RX}, \text{normal}), b, e, a)\)
- \([b, e] \subseteq \text{dom}(\text{msg\_code})\)

then \((n, (w, w)) \in V_{\text{tst}}^{\Box^{\text{gc}}}(W)\). \(\square\)
Proof. We distinguish two cases:

- \( \text{tst} = \text{untrusted} \) and \( \text{dom}(\text{ms}_\text{code}) \neq T_A \): By definition of \( \mathcal{V}_{\text{untrusted}}^{\Box,gc}(W) \), it suffices to show that:
  - \([b, e] \neq T_A \): follows directly from \([b, e] \subseteq \text{dom}(\text{ms}_\text{code}) \) and \( \text{dom}(\text{ms}_\text{code}) \neq T_A \).
  - \((n, [b, e]) \in \text{readCondition}^{\Box,gc}(\text{normal}, W) \): this follows from Lemma 26.
  - \((n, [b, e]) \in \text{readXCondition}^{\Box,gc}(W) \): by assumption and Lemma 30.
  - \((n, [b, e]) \in \text{executeCondition}^{\Box,gc}(W) \): take \( n' < n, W' \supseteq \text{purePart}(W) \), \( a' \in [b', e'] \subseteq [b, e] \) and \( w' = ((\text{RX}, \text{normal}), b', e', a') \). Then we need to show that \( (n', (w', w')) \in \mathcal{E}^{\Box,gc}(W') \). This now follows immediately from the (regular) FTLR (Theorem 2), using the two above points, Lemma 44 and Lemma 30.
  - \( \text{ms}_\text{code} \neq T_A \): By definition of \( \mathcal{V}_{\text{trusted}}^{\Box,gc}(W) \) and Lemma 30 with the assumption that \((n, \text{dom}(\text{ms}_\text{code})) \in \text{readCondition}^{\Box,gc}(W) \).

\( \square \)

Lemma 34 (FTLR for component code-values). If

\[
\mathcal{V}_{\text{ret}, \text{ret-owned}, \text{ret clos}, T_A} \vdash \text{comp-code } w
\]

and

- \( \sigma_{\text{ret}} \subseteq \sigma_{\text{glob ret}} \) and \( \sigma_{\text{clos}} \subseteq \sigma_{\text{glob clos}} \)
- \( W.\text{heap}(r_{\text{code}}) = (\text{code}, \Box, \text{ms}_\text{code}, gc) \)
- \( (\text{dom}(\text{ms}_\text{code}) \neq T_A \) and \( \text{tst} = \text{untrusted} \) or \( (\text{dom}(\text{ms}_\text{code}) \subseteq T_A \) and \( \text{tst} = \text{trusted} \))

then \( (n, [w, w]) \in \mathcal{V}_{\text{trusted}}^{\Box,gc}(W) \).

Proof. By induction on

\[
\mathcal{V}_{\text{ret}, \text{ret-owned}, \text{clos}, T_A} \vdash \text{comp-code } w
\]

There are two cases to consider:

- \( \text{ms}_\text{code}(a) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \) and \( [\sigma_b, \sigma_e] \subseteq (\sigma_{\text{ret}} \cup \sigma_{\text{clos}}) \):

We distinguish two cases:

  - \( \text{tst} = \text{trusted} \):
    By definition of \( \mathcal{V}_{\text{trusted}}^{\Box,gc} \), it suffices to prove that \( W.\text{heap}(r_{\text{code}}) = (\text{code}, \Box, \text{ms}_\text{code}, gc) \) and \( \text{dom}(\text{code}) \subseteq T_A \) and \( [\sigma_b, \sigma_e] \subseteq (\sigma_{\text{ret}} \cup \sigma_{\text{clos}}) \), all of which follow by assumption.
  - \( \text{tst} = \text{untrusted} \):
    In this case, we know that \( \sigma_{\text{ret}} = \emptyset \).
    By definition of \( \mathcal{V}_{\text{untrusted}}^{\Box,gc} \), it suffices to prove that

    \[
    \forall a' \in [\sigma_b, \sigma_e], \exists r \in \text{dom}(W.\text{heap}). W.\text{heap}(r) = (\text{pure}, ... H_\sigma) \text{ and } H_\sigma \sigma' \in \mathcal{V}_{\text{untrusted}}^{\Box,gc}
    \]

    We take \( r = r_{\text{code}} \) and the result then follows from Lemma 28.

- \( \text{ms}_\text{code}(a) \in \mathbb{Z} \)

  \[
  ([a \cdots a + \text{call_len} - 1] \subseteq T_A \land \text{ms}_\text{code}([a \cdots a + \text{call_len} - 1]) = \text{call off } \text{pc off } \text{r1 r2}) \Rightarrow \\
  (\text{ms}_\text{code}(a + \text{off } \text{pc}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \land \sigma_b + \text{off } \sigma \in \sigma_{\text{ret owned}})
  \]

- Using \( \text{ms}_\text{code}(a) \in \mathbb{Z} \), the result follows easily by definition.

\( \square \)

Lemma 35 (FTLR for component data-values). If

\[
\text{dom}(\text{ms}_\text{code}), A_{\text{own}}, A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp-value } w
\]

and

- \( (n, A_{\text{non-linear}}) \in \text{readCondition}^{\Box,gc}(\text{normal}, W) \)
We have the following cases:

- $n, A_{\text{non-linear}} \in \text{writeCondition}^{\square, gc}(\text{normal}, W)$
- $n, A_{\text{own}} \in \text{readCondition}^{\square, gc}(\text{linear}, W)$
- $n, A_{\text{own}} \in \text{writeCondition}^{\square, gc}(\text{linear}, W)$
- $\text{W.heap}(r_{\text{code}}) = R_{\text{code}} \sqcap \text{heap}, \text{ms}_{\text{code}}, gc$
- $\text{dom}(\text{ms}_{\text{code}}) \not\subseteq T_A$ or $\text{dom}(\text{ms}_{\text{code}}) \subseteq T_A$
- $(A_{\text{own}} \cup A_{\text{non-linear}}) \not\subseteq T_A$

then $(n, (w, w)) \in V_{\text{untrusted}}(W)$

Proof. By induction on the judgement

$$\text{dom}(\text{ms}_{\text{code}}), A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma}_{\text{ret}}, \overline{\sigma}_{\text{clos}} \vdash \text{comp-value } w$$

We have the following cases:

- $w = z$: result follows trivially
- $w = ((\text{perm}, l), b, e, a)$, $\text{perm} \subseteq \text{rw}$, $l = \text{linear} \Rightarrow \emptyset \subseteq A_{\text{own}}$ and $l = \text{normal} \Rightarrow [b, e] \subseteq A_{\text{non-linear}}$

By definition of $V_{\text{untrusted}}^{\square, gc}$, it suffices to prove that $[b, e] \not\subseteq T_A$, $(n, [b, e]) \in \text{readCondition}^{\square, gc}(l, W)$ and $(n, [b, e]) \in \text{writeCondition}^{\square, gc}(l, W)$. The result then follows easily from the assumptions, with Lemma 30.

- $w = \text{sealed}(\sigma, sc)$ and

$$\text{dom}(\text{ms}_{\text{code}}), A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma}_{\text{ret}}, \overline{\sigma}_{\text{clos}} \vdash \text{comp-value } sc$$

and $\sigma \in \overline{\sigma}_{\text{clos}}$.

First, we have by induction that $(n, (sc, sc)) \in V_{\text{untrusted}}^{\square, gc}(W)$. By definition of $V_{\text{untrusted}}^{\square, gc}$ and by choosing island $r = r_{\text{code}}$, it suffices to prove that

- $(n', (sc, sc)) \in H_{\sigma} \sigma \xi^{-1}(W)$ for all $n' < n$:

  - By definition of $V_{\text{untrusted}}^{\square, gc}$ and $H_{\sigma}^{\square}$ and because we know that $\sigma \in \overline{\sigma}_{\text{clos}}$, it suffices to prove that one of the following holds:

    * $(\text{dom}(\text{code}) \not\subseteq T_A$ and $(n', (sc, sc)) \in V_{\text{untrusted}}^{\square, gc}(\xi^{-1}(W))$
    * $(\text{dom}(\text{code}) \subseteq T_A$ and $(n', (sc, sc)) \in V_{\text{trusted}}^{\square, gc}(\xi^{-1}(W))$

    Both follow since $\xi^{-1}(W) = W$ and $V_{\text{untrusted}}^{\square, gc} \subseteq V_{\text{trusted}}^{\square, gc} \subseteq W$, dom(\text{ms}_{\text{code}} \# T_A) or dom(\text{ms}_{\text{code}}) \subseteq T_A$, and the above fact that $(n, (sc, sc)) \in V_{\text{untrusted}}^{\square, gc}(W)$ and Lemma 44.

  - $(\text{isLinear}(sc) \iff \text{isLinear}(sc))$: trivially fine

    - If $\text{isLinear}(sc)$, then for all $W' \sqsubseteq W$, $W_o, n' < n$ and $(n', (sc'_{S}, sc'_{T})) \in H_{\sigma} \sigma \xi^{-1}(W_o)$, we have that

      $$(n', (sc, sc'_{S}, sc'_{T})) \in E_{\text{x-jmp}}(W' \oplus W_o)$$

      By definition of $V_{\text{untrusted}}^{\square, gc}$ and $H_{\sigma}$ and because we know that $\sigma \in \overline{\sigma}_{\text{clos}}$, we know that one of the following holds:

      * $(\text{dom}(\text{code}) \# T_A$ and $(n', (sc'_{S}, sc'_{T})) \in V_{\text{untrusted}}^{\square, gc}(\xi^{-1}(W_o))$
      * $(\text{dom}(\text{code}) \subseteq T_A$ and $\overline{\sigma}_{\text{clos}} \subseteq \overline{\sigma}_{\text{glob, clos}}$ and $\overline{\sigma}_{\text{ret}} \subseteq \overline{\sigma}_{\text{glob, ret}}$ and $(n', (sc'_{S}, sc'_{T})) \in E_{\text{x-jmp}}(\xi^{-1}(W_o))$

      with $\text{tst} = \text{trusted iff executable}(sc'_{S})$.

    Lemma 22 now allows us to conclude (using Lemma 47) that:

    $$(n', (sc, sc'_{S}, sc'_{T})) \in E_{\text{x-jmp}}(W' \oplus W_o)$$

    - If nonLinear$(sc_{S})$ then for all $W' \sqsubseteq \text{purePart}(W), W_o, n' < n, (n', (sc'_{S}, sc'_{T})) \in H_{\sigma} \sigma \xi^{-1}(W_o)$, we have that

      $$(n', (sc, sc'_{S}, sc'_{T})) \in E_{\text{x-jmp}}(W' \oplus W_o)$$

      By definition of $V_{\text{untrusted}}^{\square, gc}$ and $H_{\sigma}$ and because we know that $\sigma \in \overline{\sigma}_{\text{clos}}$, we know that one of the following holds:

      * $(\text{dom}(\text{code}) \# T_A$ and $(n', (sc'_{S}, sc'_{T})) \in V_{\text{untrusted}}^{\square, gc}(\xi^{-1}(W_o))$
We have the following cases:

1. $(n', (sc', sc)) \in H'_{\text{code}} \sigma_{\text{ret}} \overline{\sigma_{\text{clo}}} \triangleq \text{code gc} \sigma \xi^{-1}(W)$ for all $n' < n$.

   By definition of $H'_{\text{code}}$, and by choosing island $r = \text{rcode}$, it suffices to prove that
   
   * $(\text{dom}(\text{code}) \neq TA)$ and $(n', (sc', sc)) \in V_{\text{untrusted}}$ for all $n' < n$.
   
   By definition of $H'_{\text{code}}$ and $H'_{\text{code}}$ and because we know that $\sigma \in \overline{\sigma_{\text{clo}}}$, it suffices to prove that one of the following holds:
   
   * $(\text{dom}(\text{code}) \leq TA)$ and $(n', (sc', sc)) \in V_{\text{trusted}}$ for all $n' < n$.

   Take $\text{tst} = \text{untrusted}$ iff $\text{dom}(\text{mcode} \neq TA)$ and $\text{tst} = \text{trusted}$ iff $\text{dom}(\text{mcode}) \leq TA$. Since $\xi^{-1}(W) = W$ and we know by assumption that $\text{dom}(\text{mcode} \neq TA)$ or $\text{dom}(\text{mcode}) \leq TA$, it suffices to prove that $(n', (sc', sc)) \in V_{\text{tst}}$.

   This last fact follows from Lemma 33 using Lemma 44 and the definition of $\text{readXCondition}$ with the assumption that $W.\text{heap}(r_{\text{code}}) = \sigma_{\text{ret}} \overline{\sigma_{\text{clo}}} \text{mcode} \text{gc}$.

2. $(\text{isLinear}(sc) \iff \text{isLinear}(sc))$: trivially fine

   For all $W' \sqsubseteq \text{purePart}(W)$, $n' < n$, $(n', (sc', sc')) \in H'_{\text{code}} \sigma_{\text{ret}} \overline{\sigma_{\text{clo}}} \triangleq \text{code gc} \sigma \xi^{-1}(W)$, we have that
   
   $(n', (sc, sc', sc)) \in E_{\text{trf}}(W' \oplus W)$

   From one of our assumptions, we know that $sc$ behaves reasonably up to $n$ steps if $\text{dom}(\text{mcode}) \subseteq TA$. Theorem 2 now tells us that $(n, (sc, sc)) \in E_{\text{trf}}(\text{purePart}(W))$, using the definition of $\text{readXCondition}$ with the assumption that $W.\text{heap}(r_{\text{code}}) = \sigma_{\text{ret}} \overline{\sigma_{\text{clo}}} \text{mcode} \text{gc}$.

   Since $\sigma \in \overline{\sigma_{\text{clo}}}$, it follows from $(n', (sc', sc')) \in H'_{\text{code}} \sigma_{\text{ret}} \overline{\sigma_{\text{clo}}} \triangleq \text{code gc} \sigma \xi^{-1}(W)$ that

   $(n', (sc', sc')) \in V_{\text{tst}} \text{gc} (W)$

   with $\text{tst} = \text{untrusted}$ iff $\text{dom}(\text{code} \neq TA)$ or $\text{nonExecutable}(sc')$ and $\text{tst} = \text{trusted}$ iff $\text{dom}(\text{code}) \subseteq TA$ and $\text{execuable}(sc')$.

   The result now follows from Lemma 24.
In this case, the result follows from Lemma 35, using the fact that \( \text{readCondition} \Box gc() \) and \( \text{writeCondition} \Box gc() \) follow trivially for the empty \( A_{\text{own}} \) and a similar observation about the empty imports.

Lemma 37 (FTLR for components). If

- \( gc = (T_A, \text{stk} \_ \text{base}, \sigma_{\text{glob} \_ \text{ret}}, \sigma_{\text{glob} \_ \text{clos}}) \)
- \( \text{comp} \) is a well-formed component, i.e. \( T_A \vdash \text{comp} \)
- One of the following holds:
  - \( \text{dom}(\text{comp}.ms_{\text{code}}) \subseteq T_A \) and \( \text{comp} \) is a reasonable component (see Section 3.7)
  - \( \text{dom}(\text{comp}.ms_{\text{code}}) \nsubseteq T_A \)
- \( \sigma_{\text{ret}} \subseteq \sigma_{\text{glob} \_ \text{ret}} \) and \( \sigma_{\text{clos}} \subseteq \sigma_{\text{glob} \_ \text{clos}} \)

Then there exists a \( W \) such that

- \( (n, (\text{comp}, \text{comp})) \in C^{\Box gc}(W) \)
- \( \text{dom}(W.\text{heap}) \) can be chosen to not include any finite set of region names
- \( \text{dom}(W.\text{free}) = \text{dom}(W.\text{priv}) = \emptyset \)

Proof. If \( \text{comp} \) is of the form \((\text{comp}_0, e_{\text{main} \_ c}, e_{\text{main} \_ d})\), then we have (by definition of component well-formedness) that \( e_{\text{main} \_ c}, e_{\text{main} \_ d} \subseteq \text{comp}_0.\text{export} \), as required by \( C^{\Box gc}(W) \). Hence, we can restrict our attention to components of the form \( \text{comp}_0 \).

Take

\[
\text{comp} = (ms_{\text{code}} \uplus ms_{\text{pad}}, ms_{\text{data}}, import, export, \sigma_{\text{ret}}, \sigma_{\text{clos}}, A_{\text{linear}})
\]

Note that confusingly, \( \text{comp}.ms_{\text{code}} = ms_{\text{code}} \uplus ms_{\text{pad}} \), so the assumption about \( \text{dom}(\text{comp}.ms_{\text{code}}) \) should be interpreted properly. We then know from \( T_A \vdash \text{comp} \) that

- \( \text{dom}(ms_{\text{code}}) = [b, e] \)
- \( [b - 1, e + 1] \nsubseteq \text{dom}(ms_{\text{data}}) \)
- \( ms_{\text{pad}} = [b - 1 \mapsto 0] \uplus [e + 1 \mapsto 0] \)
- \( \sigma_{\text{ret}}, \sigma_{\text{clos}}, T_A \vdash \text{comp} - \text{code} ms_{\text{code}} \)
- \( \exists A_{\text{own}} : \text{dom}(ms_{\text{data}}) \rightarrow \mathcal{P}(\text{dom}(ms_{\text{data}})) \)
- \( \text{dom}(ms_{\text{data}}) = A_{\text{non} \_ \text{linear}} \uplus A_{\text{linear}} \)
- \( A_{\text{linear}} = \bigcup_{a \in \text{dom}(ms_{\text{data}})} A_{\text{own}}(a) \)
- \( \forall a \in \text{dom}(ms_{\text{data}}), \text{dom}(ms_{\text{code}}), A_{\text{own}}(a), A_{\text{non} \_ \text{linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp} - \text{value} ms_{\text{data}}(a) \)

\[
\text{dom}(ms_{\text{code}}), A_{\text{non} \_ \text{linear}}, \sigma_{\text{ret}}, \sigma_{\text{clos}} \vdash \text{comp} - \text{export} e_{\text{export}}
\]

- \( (\text{dom}(ms_{\text{code}}) \subseteq T_A) \lor (\text{dom}(ms_{\text{code}}) \nsubseteq T_A \land \sigma_{\text{ret}} = \emptyset) \)
- \( \text{dom}(ms_{\text{data}}) \nsubseteq T_A \)

Take \( import = a_{\text{import}} \leftrightarrow s_{\text{import}} \).

Now take \( W \) such that

- \( \text{dom}(W.\text{free}) = \text{dom}(W.\text{priv}) = \emptyset \)
- \( W.\text{heap}(r_{\text{code}}) = \iota_{\text{code}, \Box gc, r_{\text{code}}.ms_{\text{code}}, gc} \)
- there exist \( r_{\text{addr}} : \text{dom}(ms_{\text{data}}) \leftrightarrow \text{dom}(W.\text{heap}) \) such that
For all $a \in \text{dom}(ms_{\text{data}})$, $W.\text{heap}(r_{\text{addr}}(a)) = i_{\text{std},l,\Box}^{\ell}$ with $(l = \text{pure} \text{ if } a \notin A_{\text{linear}})$ and $(l = \text{spatial\_owned} \text{ if } a \in A_{\text{linear}})$

with $r_{\text{code}}$ and $r_a$ chosen according to the given restriction.

It now remains to prove that $(n, (\text{comp}, \text{comp})) \in C^{\Box,gc} W$. Take $W' \supseteq W$ and assume that

$$\langle n', (w_{\text{import}}, w_{\text{import}}) \rangle \in \mathcal{V}^{\Box,gc}_{\text{untrusted}}(\text{purePart}(W')) \text{ for all } n' < n$$

and

$$ms'_{\text{data}} = ms_{\text{data}}[w_{\text{import}} \mapsto w_{\text{import}}]$$

We need to show

- \((n, (\sigma_{\text{ret}} \sqcup \sigma_{\text{cloi}}, ms_{\text{pad}} \sqcup ms_{\text{code}} \sqcup ms'_{\text{data}}, ms_{\text{pad}} \sqcup ms_{\text{code}} \sqcup ms'_{\text{data}})) \in \mathcal{H}(W.\text{heap})(W')\)

Take $R_{ms} : \text{dom}(active(W.\text{heap})) \rightarrow \text{MemorySegment} \times \text{MemorySegment}$ such that

- $R_{ms}(r_a) = (ms_{\text{data}}|_a, ms_{\text{data}}|_a)$.
- $R_{ms}(r_{\text{code}}) = (ms_{\text{pad}} \sqcup ms_{\text{code}}, ms_{\text{pad}} \sqcup ms_{\text{code}})$

Take $R'_W' : \text{dom}(active(W.\text{heap})) \rightarrow \text{World}$ such that

- $R'_W'(r_a) = \text{purePart}(W)[\text{heap}, r_a' \mapsto i_{\text{std},

| spatial\_owned,\Box}^{\ell}|_a \in A_{\text{non}}(a)$
- $R'_W'(r_{\text{code}}) = \text{purePart}(W)$

Since then $W = \bigoplus_{r \in \text{dom}(active(W.\text{heap}))} R'_W'(r)$, we can use Lemma 5 to construct an $RW$ with $W' = \bigoplus_{r \in \text{dom}(active(W.\text{heap}))} R'_W'(r)$ and $RW(r) \supseteq R'_W'(r)$ for all $r$. Finally, take $R_{\text{rel}} : \text{dom}(active(W.\text{heap})) \rightarrow \mathcal{F}(\text{Seal})$ to satisfy

- $R_{\text{rel}}(r_a) = \emptyset$
- $R_{\text{rel}}(r_{\text{code}}) = \sigma_{\text{ret}} \sqcup \sigma_{\text{cloi}}$

We then need to prove that

- \((n', (ms'_{\text{data}}|_a), ms_{\text{data}}|_a)) \in i_{\text{std},l,\Box}^{\ell}(H^{-1}(RW(r_a))) \text{ for all } n' < n\): Take $n' < n$. By definition, it suffices to prove that, \((n', (ms'_{\text{data}}|_a), ms_{\text{data}}|_a)) \in \mathcal{V}_{\text{untrusted}}^{\Box,gc}(RW(r_a))\).

If \((a \leftarrow_{\text{import}}) \in ms_{\text{import}}\), then we know that $ms'_{\text{data}}|_a$ is the corresponding $w_{\text{import}}$ and the result is fine by the assumption that

$$\langle n', (w_{\text{import}}, w_{\text{import}}) \rangle \in \mathcal{V}_{\text{untrusted}}^{\Box,gc}(\text{purePart}(W'))$$

together with \((\text{purePart}(W'))\) and the fact that $RW(a) \supseteq \text{purePart}(W')$ (by Lemma 13).

Otherwise, $ms_{\text{data}}(a) = ms_{\text{data}}(a)$ and we know from our assumptions that

$$\text{dom}(ms_{\text{code}}), A_{\text{own}}(a), A_{\text{non-linear}}, \sigma_{\text{ret}}, \sigma_{\text{cloi}} \vdash \text{comp-value } ms_{\text{data}}(a)$$

By Lemma 55 it then suffices to prove that

- \((n', A_{\text{non-linear}}) \in \text{readCondition}^{\Box,gc}(\text{normal}, RW(a))\): follows by Lemma 47 using the fact that $RW(a) \supseteq R'_W(a)$ and by definition, using the choice of $W$, $R'_W(a)$
- \((n', A_{\text{non-linear}}) \in \text{writeCondition}^{\Box,gc}(\text{normal}, RW(a))\): follows by Lemma 47 using the fact that $RW(a) \supseteq R'_W(a)$ and by definition, using the choice of $W$, $R'_W(a)$
- \((n', A_{\text{own}}(a)) \in \text{readCondition}^{\Box,gc}(\text{linear}, RW(a))\): follows by Lemma 47 using the fact that $RW(a) \supseteq R'_W(a)$ and by definition, using the choice of $W$, $R'_W(a)$
- \((n', A_{\text{own}}(a)) \in \text{writeCondition}^{\Box,gc}(\text{linear}, RW(a))\): follows by Lemma 47 using the fact that $RW(a) \supseteq R'_W(a)$ and by definition, using the choice of $W$, $R'_W(a)$
- \(RW(a).\text{heap}(r_{\text{code}}) = i_{\text{code},\Box}^{\ell}(\text{code}, \text{code}, ms_{\text{code}}, \Box)^{\Box,gc}\): follows from the choice of $W$ and $R'_W(a)$ and the fact that $RW(a) \supseteq R'_W(a)$
- \(\text{dom}(ms_{\text{code}}) \# T_A \text{ or } \text{dom}(ms_{\text{code}}) \subseteq T_A\): by assumption.
- \((n', (ms_{\text{pad}} \sqcup ms_{\text{code}}, ms_{\text{pad}} \sqcup ms_{\text{code}})) \in \mathcal{V}_{\text{untrusted}}^{\Box,gc}(H^{-1}(\text{purePart}(W'))) \text{ for all } n' < n\): Take $n' < n$. We take $tst = \text{trusted}$ if $[b - 1, e + 1] \subseteq T_A$ and $tst = \text{untrusted}$ otherwise. By definition, we need to prove that:

- \(\text{dom}(\text{code}) = [b, e]\): by assumption.
- \(([b - 1, e + 1] \subseteq T_A \land \text{tst} = \text{trusted}) \lor ([b - 1, e + 1] \# T_A \land \text{tst} = \text{untrusted})\): by assumption and choice of $tst$.\)
* \( \sigma_{ret}, \sigma_{clo}, T_A \vdash \text{comp-code code} \): by assumption.
* \( m_{\text{pad}} = \{ b - 1 \mapsto 0 \cup \{ c + 1 \mapsto 0 \} \): by assumption
* \( \forall a \in \text{dom(code)}.(n', (\text{code}(a), \text{code}(a))) \in Y_{\text{trust}}^{\text{gc}}(\text{purePart}(\xi^{-1}(\text{purePart}(W')))) \): Note first that \( \text{purePart}(\xi^{-1}(\text{purePart}(W'))) = \text{purePart}(W') \).

By Lemma 34 it suffices to prove that:

1. \( W'.\text{heap}(r_{\text{code}}) = \text{code}_W[A_{\text{trust}}, g_{\text{code}}] \): follows by choice of \( W \) and the fact that \( W' \supseteq W \).
2. \( \text{dom}(m_{\text{code}}) \# T_A \) and \( \text{tst} = \text{untrusted} \) or \( \text{dom}(m_{\text{code}}) \subseteq T_A \) and \( \text{tst} = \text{trusted} \): by assumption and choice of \( \text{tst} \)

\[
\text{dom}(\text{code}_W[A_{\text{trust}}, g_{\text{code}}], H_A) = \sigma_{ret} \uplus \sigma_{clo}
\]

This follows easily from the definition.

\( (n, (w_{\text{export}}, w_{\text{export}})) \in \mathcal{Y}_{\text{trust}}^{\text{gc}}(\text{purePart}(W')) \):

We know from our assumptions that

\[
\text{dom}(m_{\text{code}}), A_{\text{non-linear}}, \sigma_{ret}, \sigma_{clo} \vdash \text{comp-export} w_{\text{export}}
\]

By Lemma 36 it then suffices to prove that

- \((n, A_{\text{non-linear}}) \in \text{readCondition}^{\text{gc}}(\text{normal}, \text{purePart}(W')) \): follows by Lemma 47 using the fact that \( R_W(a) \supseteq R_W(a) \) and by definition, using the choice of \( W, R_W(a) \)
- \((n, A_{\text{non-linear}}) \in \text{writeCondition}^{\text{gc}}(\text{normal}, \text{purePart}(W')) \): follows by Lemma 47 using the fact that \( R_W(a) \supseteq R_W(a) \) and by definition, using the choice of \( W, R_W(a) \)
- \( \text{purePart}(W').\text{heap}(r_{\text{code}}) = \text{code}_W[A_{\text{trust}}, g_{\text{code}}] \): follows from the choice of \( W \) and the fact that \( W' \supseteq W \)
- \( \text{dom}(m_{\text{code}}) \# T_A \) or \( \text{dom}(m_{\text{code}}) \subseteq T_A \): by assumption.
- If \( w_{\text{export}} = \text{sealed}(\sigma, sc) \) with \( \text{dom}(m_{\text{code}}) \subseteq T_A, \sigma \in \sigma_{clo} \) and \( \text{executable}(sc) \), then \( sc \) behaves reasonably up to \( n \) steps:
  - This follows directly from the fact that \( \text{comp} \) is a reasonable component.
- \( A_{\text{non-linear}} \# T_A \): This follows from the facts that \( A_{\text{non-linear}} \subseteq \text{dom}(m_{\text{data}}) \# T_A \)

\( \square \)

Note: the trusted case of the above lemma can be considered as a compiler correctness result. The untrusted case can be considered as a back-translation correctness result.

### 3.11 Related execution configurations

\[
\mathcal{E}^{\text{comp}, g_{\text{fc}}}(W) = \left\{ \left( n, \left( m_{S}, r_{S}, g_{S}, m_{\text{stk}} \right), \left( m_{T}, r_{T} \right) \right) \mid \begin{array}{l}
gc = (T_A, \text{stk}_{\text{base}}) \text{ and} \\
\exists W_M, W_R, W_{\text{pc}}, W = W_M \oplus W_R \oplus W_{\text{pc}} \text{ and} \\
(n, (\text{reg}_{\text{pc}}, \text{reg}_{S}(r_{\text{data}})), (\text{reg}_{T}(r_{\text{data}}))) \in \mathcal{E}_{\text{simp}}^{\text{gc}}(W_{\text{pc}}) \text{ and} \\
\text{reg}_{S}(\text{pc}) \neq \text{ret-ptr-code}() \land \text{reg}_{S}(r_{\text{data}}) \neq \text{ret-ptr-data}() \land \text{nonExecutable}(\text{reg}_{S}(r_{\text{data}})) \land \text{nonExecutable}(\text{reg}_{T}(r_{\text{data}})) \land \\
ms_{S}, m_{\text{stk}}, g_{S}, g_{T} \text{ WC M and} \\
(n, (\text{reg}_{S}, \text{reg}_{T})) \in \mathcal{R}_{\text{untrusted}}(\{r_{\text{data}}\})(W_R)
\end{array} \right\}
\]

**Lemma 38** (Compatibility lemma for initial execution configuration construction). If

- \((n, (\text{comp}, \text{comp})) \in \mathcal{E}^{\text{comp}, g_{\text{fc}}}(W)\)
- \(\text{dom}(W.\text{free}) = \text{dom}(W.\text{priv}) = \emptyset\)
- \(\text{comp} \rightsquigarrow \Phi\),

then \( \exists W' \supseteq W \) such that for all \( n' < n \)

- \((n', (\Phi, \Phi)) \in \mathcal{E}^{\text{comp}, g_{\text{fc}}}(W')\)

\( \blacksquare \)

**Proof.** Take \( \text{comp} = ((m_{\text{code}}, m_{\text{data}}, \text{import}, \text{export}, \sigma_{\text{ret}}, \sigma_{\text{clo}}, A_{\text{linear}}), c_{\text{main},e}, c_{\text{main},d}) \). Then we know from \( \text{comp} \rightsquigarrow \Phi \) that

- \( c_{\text{main},e} = \text{sealed}(\sigma_1, c'_{\text{main},e}) \)
\[ \text{range}(ms_{sstk}) = \{0\} \]
\[ \text{mem} = ms_{code} \uplus ms_{data} \uplus ms_{stk} \]
\[ [b_{stk}, e_{stk}] = \text{dom}(ms_{stk}) \]
\[ [b_{stk} - 1, e_{stk} + 1] \neq \text{dom}(ms_{code}) \cup \text{dom}(ms_{data}) \]
\[ \text{import} = 0 \]
\[ \Phi = (\text{mem}, \text{reg}, 0, ms_{stk}) \]

Now take \( \sigma_{main,c} = b_{stk} - n, c_{stk}, e_{stk} \) (by Lemmas 6, 5, 2 and 4), Lemma 22 tells us that for every \( W' \subseteq W \), if

\[ n' = n - 1 \]
\[ n', (\text{import}, \text{import}) \in V^{\square, gc}_{\text{untrusted}}(\text{purePart}(W')) \]

and \( ms'_{data} = ms_{data} \uplus \text{import} \uplus \text{import} \) then

\[ (n, (\sigma_{\text{ret}} \uplus \sigma_{\text{clos}}, ms_{code} \uplus ms'_{data}, ms_{code} \uplus ms'_{data})) \in H(W.\text{heap})(W') \]

and

\[ (n, (w_{\text{export}}, w_{\text{export}})) \in V^{\square, gc}_{\text{untrusted}}(\text{purePart}(W')) \]

Since \( \text{import} = 0 \), the former holds vacuously, \( ms'_{data} = ms_{data} \) and the latter two results follow for every \( W' \subseteq W \).

Now take \( r_{a} \) for \( a \in [b_{stk}, e_{stk}] \) arbitrary and

\[ W_{stk}.\text{heap} = 0 \]
\[ W_{stk}.\text{priv} = 0 \]
\[ \text{dom}(W_{stk}.\text{free}) = \{ r_{a} \mid a \in [b_{stk}, e_{stk}] \} \]
\[ W_{stk}.\text{free}(r_{a}) = \ell_{\text{std.spatial.owned}, \square} \]
\[ W' = W \uplus W_{stk} \]
\[ W_{pc} = \text{purePart}(W') \]
\[ W_{R} = \text{purePart}(W) \uplus W_{stk} \]
\[ W_{M} = W \uplus \text{purePart}(W_{stk}) \]

Then \( W' = W_{M} \uplus W_{R} \uplus W_{pc} \) (by Lemmas 6, 5, 2 and 4), \( W_{M} \subseteq W \) (by Lemma 11) and \( W_{pc} \subseteq \text{purePart}(W) \) (by Lemma 11 and 8).

By definition of \( EC^{\square, gc} \), it suffices to prove that

\[ (n', ((\text{reg}(pc), \text{reg}(r_{data})), (\text{reg}(pc), \text{reg}(r_{data})))) \in E_{\text{xjmp}}^{\square, gc}(W_{pc}) \]: Since \( W' \subseteq W \), we know from above that:

\[ (n, (w_{\text{export}}, w_{\text{export}})) \in V^{\square, gc}_{\text{untrusted}}(\text{purePart}(W')) \]

and we have defined \( W_{pc} = \text{purePart}(W') \) and we know that \( \{c_{main,c}, c_{main,d}\} \subseteq c_{\text{export}} \). It follows that

\[ (n, (c_{main,c}, c_{main,c})) \in V^{\square, gc}_{\text{untrusted}}(W_{pc}) \]
\[ (n, (c_{main,d}, c_{main,d})) \in V^{\square, gc}_{\text{untrusted}}(W_{pc}) \]

Since \( W_{pc} = \text{purePart}(W') \), Lemma 22 tells us that for \( n' < n \),

\[ (n', ((c_{main,c}, c_{main,c}), (c_{main,c}, c_{main,d}))) \in E_{\text{xjmp}}^{\square, gc}(W_{pc}) \]

Since \( \text{reg}(pc) = c'_{main,c} \) and \( \text{reg}(r_{data}) = c'_{main,d} \), this is what we set out to prove.
• \( \text{reg}(\text{pc}) \neq \text{ret-ptr-code}(\_), \text{reg}(\text{r}_\text{data}) \neq \text{ret-ptr-data}(\_), \text{nonExecutable}(\text{reg}(\text{r}_\text{data})) \) and \( \text{nonExecutable}(\text{reg}(\text{r}_\text{data})) \): This follows from the facts that \( \text{reg}(\text{pc}) = \epsilon'_\text{main,c}, \text{reg}(\text{r}_\text{data}) = \epsilon'_\text{main,d}, \text{nonExecutable}(\epsilon'_\text{main,d}), \{\epsilon'_\text{main,c}, \epsilon'_\text{main,d}\} \subseteq \mathcal{c}_{\text{export}} \) and the fact that \( \mathcal{c}_{\text{export}} \) are also valid

• \( ms_{\xi}, ms_{\text{stk}}, stk, ms_{\text{T}} \vdash_{\xi}^{gc} W_M \): Since \( W_M \models W \), we have seen above that
\[
(n, (\sigma^\text{reg} \uplus \sigma^\text{clos}, ms_{\text{code}} \uplus ms_{\text{data}}, ms_{\text{code}} \uplus ms_{\text{data}})) \in \mathcal{H}(W_M.\text{heap})(W_M)
\]
and by Lemma \([4]\) also
\[
(n', (\sigma^\text{reg} \uplus \sigma^\text{clos}, ms_{\text{code}} \uplus ms_{\text{data}}, ms_{\text{code}} \uplus ms_{\text{data}})) \in \mathcal{H}(W_M.\text{heap})(W_M)
\]
It suffices to prove that also
\[
- (n', (\emptyset, \emptyset)) \in S^{gc}(\text{purePart}(W_M))
- (n', (ms_{\text{stk}}, ms_{\text{stk}})) \in \mathcal{F}^{gc}(\text{purePart}(W_M))
\]
The former follows vacuously. The latter follows by taking \( R_{ms}(r_a) = (ms_{\text{stk}} | \{a\}, ms_{\text{stk}} | \{a\}) \), \( Rw(r_a) = \text{purePart}(W_M) \) for \( a \in [b_{\text{stk}}, e_{\text{stk}}] \) by Lemma \([1]\) and \([7]\) if we can show that:
\[
(n'', (ms_{\text{stk}} | \{a\}, ms_{\text{stk}} | \{a\})) \in t_{(a)}^{\xi \cdot gc}.H \models (\text{purePart}(W_M))
\]
for all \( n'' < n' \). By definition, it suffices to show that for all \( n'' < n' \), we have that:
\[
(n'', (ms_{\text{stk}}(a), ms_{\text{stk}}(a)) \in V_{\text{untrusted}}^{\xi \cdot gc}(\text{purePart}(W_M))
\]
But \( ms_{\text{stk}}(a) = 0 \), so this follows easily by definition.

• \( (n', (\text{reg}, \text{reg})) \in \mathcal{T}_{\text{untrusted}}^{\xi \cdot gc}(\{\text{r}_\text{data}\})(W_R) \):

We have that \( \text{reg}(\text{RegisterName} \setminus \{\text{pc}, \text{r}_\text{data}, \text{r}_\text{stk}\}) = 0 \), so by definition, it suffices to prove that
\[
(n', (\text{reg}(\text{r}_{\text{stk}}), \text{reg}(\text{r}_{\text{stk}}))) \in V_{\text{untrusted}}^{\xi \cdot gc}(W_R)
\]
Since \( \text{reg}(\text{r}_{\text{stk}}) = \text{stack-ptr}(\text{rw}, b_{\text{stk}}, e_{\text{stk}}, e_{\text{stk}}) \) and \( \text{reg}(\text{r}_{\text{stk}}) = ((\text{rw}, \text{linear}), b_{\text{stk}}, e_{\text{stk}}, e_{\text{stk}}) \), it suffices to prove (by definition) that
\[
(n', [b_{\text{stk}}, e_{\text{stk}}]) \in \text{stackReadCondition}_{\xi \cdot gc}(W_R)
\]
\[
(n', [b_{\text{stk}}, e_{\text{stk}}]) \in \text{stackWriteCondition}_{\xi \cdot gc}(W_R)
\]
For both, we can take \( S = \text{dom}(W_{\text{stk}}.\text{free}) = \{r_a | a \in [b_{\text{stk}}, e_{\text{stk}}]\}, R(r_a) = \{a\} \), and then it suffices to prove that for all \( r_a, W_{\text{r}.\text{free}}(r_a).H \models n^{\text{std.spatial-owned},\xi \cdot gc} \) resp. \( W_{\text{r}.\text{free}}(r_a).H \models n^{\text{std.spatial-owned},\xi \cdot gc} \). Both follow easily since \( W_{\text{r}.\text{free}}(r_a) = W_{\text{stk}}.\text{free}(r_a) = (\xi \cdot gc)_{a \cdot gc} \).

\( \square \)

Lemma \( 39 \) (Adequacy of execution configuration LR). If

\[
- (n, (\Phi_S, \Phi_T)) \in \mathcal{EC}^{\xi \cdot gc}(W)
- i \leq n
- \Phi_S \uparrow_{\xi \cdot gc}
- \Phi_T \downarrow_{\xi \cdot gc}
\]
then \( \Phi_T \downarrow_{\xi \cdot gc} \).

Also, if

\[
- (n, (\Phi_S, \Phi_T)) \in \mathcal{EC}^{\xi \cdot gc}(W)
- i \leq n
- \Phi_T \uparrow_{\xi \cdot gc}
- \Phi_S \downarrow_{\xi \cdot gc}
\]
then \( \Phi_S \uparrow_{\xi \cdot gc} \).

\( \square \)

Proof. First, assume that

\[
- (n, (\Phi_S, \Phi_T)) \in \mathcal{EC}^{\xi \cdot gc}(W)
\]
• $i \leq n$
• $\Phi_S^i \vdash_{gc}$

Assume w.l.o.g. that $\Phi_S = (ms_S, \text{reg}_S, stk, ms_{stk})$, $\Phi_T = (ms_T, \text{reg}_T)$ and $gc = (T_A, stk_{base})$. Then it follows from $(n, (\Phi_S, \Phi_T)) \in EC^{\leq}(W)$ that there exist $W_M, W_R, W_{pc}$ such that

- $W = W_M \oplus W_R \oplus W_{pc}$
- $(n, ((\text{reg}_S(pc), \text{reg}_S(r_{data})), (\text{reg}_T(pc), \text{reg}_T(r_{data})))) \in EC^{\leq, gc}(W_{pc})$
- $\text{reg}_S(pc) \neq \text{ret-ptr-code}(\_), \text{reg}_S(r_{data}) \neq \text{ret-ptr-data}(\_), \text{nonExecutable}(\text{reg}_S(r_{data}))$ and $\text{nonExecutable}(\text{reg}_T(r_{data}))$:
  - $ms_S, ms_{stk}, stk, ms_T : gc \triangleright \_ W_M$
  - $(n, (\text{reg}_S, \text{reg}_T)) \in R^{\leq, gc}W_{pc}$

We can then instantiate the conditions from $EC^{\leq, gc}$ with the other conditions from this list to obtain $\Phi'_S, \Phi'_T$ such that

- $\Phi'_S = xjumpResult(\text{reg}_S(pc), \text{reg}_S(r_{data}), \Phi_S)$
- $\Phi'_T = xjumpResult(\text{reg}_T(pc), \text{reg}_T(r_{data}), \Phi_T)$
- $(n, (\Phi'_S, \Phi'_T)) \in O^{\leq}(T_A, stk_{base}, reg_{pc}, ret_{gc}, gc)$

Using the facts that $\text{reg}_S(pc) \neq \text{ret-ptr-code}(\_), \text{reg}_S(r_{data}) \neq \text{ret-ptr-data}(\_)$ and $\text{nonExecutable}(\text{reg}_S(r_{data}))$, we know (by definition of $xjumpResult(\_), \_)$ that

- $\Phi'_S = xjumpResult(\text{reg}_S(pc), \text{reg}_S(r_{data}), \Phi_S) = \Phi_S[\text{reg}.pc \mapsto \text{reg}_S(pc)][\text{reg}.r_{data} \mapsto \text{reg}_S(r_{data})] = \Phi_S$
- $\Phi'_T = xjumpResult(\text{reg}_T(pc), \text{reg}_T(r_{data}), \Phi_T) = \Phi_T[\text{reg}.pc \mapsto \text{reg}_T(pc)][\text{reg}.r_{data} \mapsto \text{reg}_T(r_{data})] = \Phi_T$

From $(n, (\Phi'_S, \Phi'_T)) \in O^{\leq}(T_A, stk_{base}, reg_{pc}, ret_{gc}, gc)$, it follows immediately that if $\Phi_S^i \vdash_{T_A, stk_{base}}^1$ with $i \leq n$, then $\Phi_T^i \vdash_{\_}$ as required.

The proof in the other direction is directly analogous.  

\begin{lemma}[Compatibility lemma for context plugging] If $(n, (C_S, C_T)) \in C^{\leq, gc}(W_1)$ and $(n, (P_S, P_T)) \in C^{\leq, gc}(W_2)$, then $(n', (C_S[P_S], C_T[P_T])) \in EC^{\leq, gc}(W_1 \oplus W_2)$, for all $n' < n$.
\end{lemma}

\begin{proof}
By definition, we have that $C_S \triangleright P_S \triangleright C_S[P_S]$ and $C_T \triangleright P_T \triangleright C_T[P_T]$.

The result follows directly from Lemmas 23 and 38.
\end{proof}

\section{4 Full Abstraction}

\begin{definition}[Source language contextual equivalence] In the source language, we define that $comp_1 \approx_{ctx} comp_2$ iff
\[ \forall E, \emptyset \vdash E \Rightarrow E[\text{comp}_1] \vdash_{\_}^{T_A, i, stk_{base}} E[\text{comp}_2] \vdash_{\_}^{T_A, i, stk_{base}} \]

with $T_A, i = \text{dom}(\text{comp}_i, ms_{\text{code}})$.
\end{definition}

Note that we define source language contextual equivalence with respect to contexts that are not in $T_A$. This means that they are unable to perform calls. We believe this fits with the goal of this work: allow programmers (or better: authors of previous compiler passes) to reason about their target language programs under a special perspective, where all calls can be interpreted as calls that actually behave in a well-bracketed way by the operational semantics. This special perspective is defined by overlaying a different operational semantics on the existing code: the source language semantics. The fact that source contexts cannot make calls themselves is no problem: authors of previous compiler passes should only be able to take the perspective that their own calls are guaranteed to be well-bracketed. It does not matter for them whether calls in the rest of the system are guaranteed to be well-bracketed. Note also that if the trusted code hands out closures, the context can still invoke them with an xjmp, rather than a call. That xjmp may even be the one in the implementation of call if the context uses that implementation. This works perfectly fine, except that the context does not get any well-bracketedness guarantees, but that doesn’t matter.

\begin{definition}[Target language contextual equivalence] In the target language, we define that $comp_1 \approx_{ctx} comp_2$ iff
\[ \forall E, \emptyset \vdash E \Rightarrow E[\text{comp}_1] \vdash \_ E[\text{comp}_2] \vdash \_ \]
\end{definition}

\foreachline{42}
Theorem 3. For reasonable, well-formed components \( \text{comp}_1 \) and \( \text{comp}_2 \) (with respect to \( T_{A,i} = \text{dom}(\text{comp}_i, ms_{\text{code}}) \), respectively), we have

\[
\text{comp}_1 \simeq_{\text{ctx}} \text{comp}_2
\]

\[
\Downarrow
\]

\[
\text{comp}_1 \simeq_{\text{ctx}} \text{comp}_2
\]

Proof.

- Consider first the upward arrow. Assume \( \text{comp}_1 \simeq_{\text{ctx}} \text{comp}_2 \).

Take a \( \mathcal{C} \) such that \( \emptyset \vdash \mathcal{C} \), take \( T_{A,i} = \text{dom}(\text{comp}_i, ms_{\text{code}}) \), \( \sigma_{\text{glob-ret}} = \sigma_1 \sigma_{\text{ret}} \) and \( \bar{\sigma}_{\text{glob-clos}} = \sigma_1 \sigma_{\text{clos}} \) and we will prove that \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \).

By symmetry, we can assume w.l.o.g. that \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \) and prove that \( \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \). Note that this implies that \( \mathcal{C} \) is a valid context for both \( \text{comp}_1 \) and \( \text{comp}_2 \).

First, we show that also \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \).

Take \( n \) the amount of steps in the termination of \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \). It follows from Lemma 37 that \( (n + 1, \text{comp}_1, \text{comp}_2) \in \mathcal{C}_{gc} \) for some \( W \) with \( \text{dom}(W, \text{free}) = \emptyset \). It also follows from the same Lemma 37 and Lemma 1 that \( (n + 1, \mathcal{C}, \mathcal{C}) \in \mathcal{C}_{gc} \) for some \( W \) that we can choose such that \( W \uplus W' \) is defined. Lemma 40 then tells us that \( n, (\mathcal{C}[\text{comp}_1], \mathcal{C}[\text{comp}_1]) \in \mathcal{E}\mathcal{C}_{gc}(W \uplus W') \) together with \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \), Lemma 39 tells us that \( \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \), concluding the second direction of the proof.

- The downward arrow is similar.

Assume \( \text{comp}_1 \simeq_{\text{ctx}} \text{comp}_2 \). Take \( T_{A,i} = \text{dom}(\text{comp}_i, ms_{\text{code}}) \), \( \sigma_{\text{glob-ret}} = \sigma_1 \sigma_{\text{ret}} \) and \( \bar{\sigma}_{\text{glob-clos}} = \sigma_1 \sigma_{\text{clos}} \).

Take a \( \mathcal{C} \) such that \( \emptyset \vdash \mathcal{C} \) and we will prove that \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \) and prove that \( \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \). Note that this implies that \( \mathcal{C} \) is a valid context for both \( \text{comp}_1 \) and \( \text{comp}_2 \).

First, we show that also \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \).

Take \( n \) the amount of steps in the termination of \( \mathcal{C}[\text{comp}_1] \downarrow_{\mathcal{C}} \). It follows from Lemma 37 that \( (n + 1, \text{comp}_1, \text{comp}_2) \in \mathcal{C}_{gc} \) for some \( W \) with \( \text{dom}(W, \text{free}) = \emptyset \). It also follows from the same Lemma 37 and Lemma 1 that \( (n + 1, \mathcal{C}, \mathcal{C}) \in \mathcal{C}_{gc} \) for some \( W \) that we can choose such that \( W \uplus W' \) is defined. Lemma 40 then tells us that \( n, (\mathcal{C}[\text{comp}_1], \mathcal{C}[\text{comp}_1]) \in \mathcal{E}\mathcal{C}_{gc}(W \uplus W') \) together with \( \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \), Lemma 39 tells us that \( \mathcal{C}[\text{comp}_2] \downarrow_{\mathcal{C}} \), concluding the second direction of the proof.

\[ \Box \]

5 Lemmas

Lemma 41. If \( r, r' \in \text{dom}(W, \text{heap}) \) and \( W, \text{heap}(r) = (\text{pure}, \_H_s) \) and \( W, \text{heap}(r') = (\text{pure}, \_H_s') \) and \( s \in \text{dom}(H_s) \) and \( s \in \text{dom}(H_s') \) and \( ms_S, stk, ms_{\text{stk}}, ms_{T, \mathcal{C}} \downarrow_{\mathcal{C}} W \) then

\[ H_s = H_s' \text{ and } r = r' \]

\[ \Box \]

Proof. This follows easily by definition of \( ms_S, stk, ms_{\text{stk}}, ms_{T, \mathcal{C}} \downarrow_{\mathcal{C}} W \) and \( H_s \).
Lemma 42. if \((n, [b, e]) \in \text{stackReadCondition}^n_{\text{gc}}(W)\), and \((n, [b, e]) \in \text{stackWriteCondition}^n_{\text{gc}}(W)\), and \(ms, stk, ms_{stk}, ms_{T} \in \mathbb{F}_{\text{gc}}\) W, then

\[
\exists S \subseteq \text{addressable}(\text{linear}, W.\text{free}).
\]

\[
\exists R : S \rightarrow \mathcal{P}(\mathbb{N}).
\]

\[
\bigcup R(r) = [b, e] \land
\forall r \in S . W.\text{free}(r).H \sqsubseteq _{R(r), gc} \text{std}.p.H \land |R(r)| = 1
\]

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

\[\blacklozenge R(r) = [b, e]
\]

\[
\forall r \in S . W.\text{free}(r).H \sqsubseteq _{R(r), gc} \text{std}.p.H \land |R(r)| = 1 \land W.\text{free}(r) \text{ is address-stratified}
\]

\[\text{Now pick } S = S_R \cup S_W \text{ and show}
\]

\[\forall r \in S . R(r) = R_W(r)
\]

Assuming for contradiction \(\exists r \in S . R(r) \neq R_W(r)\), we have \(W.\text{free}(r).H \sqsubseteq _{R(r), gc} \text{std}.p.H \land W.\text{free}(r).H \sqsupseteq _{R_W(r), gc} \text{std}.p.H\) for two distinct singleton sets * and -. Due to the memory satisfaction assumption \(W.\text{free}(r)\) is non-empty and it is trivial to show that so is the standard regions. From the above, we can conclude that \(W.\text{free}(r)\) should contain some \(ms\) with \(\text{dom}(ms) = *\) and \(\text{dom}(ms) = -\) which contradicts the address stratification assumption.

Now pick

\[R = R_R(r) \text{ for } r \in S
\]

We first show \(\bigcup R(r) \supseteq [b, e]\). For \(a \in [b, e]\) we know that there exists \(r_1\) and \(r_2\) such that \(R_W(r_1) = R_W(r_2) = \{a\}\). It suffices to show \(r_1 = r_2\). To this end assume the contrary. This, the address-stratification assumption on all write regions, the \(n\)-subset of a standard region for read regions, and memory satisfaction lead to a contradiction as any part of memory only can be governed by one region. Address-stratification and the singleton requirement follows trivially from the assumptions from the write region. Finally the \(n\)-equality follows from Lemma 43.

Lemma 43. If \(\iota.H \sqsubseteq _{A, gc} \text{std}.v.H\) and \(\iota.H \sqsupseteq _{A, gc} \text{std}.v.H\), then \(\iota.H \sqsubseteq _{A, gc} \text{std}.v.H\).

\[\blacklozenge
\]

Proof. By definition.

6 Proofs

6.1 Lemmas

In this section, I have listed lemmas that seem to be necessary for the FTLR proof.

Lemma 44 (Downwards closure of relations). If \(n' \leq n\), then

- If \((n, A) \in \text{readCondition}^n_{\text{gc}}(l, W)\), then \((n', A) \in \text{readCondition}^{n'}_{\text{gc}}(l, W)\).
- If \((n, A) \in \text{stackReadCondition}^n_{\text{gc}}(l, W)\), then \((n', A) \in \text{stackReadCondition}^{n'}_{\text{gc}}(l, W)\).
- If \((n, A) \in \text{writeCondition}^n_{\text{gc}}(l, W)\), then \((n', A) \in \text{writeCondition}^{n'}_{\text{gc}}(l, W)\).
- If \((n, A) \in \text{stackWriteCondition}^n_{\text{gc}}(l, W)\), then \((n', A) \in \text{stackWriteCondition}^{n'}_{\text{gc}}(l, W)\).

44
• If \((n, A) \in \text{executeCondition}^{\Box,.g_c}(l, W)\), then \((n', A) \in \text{executeCondition}^{\Box,.g_c}(l, W)\).
• If \((n, A) \in \text{readXCondition}^{\Box,.g_c}(W)\), then \((n', A) \in \text{readXCondition}^{\Box,.g_c}(W)\).
• If \((n, (w, w')) \in \mathcal{V}_{\text{tst}}^{g_c}(W)\), then \((n', (w, w')) \in \mathcal{V}_{\text{tst}}^{g_c}(W)\).
• If \((n, (w, w')) \in \mathcal{R}_{\text{tst}}^{g_c}(W)\), then \((n', (w, w')) \in \mathcal{R}_{\text{tst}}^{g_c}(W)\).
• If \((n, (ms_{stk}, ms_T)) \in \mathcal{F}^{g_c}(W)\), then \((n', (ms_{stk}, ms_T)) \in \mathcal{F}^{g_c}(W)\).
• If \((n, (ms_{stk}, ms_T)) \in \mathcal{S}^{g_c}(W)\), then \((n', (ms_{stk}, ms_T)) \in \mathcal{S}^{g_c}(W)\).
• If \((n, (\overline{\sigma}, ms, ms_T)) \in \mathcal{H}W.\text{heap}(W)\), then \((n', (\overline{\sigma}, ms, ms_T)) \in \mathcal{H}W.\text{heap}(W)\).
• \((ms, stk_{ms}, stk_{ms_T}) \prec g_c W\), then also \((ms, stk, stk_{ms}, stk_{ms_T}) \prec g_c W\).

Proof. The properties for \text{readCondition}^{\Box,.g_c}, \text{stackReadCondition}^{\Box,.g_c}, \text{writeCondition}^{\Box,.g_c}, \text{stackWriteCondition}^{\Box,.g_c}, \text{executeCondition}^{\Box,.g_c}, \text{readXCondition}^{\Box,.g_c} follow easily by definition. The property for \mathcal{V}^{\Box,.g_c}_{\text{untrusted}} follows from the others and by definition. The property for \mathcal{R}^{\Box,.g_c}_{\text{untrusted}} follows directly from the one for \mathcal{V}^{\Box,.g_c}_{\text{untrusted}}.

The property for \mathcal{H}, \((n, (\_, \_)) \in \mathcal{F}^{g_c}(\_), (n, (\_, \_)) \in \mathcal{S}^{g_c}(\_\_)\) follows by their definition from the quantifications over \(n' < n\), and the property for \((\_, \_, \_); n, W\) follows from those.

Lemma 45 (Properties of n-equality of worlds). If \(W_1 \equiv W_2\) then

- \(\text{purePart}(W_1) \equiv \text{purePart}(W_2)\).
- If \(W'_1 \sqsubseteq W_1\) then there exists a \(W'_2\) such that \(W'_2 \equiv W'_1\) and \(W'_2 \sqsubseteq W_2\).
- If \(W'_1 \oplus W_1\) is defined, then there exists a \(W'_2 \equiv W'_1\) and \(W'_2 \oplus W_2\) is defined.
- If \(W_1 = W'_1 \oplus W''_1\) then there exist \(W'_2 \equiv W'_1\) and \(W''_2 \equiv W''_1\) such that \(W_2 = W'_2 \oplus W''_2\).
- If \(W'_1 \equiv W'_2\) then \(W_1 \oplus W'_1 \equiv W_2 \oplus W'_2\).
- If \(\xi^{-1}(W_1) \equiv \xi^{-1}(W_2)\).

Proof. Easy to prove by unfolding definitions and making unsurprising choices for existentially quantified worlds.

Lemma 46 (Non-expansiveness of relations). If \(W \equiv W'\), then

- \(\text{readCondition}^{\Box,.g_c}(W) \equiv \text{readCondition}^{\Box,.g_c}(W')\).
- If \(\text{stackReadCondition}^{\Box,.g_c}(l, W) \equiv \text{stackReadCondition}^{\Box,.g_c}(l, W')\).
- If \(\text{writeCondition}^{\Box,.g_c}(l, W) \equiv \text{writeCondition}^{\Box,.g_c}(l, W')\).
- If \(\text{stackWriteCondition}^{\Box,.g_c}(l, W) \equiv \text{stackWriteCondition}^{\Box,.g_c}(l, W')\).
- If \(\text{readXCondition}^{\Box,.g_c}(l, W) \equiv \text{readXCondition}^{\Box,.g_c}(W')\).
- \(\mathcal{E}^{\Box,.g_c}(W) \equiv \mathcal{E}^{\Box,.g_c}(W')\).
- \(\mathcal{E}_{\text{jmp}}^{\Box,.g_c}(W) \equiv \mathcal{E}_{\text{jmp}}^{\Box,.g_c}(W')\).
- If \(\text{executeCondition}^{\Box,.g_c}(l, W) \equiv \text{executeCondition}^{\Box,.g_c}(l, W')\).
- If \(\mathcal{V}_{\text{tst}}^{g_c}(W) \equiv \mathcal{V}_{\text{tst}}^{g_c}(W')\).
- If \(\mathcal{R}_{\text{tst}}^{g_c}(W) \equiv \mathcal{R}_{\text{tst}}^{g_c}(W')\).
- If \((n, (ms_{stk}, ms_T)) \in \mathcal{F}^{g_c}(W)\), then \((n, (ms_{stk}, ms_T)) \in \mathcal{F}^{g_c}(W')\).
- If \((n, (ms_{stk}, ms_T)) \in \mathcal{S}^{g_c}(W)\), then \((n, (ms_{stk}, ms_T)) \in \mathcal{S}^{g_c}(W')\).
- If \((n, (\overline{\sigma}, ms, ms_T)) \in \mathcal{H}W.\text{heap}(W)\), then \((n, (\overline{\sigma}, ms, ms_T)) \in \mathcal{H}W.\text{heap}(W')\).
• \((ms_S, stk, ms_{stk}, ms_T) \succ^g \frac{n}{n'} W\) iff \((ms_S, stk, ms_{stk}, ms_T) \succ^g \frac{n}{n'} W'\).

\[\text{Proof.}\] The properties for \(\text{readCondition} \Diamond^g, \text{stackReadCondition} \Diamond^g, \text{writeCondition} \Diamond^g, \text{stackWriteCondition} \Diamond^g, \text{readXCondition} \Diamond^g\) follow from the fact that these are defined to use the world only for comparing regions using \(\frac{n}{n'} \subseteq \frac{n}{n'}\) and \(\frac{n}{n'} \geq \frac{n}{n'}\).

The property for \(\mathcal{E}^{\Diamond^g} \text{ and } \mathcal{E}^{\Diamond^g}_{j, \text{imp}}\) follow from Lemma 45.

The property for \(\text{executeCondition} \Diamond^g\) follows from Lemma 45 and the property for \(\mathcal{E}^{\Diamond^g}\).

The property for \(\mathcal{V}^{\Diamond^g}_{\text{untrusted}}\) follows from the other properties, by definition, from Lemma 45 and non-expansiveness of regions.

The property of \(\mathcal{R}^{\Diamond^g}_{\text{untrusted}}\) follows from the one for \(\mathcal{V}^{\Diamond^g}_{\text{untrusted}}\).

The property for \(\mathcal{H}, (n, (\_ , \_ )) \in \mathcal{F}^{\Diamond^g}(\_ ), (n, (\_ , \_ )) \in \mathcal{S}^{\Diamond^g}(\_ )\) and \((\_ , \_ , \_ , \_ ) : n W\) follows from the non-expansiveness of regions in the world and Lemma 45.

\[\square\]

**Lemma 47** (World monotonicity of relations). For all \(n, W' \supseteq W\), we have that

- If \((w_1, w_2) \in H_{\sigma} \sigma W\), then \((w_1, w_2) \in H_{\sigma} \sigma W'\).
- If \((n, (w_1, w_2)) \in \mathcal{V}^{\Diamond^g}_{\text{untrusted}}(W)\), then \((n, (w_1, w_2)) \in \mathcal{V}^{\Diamond^g}_{\text{untrusted}}(W')\).
- If \((n, (\sigma, ms_{S}, ms_T)) \in \mathcal{H}(W, \text{heap})(W)\), then \((n, (\sigma, ms_{S}, ms_T)) \in \mathcal{H}(W, \text{heap})(W')\).

\[\square\]

**Proof.** Follows from the definitions. Note: the proof for \(\mathcal{H}\) relies on Lemma 3.

**Lemma 48** \((\mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})})\) closed under target language antireduction). For all \(\Phi_S, \Phi_T, \Phi'_T, j, n, i\)

\[\Phi_T \rightarrow_j \Phi'_T\text{ and } (n - j, (\Phi_S, \Phi'_T)) \in \mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})},\]

then

\[(n, (\Phi_S, \Phi'_T)) \in \mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})}\]

\[\square\]

**Proof.** Special case of Lemma 50.

**Lemma 49** \((\mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})})\) closed under source language antireduction). For all \(\Phi_S, \Phi_T, \Phi'_T, j, n, i\)

\[\Phi_S \rightarrow^g_j \Phi'_S\text{ and } (n - j, (\Phi'_S, \Phi_T)) \in \mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})},\]

then

\[(n, (\Phi'_S, \Phi_T)) \in \mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})}\]

\[\square\]

**Proof.** Special case of Lemma 50.

**Lemma 50** \((\mathcal{O}^{\Diamond^g, gc}\) closed under antireduction (generalised previous lemma)). For all \(\Phi_S, \Phi'_S, \Phi_T, \Phi'_T, j_S, j_T, n, i\)

- \(\Phi_S \rightarrow^{gc}_j \Phi'_S\)
- \(\Phi_T \rightarrow^{jt}_j \Phi'_T\)
- \((n, (\Phi_S, \Phi'_T)) \in \mathcal{O}^{\Diamond^g, gc}\)

then

\[(n + \min(j_S, j_T), (\Phi_S, \Phi'_T)) \in \mathcal{O}^{\Diamond^g, gc}\]

\[\square\]

**Proof.** Our two languages are deterministic, so we have that \(\Phi_S \Downarrow^{\Diamond^g}_{j_S, \text{min}(j_S, j_T)} \Phi'_S\) and \(\Phi_T \Downarrow^{jt}_{j_T + \text{min}(j_S, j_T)} \Phi'_T\).

It is also easy to show that if \(\Phi_S \Downarrow^{\Diamond^g}_{j_S, \text{min}(j_S, j_T)} \Phi'_S\) and if \(\Phi_T \Downarrow^{jt}_{j_T + \text{min}(j_S, j_T)} \Phi'_T\), the result then follows easily by definition of \(\mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})}\) and \(\mathcal{O}^{\prec, (T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{glob}, \text{col})}\).
Lemma 51 (Capability safety doesn’t depend on address). If
\[ (n, ((\text{perm}, l), b, e, a), ((\text{perm}, l), b, e, a)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]
then
\[ (n, ((\text{perm}, l), b, e, a'), ((\text{perm}, l), b, e, a')) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]

Proof. Direct from the definition of $\mathcal{V}_{\text{tst}}^{\square, gc}$.

Lemma 52 (Stack capability safety doesn’t depend on address). If
\[ (n, ((\text{stack-ptr}(\text{perm}, b, e, a), (\text{perm}, l), b, e, a)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]
then
\[ (n, ((\text{stack-ptr}(\text{perm}, b, e, a'), (\text{perm}, l), b, e, a')) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]

Proof. Direct from the definition of $\mathcal{V}_{\text{tst}}^{\square, gc}$.

Lemma 53 (Seal safety doesn’t depend on current seal). If
\[ (n, (\text{seal}((\sigma_b, \sigma_e, \sigma), \text{seal}(\sigma_b, \sigma_e, \sigma))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]
then
\[ (n, (\text{seal}((\sigma_b, \sigma_e, \sigma'), \text{seal}(\sigma_b, \sigma_e, \sigma'))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]

Proof. Direct from the definition of $\mathcal{V}_{\text{tst}}^{\square, gc}$.

Lemma 54 (Capability safety monotone w.r.t. permission). If
\[ (n, ((\text{perm}, l), b, e, a), ((\text{perm}, l), b, e, a)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]
and
\[ \text{perm} ' \subseteq \text{perm} \]
then
\[ (n, ((\text{perm}', l), b, e, a), ((\text{perm}', l), b, e, a)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W) \]

Proof. Direct from the definition of $\subseteq$ and $\mathcal{V}_{\text{tst}}^{\square, gc}$.

Lemma 55 (Capability splitting retains safety for normal capabilities). If
\begin{itemize}
  \item $(n, (c, c)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W)$
  \item $c = ((\text{perm}, l), b, e, a)$
  \item $b \leq s < e$
  \item $c_1 = ((\text{perm}, l), b, s, a)$
  \item $c_2 = ((\text{perm}, l), s + 1, e, a)$
  \item $c_3 = \text{linearityConstraint}(c)$
  \item $\text{msg}, \text{stk}, \text{msg}_{\text{stk}}, \text{msg}_{\text{T}} : b_{\text{gc}}^c W \oplus \text{purePart}(W_1 \oplus W_2 \oplus W_3)$
\end{itemize}
then there exist $W_1, W_2, W_3$ such that
\begin{itemize}
  \item $W = W_1 \oplus W_2 \oplus W_3$
  \item $(n, (c_1, c_1)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W_1)$
  \item $(n, (c_2, c_2)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W_2)$
  \item $(n, (c_3, c_3)) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W_3)$
\end{itemize}
Proof. If \( l = \text{normal} \), then we pick \( W_3 = W \) and \( W_1 = W_2 = \text{purePart}(W_3) \) which easily satisfies \( W = W_1 \oplus W_2 \oplus W_3 \).

Assuming \( gc = (T_A, gc, \sigma_{\text{glob,ext}}, \sigma_{\text{glob,slot}}) \), we know by assumption \((n, (c, e)) \in \mathcal{V}_{\text{tst}}^{\sqcap, gc}(W) \) that either

- \( T_A \subseteq [b, e] \); or
- \( T_A \# [a, e] \)

In the former case, the result follows from Lemma 29 and Lemma 30.

In the latter case, we know that \([b, s], [s + 1, e] \neq \emptyset \) because \( b \leq s < e \), so the result follows from Lemma 29 and Lemma 30.

If \( l = \text{linear} \), then we know from \((n, (c, e)) \in \mathcal{V}_{\text{tst}}^{\sqcap, gc}(W) \) and w.l.o.g we can assume

- \((n, [b, e]) \in \text{readCondition}^{\sqcap, gc}(\text{linear}, W) \)
- \((n, [b, e]) \in \text{writeCondition}^{\sqcap, gc}(\text{linear}, W) \)

from this, we get \( S_{\text{read}} \subseteq \text{addressable}(\text{linear}, W) \cdot \text{heap}W \) and \( R_{\text{read}} : S_{\text{read}} \rightarrow \text{World} \cdot \text{private stack} \) such that

- \( \bigcup_{r \in S_{\text{read}}} R_{\text{read}}(r) \supseteq A \)
- \( \forall r. |R_{\text{read}}(r)| = 1 \)
- \( \forall r \in S_{\text{read}}, \text{heap}(r).H \supseteq \text{std}(r).W.\text{heap}(r).H \)

and \( S_{\text{write}} \subseteq \text{addressable}(\text{linear}, W) \cdot \text{heap}W \) and \( R_{\text{write}} : S_{\text{write}} \rightarrow \text{World} \cdot \text{private stack} \) such that

- \( \bigcup_{r \in S_{\text{write}}} R_{\text{write}}(r) \supseteq A \)
- \( \forall r. |R_{\text{write}}(r)| = 1 \)
- \( \forall r \in S_{\text{write}}, \text{heap}(r).H \supseteq \text{std}(r).W.\text{heap}(r).H \wedge \text{heap}(r) \) is address-stratified

Now we would like to show \( R_{\text{read}}^{-1}([b, e]) = R_{\text{write}}^{-1}([b, e]) \). We know that the two sets are the same size as both \( R \)'s map to singleton sets. This means that there exists \( r \neq r' \) s.t. \( R_{\text{read}}(r) = R_{\text{write}}(r') = \{a'\} \) for \( a' \in [b, e] \).

By definition of the standard region, we know that for any \( W \) and any \( ms \) and \( ms' \) where \((n-1, (ms, ms')) \in W.\text{heap}(r) \) we have \( \text{dom}(ms) = \text{dom}(ms') = \{a'\} \).

By assumption \( \text{heap}(r) \) is address-stratified which means that for any \( W \) and any \( ms \) and \( ms' \) where \((n-1, (ms, ms')) \in W.\text{heap}(r) \) we have \( \text{dom}(ms)e = \text{dom}(ms')e = \{a'\} \).

By the memory satisfaction, the memory must be split into disjointed parts that each satisfy a region. With two regions that require a memory segment pair with the same domain, we cannot satisfy all the regions, so we must have

Now pick \( W_1 \) as the world that owns \( R_{\text{read}}^{-1}([b, s]), W_2 \) the world that owns \( R_{\text{read}}^{-1}([s+1, e]) \), and \( W_3 \) as the world that owns the remaining regions of \( W \).

It is clearly the case that \( W = W_1 \oplus W_2 \oplus W_3 \).

In this case, we need to show

\[(n, (0, 0)) \in \mathcal{V}_{\text{tst}}^{\sqcap, gc}(W_3)\]

which is trivially the case.

For the remaining, it suffices to show

- \((n, [b, s]) \in \text{readCondition}^{\sqcap, gc}(\text{linear}, W_1) \)
- \((n, [b, s]) \in \text{writeCondition}^{\sqcap, gc}(\text{linear}, W_1) \)
- \((n, [s+1, e]) \in \text{readCondition}^{\sqcap, gc}(\text{linear}, W_2) \)
- \((n, [s+1, e]) \in \text{writeCondition}^{\sqcap, gc}(\text{linear}, W_2) \)

which follows by assumption. 

\[ \square \]

Lemma 56 (Capability splitting retains safety for stack capabilities). If

- \((n, (c_S, c_T)) \in \mathcal{V}_{\text{tst}}^{\sqcap, gc}(W) \)
- \( c_S = \text{stack-ptr}(\text{perm}, b, e, a) \)
- \( c_T = ((\text{perm}, \text{linear}), b, e, a) \)
then there exist $W_1, W_2, W_3$ such that

- $W = W_1 \oplus W_2 \oplus W_3$
- $(n, (c_1, c_1)) \in V_{\text{tst}}^{\text{gc}}(W_1)$
- $(n, (c_2, c_2)) \in V_{\text{tst}}^{\text{gc}}(W_2)$
- $c_1 = ((\text{perm}, l), b, m, a)$
- $c_2 = ((\text{perm}, l), m+1, e, a)$
- $c = ((\text{perm}, l), b, e, a)$
- $b \leq m \leq e$
- $c'_1 = \text{linearityConstraint}(c_1)$
- $c'_2 = \text{linearityConstraint}(c_2)$
- $W_1 \oplus W_2 \oplus W_M$ is defined
- $ms', stk, ms_{\text{stk}}, ms_{\text{t}} : W_M$

then there exist $W'_1, W'_2, W'_3$ such that

- $W_1 \oplus W_2 = W'_1 \oplus W'_2 \oplus W'_3$
- $(n, (c, c)) \in V_{\text{tst}}^{\text{gc}}(W'_1)$
- $(n, (c'_1, c'_1)) \in V_{\text{tst}}^{\text{gc}}(W'_2)$
- $(n, (c'_2, c'_2)) \in V_{\text{tst}}^{\text{gc}}(W'_3)$

**Proof.** Similar to the proof of Lemma 55.

**Lemma 57** (Capability splicing retains safety for normal capabilities). If

- $(n, (c_1, c_1)) \in V_{\text{tst}}^{\text{gc}}(W_1)$
- $(n, (c_2, c_2)) \in V_{\text{tst}}^{\text{gc}}(W_2)$
- $c_1 = ((\text{perm}, l), b, m, a)$
- $c_2 = ((\text{perm}, l), m+1, e, a)$
- $c = ((\text{perm}, l), b, e, a)$
- $b \leq m \leq e$
- $c'_1 = \text{linearityConstraint}(c_1)$
- $c'_2 = \text{linearityConstraint}(c_2)$
- $W_1 \oplus W_2 \oplus W_M$ is defined
- $ms', stk, ms_{\text{stk}}, ms_{\text{t}} : W_M$

then there exist $W'_1, W'_2, W'_3$ such that

- $W_1 \oplus W_2 = W'_1 \oplus W'_2 \oplus W'_3$
- $(n, (c, c)) \in V_{\text{tst}}^{\text{gc}}(W'_1)$
- $(n, (c'_1, c'_1)) \in V_{\text{tst}}^{\text{gc}}(W'_2)$
- $(n, (c'_2, c'_2)) \in V_{\text{tst}}^{\text{gc}}(W'_3)$

**Proof.** From $(n, (c_1, c_1)) \in V_{\text{tst}}^{\text{gc}}(W_1)$ and $(n, (c_2, c_2)) \in V_{\text{tst}}^{\text{gc}}(W_2)$, it follows that either $[b, m] \# T_A$ or $[b, m] \subseteq T_A$ and also either $(m+1, e] \# T_A$ or $[m+1, e] \subseteq T_A$.

Consider first the case where either $[b, m] \subseteq T_A$ or $[m+1, e] \subseteq T_A$. Then by definition of $V_{\text{tst}}^{\text{gc}}$, we have that $l = \text{normal}$, $\text{tst} = \text{trusted}$ and $(n, [b, m]) \in \text{readXCondition}^{\text{gc}}(W_1)$ or $(n, [m+1, e]) \in \text{readXCondition}^{\text{gc}}(W_2)$, respectively. It follows by definition of $\text{readXCondition}^{\text{gc}}$ and $\epsilon^{\text{code}}$ that $[b-1, m+1]$ or $[m, e+1] \subseteq T_A$, respectively. So that it is impossible that $[m+1, e] \# T_A$ or $[b, m] \# T_A$ respectively. In other words, we must have that both $[b, m] \subseteq T_A$ and $[m+1, e] \subseteq T_A$.

From Lemma 32, it follows that also $(n, [b, e]) \in \text{readXCondition}^{\text{gc}}(W_1 \oplus W_2)$ and from Lemma 29, it follows that $(n, [b, e]) \in \text{readXCondition}^{\text{gc}}(\text{purePart}(W_1 \oplus W_2))$. Finally, we have that $(n, (c, e)) \in V_{\text{tst}}^{\text{gc}}(\text{purePart}(W_1 \oplus W_2))$. We can take $W'_3 = \text{purePart}(W_1 \oplus W_2)$, $W'_1 = W_1$ and $W'_2 = W_2$ and the remaining proof obligations follow by assumption and by Lemma 9 and 21.

Now consider the case that both $[b, m] \# T_A$ and $[m+1, e] \# T_A$. The results now follow by definition of $V_{\text{untrusted}}$, using Lemma 32 and 29, taking $W'_3 = W_1 \oplus W_2$ and $W'_1 = \text{purePart}(W_1)$ and $W'_2 = \text{purePart}(W_2)$.
Lemma 58 (Capability splicing retains safety for stack capabilities). If

- \((n, (c_1, S, c_{1,T})) \in \mathcal{V}^{\sqcap,gc}_{\text{rst}}(W_1)\)
- \((n, (c_2, S, c_{2,T})) \in \mathcal{V}^{\sqcap,gc}_{\text{rst}}(W_2)\)
- \(b \leq m \leq e\)
- \(c_1, S = \text{stack-ptr}(\text{perm}, b, m, a)\)
- \(c_1, T = ((\text{perm}, \text{linear}), b, m, a)\)
- \(c_2, S = \text{stack-ptr}(\text{perm}, m + 1, e, a)\)
- \(c_2, T = ((\text{perm}, \text{linear}), m + 1, e, a)\)
- \(c_S = \text{stack-ptr}(\text{perm}, b, e, a)\)
- \(c_T = ((\text{perm}, \text{linear}), b, e, a)\)

then we have that

- \((n, (c_S, c_T)) \in \mathcal{V}^{\sqcap,gc}_{\text{rst}}(W_1 \oplus W_2)\)

Proof. This follows easily by definition of \(\mathcal{V}^{\sqcap,gc}_{\text{rst}}\) and using Lemma 32.

Lemma 59 (Stack capability safety monotone w.r.t. permission). If

\[(n, (\text{stack-ptr}(\text{perm}, b, e, a), ((\text{perm}, l), b, e, a))) \in \mathcal{V}^{\sqcap,gc}_{\text{rst}}(W)\]

and

- \(\text{perm}' \subseteq \text{perm}\)

then

\[(n, (\text{stack-ptr}(\text{perm}', b, e, a), ((\text{perm}', l), b, e, a))) \in \mathcal{V}^{\sqcap,gc}_{\text{rst}}(W)\]

Proof. Follows directly from the definition.

Lemma 60 (readCondition works). If

- \((m, S, stk, m_{stk}, m_{T}) \cdot \text{perm} \cdot \text{stack} : W_M\)
- \((n, (b, e)) \in \text{readCondition}^\sqcup,gc(l, W)\)
- \(a \in [b, e]\)
- \(W \oplus W_M\) is defined
- \(n' < n\)

Then \((n', (m, S(a), m_{T}(a))) \in \mathcal{V}^{\sqcap,gc}_{\text{untrusted}}(W)\) for some \(W'\) such that \(W_M = W' \oplus W'_M\). Additionally, if

- \((n, (b, e)) \in \text{writeCondition}^\sqcap,gc(l, W)\)

Then \((m, S[a \to 0], stk, m_{stk}, m_{T}[a \to 0]) \cdot \text{perm} \cdot \text{stack} : W'_M\).

Proof. From \((n, (b, e)) \in \text{readCondition}^\sqcup,gc(l, W)\), we get an \(S \subseteq \text{addressable}(l, W.\text{heap})\), an \(R : S \to \mathcal{P}(\mathbb{N})\) with \(\bigcup_{r \in S} R(r) \supseteq [b, e] \) and \(W.\text{heap}(r), H \subseteq \mathcal{L}_{n, m, \text{std}, n, \text{stack}, \text{perm}, l, \text{heap}, H}\) for all \(r \in S\).

Since \(a \in [b, e]\), there is a unique \(r \in S\) such that \(a \in R(r)\).

Since \(W \oplus W_M\) is defined, we have that \(r \in \text{dom}(W.\text{heap}) = \text{dom}(W_M.\text{heap})\) and \(W.\text{heap}(r) \oplus W_M.\text{heap}(r)\) is defined.

From \((m, S, stk, m_{stk}, m_{T}) \cdot \text{perm} \cdot \text{stack} : W_M\), we get that \(stk = (\text{opc}_0, m_{s0}) : \cdots : (\text{opc}_m, m_{sm})\), \(m_{s} \sqcup m_{stk} \sqcup m_{s0} \sqcup \cdots \sqcup m_{sm}\) is defined, \(W_M = W_{\text{stack}} \oplus W_{\text{free-stack}} \oplus W_{\text{heap}}\) and \(\exists m_{T, \text{stack}}, m_{T, \text{free-stack}}, m_{T, \text{heap}}, m_{T, f}, m_{s}, m_{T, s}\) such that

- \(m_{s} = m_{s, f} \sqcup m_{s}'\)
- \(m_{T, s} = m_{T, \text{stack}} \sqcup m_{T, \text{free-stack}} \sqcup m_{T, \text{heap}} \sqcup m_{T, f}\)
\( (n, (\sigma, ms_s', ms_{T.heap})) \in H(W_M)(W_{heap}) \), we get an \( R_{ms} : \text{dom}(active(W_{heap}.heap)) \rightarrow \text{MemorySegment} \times \text{MemorySegment} \), such that

\[
\begin{align*}
W.\text{heap} & = \bigoplus_{r \in \text{dom}(active(W_{heap}.heap))} R_W(r), \forall r \in \text{active}(W_M.heap), \\
\text{we have that} (n', R_{ms}(r)) \in W_{heap}.heap(r).H \xi^{-1}(R_W(r)) \text{ for all } n' < n. \text{ We also get an } R_{seal} : \text{dom}(active(W_{heap}.heap)) \rightarrow \mathcal{P} (\text{Seal}) \text{ such that } (\bigcup_{r \in \text{dom}(active(W_{heap}.heap))} R_{seal}(r)) \subseteq \sigma \text{ and } \text{dom}(W_{heap}(r), H_{\sigma}) = R_{seal}(r). \\
\text{We have that } r \in addressable(l, W_{heap}) \subseteq active(W_{heap}.heap), \text{ so } (n', R_{ms}(r)) \in W_{M}.heap(r).H \xi^{-1}(R_W(r)). \\
\text{Because } W_{heap}(r).H \subseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H \text{ and } W \oplus W_M \text{ is defined, it follows that also } W_{M}.heap(r).H \subseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H. \text{ This means that } (n', R_{ms}(r)) \in H^{\mathbb{R}(r) \times [std]} \xi^{-1}(R_W(r)). \\
\text{From this, it follows that } \text{dom}(R_{ms}(r).1) = \text{dom}(R_{ms}(r).2) = R(r) \text{ and we get a } S : R(r) \rightarrow \text{World with } \\
\xi(\xi^{-1}(R_W(r))) = \bigoplus_{a \in R(r)} S(a) \text{ and } S(a) \subseteq R(r), R_{(ms_s(a), ms_{T}(a))}) \in \bigcup_{\text{untrusted}}^{\mathbb{R}(r) \times [std]} S(a). \\
\text{Since } a \in R(r), \text{ we have that } (n', (ms_s(a), ms_{T}(a))) \in \bigcup_{\text{untrusted}}^{\mathbb{R}(r) \times [std]} S(a) \text{ and we can take } W'_M = W'_r \oplus W_{heap}' \oplus (W_{stack} \oplus W_{free,stack}) \text{ with } W'_r = \bigoplus_{a \in R(r)} S(a) \text{ and } W'_{heap} = \bigoplus_{r \in \text{dom}(active(W_{heap}.heap))} W(r) \oplus (W_{stack} \oplus W_{free,stack}) = W_{heap} \oplus (W_{stack} \oplus W_{free,stack}) = W_M. \\
\text{Additionally, if } (n, (b, c)) \in \text{writeCondition}^{\mathbb{R}(r) \times [std]} (l, W), \text{ then we get an } S' \subseteq addressable(l, W_{heap}) \subseteq active(W_{M}.heap), \text{ an } R' : S' \rightarrow \mathcal{P}(\mathbb{N}) \text{ such that } \bigcup_{r \in S'} R'(r) \supseteq [b, c] \text{ such that for all } r \in S', W_{heap}(r).H \subseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H \text{ and } W_{heap}(r) \text{ is address-stratified.} \\
\text{Since } a \in [b, c], \text{ there is an } r' \in S' \text{ such that } a \in R'(r'). \\
\text{Because } W \oplus W_M \text{ is defined, it follows that also } W_{M}.heap(r').H \subseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H \text{ and } W_{heap}(r') \text{ is address-stratified.} \\
\text{It follows that } r = r' \text{ because } \text{dom}(R_{ms}(r).1) = \text{dom}(R_{ms}(r).2) = R(r) \supseteq a \text{ and } \text{dom}(R_{ms}(r').2) = \text{dom}(R_{ms}(r').1) = R'(r') \supseteq a \text{ and all the } R_{ms}(r).1 \text{ and } R_{ms}(r).2 \text{ are disjoint.} \\
\text{We have that } (n', (R_{ms}(r).1[a \mapsto 0], R_{ms}(r).2[a \mapsto 0])) \in W_{heap}.heap(r).H \xi^{-1}(W'_r) \text{ for all } n' < n \text{ because } W_{M}.heap(r) \text{ is address-stratified and } W_{M}.heap(r).H \supseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H. \\
\text{From this, it follows that } (n, (\sigma, ms_s(a \mapsto 0), ms_{T, heap}(a \mapsto 0))) \in H(W_M)(W'_r \oplus W'_{heap}) \text{ and finally } (n, (\sigma, ms_s(a \mapsto 0), stk, ms_{T, heap}(a \mapsto 0))) \in H(W'_M)(W'_r \oplus W'_{heap}) \text{.} \\
\text{Lemma 61 (stackReadCondition works). If } \\
\begin{align*}
( (n, stk, ms_{stk}, ms_{T}) & : n \rightarrow W_M, \\
( (n, (b, c)) & \in \text{stackReadCondition}^{\mathbb{R}(r) \times [std]} (W) \\
a & \in [b, c], \\
W & \oplus W_M \text{ is defined} \\
n' & < n
\end{align*}
\text{Then } (n', (ms_{stk}(a), ms_{T}(a))) \in \bigcup_{\text{untrusted}}^{\mathbb{R}(r) \times [std]} S \text{ for some } W' \text{ such that } W_M = W' \oplus W'_M. \text{ Additionally, if } \\
\begin{align*}
( (n, (b, c)) & \in \text{stackWriteCondition}^{\mathbb{R}(r) \times [std]} (W) \\
\text{Then } (ms_s, stk, ms_{stk}[a \mapsto 0], ms_{T}[a \mapsto 0]) & : n \rightarrow W'_M. \\
\text{Proof. From } (n, (b, c)) \in \text{stackReadCondition}^{\mathbb{R}(r) \times [std]} (W), \text{ we get an } S \subseteq addressable(l, W_{free}) \text{, an } R : S \rightarrow \mathcal{P}(\mathbb{N}) \text{ with } \\
\bigcup_{r \in S} R(r) \supseteq [b, c] \text{ and } W_{free}(r).H \subseteq i_{[std,p,gc]}^{\mathbb{R}(r) \times [std]} H \text{ for all } r \in S. \\
\text{Since } a \in [b, c], \text{ there is a unique } r \in S \text{ such that } a \in R(r). \\
\text{Since } W \oplus W_M \text{ is defined, we have that } r \in \text{dom}(W_{free}) = \text{dom}(W_M.\text{free}) \text{ and } W_{free}(r) \oplus W_M.\text{free}(r) \text{ is defined.} \\
\text{□}
From $(ms_S, stk, ms_{stk}, ms_T) : \delta_n W_M$, we get that $stk = (opc_0, ms_0) : \cdots : (opc_m, ms_m)$, $ms_g \oplus ms_{stk} \oplus ms_f \oplus \cdots \oplus ms_n$ is defined, $W_M = W_{stack} \oplus W_{free_stack} \oplus W_{heap}$ and $\exists ms_T, stack, ms_{T, free_stack}, ms_{T, heap}, ms_T, f, ms_S, f, ms_g$ such that

- $ms_g = ms_{S, f} \oplus ms_g$
- $ms_T = ms_{T, stack} \oplus ms_{T, free_stack} \oplus ms_{T, heap} \oplus ms_T$

$(n, (stk, ms_{T, stack})) \in S^{\delta_n}(W_{Stack})$

$(n, (ms_{stk}, ms_{T, free_stack})) \in F^{gc}(W_{free_stack})$

$(n, (\pi, ms_{T, heap})) \in H(W_M, heap)(W_{Heap})$.

From $(n, (ms_{stk}, ms_{T, free_stack})) \in F^{gc}(W_{free_stack})$, we get an $R_{ms} : dom(\text{active}(W_{free_stack} \oplus heap)) \rightarrow MemorySegment \times MemorySegment, ms_{T, free_stack} = \bigcup_{r \in dom(\text{active}(W_{free_stack} \oplus heap))} \pi_1(R_{ms}(r)), stk_base = dom(ms_{T, free_stack}) \setminus stk_base \in dom(ms_{stk}), \exists R_{W} : dom(\text{active}(W_{free_stack} \oplus heap)) \rightarrow World, W_{free_stack} = \bigcup_{r \in dom(\text{active}(W_{free_stack} \oplus heap))} R_{W}(r), \forall r \in dom(\text{active}(W_{free_stack} \oplus heap)), we have that $(n', R_{ms}(r)) \in W_{free_stack} \oplus heap$. $H_{X^{-1}}(R_{W}(r))$ for all $n' < n$.

We have that $r \in \text{addressable}(l, W_{free}) \subseteq active(W_{free_stack} \oplus heap), so (n', R_{ms}(r)) \in W_{free_stack} \oplus heap). H_{X^{-1}}(R_{W}(r))$. Because $W_{free}(r), H \equiv H_{std}^{\oplus}(R_{W}(r), gc), W \oplus W_M = W \oplus W_{free_stack} \oplus W_{free_stack} \oplus W_{heap}$ is defined, it follows that also $W_{free_stack} \oplus heap. H \equiv H_{std}^{\oplus}(R_{W}(r)). This means that also $(n', R_{ms}(r)) \in H_{std}^{\oplus}(R_{W}(r))$. From this, it follows that $dom(R_{ms}(r).1) = dom(R_{ms}(r).2) = R(r)$ and we get a $S : R(r) \rightarrow World$ with $\xi^{-1}(R_{W}(r)) = \oplus_{a \in R(r)} S(a)$ and $\forall a \in R(r). (n', (ms_{stk}(a), ms_f(a))) \in V_{untrusted}^{\delta_n}(S(a))$. Since $a \in R(r)$, we have that $(n', (ms_{stk}(a), ms_f(a))) \in V_{untrusted}^{\delta_n}(S(a))$ and we can take $W'_M = W'_r \oplus W_{heap} \oplus (W_{stack} \oplus W'_{free_stack})$ with $W'_r = \oplus_{a \in R(r)(\{a\})} S(a)$ and $W'_{free_stack} = \oplus_{r \in dom(\text{active}(W_{heap} \oplus heap))(\{r\})} R_{W}(r')$, and get $S(a) \oplus W_M = S(a) \oplus W'_r \oplus W'_{free_stack} \oplus (W_{stack} \oplus W_{heap})$

Additionally, if $(n, (b, c)) \in stackWriteCondition(W)$, then we get an $S' \subseteq \text{addressable}(l, W_{free}) \subseteq active(W_{free_stack} \oplus heap)$.

Since $a \in [b, c]$, there is an $r' \in S'$ such that $a \in R'(r')$.

Because $W \oplus W_M$ is defined, it follows that also $W_{free_stack} \oplus heap (r'). H_n \equiv H_{std}^{\oplus}(R'(r'), gc), W_{free} \oplus heap (r')$ is address-stratified.

It follows that $r = r'$ because $dom(R_{ms}(r).1) = dom(R_{ms}(r).2) = R(r) \supset a$ and $dom(R_{ms}(r').2) = dom(R_{ms}(r').1) = R'(r') \supset a$ and all the $R_m(r).1$ and $R_m(r).2$ are disjoint.

We have that $(n', (R_{ms}(r).1) \in dom \in 0, R_{ms}(r).2 \in dom \in 0) \in W_{free_stack} \oplus heap, H \equiv H_{std}^{\oplus}(W'), W'_{free_stack} \oplus heap (r') \subseteq H_{X^{-1}}(W')$ for all $n' < n$ because $W_{free_stack} \oplus heap (r')$ is address-stratified and $W_{free_stack} \oplus heap (r')$. $H_n \equiv H_{std}^{\oplus}(W, W')$.

From this, it follows that $(n, (ms_{stk}[a \in 0], ms_{T, heap}[a \in 0])) \in F^{gc}(W \oplus W_{free_stack} \oplus W'_{free_stack})$ and finally $(ms_S, stk, ms_{stk}[a \in 0], ms_{T, heap}[a \in 0]) : \delta_n W_M$.

**Lemma 62** (load from regular capability works). If

- $(ms_S, stk, ms_{stk}, ms_T) : \delta_n W_M$
- $c = ((perms, l), b, e, a)$
- $c' = ((perms', l'), b', c', a')$
- $perm \in readExtended, perm' \in readAccessed$
- $(n, (c, c')) \in V_{std}^{\delta_n}(H, W)$
- $W \oplus W_M$ is defined
- $n' < n$
\[ w_S = \text{linearityConstraint}(ms_S(a)), \ w_T = \text{linearityConstraint}(ms_T(a')) \]

\[ \text{linearityConstraintPerm}(\text{perm}, ms_S(a)), \ \text{linearityConstraintPerm}(\text{perm}', ms_T(a')) \]

\[ a \in [b, e] \]

\[ a' \in [b', e'] \]

Then \( \exists W', W_M \).

\[ W_M = W' \oplus W_M' \]

\[ (ms_S[a \mapsto w_S], stk, ms_{stk}, ms_T[a' \mapsto w_T]) : S_n W_M' \]

\[ (n', (ms_S(a), ms_T(a'))) \in V_{\text{inst}}^{\text{gc}}(W') \]

Proof. Consider first the case that \((n, (c', c')) \in V_{\text{inst}}^{\text{gc}}(W)\).

From \((n, (c', c')) \in V_{\text{inst}}^{\text{gc}}(H_\sigma, W)\) with \(c = ((\text{perm}, f), b, e, a), c' = ((\text{perm}', f'), b', e', a'), \text{perm} \in \text{readAllowed}\) and \(\text{perm}' \in \text{readAllowed}\), we get that \(b = b', e = e'\) and \(a = a'\) and \((n, (b, e)) \in \text{readCondition}_{\text{inst}}^{\text{gc}}(l, W)\).

Lemma 60 then gives us a \(W'\) and \(W'_M\) such that \(W_M = W' \oplus W'_M\) and \((n', (ms_S(a), ms_T(a'))) \in V_{\text{inst}}^{\text{gc}}(W')\).

It remains to prove that \((ms_S[a \mapsto w_S], stk, ms_{stk}, ms_T[a' \mapsto w_T]) : S_n W'_M\). We have to distinguish the case that \(\text{isLinear}(ms_S(a))\) and the opposite case.

- case \(\text{isLinear}(ms_S(a))\): then \(\text{linearityConstraint}(ms_S(a)) = 0\) and it follows from \((n', (ms_S(a), ms_T(a'))) \in V_{\text{inst}}^{\text{gc}}(W)\) that also \(\text{isLinear}(ms_T(a'))\) and \(\text{linearityConstraint}(ms_T(a')) = 0\). From \(\text{linearityConstraintPerm}(\text{perm}, ms_S(a))\) and \(\text{linearityConstraintPerm}(\text{perm}', ms_T(a'))\), we then also get that \(\text{perm}, \text{perm}' \in \text{writeAllowed}\) and \((n, (c', c')) \in \text{inst}_\sigma^{\text{gc}}(W)\). Then \((n, (c, eaddr)) \in \text{writeCondition}_{\text{inst}}^{\text{gc}}(l, W)\).

Additionally, if..." case in Lemma 60 with Lemma 44 we then get that \((ms_S[a \mapsto w_S], stk, ms_{stk}, ms_T[a' \mapsto w_T]) : S_n W'_M\).

- case \(\neg\text{isLinear}(ms_S(a))\): then \(\text{linearityConstraint}(ms_S(a)) = ms_S(a)\) and it follows from \((n', (ms_S(a), ms_T(a'))) \in V_{\text{inst}}^{\text{gc}}(W)\) that also \(\neg\text{isLinear}(ms_T(a'))\) and \(\text{linearityConstraint}(ms_T(a')) = ms_T(a')\). The fact that \((ms_S[a \mapsto w_S], stk, ms_{stk}, ms_T[a' \mapsto w_T]) : S_n W'_M\) then follows simply by downwards closure of memory satisfaction, i.e. Lemma 44.

Now consider the case that \((n, (c', c')) \in (V_{\text{inst}}^{\text{gc}}(W) \setminus V_{\text{inst}}^{\text{gc}}(W))\). then we have that \([b, e] \subseteq T_A\) and \((n, [b, e]) \in \text{readCondition}_{\text{inst}}^{\text{gc}}(W)\). By definition of \(\text{readCondition}_{\text{inst}}^{\text{gc}}\), there is an \(r\) such that \(W.\text{heap}(r) \triangleleft \text{inst}_{\text{code}}\), \(a \in \text{dom}(\text{code})\). By definition of \(\underline{\text{code}}\) and using the fact that \(\text{dom}(\text{code}) \ni a \subseteq [b, e] \subseteq T_A\), we know that \(\text{dom}(\text{code}) \subseteq T_A\) and \(\underline{\text{inst}}_{\text{comp-code}}\text{ code}\). From the fact that \((ms_S, stk, ms_{stk}, ms_T) : S_n W_M\), together with the definition of \(\underline{\text{code}}\), we know that \((n', \Phi, \text{mem}(a), \Phi, \text{mem}(a))) \in V_{\text{inst}}^{\text{gc}}(\text{purePart}(W_M))\) and we have that \(\text{purePart}(W_M) = \text{purePart}(W)\) by Lemma 6. From the fact that \(\underline{\text{inst}}_{\text{comp-code}}\text{ code}\), we get that \(\neg\text{isLinear}(\text{code}(a))\).

We can then take \(W_M = W_M', W' = W\) and get the required results from what we have proven above and using Lemma 44.

\[ \text{Lemma 63 (load from stack capability works). If} \]

\[ \text{linearityConstraintPerm}(\text{perm}, ms_S(a)), \ \text{linearityConstraintPerm}(\text{perm}', ms_T(a')) \]

\[ a \in [b, e] \]

\[ a' \in [b', e'] \]

\[ w_S = \text{linearityConstraint}(ms_S(a)), \ w_T = \text{linearityConstraint}(ms_T(a')) \]
Then $\exists W', W_M'$.

- $W_M = W' \oplus W_M' \in V_{\neg stackReadCondition}^{\neg gc}(H_\sigma, W)$
- $(ms_S, stk, ms_stk[a \mapsto w_S], ms_T[a' \mapsto w_T]) \vdash_n^{gc} W_M'$
- $(n', (ms_stk(a), ms_T(a))) \in \mathcal{V}_{\neg stackWriteCondition}^{\neg gc}(W')$

Proof. From $(n, (c, c')) \in \mathcal{V}_{untrusted}^{\neg gc}(H_\sigma, W)$ with $c = stackPtr(perm, b, e, a)$, $c' = ((perm', l'), b', e', a')$, $(perm \in readAllowed$ or $perm' \in readAllowed), we get that $perm = perm'$, $l' = linear, b = b', e = e'$ and $a = a'$ and $(n, (b, e)) \in stackWriteCondition_{\neg gc}(W)$.

From Lemma 61 we then get that $(n', (ms_stk(a), ms_T(a))) \in \mathcal{V}_{\neg stackWriteCondition}^{\neg gc}(W')$ for some $W'$ such that $W_M = W' \oplus W_M'$.

It remains to prove that $(ms_S, stk, ms_stk[a \mapsto w_S], ms_T[a' \mapsto w_T]) \vdash_n^{gc} W_M'$. We have to distinguish the case that $isLinear(ms_stk(a))$ and the opposite case.

- case $\neg isLinear(ms_stk(a))$: then $linearityConstraint(ms_stk(a)) = 0$ and it follows from $(n', (ms_stk(a), ms_T(a))) \in \mathcal{V}_{\neg stackWriteCondition}^{\neg gc}(H_\sigma, W)$ that also $isLinear(ms_T(a'))$ and $linearityConstraint(ms_T(a')) = 0$. From $linearityConstraintPerm(perm)$ and $linearityConstraintPerm(perm', ms_T(a'))$, we then also get that $perm = perm' \in writeAllowed$ and from $(n, (c, c')) \in \mathcal{V}_{untrusted}^{\neg gc}(H_\sigma, W)$, it then follows that $(n, (b, caddr)) \in stackWriteCondition_{\neg gc}(W)$. From the “Additionally, if..” case in Lemma 60 with Lemma 44 we then get that $(ms_S, stk, ms_stk[a \mapsto 0], ms_T[a' \mapsto 0]) \vdash_n^{gc} W_M'$.

- case $\neg isLinear(ms_stk(a))$: then $linearityConstraint(ms_stk(a)) = ms_stk(a)$ and it follows from $(n', (ms_stk(a), ms_T(a))) \in \mathcal{V}_{untrusted}^{\neg gc}(H_\sigma, W)$ that also $\neg isLinear(ms_T(a'))$ and $linearityConstraint(ms_T(a')) = ms_T(a')$. The fact that $(ms_S[a \mapsto w_S], stk, ms_stk, ms_T[a' \mapsto w_T]) \vdash_n^{gc} W_M'$ then follows simply by downwards closure of memory satisfaction, i.e. Lemma 44.

Lemma 64 (Store to regular capability works). If

- $(ms_S, stk, ms_stk, ms_T) \vdash_n^{gc} W_M$
- $c_1 = ((perm, l), b, e, a)$
- $c'_1 = ((perm', l'), b', e', a')$
- $perm \in writeAllowed\, \land \, perm' \in writeAllowed$
- $a \in [b, e]$
- $a' \in [b', e']$
- $(n, (c_1, c'_1)) \in \mathcal{V}_{untrusted}^{\neg gc}(H_\sigma, W_1)$
- $(n, (c_2, c'_2)) \in \mathcal{V}_{untrusted}^{\neg gc}(H_\sigma, W_2)$
- $W_1 \oplus W_2 \oplus W_M$ is defined
- $n' < n$
- $c_3 = linearityConstraint(c_2), c'_3 = linearityConstraint(c'_2)$

Then $\exists W'_2, W_M'$.

- $W_2 \oplus W_M = W'_2 \oplus W_M'$
- $(ms_S[a \mapsto c_2], stk, ms_stk, ms_T[a' \mapsto c'_2]) \vdash_n^{gc} W_M'$
- $(n', (c_3, c'_3)) \in \mathcal{V}_{untrusted}^{\neg gc}(W'_2)$
Proof. From \((n, (c_1, c_1')) \in \mathcal{V}_{trusted}^{\square,gc}(W_1)\) and \(\text{perm} \in \text{writeAllowed}\), it follows that \(c_1 = c_1'\) and \((n, [b, e]) \in \text{writeCondition}^{\square,gc}(I, W_1)\).

We then get a \(r\) and \(A\), such that \(a \in A, W_1.\text{heap}(r), H \unlhd \iota_{A,gc}^{std,v,\square}.H\) and \(W_1.\text{heap}(r)\) is address-stratified.

If we decompose the judgement that \((\text{ms}_S, \text{stk}, \text{ms}_{stk}, \text{ms}_{T}) : \chi_n \xrightarrow{\text{gc}} W_M\), then we get some \((n - 1, \text{ms}_S|_A, \text{ms}_{T}|_A) \in \iota_{A,gc}^{std,v,\square} W_{M,A}\) for some \(W_M = W_{M,A} \oplus W_{M,R}\). If we define \(W'_M\) as \(W_M \oplus W_2\) and \(W'_2 = \text{purePart}(W_2)\), then we can use the properties about \(W_1.\text{heap}(r)\) above to show that \((\text{ms}_S[a \mapsto c_2], \text{stk}, \text{ms}_{stk}, \text{ms}_{T}[a' \mapsto c_2']) : \chi_n' W'_M\).

The fact that \((n', (c_3, c_3')) \in \mathcal{V}_{untrusted}^{\square,gc}(W'_2)\) follows from Lemma \ref{lem:noaccess}.

\begin{lemma}[Store to stack capability works.]

If

\begin{itemize}
  \item \((\text{ms}_S, \text{stk}, \text{ms}_{stk}, \text{ms}_{T}) : \chi_n \xrightarrow{\text{gc}} W_M\)
  \item \(c_1 = \text{stack-ptr}(\text{perm}, b, e, a)\)
  \item \(c_1' = ((\text{perm}', \text{linear}), b', e', a')\)
  \item \(\text{perm} \in \text{writeAllowed}, \text{perm}' \in \text{writeAllowed}\)
  \item \(a \in [b, e]\)
  \item \(a' \in [b', e']\)
  \item \((n, (c_1, c_1')) \in \mathcal{V}_{trusted}^{\square,gc}(H_S, W_1)\)
  \item \((n, (c_2, c_2')) \in \mathcal{V}_{untrusted}^{\square,gc}(H_S, W_2)\)
  \item \(W_1 \oplus W_2 \oplus W_M\) is defined
  \item \(n' < n\)
  \item \(c_3 = \text{linearityConstraint}(c_2), c_3' = \text{linearityConstraint}(c_2')\)
\end{itemize}

Then \(\exists W'_2, W'_M\).

\begin{itemize}
  \item \(W_2 \oplus W_M = W'_2 \oplus W'_M\)
  \item \((\text{ms}_S, \text{stk}, \text{ms}_{stk}[a \mapsto c_2], \text{ms}_{T}[a' \mapsto c_2']) : \chi_n' W'_M\)
  \item \((n', (c_3, c_3')) \in \mathcal{V}_{untrusted}^{\square,gc}(W'_2)\)
\end{itemize}

\end{lemma}

\begin{proof}

From \((n, (c_1, c_1')) \in \mathcal{V}_{trusted}^{\square,gc}(W_1)\) and \(\text{perm} \in \text{writeAllowed}\), it follows that \(\text{perm} = \text{perm}'\), \(b = b'\), \(e = e'\), \(a = a'\) and \((n, [b, e]) \in \text{stackWriteCondition}^{\square,gc}(W_1)\).

We then get a \(r\) and \(A\), such that \(a \in A, W_1.\text{free}(r), H \unlhd \iota_{A,gc}^{std,v,\square}.H\) and \(W_1.\text{free}(r)\) is address-stratified.

If we decompose the judgement that \((\text{ms}_S, \text{stk}, \text{ms}_{stk}, \text{ms}_{T}) : \chi_n \xrightarrow{\text{gc}} W_M\), then we get some \((n - 1, \text{ms}_S|_A, \text{ms}_{T}|_A) \in \iota_{A,gc}^{std,v,\square} W_{M,A}\) for some \(W_M = W_{M,A} \oplus W_{M,R}\). If we define \(W'_M\) as \(W_M \oplus W_2\) and \(W'_2 = \text{purePart}(W_2)\), then we can use the properties about \(W_1.\text{free}(r)\) above to show that \((\text{ms}_S[a \mapsto c_2], \text{stk}, \text{ms}_{stk}, \text{ms}_{T}[a' \mapsto c_2']) : \chi_n' W'_M\).

The fact that \((n', (c_3, c_3')) \in \mathcal{V}_{untrusted}^{\square,gc}(W'_2)\) follows from Lemma \ref{lem:noaccess}.

\begin{lemma}

If

\begin{itemize}
  \item \((n, (\text{sealed}(\sigma, sc_1), \text{sealed}(\sigma, sc_1'))) \in \mathcal{V}_{int}^{\Box,gc}(W_{R,1})\)
  \item \((n, (\text{sealed}(\sigma, sc_2), \text{sealed}(\sigma, sc_2'))) \in \mathcal{V}_{int}^{\Box,gc}(W_{R,2})\)
  \item \text{nonExecutable}(sc_1') and \text{nonExecutable}(sc_2')
  \item \(W_{R,1} \oplus W_{R,2} \oplus W_M\) is defined
  \item \(\text{ms}_S, \text{stk}, \text{ms}_{stk}, \text{ms}_{T} : \chi_n^{gc} W_M\)
\end{itemize}

Then

\begin{itemize}
  \item \((n - 1, (sc_1, sc_2, sc_1', sc_2')) \in \mathcal{E}_{x_{int}}^{\Box,gc}(W_{R,1} \oplus W_{R,2})\)
\end{itemize}

\end{lemma}

\begin{proof}

From \((n, (\text{sealed}(\sigma, sc_1), \text{sealed}(\sigma, sc_1'))) \in \mathcal{V}_{int}^{\Box,gc}(W_{R,1})\), we know that
• \((\text{isLinear}(sc_1) \iff \text{isLinear}(sc'_1))\)  
• \(\exists r \in \text{dom}(W.\text{heap}),\overline{\sigma_{\text{ret}}} , \overline{\sigma_{\text{close}}} , ms_{\text{code}}.\)  
• \(W_{R,1}.\text{heap}(r) = (\text{pure}, _-, H_\sigma)\)  
• \(H_\sigma \vdash H^s_{\sigma, \boxdot} \overline{\sigma_{\text{ret}}} \overline{\sigma_{\text{close}}} ms_{\text{code}} gc\)  
• \((n', (sc_1, sc'_1)) \in H_\sigma \sigma \xi^{-1}(W_{R,1})\) for all \(n' < n\)  
• If \((\text{isLinear}(sc_1)\) then for all \(W' \supseteq W_{R,1}, W_o, n' < n\), \((n', (sc_2, sc'_2)) \in H_\sigma \sigma \xi^{-1}(W_o)\), we have that \((n', sc_1, sc_2, sc'_1, sc'_2) \in \mathcal{F}_{\text{xJump}}(W' \oplus W_o)\)  
• If \((\text{nonLinear}(sc_1)\) then for all \(W' \supseteq \text{purePart}(W_{R,1}), W_o, n' < n\), \((n', (sc_2, sc'_2)) \in H_\sigma \sigma \xi^{-1}(W_o)\), we have that \((n', sc_1, sc_2, sc'_1, sc'_2) \in \mathcal{F}_{\text{xJump}}(W' \oplus W_o)\)  

From \((n, (\text{sealed}(\sigma, sc_2), \text{sealed}(\sigma, sc'_2))) \in \mathcal{V}^{\boxdot, gc}_{\text{rst}}(W_{R,2})\), we know that  
• \((\text{isLinear}(sc_2) \iff \text{isLinear}(sc'_2))\)  
• \(\exists r' \in \text{dom}(W.\text{heap}),\overline{\sigma_{\text{ret}}} , \overline{\sigma_{\text{close}}} , ms_{\text{code}}.\)  
• \(W_{R,2}.\text{heap}(r') = (\text{pure}, _-, H_\sigma)\)  
• \(H_\sigma \vdash H^s_{\sigma, \boxdot} \overline{\sigma_{\text{ret}}} \overline{\sigma_{\text{close}}} ms_{\text{code}} gc\)  
• \((n', (sc_2, sc'_2)) \in H_\sigma \sigma \xi^{-1}(W_{R,2})\) for all \(n' < n\)  

From \(ms_{S}, stk, ms_{stk}, ms_{T} : \Phi^c_w M\), we know that two different regions cannot have \(H_\sigma\) defined for the same \(\sigma\), so that \(r = r'\). Both when \(\text{isLinear}(sc_1)\) and when \(\text{nonLinear}(sc_1)\) (using Lemma 12), we know that for all \(W' \supseteq W_{R,1}, W_o, n' < n, sc_2, sc'_2, (n', (sc_2, sc'_2)) \in H_\sigma \sigma \xi^{-1}(W_o)\), we have that \((n', sc_1, sc_2, sc'_1, sc'_2) \in \mathcal{F}_{\text{xJump}}(W' \oplus W_o)\)  

We can now instantiate this fact with \(W' = W_{R,1}, W_o = W_{R,2}, n' = n - 1\), and the fact that \((n - 1, (sc_2, sc'_2)) \in H_\sigma \sigma \xi^{-1}(W_{R,2})\) (see above) to conclude \((n - 1, sc_1, sc_2, sc'_1, sc'_2) \in \mathcal{F}_{\text{xJump}}(W_{R,1} \oplus W_{R,2})\). 

\[\blacksquare\]

### 6.2 FTLR proof

**Lemma 67.** If  

• One of the following sets of requirements holds:  
  - \(\text{tst} = \text{trusted}, \Phi_S\) is reasonable up to \(n\) steps and \([b, e] \subseteq T_A\)  
  - \(\text{tst} = \text{untrusted}\) and \([b, e] \neq T_A\) and \((n, [b, e]) \in \text{readCondition}^{\boxdot, gc}(\text{normal}, W_{pc})\)  
• \(\Phi_S(pc) = \Phi_T(pc) = ([\text{rx}, \text{normal}], b, e, a)\)  
• \((n, [b, e]) \in \text{readXCondition}^{\boxdot, gc}(W_{pc})\)  
• \((n, (\Phi_S.\text{reg}, \Phi_T.\text{reg})) \in \mathcal{R}^{\boxdot, gc}_\text{rst}(W_R)\)  
• \(\Phi_S.\text{mem}, \Phi_S.\text{stk}, \Phi_S.\text{ms}_{\text{stk}}, \Phi_T.\text{mem} : \Phi^c_w M\)  
• \(W_{pc} \oplus W_R \oplus W_M\) is defined.  
• **Theorem 2** holds for all \(n' < n\).  

Then  

\((n, (\Phi_S, \Phi_T)) \in \mathcal{O}^{\boxdot, gc}\)  

\[\blacksquare\]
Proof. In the following we will use the following: \( \text{reg}_S = \Phi_S.\text{reg} \), \( \text{reg}_T = \Phi_T.\text{reg} \), \( \text{ms}_S = \Phi_S.\text{mem} \), \( \text{ms}_\text{stk} = \Phi_S.\text{ms}_\text{stk} \), and \( \text{stk} = \Phi_S.\text{stk} \).

By complete induction on \( n \), i.e. we can assume that the lemma already holds for \( n' < n \).

In order to prove this, we first, we prove that one of the following holds:

1. \( \Phi_S \rightarrow \rightarrow \) failed and \( \Phi_T \rightarrow \rightarrow \) failed for some \( i,j \).
2. \( \Phi_S \rightarrow \rightarrow \) halted and \( \Phi_T \rightarrow \rightarrow \) halted

3. All of the following hold: (includes simple cases: getype, geta, getb, gete, getp, getl, it, plus, minus, one case of move that can be handled uniformly)
   
   - \( \Phi_S \rightarrow \rightarrow \Phi'_S \)
   - \( \Phi_T \rightarrow \Phi'_T \)
   - \( \Phi_S \) does not point to \( \text{call}^{off}\text{pc}^{off} r_1 r_2 \) or \( \text{xjmp} r_1 r_2 \)
   - \( \Phi'_S = \text{updatePc}(\Phi_S[\text{reg} \rightarrow z]) \neq \text{failed} \)
   - \( \Phi'_T = \text{updatePc}(\Phi_T[\text{reg} \rightarrow z]) \neq \text{failed} \)
   - \( r \neq \text{pc} \)
   - \( z \in \mathbb{Z} \)

4. All of the following hold:
   
   - \( \text{callCondition}(\Phi_S, r_1, r_2, \text{off}_{pc}, \text{off}_{\sigma}, a) \)
   - \( r_1 \neq r_{t1} \) and \( r_2 \neq r_{t1} \)
   - \( b \leq a \) and \( a + \text{call}\_\text{len} - 1 \leq e \)
   - \( \text{executable}(\Phi_S(\text{pc})) \)
   - for all \( i = 0..\text{call}\_\text{len} - 1 \), \( \text{ms}_T(a + i) = \text{ms}_S(a + i) \in \mathbb{Z} \).
   - \( [b, e] \subseteq T_A \)
   - \( \Phi_S \rightarrow \rightarrow \Phi'_S \neq \text{failed} \)
   - \( \Phi_T \rightarrow \Phi'_T \neq \text{failed} \)
   - \( \Phi_S(r_1) = \text{sealed}(\sigma, c_1) \)
   - \( \Phi_S(r_2) = \text{sealed}(\sigma, c_2) \)
   - \( \text{nonExecutable}(c_2) \)
   - \( \Phi_S(r_{\text{stk}}) = \text{stack-ptr}((\text{rw}, b_{\text{stk}}, c_{\text{stk}}, a_{\text{stk}})) \)
   - \( b_{\text{stk}} < a_{\text{stk}} \leq c_{\text{stk}} \)
   - \( \Phi_S(\text{pc}) = ((\text{perm}, \text{normal}), b, e, a) \)
   - \( w_1 = \text{linearityConstraint}(c_1) \) and \( w_2 = \text{linearityConstraint}(c_2) \)
   - \[
   \Phi''_S.\text{reg} = \Phi_S.\text{reg}[r_{\text{stk}} \mapsto \text{stack-ptr}((\text{rw}, b_{\text{stk}}, a_{\text{stk}} - 1, a_{\text{stk}} - 1))] \\
   [r_{\text{retcode}} \mapsto \text{sealed}(\sigma', \text{ret-ptr-code}(b, e, a + \text{call}\_\text{len}))] \\
   [r_{\text{retdata}} \mapsto \text{sealed}(\sigma', \text{ret-ptr-data}(a_{\text{stk}}, c_{\text{stk}}))] \\
   [r_{t1} \mapsto w_1, w_2] \\
   [r_{t1} \mapsto 0]
   \]
   - \( \Phi''_S.\text{mem} = \Phi_S.\text{mem} \)
   - \( \text{ms}_{\text{stk}}.\text{prev}, S = \Phi_S.\text{ms}_{\text{stk}}[a_{\text{stk}}, c_{\text{stk}}][a_{\text{stk}} \mapsto 42] \)
   - \( \Phi''_S.\text{stk} = ((a + \text{call}\_\text{len}), \text{ms}_{\text{stk}}.\text{prev}, S) :: \Phi_S.\text{stk} \)
   - \( \Phi''_S.\text{ms}_{\text{stk}} = \Phi_S.\text{ms}_{\text{stk}} - \Phi_S.\text{ms}_{\text{stk}}[a_{\text{stk}}, c_{\text{stk}}] \)
   - \( b \leq a + \text{off}_{pc} \leq e \)
   - \( \text{mem}(a + \text{off}_{pc}) = \text{sealed}(\sigma_b, \sigma_c, \sigma_a) \)
   - \( \sigma' = \sigma_a + \text{off}_{\sigma} \)
   - \( \sigma_a \leq \sigma' \leq \sigma_c \)
   - \( \Phi'_S = \text{xjumpResult}(c_1, c_2, \Phi''_S) \)
5. All of the following hold: (includes cap-manipulation cases: move, cca, restrict, seta2b, cseal, split, splice, that can be handled mostly uniformly)

- \( \Phi_S \rightarrow^{sc} \Phi'_S \)
- \( \Phi_T \rightarrow \Phi'_T \)
- \( \Phi_S \) does not point to call \( \text{off} \) \( r_1, r_2 \) or xjmp \( r_1, r_2 \)
- \( \Phi'_S = \text{updatePc}(\Phi_S[\text{reg}.r_1 \cdots r_k \rightarrow w_1 \cdots w_k]) \neq \text{failed} \)
- \( \Phi'_T = \text{updatePc}(\Phi_T[\text{reg}.r_1 \cdots r_k \rightarrow w'_1 \cdots w'_k]) \neq \text{failed} \)
- \( r_i \neq \text{pc} \) for all \( i \)

One of the following holds:

5.1. (restrict, cca, seta2b) \( w_1 = ((\text{perm}', l), b, e, a'), w'_1 = ((\text{perm}', l), b, e, a'), \Phi_S(r_1) = ((\text{perm}, l), b, e, a) \) and \( \Phi_T(r_1) = ((\text{perm}, l), b, e, a) \) and \( \text{perm} \subseteq \text{perm} \), \( k = 1 \)

5.2. (restrict, cca, seta2b) \( w_1 = \text{stack-ptr}(\text{perm}', b, e, a'), w'_1 = ((\text{perm}', \text{linear}), b, e, a'), \Phi_S(r_1) = \text{stack-ptr}(\text{perm}, b, e, a) \) and \( \Phi_T(r_1) = ((\text{perm}, \text{linear}), b, e, a) \) and \( \text{perm} \subseteq \text{perm} \), \( k = 1 \)

5.3. (cca, seta2b) \( w_1 = \text{seal}(\sigma_b, \sigma_e, \sigma'), w'_1 = \text{seal}(\sigma_b, \sigma_e, \sigma') \), \( \Phi_S(r_1) = \text{seal}(\sigma_b, \sigma_e, \sigma) \) and \( \Phi_T(r_1) = \text{seal}(\sigma_b, \sigma_e, \sigma), k = 1 \)

5.4. (move) \( w_1 = \Phi_S(r_2), w'_1 = \Phi_T(r_2) \), and \( w_2 = \text{linearityConstraint}(w_1), w'_2 = \text{linearityConstraint}(w'_1) \) and \( r_1 \neq r_2 \) and \( k = 2 \)

5.5. (cseal) \( w_1 = \text{sealed}(\sigma, \Phi_S(r_1)), w'_1 = \text{sealed}(\sigma, \Phi_T(r_1)) \), \( \Phi_S(r_2) = \Phi_T(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma), \sigma_b \leq \sigma \leq \sigma_e, k = 1 \), and \( \Phi_S \) points to cseal \( r_1 \) \( r_2 \)

5.6. (split) \( \Phi_T(r_3) = \Phi_S(r_3) = ((\text{perm}, l), b, e, a), b \leq s \leq e, w_1 = w'_1 = ((\text{perm}, l), b, n, a), w_2 = w'_2 = ((\text{perm}, l), n + 1, e, a), w_3 = w'_3 = \text{linearityConstraint}(\Phi_S(r_3)), k = 3 \)

5.7. (split) \( \Phi_T(r_3) = \Phi_S(r_3) = \text{sealed}(\sigma_b, \sigma_e, \sigma), \sigma_b \leq s \leq \sigma_e, w_1 = w'_1 = \text{sealed}(\sigma_b, \sigma_e, \sigma), w_2 = w'_2 = \text{sealed}(s + 1, \sigma_e, \sigma), k = 2 \)

5.8. (split) \( \Phi_S(r_3) = \text{stack-ptr}(\text{perm}, b, e, a), \Phi_T(r_3) = ((\text{perm}, \text{linear}), b, e, a), b \leq n, n < e, w_1 = \text{stack-ptr}(\text{perm}, b, n, a), w'_1 = ((\text{perm}, \text{linear}), b, n, a), w_2 = \text{stack-ptr}(\text{perm}, n + 1, e, a), w'_2 = (\text{perm}, \text{linear}), n + 1, e, a), k = 2 \)

5.9. (split) \( \Phi_T(r_2) = ((\text{perm}, \text{linear}), b_2, e_2, \ldots), \Phi_S(r_2) = \text{stack-ptr}(\text{perm}, b_2, e_2, \ldots), \) and \( \Phi_T(r_3) = ((\text{perm}, \text{linear}), e_2 + 1, e_3, a_3), \Phi_S(r_3) = \text{stack-ptr}(\text{perm}, e_2 + 1, e_3, a_3), b_2 \leq e_2, e_2 + 1 \leq e_3, w_1 = ((\text{perm}, \text{linear}), b_2, e_3, a_3), w'_1 = \text{stack-ptr}(\text{perm}, b_2, e_3, a_3), \) and \( w_2 = w'_2 = w_3 = w'_3 = 0, k = 3 \)

5.10. (split) \( \Phi_T(r_2) = \Phi_S(r_2) = ((\text{perm}, \text{linear}), b_2, e_2, \ldots), \Phi_T(r_3) = \Phi_S(r_3) = ((\text{perm}, \text{linear}), e_2 + 1, e_3, a_3), \) and \( b_2 \leq e_2, e_2 + 1 \leq e_3, w_1 = w'_1 = (\text{perm}, \text{linear}), b_2, e_3, a_3), \) and \( w_2 = w'_2 = \text{linearityConstraint}(\Phi_S(r_2)) \) and \( w_3 = w'_3 = \text{linearityConstraint}(\Phi_S(r_3)), k = 3 \)

5.11. (split) \( \Phi_T(r_2) = \Phi_S(r_2) = \text{sealed}(\sigma_{b,2}, \sigma_{e,2}, \ldots), \Phi_T(r_3) = \Phi_S(r_3) = \text{sealed}(\sigma_{e,2} + 1, \sigma_{e,3}, \sigma_{a,3}), \) and \( \sigma_{b,2} \leq \sigma_{e,2}, \sigma_{e,2} + 1 \leq \sigma_{e,3} \text{and} w_1 = w'_1 = \text{sealed}(\sigma_{b,2}, \sigma_{e,3}, \sigma) \)

5.12. (juz zero case, noop move) \( k = 0 \)

6. All of the following hold: (includes memory-manipulation cases: store, load, that can be handled mostly uniformly)

- \( \Phi_S \rightarrow^{sc} \Phi'_S \)
- \( \Phi_T \rightarrow \Phi'_T \)
- \( \Phi_S \) does not point to call \( \text{off} \) \( r_1, r_2 \) or xjmp \( r_1, r_2 \)
- \( \Phi'_S = \text{updatePc}(\Phi_S[\text{reg}.r_1 \rightarrow w_1, w_2[[\text{mem}.a \rightarrow w]]) \)
- \( \Phi'_T = \text{updatePc}(\Phi_T[\text{reg}.r'_1 \rightarrow w'_1, w'_2[[\text{mem}.a \rightarrow w']]) \)
- \( r_i \neq \text{pc} \) for all \( i \)

One of the following holds:

6.1. (store) \( w_1 = w'_1 = \Phi_S(r_1) = \Phi_T(r_1) = ((\text{perm}, l), b, e, a), \text{perm} \in \text{writeAllowed}, \text{and} \) \( \text{withinBounds}(w_1), \) and \( w = \Phi_S(r_2), w' = \Phi_T(r_2), \) and \( w_2 = \text{linearityConstraint}(\Phi_S(r_2)), w'_2 = \text{linearityConstraint}(\Phi_T(r_2)). \)

6.2. (load) \( w_2 = w'_2 = \Phi_T(r_2) = \Phi_S(r_2) = ((\text{perm}, l), b, e, a), \text{perm} \in \text{readAllowed}, \text{withinBounds}((\text{perm}, l), b, e, a), \) and \( w_1 = \Phi_S(\text{mem}.a), w'_1 = \Phi_T.\text{mem}(a), \) and \( w = \text{linearityConstraint}(w_1), w' = \text{linearityConstraint}(w'_1), \text{linearityConstraintPerm}(\text{perm}, w_1), \text{linearityConstraintPerm}(\text{perm}, w'_1) \)

58
7. All of the following hold: (includes memory-manipulation cases: store, load, that can be handled mostly uniformly (stack))

- \( \Phi_S \rightarrow^{sc} \Phi'_S \)
- \( \Phi_T \rightarrow \Phi'_T \)
- \( \Phi_S \) does not point to \( \text{call}^{off\_pc, \_off\_r} \) \( r_1 \) \( r_2 \) or \( \text{xjmp} \) \( r_1 \) \( r_2 \)
- \( \Phi'_S = \text{updatePc}(\Phi_S[reg.r_1, r_2 \mapsto w_1, w_2][\text{ms}_{stk}.a \mapsto w]) \)
- \( \Phi'_T = \text{updatePc}(\Phi_T[reg.r'_1, r'_2 \mapsto w'_1, w'_2][\text{ms}_{stk}.a \mapsto w']) \)
- \( r_i \neq \text{pc} \) for all \( i \)

One of the following hold:

7.1. (store) \( w_1 = \Phi_T(r_1) = ((\text{perm}, \text{linear}), b, e, a), w'_1 = \Phi_S(r_1) = \text{stack-ptr}(\text{perm}, b, e, a), \) and \( \text{perm} \in \text{writeAllowed}, \text{withinBounds}(w_1), \) and \( w = \Phi_S(r_2), \) and \( w' = \Phi_T(r_2), \) and \( w_2 = \text{linearityConstraint}(\Phi_S(r_2)), w'_2 = \text{linearityConstraint}(\Phi_T(r_2)). \)

7.2. (load) \( w'_2 = \Phi_T(r_2) = ((\text{perm}, \text{linear}), b, e, a), \) and \( w_2 = \Phi_S(r_2) = \text{stack-ptr}(\text{perm}, b, e, a), \) and \( \text{perm} \in \text{readAllowed}, \text{withinBounds}(((\text{perm}, l, b, e, a)), \) and \( a \in \text{dom}(\Phi.(\text{ms}_{stk})), \) and \( a \in \text{dom}(\Phi.(\text{ms}_{stk})), \) and \( w_1 = \Phi_S.(\text{ms}_{stk})(a), \) and \( w'_1 = \Phi_T.(\text{ms}_{stk})(a), \) and \( w = \text{linearityConstraint}(w_1), w' = \text{linearityConstraint}(w'_1), \) \( \text{linearityConstraintPerm}(\text{perm}, w_1) \) and \( \text{linearityConstraintPerm}(\text{perm}, w'_1) \)

8. All of the following hold: (includes control-flow manipulation cases: jmp, jnz, xjmp, that can be handled mostly uniformly)

- \( \Phi_S \rightarrow^{sc} \Phi'_S \)
- \( \Phi_T \rightarrow \Phi'_T \)
- \( \Phi_S \) does not point to \( \text{call}^{off\_pc, \_off\_r} \) \( r_1 \) \( r_2 \) or \( \text{xjmp} \) \( r_1 \) \( r_2 \)

One of the following holds:

8.1. (jmp,jnz) \( \Phi'_S = \Phi_S.[\text{reg-pc}, r_1 \mapsto \Phi_S(r_1), w_1] \) and \( \Phi'_T = \Phi_T.[\text{reg-pc}, r'_1 \mapsto \Phi_T(r_1), w'_1] \) and \( \Phi_S(r_1) = \Phi_T(r_1) = ((\text{perm}_1, h_1), b_1, e_1, a_1), \) \( \text{executable}(\Phi_S(r_1)), \) \( \text{withinBounds}(\Phi_S(r_1)), w_1 = \text{linearityConstraint}(\Phi_S(r_1)) \) and \( w'_1 = \text{linearityConstraint}(\Phi_T(r_1)) \)

8.2. (xjmp) \( \Phi_S(r_1) = \text{sealed}(\sigma, c_1) \) and \( \Phi_S(r_2) = \text{sealed}(\sigma, c_2) \) and \( \Phi_T(r_1) = \text{sealed}(\sigma, c'_1) \) and \( \Phi_T(r_2) = \text{sealed}(\sigma, c'_2) \) and \( c'_1 \neq \text{ret-ptr-code}(\cdot) \) and \( c'_2 \neq \text{ret-ptr-data}(\cdot) \) and \( \text{nonExecutable}(\Phi_S(r_2)) \) and \( \text{nonExecutable}(\Phi_T(r_2)) \) and \( \Phi'_S = \Phi_S.[\text{reg-pc}, r_1 \mapsto \text{linearityConstraint}(c_1), \text{linearityConstraint}(c_2)] \) and \( \Phi'_T = \Phi_T.[\text{reg-pc}, r_2 \mapsto \text{linearityConstraint}(c'_1), \text{linearityConstraint}(c'_2)] \) and \( \Phi'_T = \text{xjumpResult}(c'_1, c_2, \Phi'_S) \) and \( \Phi'_T = \Phi_T.[\text{reg-pc}, r_2 \mapsto \text{linearityConstraint}(c'_1), \text{linearityConstraint}(c'_2)] \) and \( \Phi'_T = \text{xjumpResult}(c'_1, c'_2, \Phi'_T) \)

The above follows from a careful analysis of the cases of the operational semantics, using the following facts:

- \( [b, e] \subseteq T_A \) or \( [b, e] \neq T_A \) (by assumption of this lemma)
- \( \Phi_S.(\text{reg-pc}) = \Phi_T.(\text{reg-pc}) \) (by assumption of this lemma)
- \( \Phi_S.(\text{mem})(a) = \Phi_T.(\text{mem})(a) \) if \( a \in [b, e] \): follows from \( (n, [b, e]) \in \text{readXCondition}(\square_{nc}(W_{pc})), \) the fact that \( W_{pc} \oplus W_R \oplus W_M \) is defined and \( \Phi_S.(\text{mem}), \Phi_S.(\text{stk}), \Phi_S.(\text{ms}_{stk}), \Phi_T.(\text{mem}), \Phi_T.(\text{ms}_{stk}) \)
- \( \Phi_S.(\text{mem})(a \cdots a + \text{call_len} - 1) = \Phi_T.(\text{mem})(a \cdots a + \text{call_len} - 1) \) if \( [a \cdots a + \text{call_len} - 1] \subseteq [b, e] \) (follows similarly).
- \( \Phi_S.(\text{reg})(r) = ((\text{perm}_1, h_1), b_1, e_1, a_1) \) implies that \( \Phi_S.(\text{reg})(r) = \Phi_T.(\text{reg})(r) \) and \( a_1 \in \text{dom}(\Phi_S.(\text{mem})) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) = z \) implies that \( \Phi_S.(\text{reg})(r) = \Phi_T.(\text{reg})(r) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) = \text{sealed}(\sigma_0, \sigma, \cdot) \) implies that \( \Phi_S.(\text{reg})(r) = \Phi_T.(\text{reg})(r) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) = \text{sealed}(\sigma, \cdot) \) implies that \( \Phi_T.(\text{reg})(r) = \text{sealed}(\sigma, \cdot) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) \neq \text{ret-ptr-data}(\cdot, \cdot) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) \neq \text{ret-ptr-code}(\cdot, \cdot) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
- \( \Phi_S.(\text{reg})(r) = \text{stack-ptr}(\text{perm}_1, b_1, e_1, a_1) \) implies that \( \Phi_T.(\text{reg})(r) = ((\text{perm}_1, \text{linear}), b_1, e_1, a_1) \) and \( a \in \text{dom}(\Phi_S.(\text{ms}_{stk})) \) (follows from \((n, (\Phi_S.(\text{reg}), \Phi_T.(\text{reg}))) \in \mathcal{R}_{\tau_{pc}, \square_{nc}}(W_R))\).
• Similar facts about the current address, base and end pointer, permissions and linearity of all register capabilities being equal (follows from \((n, (\Phi_S, \text{reg}, \Phi_T, \text{reg})) \in R^{\text{t}_{\text{reg}}}_{\text{trusted}}(W_R))\).

By the above observation, we know that \(\Phi_S \rightarrow^g c \Phi_S'\) and \(\Phi_T \rightarrow \Phi_T'\) for some \(\Phi_S'\) and \(\Phi_T'\). According to Lemma 50 it suffices to show:

\[(n-1, (\Phi_S', \Phi_T')) \in \mathcal{O}^{\square,gc}\]

Consider each of the possible cases:

In case 1, both executions go to failed. In this case, the result follows vacuously by definition of \(\mathcal{O}^{\square,gc}(T_A, \text{stk}, \text{base}, \text{glob}, \text{ret}, \text{stk}, \text{reg}, \text{hlib}, \text{close})\).

In case 2, both source and target configuration halts in 0 steps, so both directions of \(\mathcal{O}^{\square,gc}\) are trivially satisfied.

In case 3, we use the induction hypothesis to conclude

\[(n-1, (\Phi_S', \Phi_T')) \in \mathcal{O}^{\square,gc}\]

from the following facts:

• One of the following sets of requirements holds:
  
  - \(\text{tst} = \text{trusted}\), \(\Phi_S\) is reasonable up to \(n-1\) steps and \([b, c] \subseteq T_A\)
  
  - \(\text{tst} = \text{untrusted}\) and \([b, c] \neq T_A\) and \((n-1, [b, c]) \in \text{readCondition}^{\square,gc}(\text{normal}, W_{pc})\)

This follows from the corresponding assumption of this lemma, the fact that \(\Phi_S \rightarrow^g c \Phi_S'\) and \(\Phi_S\) does not point to \(\text{call}\)^\(gf^0_{\text{pc}}\), \(\text{xjmp} r_1 r_2\) or \(\text{xjmp} r_1 r_2\) and 44

• \(\Phi_S'(pc) = \Phi_T'(pc) = ([\text{ix}, \text{normal}], b, c, \_):\) Follows from the definition of \(\text{updatePc}\) and the assumptions of this case.

• \((n-1, [b, c]) \in \text{readXCondition}^{\square,gc}(W_{pc})\): Follows from the corresponding assumption of this lemma using Lemma 44

• \((n-1, (\Phi_S, \text{reg}, \Phi_T, \text{reg})) \in R^{\text{t}_{\text{reg}}}_{\text{trusted}}(W_R)\): Follows from the corresponding assumption of this lemma using Lemma 44 and the definition of \(R^{\text{t}_{\text{reg}}}_{\text{trusted}}\) and the fact that integers are always in \(V^{\text{t}_{\text{reg}}}_{\text{trusted}}\).

• \(\Phi_S, \text{ms}, \text{stk}, \Phi_S, \text{stk}, \Phi_T, \text{mem}, n-1 \in W_M\): Follows from the corresponding assumption of this lemma using Lemma 44

• Theorem 2 holds for all \(n'' < n-1\) : follows from the corresponding assumption of this lemma, since \(n-1 < n\).

In case 4, we may assume \(r_1 \neq r_2\) \(4\) and \(r_1 \neq \text{pc} \in \{1, 2\}\) as this will cause the execution to fail. We need to let the target execution catch up. That is \(\Phi_T' \rightarrow_\alpha \Phi_T''\) for

\[
\Phi_T'' = \Phi_T [\text{mem}, a_{\text{stk}} \mapsto 42] \\
[\text{reg}, \text{reg}, a_{\text{stk}} \mapsto ((\text{rw}, \text{linear}), b_{\text{stk}}, a_{\text{stk}} - 1, a_{\text{stk}} - 1)] \\
[\text{reg}, \text{data} \mapsto \text{sealed}(\sigma', ((\text{rw}, \text{linear}), a_{\text{stk}}, e_{\text{stk}}, a_{\text{stk}} - 1))] \\
[\text{reg}, \text{pc} \mapsto c_1''] \\
[\text{reg}, \text{data} \mapsto c_2']
\]

where \(\Phi_T(r_1) = \text{sealed}(\sigma, c_i)\) for \(i \in \{1, 2\}\) and \(\text{reg}, \text{pc} \mapsto 15\) which is the offset to the return code. Now using Lemma 50 again, it suffices to show

\[(n-1, (\Phi_S', \Phi_T'')) \in \mathcal{O}^{\square,gc}\]

By assumption, we have \((n, (\text{reg}(r_1), \text{reg}(r_2))) \in V^{\text{t}_{\text{reg}}}_{\text{trusted}}(W_{R,i})\) for some \(W_{R,i}\) with \(i \in \{1, 2\}\). We know the capabilities in \(r_1\) and \(r_2\) are sealed capabilities, and by Lemma 44 and the definition of \(V^{\text{t}_{\text{reg}}}_{\text{trusted}}\) we get \((n-1, (c_1, c_2')) \in H_\sigma \sigma^{-1}(W_{R,i})\) and w.l.o.g.

\[
\forall W' \supseteq W_{R,1}, W_o, n'' < n, (n', (c_2, c_2')) \in H_\sigma \sigma^{-1}(W_o), (n', c_1, c_2, c_1, c_2) \in \text{xjmp}(W' \oplus W_o)
\]

\[\text{(1)}\]

If the register contains a data capability, then the execution fails in the step after the jump. If it is an executable capability, then the xjump fails as it does not permit executable capabilities for the data part.

The pc is executable which causes the xjump to fail.
Now take $W'_0 = W_{R,2}$, take $n' = n - 1$ and construct $W'_{R,1}$ as follows:

By Lemma 14 and the safety assumption on the register-file, there exists $S \supseteq [b_{stk}, e_{stk}]$ such that for some $R: S \to P(\mathbb{N})$ we have $\cup_{r \in S} R(r) \supseteq [b_{stk}, e_{stk}]$ and for all $r \in S$, $W_{R,1}.free(r).H \supseteq t_{R(r), gc}.H$ and $|R(r)| = 1$ and $W_{R,1}.free(r)$ is address-stratified. Now take $r_{priv, stk}$ fresh and define

$$W'_{R,1} = W_{R,1}[free.R^{-1}([a_{stk}, e_{stk}]) \mapsto \text{revoked}][\text{priv}.r_{priv, stk} \mapsto ([\sigma_{sta,s, \square}^{\text{prim, r_{priv, stk}}} \text{mem}[\text{data, i}, \sigma_{stat}]), gc, a + \text{call_len}]]$$

We know $W'_{R,1} \supseteq W_{R,1}$ as the revoked regions must have been spatial in $W_{R,0}$ (as they are owned by the part of the world assigned to the stack-register in the register-file relation). The static region for the private stack is an extension of the old world.

Pick this world as $W'$ in Eq 1. Let $W'_{R,2}$ be the same world but with the ownership of $W_{R,2}$ and pick it for $W'_r$. Now observe that also $W'_{R,2} \supseteq W_{R,2}$ and use monotonicity of $H_r$ with the above facts to get

$$(n - 1, c_1, c_2, c_1', c_2') \in c_{xcmp}^{\square, gc}(W'_{R,1} \sqcup W'_{R,2})$$

Now pick register files and memories such that they form $\Phi_S^g$ (defined in the assumptions) and $\Phi^T_T$ and for $W'_r$ and $W'_M$ (we define them below) show

i. $W'_{R,1} \sqcup W'_{R,2} \sqcup W'_r \sqcup W'_M$ is defined.

ii. $(n - 1, (\Phi_S^g\text{-reg}, \Phi_T^T\text{-reg})) \in R_{\text{untrusted}}^{\square, gc}(\{r_{data}\})(W'_r)

iii. $\Phi_S^g\text{-mem}, \Phi_S^g\text{-stk}, \Phi_T^T\text{-stk}, \Phi_T^T\text{-mem}: n - 1 \leftarrow W'_M$

to get

$$(n - 1, (\Phi_S^g, \Phi_T^T)) \in c_{xcmp}^{\square, gc}$$

as desired.

It remains to show Eq 2, but first we note that we can deduce the following: From the assumption $(n, [b, e]) \in \text{readXCondition}^{\square, gc}(W)$ we get $r \in \text{addressable}(\text{normal, W. heap})$ and $m_{\text{code}}$ such that $\text{dom}(m_{\text{code}}) \supseteq [b, e]$ and $W.\text{heap}(r) \supseteq t_{r_{\text{stak}}, gc}.m_{\text{code}}.gc$. Further by $m_{\text{stk}}, m_{\text{stk}}.m_{\text{reg}}: n - 1 \leftarrow W'$, we know that

$$(n, (m_{\text{code}} \sqcup m_{\text{pad}}, m_{\text{code}} \sqcup m_{\text{pad}})) \in H_{\text{code}}^{\square}.\sigma_{\text{ret}}.\sigma_{\text{global}}.m_{\text{code}}(T_A, -\sigma_{\text{global}}.\sigma_{\text{global}}.\sigma_{\text{code}})W'_r$$

(2)

where $\sigma' \in \sigma_{\text{ret}}'$ and $\text{dom}(m_{\text{code}}) \supseteq [b, e]$ and $a + \text{off}_{pc} \in \text{dom}(m_{\text{code}})$ and $W = W_r \sqcup \ldots$. That is: $m_{\text{code}}$ contains the call we are considering. This also entails

$$\sigma_{\text{ret}}'.\sigma_{\text{global}}.\sigma_{\text{code}}.\sigma_{\text{comp}}.m_{\text{code}}$$

(3)

from Eq 2 we also get

- $\text{dom}(m_{\text{code}} \sqcup m_{\text{pad}}) \subseteq T_A$

Otherwise we would have $\text{dom}(m_{\text{code}} \sqcup m_{\text{pad}}) \# T_A$ which would contradict $T_A \supseteq [b, e] \subseteq \text{dom}(m_{\text{code}} \sqcup m_{\text{pad}})$

- $\sigma_{\text{ret}}' \subseteq \sigma_{\text{global}}.\sigma_{\text{ret}}$

Follows from the above.

Case 4. Pick $W'_r$ and $W'_M$ to have the regions of $W'_{R,1}$, but where $W'_r$ owns $r_{priv, stk}$ and otherwise has the ownership of $W_r$ and $W$ with the exception of the regions owned by $W'_{R,1}$ and $W'_{R,2}$. $W'_M$ has the ownership of $W_M$. Case 4 follows from assumption $W \sqcup W' \sqcup W_M$. The only changes to the worlds is that some ownership has been shifted from $W_r$ to $W'_{R,1}$ and $W'_{R,2}$ and the ownership for $W$ now belongs to $W'_r$. In other words, no ownership has been duplicated.

Case 4.

First, from reasonability of $\Phi_S$, we get that $\Phi.\text{reg}(r)$ is reasonable in memory $\Phi.\text{mem}$ and free stack $\Phi.\text{ms stk}$ up to $n - 1$ steps for all $r \neq pc$. Lemma 20 then tells us that $(n - 1, (\Phi_S.\text{reg}, \Phi_T.\text{reg})) \in R_{\text{untrusted}}^{\square, gc}(W_R)$.

We then need to split the ownership of $W'_r$. From assumption $(n, (\Phi_S.\text{reg}, \Phi_T.\text{reg})) \in R_{\text{untrusted}}^{\square, gc}(W'_r)$, we get a way to split the ownership of $W_r$. We take this as the starting point, but with the following changes: regions $r_1$ and $r_2$ maps to worlds with no ownership (i.e. purePart$(W'_r)$). region $\text{reg}$ maps to a world with the same ownership, but of course without the now revoked regions. Region $r_{\text{data}}$ maps a world that owns private $r_{priv, stk}$ region. Finally, $r_{\text{code}}$ maps to a world with the ownership of $W_r$. We split the world in the same way for the registers that remain unchanged, and we get from Lemma 14 and 47 that $(n - 1, (\Phi_S.\text{reg}, \Phi_T.\text{reg})) \in R_{\text{untrusted}}^{\square, gc}(\{r_{data}, \text{reg}, r_{data}, r_{data}, r_{code}\})(W'_R)$ for the appropriate $W'_R$.

To obtain $(n - 1, (\Phi_S.\text{reg}, \Phi_T.\text{reg})) \in R_{\text{untrusted}}^{\square, gc}(\{r_{data}\})(W'_R)$, it remains to prove the following cases:

61
Case \( r_{stk} \): Show:

\[
(n - 1, (\text{stack}-\text{ptr}(rw, b_{stk}, a_{stk} - 1), (\text{rwl}, \text{linear}, b_{stk}, a_{stk} - 1, a_{stk} - 1)) \in \mathcal{V}^{\sqcup,gc}_{\text{untrusted}}(W'_{R,r_{stk}})
\]

We know by Assumption \((n, (\Phi_S.reg, \Phi_T.reg)) \in \mathcal{R}^{\sqcup,gc}_{\text{trust}}(W_R)\), we know

\[
(n, (\text{stack}-\text{ptr}(rw, b_{stk}, a_{stk} - 1), (\text{rwl}, \text{linear}, b_{stk}, e_{stk}, a_{stk})) \in \mathcal{V}^{\sqcup,gc}_{\text{trust}}(W_{R,stk})
\]

which by the stackReadCondition gives \(S \subseteq \text{addressable}(linear, W, \text{free})\) and \(R : S \rightarrow \mathcal{P}(\mathbb{N})\). We need to pick an \(S'\) to argue

\[
(n - 1, [b_{stk}, a_{stk} - 1]) \in \text{stackReadCondition}^{\sqcup,gc}_{\text{trust}}(W'_{R,r_{stk}})
\]

To this end pick \(S' = R^{-1}(\bigcup_{r \in S} R(r)) \setminus [a_{stk}, e_{stk}]\) and \(R'\) to be \(R\) limited to \(S'\). As we exclude all the revoked regions and the ownership otherwise remains the same in \(W'_{R,r_{stk}}\), as in \(W_{R,stk}\), the regions in \(S'\) are exactly the same in the new world as they were in \(W_{R,stk}\). So what we need to show follows immediately from

\[
(n, [b_{stk}, e_{stk}] \in \text{stackReadCondition}^{\sqcup,gc}_{\text{trust}}(W'_{R,r_{stk}})
\]

and by Lemma \[12\]

We show

\[
(n - 1, [b_{stk}, a_{stk} - 1]) \in \text{stackWriteCondition}^{\sqcup,gc}_{\text{trust}}(W'_{R,r_{stk}})
\]

in the same way.

Case \( r_{11} \): Show:

\[
(n - 1, (0, 0)) \in \mathcal{V}^{\sqcup,gc}_{\text{untrusted}}(W'_{R,r_{11}})
\]

Follows immediately from the definition.

Case \( r_{1, r_2} \): The two cases are symmetric, so we just show the \( r_1\) case:

\[
(n - 1, (\text{linearityConstraint}(c_1), \text{linearityConstraint}(\Phi_T(r_1)))) \in \mathcal{V}^{\sqcup,gc}_{\text{untrusted}}(W'_{R,r_1})
\]

Follows from the related register files assumption and Lemma \[13\] and the fact that for sealed capabilities if they are in \(\mathcal{V}^{\sqcup,gc}_{\text{trusted}}\) then they are in the \(\mathcal{V}^{\sqcup,gc}_{\text{untrusted}}\) part.

Case \( r_{\text{stored}} \): We have to show

\[
(n - 1, (\text{sealed}(\sigma', \text{ret-\text{ptr-data}}(a_{stk}, e_{stk})), \text{sealed}(\sigma', ((\text{rwl}, \text{linear}), a_{stk}, e_{stk}, a_{stk} - 1)))) \in \mathcal{V}^{\sqcup,gc}_{\text{untrusted}}(W'_{R,r_{\text{stored}}})
\]

where \(W'_{R,r_{\text{stored}}}\) is the part of \(W_R\) with ownership over \(r_{\text{priv-stk}}\).

Use the \(r = r_{\text{ms-code}}\) as the witness. The readXCondition \(\sqcup,gc\) gives us \(W.\text{heap}(r) = \text{r-code}^{\sqcup,gc}_{\text{ms-code},gc}\). As it is a pure region, it is also present in the future world we consider. We now have to show:

a) for \(n'' < n - 1\) we have

\[
(n'', (\text{ret-\text{ptr-data}}(a_{stk}, e_{stk})), ((\text{rwl}, \text{linear}), a_{stk}, e_{stk}, a_{stk} - 1)) \in H^{\text{code}}_{\text{r-code}}(\sigma_{\text{r-ret}} \sigma_{\text{r-clos}} m'_{\text{ms-code}}(T_A, \text{stk-base}) \sigma' \xi^{-1}(W'_{R,r_{\text{stored}}}))
\]

First, we already know dom\((m'_{\text{ms-code}}) \subseteq T_A\) and \(\sigma_{\text{r-ret}} \subseteq \sigma_{\text{r-glob,ret}}\). Now pick \(r_{\text{priv-stk}}\) as the witness. We immediately get dom\((m'_{\text{ms-code}}(\text{a, a + call-len - 1})) = \text{call-off}_{\text{pc, off}}(r_{1} \text{r'}_2)\) follows from callCondition\((\Phi_S, r_1, r_2, \text{off}_{\text{pc}}, \text{off}_{\sigma, a})\). Finally, \(m'_{\text{ms-code}}(\text{a + off}_{\text{pc}}) = \text{sealed}(\sigma_{\omega}, \sigma_{e})\) with \(\sigma_{\omega} = \sigma_{\omega} + \text{off-pc}\) which follows from \(\sigma_{\text{r-ret}}, \sigma_{\text{r-clos}}, \text{r-code, ms-code}\) and the fact that the call is there.

b) isLinear\((\text{sealed}(\sigma', ((\text{rwl}, \text{linear}), a_{stk}, e_{stk}, a_{stk})))\) if isLinear\((\text{sealed}(\sigma', ((\text{rwl}, \text{linear}), a_{stk}, e_{stk}, a_{stk})))\)

Trivial, both are linear.

c) \(W'' \sqsubseteq \text{purePart}(W'_{R,r_{\text{stored}}})\), \(W''_{o, n'' < n - 1,}
\]

\[
(n'', (sc_{\omega}, sc'_{\text{r}})) \in H^{\text{code}}_{\text{r-code}}(\sigma_{\text{r-ret}} \sigma_{\text{r-clos}} m'_{\text{ms-code}}(T_A, \text{stk-base}) \sigma' \xi^{-1}(W_0))
\]

Trivial as both configurations fail.
Case $\text{rcode}$: We have to show

$$\left( n - 1, \left( \text{sealed}(\sigma', \text{ret-ptr-code}(b, e, a + \text{call}_\text{len})), \text{sealed}(\sigma', ((\text{RX}, \text{normal}), b, e, a + \text{ret-pt_offset})) \right) \right) \in \mathcal{V}_{\text{untrusted}}^\square(W_{R, \text{rcode}}')$$

where $W_{R, \text{rcode}}'$ has the ownership of $W$. Just as in the previous case, we know that for some region $r$ there exists an $ms_{\text{code}}$ such that: $\text{W.heap}(r) = \sigma_{\text{code}} \triangleq ms_{\text{code}}(T_A, \text{stk_base})$ where $\sigma' \in \sigma_{\text{ret}}$ and $\text{dom}(ms_{\text{code}}) \supseteq [b, e]$ and $a + \text{off}_{\text{pc}} \in \text{dom}(ms_{\text{code}})$ and $\sigma_{\text{ret}} \leq \sigma_{\text{glob} \cdot \text{ret}}$. It follows easily from the definition of $H^\square_{\text{code}}$ that

$$\left( n'', \left( \text{ret-ptr-code}(b, e, a + \text{call}_\text{len}), ((\text{RX}, \text{normal}), b, e, a + \text{ret-pt_offset})) \right) \right) \in H_{\sigma}^\square \sigma_{\text{ret}} \sigma_{\text{close}} ms_{\text{code}}(T_A, \text{stk_base}) \sigma' \xi^{-1}(W_{R, \text{rcode}}')$$

for $n'' < n - 1$. Both capabilities are non-linear, so

$$\text{isLinear}(\text{ret-ptr-code}(b, e, a + \text{call}_\text{len})) \text{ iff } \text{isLinear}(((\text{RX}, \text{normal}), b, e, a + \text{ret-pt_offset}))$$

is indeed the case.

Finally we need to show:

$$\forall n'' \equiv \text{purePart}(W_{R, \text{rcode}}'), W_o, n'' < n - 1, \left( n'', (sc'_S, sc'_T) \right) \in H_{\sigma}^\square \sigma_{\text{ret}} \sigma_{\text{close}} ms_{\text{code}}(T_A, \text{stk_base}) \sigma' \xi^{-1}(W_o).$$

$\left( n'', \text{ret-ptr-code}(b, e, a + \text{call}_\text{len}), sc'_S, ((\text{RX}, \text{normal}), b, e, a + \text{ret-pt_offset}), sc'_T \right) \in \mathcal{E}_{\text{xjump}}(W'' \oplus W_o)$

To this end let $W'' \equiv \text{purePart}(W_{R, \text{rcode}}')$ and $W_o$ be given s.t. $W'' \oplus W_o$ is defined. Further, let $\left( n'', (sc'_S, sc'_T) \right) \in H_{\sigma}^\square \sigma_{\text{ret}} \sigma_{\text{close}} ms_{\text{code}}(T_A, \text{stk_base}) \sigma' \xi^{-1}(W_o)$ be given and show

$$\left( n'', \text{ret-ptr-code}(b, e, a + \text{call}_\text{len}), sc'_S, ((\text{RX}, \text{normal}), b, e, a + \text{ret-pt_offset}), sc'_T \right) \in \mathcal{E}_{\text{xjump}}(W'' \oplus W_o)$$

Now let $n'' \leq n''$ be given along with $\text{reg}_S(3), \text{reg}_T(3), ms_S(3), ms_T(3), ms_{\text{stk}}(3), stk(3), W''_R, W''_M$ such that

- $W'' \oplus W_o \oplus W''_R \oplus W''_M$ is defined
- $ms_S(3), ms_{\text{stk}}(3), stk(3), ms_T(3), ms_{\text{stk}}(3), W''_M$
- $\left( n'', (\text{reg}_S(3), \text{reg}_T(3)) \right) \in \mathcal{R}_{\text{untrusted}}(W_R)$

Based on $H_{\text{code}}^\square$, there are three possible values for $sc'_S$ and $sc'_T$. In the first case, $sc'_S$ is a ret-ptr-code and $sc'_T$ is a capability with permission RX. In this case, xjumpResult will produce failed configurations which are trivially in the observation relation. In the next case, it is required that $\sigma' \in \sigma_{\text{ret}}$, but this cannot be the case as $\sigma' \in \sigma_{\text{ret}}$ and we have $\sigma_{\text{ret}} \sigma_{\text{close}} = \text{comp-code} ms_{\text{code}}$, which implies that $\sigma_{\text{close}} \neq \sigma_{\text{ret}}$.

This leaves us with one final case, namely $sc'_S = \text{ret-ptr-data}(b'_{\text{stk}}, e'_{\text{stk}})$ and $sc'_T = ((\text{RX}, \text{linear}), b'_{\text{stk}}, e'_{\text{stk}}, b'_{\text{stk}} - 1)$. Further we know

$$\exists r \in \text{addressable}(\text{linear}, W_o, \text{priv}). r.H = (\text{sta}_a, a', \text{call}_\text{len} \text{ and dom}(ms_{\text{priv}, r}) = \text{dom}(ms_{\text{priv}, r}) = [b', e'] \text{ and decodeInstruction(code([a', a' + \text{call}_\text{len} - 1]))}$$

and $\text{rcode} = \text{call}_{\text{off}_{\text{pc}}, a'} r_1, r_2$ and $\text{rcode} = \text{call}_{\text{off}_{\text{pc}}, a'} r_1, r_2$. We get $a' = a$. This means that $W''_M$ and $W''_R$ have this region.

We know that the two register-files are related which in particular means that the values in register $\text{rstk}$ are related. Now consider the following cases:

- $\text{reg}_S(3)(\text{rstk}) \neq \text{stack-ptr}(\ldots, \ldots)$

  In this case due to $\text{reg}_S(3)$ being related to $\text{reg}_T(3)$, there are three cases we need to consider. In all cases, the source configuration will fail because the value in the stack register is not a stack capability. If we can argue that the target configuration will also fail, then the two are in the observation relation. First, if $\text{reg}_S(3) = \text{sealed}(\sigma_{\text{ret} \cdot \text{stk}}, sc'_S)$, then the return code will fail when the base address (a sealed capability has no base address, so the instruction returns $-1$) is compared to $\text{stk_base}$. Second, if $\text{reg}_T(3) = \text{sealed}(\ldots, \ldots)$, then the target execution fails when it attempts to splice this seal with $sc'_T$ (which we know is not a seal capability). Finally, $\text{reg}_T(3) = (\text{perm}_{\text{ret} \cdot \text{stk}}, b_{\text{ret} \cdot \text{stk}}, b_{\text{ret} \cdot \text{stk}}, e_{\text{ret} \cdot \text{stk}}, \ldots)$ and $\left( n'', [b_{\text{ret} \cdot \text{stk}}, e_{\text{ret} \cdot \text{stk}}] \right) \in \text{readCondition}^{\square} ms_{\text{stk}}(\text{stk_base}, W_{R, \text{stk}})$ is satisfied. This means that it is satisfied by some heap region, but by the memory satisfaction assumption $\text{stk_base}$ must be in the free stack part of the world. This means that the execution will fail that $\text{stk_base}$ check.
• \( \text{reg}_{S}^{(3)}(r_{stk}) = \text{stack-ptr}(\text{perm}_{\text{ret}_{stk}}, b_{\text{ret}_{stk}}, \ldots) \) and \( b_{\text{ret}_{stk}} \neq \text{stk}_{\text{base}} \)

Here the source side will fail the xjump as the base address is not \( \text{stk}_{\text{base}} \). Similarly on the target side, the return code will fail the \( \text{stk}_{\text{base}} \) check.

• \( \text{reg}_{S}^{(3)}(r_{stk}) = \text{stack-ptr}(\ldots b_{\text{ret}_{stk}}, e_{\text{ret}_{stk}}, \ldots) \) and \( b_{\text{ret}_{stk}} = \text{stk}_{\text{base}} \) and either \( e_{\text{ret}_{stk}} + 1 \neq b'_{stk} \) or \( \text{perm}_{\text{ret}_{stk}} \neq \text{rw} \) or \( b_{\text{ret}_{stk}} > e_{\text{ret}_{stk}} \).

In this case, the source configuration will fail as one of the conditions in \( \text{xjumpResult} \) will not be met. On the target side, the splice will fail as either the two capabilities being spliced don’t line up, the permissions don’t match, or the range of authority is empty, respectively.

• \( \text{reg}_{S}^{(3)}(r_{stk}) = \text{stack-ptr}(\text{perm}_{\text{ret}_{stk}}, b_{\text{ret}_{stk}}, e_{\text{ret}_{stk}}, \ldots) \) and \( b_{\text{ret}_{stk}} = \text{stk}_{\text{base}} \) and \( e_{\text{ret}_{stk}} + 1 = b'_{stk} \) and \( \text{perm}_{\text{ret}_{stk}} = \text{rw} \) and \( b_{\text{ret}_{stk}} = e_{\text{ret}_{stk}} \).

We would like to show that \( ms_{\text{priv},S}^{(3)} \) is the top most stack frame and that \( r' \) governs it. By the memory satisfaction assumption and the presence of \( r' \) in \( W_{M}^{r} \) we know that \( \text{stk}_{(3)} \) is non-empty. By the memory satisfaction on the private stack, the following must be the case:

\[
\text{stk}_{(3)} = (\text{opc}_0, ms_0), \ldots, (\text{opc}_m, ms_m) \land
\forall i \in \{0, \ldots, m\}, (\text{dom}(ms_i) \neq \emptyset) \land
\forall i < j, \forall a \in \text{dom}(ms_i), \forall a' \in \text{dom}(ms_j), \text{stk}_{\text{base}} < a < a' \
\]

Assume for contradiction \( ms_{\text{priv},S}^{(3)} \) is not the top frame. In that case \( \text{dom}(ms_0) \neq \emptyset \) and \( \forall a \in \text{dom}(ms_0) \ldots \text{stk}_{\text{base}} < a < b_{stk} \) at the same time, we know

\[
(n'', (\text{stack-ptr}(\text{rw}, \text{stk}_{\text{base}}, e_{\text{ret}_{stk}}, \ldots), (\text{rw}, \text{linear}), \text{stk}_{\text{base}}, e_{\text{ret}_{stk}}, \ldots))) \in W_{R,t_{stk}}^{\gamma, gc} \ \text{untrusted}(W_{R,t_{stk}}^{\gamma, gc})
\]

which means that the free stack part of the world contains a region that at least governs \( \text{stk}_{\text{base}}, e_{\text{ret}_{stk}} \).

Combine this with \( e_{\text{ret}_{stk}} + 1 = b'_{stk} \), we can conclude that no such address can exist in \( ms_0 \), so it must be empty, but this cannot be the case either. Therefore, the top stack frame must contain \( ms_{\text{priv},S}^{(3)} \).

Further, due to the disjointedness required by memory satisfaction, it must be \( r' \) that governs this stack frame. This also means that we have \( \text{opc}_0 = a + \text{call}_len \). With this, we have all the requirements for \( \text{xjumpResult} \) satisfied on both sides which allows us to pick the necessary configurations:

\[
\Phi_{S}^{(4)} = \text{xjumpResult}(r_{1}, r_{2}, (ms_{S}^{(3)}, \text{reg}_{S}^{(3)}, (ms_{\text{priv},S}^{(3)}, a + \text{call}_len) :: \text{stk}_{\text{ret}}, ms_{stk}^{(3)}))
\]

\[
\text{reg}_{S}^{(3)}[\text{reg}.\text{pc} \mapsto ((\text{RX}, \text{normal}), b, e, a + \text{call}_len)]
\text{reg}.\text{r}_{\text{data}} \mapsto 0]
\text{reg}.\text{r}_{\text{stk}} \mapsto \text{stack-ptr}(\text{rw}, \text{stk}_{\text{base}}, e_{stk}, e_{\text{ret}_{stk}} + 1)
\text{reg}.r_{1} \mapsto 0
\text{reg}.r_{2} \mapsto 0
\]

and

\[
\Phi_{T}^{(4)} = \text{xjumpResult}(r_{1}, r_{2}, (ms_{T}^{(3)}, \text{reg}_{T}^{(3)}))
\]

\[
\text{reg}_{T}^{(3)}[\text{reg}.\text{pc} \mapsto ((\text{RX}, \text{normal}), b, e, a + \text{ret}_p_{\text{offset}})]
\text{reg}.\text{r}_{\text{data}} \mapsto ((\text{rw}, \text{linear}), \text{stk}_{\text{base}}, e_{\text{ret}_{stk}})]
\]

It now remains to show

\[
(n'', (\Phi_{S}^{(4)}, \Phi_{T}^{(4)})) \in O_{\gamma, gc}^{\square}
\]

Use Lemma [50] by which it suffices to show the following two things:

- \( \Phi_{T}^{(4)} \rightarrow^{l} \Phi_{T}^{(5)} = \text{ms}_{T}^{(3)}, \text{reg}_{S}^{(3)}[\text{reg}.\text{pc} \mapsto ((\text{RX}, \text{normal}), b, e, a + \text{call}_len)]
\text{reg}.\text{r}_{\text{data}} \mapsto 0]
\text{reg}.\text{r}_{\text{stk}} \mapsto ((\text{rw}, \text{linear}), \text{stk}_{\text{base}}, e_{\text{ret}_{stk}} + 1)
\text{reg}.r_{1} \mapsto 0
\text{reg}.r_{2} \mapsto 0
\)

for some number of steps \( l \). This follows immediately from the operational semantics.

- \( \left( n'', (\Phi_{S}^{(4)}, \Phi_{T}^{(5)}) \right) \in O_{\gamma, gc}^{\square} \)

For fresh \( r_{u_{stk}}^{r'}, \ldots, r_{c_{stk}}^{r'} \) and \( W''^{r} \) defined as

\[ W''^{r} = W''^{r}[\text{priv}, r' \mapsto \text{revoked}, \text{free}, r_{u_{stk}}^{r'}, \ldots, r_{c_{stk}}^{r'} \mapsto ( l_{\text{std}, S}^{u_{stk}}), gc \ldots ( l_{\text{std}, S}^{c_{stk}}), gc ] \]

and \( W_{M}^{r} \) the same as \( W''^{r} \), but with the ownership of \( W''^{r} \) as well as for the regions \( r_{u_{stk}}^{r'} \ldots r_{c_{stk}}^{r'} \), and \( W_{M}^{r} \) the same as \( W''^{r} \) but with the ownership of \( W''_{M}^{r} \), we show the following:
1. \( (n', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W'') \)

2. \( \Phi^S_{\text{mem}, \text{stk}}, \Phi^S_{\text{ms,stk}}, \Phi^T_{\text{mem, stk}} \in W'' \)

3. \( W'' \oplus W' \oplus W'_M \) defined

Assuming the above, we use our assumption that Theorem 2 holds for all \( n' < n \) to get

\[ (n'', (((\text{rx}, \text{normal}), b, e, a + \text{call_len}), ((\text{rx}, \text{normal}), b, e, a + \text{call_len})) \in \mathcal{E}^\square_{\text{gc}}(W'') \]

Note that from assumption “\( \Phi_S \) reasonable up to \( n \) steps” and Lemma 16 and \( n'' \leq n' < n - 1 \) we get “\( \Phi_S \) reasonable up to \( n'' + 1 \) steps” from which it follows that “\( \Phi_S(\text{pc}) + \text{call_len} \) reasonable up to \( n'' \) steps”. Using this along with the register-file safety and memory satisfaction, we get

\[ (n'', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W'') \]

as desired. We need to show the three things we skipped:

Show:

\[ (n'', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W'') \]

We will split the world like in \( (n'', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W) \), but where \( r_{\text{stk}} \) also get the ownership of \( r_{\text{stk}}' \). We need to show the following:

* Case \( r_{\text{data}}, r_1, r_2 \):
  
  Trivial.

* Case \( r \notin \{r_{\text{data}}, r_1, r_2, \text{pc, stk} \} \):
  
  We have \( \Phi^S_{\text{reg}}(r) = \Phi^T_{\text{reg}}(r) \) and \( \Phi^T_{\text{reg}}(r) = \Phi^T_{\text{reg}}(r) \).
  
  We already know:

\[ (n'', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W'') \]

This is true for some \( W''_{R,r} \) which does not have the ownership of the regions that are revoked in \( W''_{R,r} \), so that we can take the corresponding \( W''_{R,r} \) and have \( W''_{R,r} \equiv W''_{R,r} \). From Lemma 47 we then get

\[ (n'', (\Phi^S_{\text{reg}}, \Phi^T_{\text{reg}})) \in \mathcal{R}^\square_{\text{untrusted}}(W'') \]

as desired.

* Case \( r_{\text{stk}} \):
  
  For this case, we need to show

\[ (n'', [\text{stk_base}, e_{\text{stk}}]) \in \text{stackReadCondition}^\square_{\text{gc}}(W'') \]

and

\[ (n'', [\text{stk_base}, e_{\text{stk}}]) \in \text{stackWriteCondition}^\square_{\text{gc}}(W'') \]

For \( W''_{R,\text{stk}} \) that owns the same as \( W''_{R,\text{stk}} \) as well as \( r_{\text{stk}}' \).

For the first part, we use

\[ (n'', (\mathcal{S}_{\text{reg}}(r_{\text{stk}}), \mathcal{S}_{\text{reg}}(r_{\text{stk}}))) \in \mathcal{Y}^\square_{\text{untrusted}}(W''_{R,\text{stk}}) \]

and the fact that the stack capability must have \( \text{rw} \) permission from which it follows that

\[ (n'', [\text{stk_base}, e_{\text{ret,stk}}]) \in \text{stackReadCondition}^\square_{\text{gc}}(W'') \]

which gives us \( S_{\text{free}} \subseteq \text{addressable}(\text{linear}, W, \text{free}) \) and \( R_{\text{free}} : S_{\text{free}} \to \mathcal{P}(\mathbb{N}) \) such that

- \( \forall r \in S_{\text{free}}, |R(r)| = 1 \)
- \( \bigcup_{r \in S_{\text{reg}}} R(r) \subseteq [\text{stk_base}, e_{\text{ret,stk}}] \)
- \( \forall r \in S_{\text{free}}, W_{\text{free}}(r) \supseteq n_{\text{std,so}, \square} \)

Now pick \( S_{\text{read}} = S_{\text{free}} \cup \{ r_{\text{stk}}', \ldots, r_{\text{stk}}'' \} \) and

\[ R_{\text{read}}(r) = \begin{cases} R_{\text{read}}(r) & r \in S_{\text{read}} \\ \{ a \} & r = r_{\text{stk}} \end{cases} \ 	ext{such that} \ r = r_{\text{stk}} \]

It is clearly the case that \( \forall r \in S_{\text{read}}, |R_{\text{read}}(r)| = 1 \) and \( \bigcup_{r \in S_{\text{read}}} R(r) \subseteq [\text{stk_base}, e_{\text{ret,stk}}] \). For

\[ \forall r \in S_{\text{free}}, W_{\text{free}}(r) \supseteq n_{\text{std,so}, \square} \]

it follows from \( \forall r \in S_{\text{free}}, W_{\text{free}}(r) \supseteq n_{\text{std,so}, \square} \)

and \( W_{\text{free}}(r) \supseteq n_{\text{std,so}, \square} \) for \( r \in \{ r_{\text{stk}}', \ldots, r_{\text{stk}}'' \} \).

The stackReadCondition is shown in the same way, but we also use Lemma 70 to argue that the new regions are address-stratified.
Show:

\[ ms^{(3)}_S, \Phi^{(4)}_S, stk, ms^{(3)}_{stk} \uplus ms^{(3)}_{priv, S}, ms'^{\tau}_T, gc \ W''_M \]

From the assumption \( ms^{(3)}_S, ms^{(3)}_{stk}, stk^{(3)}_T, gc_T, W''_M \), we know

\[ stk = (a + \text{call}\_len, ms^{(3)}_{priv, S}) :: (\text{opc}_1, ms_1) :: \cdots :: (\text{opc}_m, ms_m) \]

\[ ms^{(3)}_S \uplus ms^{(3)}_{stk} \uplus ms^{(3)}_{priv, S} \uplus ms_1 \uplus \cdots \uplus ms_m \]

\[ W''_M = W''_{stack} \uplus W''_{free\_stack} \uplus W''_{heap} \]

\[ ms_{T, stack}, ms_{T, free\_stack}, ms_{T, heap}, ms_{T, f}, ms_{S, f}, ms'_S, \sigma \]

such that

\[ ms^{(3)}_S = ms_{f, S} \uplus ms'_{S} \]

\[ ms^{(3)}_T = ms_{T, stack} \uplus ms_{T, free\_stack} \uplus ms_{T, heap} \uplus ms_{T, f} \]

\[ (n''_m, (stk, ms_{T, stack})) \in S^{gc}(W''_{stack}) \]

\[ (n''_m, (ms^{(3)}_{stk}, ms_{T, free\_stack})) \in F^{gc}(W''_{free\_stack}) \]

\[ (n''_m, (\sigma, ms'_S, ms_{T, heap})) \in H(W, heap)(W''_{heap}) \]

We will pick the same things to show 5 with a few changes. We have to show

\[ \Phi^{(4)}_S, stk = (\text{opc}_1, ms_1) :: \cdots :: (\text{opc}_m, ms_m) \]

By the memory satisfaction assumption and the change to the stack.

\[ ms^{(3)}_S \uplus ms^{(3)}_{stk} \uplus ms^{(3)}_{priv, S} \uplus ms_1 \uplus \cdots \uplus ms_m \]

By the memory satisfaction assumption.

\[ W''_M = W''_{stack} \uplus W''_{free\_stack} \uplus W''_{heap} \]

Define the new worlds to have the ownership of their \( W''_M \) counterparts except \( W''_{stack} \) does not take ownership of the regions used for the safety of addresses \( b'_{stk}, \ldots, e'_{stk} \). This ownership goes to \( W''_{free\_stack} \) instead.

We need to pick partitions of \( ms^{(3)}_T \) and a frame for \( ms^{(3)}_S \). We pick the same as we get from the memory satisfaction assumption except we pick the free stack partition of the target memory to be \( ms_{T, free\_stack} \uplus ms^{(3)}_{priv, S} \) and the stack partition to be \( ms_{T, stack} \uplus ms_{T, stack}|_{dom(ms^{(3)}_{priv, S})} \)

\[ ms^{(3)}_S = ms_{f, S} \uplus ms'_{S} \]

By assumption.

\[ ms^{(3)}_T = ms_{T, stack} \uplus ms_{T, free\_stack} \uplus ms_{T, heap} \uplus ms_{T, f} \]

By assumption and the fact that the only change is that we moved part of the stack to the free stack.

\[ (n''_m, (\text{opc}_1, ms_1) :: \cdots :: (\text{opc}_m, ms_m), ms_{T, stack} \uplus ms_{T, stack}|_{dom(ms^{(3)}_{priv, S})}) \in S^{gc}(W''_{stack}) \]

Follows easily from the private stack satisfaction assumption. The distribution functions from the assumption are simply limited to forget about the now revoked region and extend the world partition to be on \( W''_M \) but with the same ownership as in the one we had in the assumption.

All the new partitions are future worlds of the old ones as none of them owned the revoked region (it was owned by \( W''_R \)).

\[ (n''_m, (ms^{(3)}_{stk} \uplus ms^{(3)}_{priv, S}, ms_{T, free\_stack} \uplus ms_{T, stack}|_{dom(ms^{(3)}_{priv, S})})) \in F^{gc}(W''_{free\_stack}) \]

From the \( (n''_m, (ms^{(3)}_{stk}, ms_{T, free\_stack})) \in F^{gc}(W''_{free\_stack}) \) assumption we get \( R_{ms} : dom(\text{active}(W''_{free\_stack})) \)

MemorySegment \times MemorySegment and \( R_{W} : dom(\text{active}(W''_{free\_stack}), free) \rightarrow \text{World}_{\text{private stack}} \)

for which

\[ ms^{(3)}_{stk} = \bigcup_{r \in dom(\text{active}(W''_{free\_stack}))} \pi_1(R_{ms}(r)) \]

\[ ms_{T, free\_stack} = \bigcup_{r \in dom(\text{active}(W''_{free\_stack}))} \pi_2(R_{ms}(r)) \]

\[ stk_{base} \in dom(ms^{(3)}_{stk}) \text{ and } stk_{base} \in dom(ms_{T, free\_stack}) \]

\[ W''_{free\_stack} = \bigcup_{r \in dom(\text{active}(W''_{free\_stack}))} R_{W}(r) \]

\[ \forall r \in dom(\text{active}(W''_{free\_stack})), n''_m < n''_m, R_{ms}(r)) \in W''_{free\_stack}-free(r).H \xi^{-1}(R_{W}(r)) \]

Now pick

\[ R'_{ms}(r) = \begin{cases} (ms^{(3)}_{priv,S}|_{a'}, ms_{T, stack}|_{a'}) & r_{a'} \in \{r_{a_1} \ldots r_{a_k}\} \\ R_{ms}(r) & \text{otherwise} \end{cases} \]

and

\[ R'_{W}(r) = \begin{cases} W''_{a'} & r_{a'} \in \text{revoked, free, } r_{a_1} \ldots r_{a_k} \\ R_{W}(r) & \text{otherwise} \end{cases} \]
for $W''_a$ constructed as follows: $W''_{\text{stack}}$ is the part of the world given to the stack judgement in the assumption. This world is split into a number of parts to satisfy the memory interpretation of each frame. Say $W''_{\text{top,stack}}$ is used for the top stack frame in the assumption. The top stack frame is governed by a static region, so by definition it is split into parts that satisfy each of the addresses. That is for $a' \in \{b_{\text{stk}} \ldots e_{\text{stk}}\}$, $W''_o$ is the part that makes the value in memory satisfy the value relation. Now let $W''_a$ be $W''$ but with the ownership of $W''_o$.

It is easy to see that the $R_{ms}$s constructs the two memories, and from the assumption, we also get that the base stack address is in there.

It remains to show

$$\forall r \in \text{dom}(\text{active}(W''_{\text{free,stack}}.\text{free})), n'' < n''. (n'', R_{ms}(r)) \in W''_{\text{free,stack}}.\text{free}(r).H \xi^{-1}(r_W(r))$$

for $r \in \text{dom}(\text{active}(W''_{\text{free,stack}}.\text{free})) \setminus \{r_{\text{bk}} \ldots r_{\text{en}}\}$, it follows from monotonicity of the $H$ (memory interpretation) function. For $r_{a'} \in \{r_{\text{bk}} \ldots r_{\text{en}}\}$ and $n'' < n''$, we need to show

$$\left( n'', (ms^{(3)}_{\text{prev,S}}(a'), ms_{T,\text{stack}}(|a'|)\right) \in i'_{\text{std},a,\text{gc}}(H(\xi^{-1}r_W(r_{a'})))$$

which amounts to showing $\text{dom}(ms^{(3)}_{\text{prev,S}}(\{a'\})) = \text{dom}(ms_{T,\text{stack}}(|\{a'\}|)(a'))$, which is the case, and

$$\left( n'', (ms^{(3)}_{\text{prev,S}}(a'), ms_{T,\text{stack}}(|a'|)\right) \in \gamma_{\text{std},a,\text{gc}}(W''_a)$$

Which follows from Lemma 17 and the fact that we have a memory satisfaction assumption in which $ms^{(3)}_{\text{prev,S}}$ is governed by a standard static region.

* $(n'', (\sigma, ms, ms_{T,\text{heap}})) \in H(W_{\text{heap}}(W''_a))$

Following by Lemma 17 and the heap satisfaction assumption.

Argue:

$W'' \oplus W''_R \oplus W''_M$ is defined.

This follows from the assumption $W'' \oplus W''_R \oplus W''_M$ and the fact that each of the new worlds is constructed from one of the past worlds and only one of them claims the ownership of the new regions.

This concludes case 11.

Case 12: we need to show:

$$\Phi_{\sigma,\text{mem}, \Phi_{\sigma,\text{stk}}, \Phi_{\sigma,\text{ms,stk}}, \Phi_{T,\text{mem}}^{gc} n-1 W'_M$$

which amounts to

$$\Phi_{\sigma,\text{mem}, ((a + \text{call_len})}, ms_{\text{stack,prev,S}}) :: \Phi_{\sigma, \Phi_{\sigma,\text{ms,stk}} - \Phi_{\sigma,\text{ms,stk}}[a_{\text{stk}}, e_{\text{stk}}]}, \Phi_{T,\text{mem}}^{gc} n-1 W'_M$$

for $ms_{\text{stack,prev,S}} = \Phi_{\sigma,\text{ms,stk}}[a_{\text{stk}}, e_{\text{stk}}]$.

In order to show this, we will first show the following:

- $b_{\text{stk}} = \text{stk_base}$

  We know $\Phi_S$ is reasonable up to $n$ steps. Further, from $\text{callCondition}(\Phi_S, r_1, r_2, \text{off}_{pc}, \text{off}_{pc}, a)$ and $b \leq a$ and $a + \text{call_len} - 1 \leq e$ and $[b, e] \subseteq T_A$ we can conclude that $\Phi_S$ points to $\text{call}_{\text{off}_{pc}, \text{off}_{pc}, r_1, r_2}$ in $T_A$. By the Guarantee stack base address before call we then know $\Phi_S(r_{\text{stk}}) = \text{stack-ptr}(\omega, \text{stk_base}, \omega)$.

By assumption we have $\Phi_{\sigma,\text{mem}, \Phi_{\sigma,\text{stk}}, \Phi_{\sigma,\text{ms,stk}}, \Phi_{T,\text{mem}}^{gc} n W_M$ which gives us $ms_{T,\text{stack}}, ms_{T,\text{free,stack}}, ms_{T,\text{heap},}, ms_{T,f}, ms_{F}, ms_{F'}, W_{M,\text{stack}}, W_{M,\text{free,stack}}, W_{M,\text{heap}}$ such that:

- $\Phi_{\sigma,\text{stk}} = (\text{opc}_o, ms_o) \ldots (\text{opc}_m, ms_m)$
- $ms_o \cup \ldots \cup ms_m \cup \Phi_{\sigma,\text{mem}} \cup \Phi_{\sigma,\text{ms,stk}}$
- $W_M = W_{M,\text{stack}} \oplus W_{M,\text{free,stack}} \oplus W_{M,\text{heap}}$
- $\Phi_{\sigma,\text{mem}} = ms_{F} \cup ms_{F'}$
- $\Phi_{T,\text{mem}} = ms_{T,\text{stack}} \cup ms_{T,\text{free,stack}} \cup ms_{T,\text{heap}} \cup ms_{T,\text{f}}$
- $(n, (\Phi_{\sigma,\text{stk}}, ms_{T,\text{stack}})) \in S^{gc}(W_{M,\text{stack}})$
- $(n, (\Phi_{\sigma,\text{ms,stk}}, ms_{T,\text{free,stack}})) \in S^{gc}(W_{M,\text{free,stack}})$
- $(n, (\sigma, ms'_{\sigma, ms_{T,\text{heap}}})) \in H(W_{M,\text{heap}}(W_{M,\text{heap}})$
In order to prove the memory satisfaction, we pick the same memories except for the following changes:

- The target free stack partition: \( ms_T,free_{stack} \setminus ms_T,free_{stack}|_{\a_{st},e_{st}} \)
- The target private stack partition: \( ms_T,stack \setminus ms_T,free_{stack}|_{\a_{st},e_{st}} [a_{stk} \mapsto 42] \)

and the worlds

- Free stack world: \( W'_M,free_{stack} \) is \( W'_R,1 \) with the ownership of \( W_M,free_{stack} \) except that it gives up any ownership that we used for the safety of the addresses \( [e_{st}, e_{st}] \).
- Private stack world: \( W'_M,stack \) is \( W'_R,1 \) with the ownership of \( W_M,stack \) except that it takes the ownership that \( W_M,free_{stack} \) used for the safety of the addresses \( [a_{st}, e_{st}] \).
- Heap world: \( W'_M,heap \) is \( W'_R,1 \) with the ownership of \( W_M,heap \) which gives us \( W'_M,heap \sqsubset W_M,heap \).

We now need to show:

- \( \text{stk} = ((a + \text{call}_\text{len}), ms_{stk,priv,S}) :: \text{opc}_0, ms_0 \ldots (\text{opc}_m, ms_m) \)
  Trivial.
- \( ms_{stk,priv,S} \sqcup ms_0 \sqcup \ldots \sqcup ms_m \sqcup \Phi_G,mem \sqcup \Phi_G,ms_{stk} = \Phi_G,ms_{stk}|_{\a_{st},e_{st}} \)
  Follows by assumption and the fact that we have shuffled around some memory\(^5\)
- \( W'_M = W'_M,stack \sqcup W'_M,free_{stack} \sqcup W'_M,heap \)
  This follows by assumption and the fact that all ownership we have added to a world has been removed from another.
- \( \Phi_G,mem = ms_{f,S} \sqcup ms_S \)
  By assumption.
- \( \Phi_T,mem = ms_{T,stack} \sqcup ms_{T,free_{stack}}|_{\a_{st},e_{st}} [a_{stk} \mapsto 42] \sqcup ms_{T,free_{stack}} \sqcup ms_{T,heap} \sqcup ms_{T,f} \)
  Follows by assumption and the fact that we have shuffled around some memory\(^6\)
- \( \{n - 1, ((a + \text{call}_\text{len}), ms_{stk,priv,S}) :: \Phi_G,ms_{T,stack} \sqcup ms_{T,free_{stack}}|_{\a_{st},e_{st}} [a_{stk} \mapsto 42]) \} \in S^{\preceq}(W'_M,stack) \)

For most of the conditions, they follow from the previous stack satisfaction assumption. The only challenge is to argue that the new stack frame satisfies the conditions. First of all, we know \( ms_{stk,priv,S} \) is non-empty as it at least contains address \( a_{st} \). Next, we need to argue that

\[
\forall i \in \{0, \ldots, m\}, \\
\forall a \in \text{dom}(ms_{stk,priv,S}). \forall a' \in \text{dom}(ms_j). \text{stk}_\text{base} < a < a' \wedge
\]

The first bit, \( \text{stk}_\text{base} < a \) for \( a \in \text{dom}(ms_{stk,priv,S}) \) follows from the fact that \( a_{st} \) is the smallest address of \( a \in \text{dom}(ms_{stk,priv,S}) \) and by assumption \( \text{stk}_\text{base} < a_{st} \).

Now assume for contradiction that there exists \( a' \in \text{dom}(ms_j) \) for some \( j \) such that \( a' \leq a \) for some \( a \in \text{dom}(ms_{stk,priv,S}) \). By assumption we have \( \text{stk}_\text{base} < a' \) so \( a' \in [\text{stk}_\text{base} + 1, a] \subseteq [\text{stk}_\text{base} + 1, a_{st}] \subseteq [\text{stk}_\text{base}, e_{st}] \) which means that \( a' \) is an address governed by a stack pointer. By the register-safety assumption this must mean that it is an address of the free part of the stack. At the same time, it must be an address of the private stack because \( ms_j \) is part of the stack from the original configuration. This contradicts the initial memory satisfaction assumption as the different parts must be disjointed.

From the stack satisfaction assumption, we get \( R_{ms} \) and \( R_W \). Pick

\[
R_{ms}(r) = \begin{cases} 
(ms_{stk,priv,S}, (a + call_len), ms_{T,free_{stack}}|_{\a_{st},e_{st}}|_{a_{stk} \mapsto 42}) & \text{for } r = r_{priv_{stk}} \\
R_{ms}(r) & \text{otherwise}
\end{cases}
\]

and for \( R_W' \) pick

\[
R_W'(r) = \begin{cases} 
W_{M,free_{stack}}|_{\a_{st},e_{st}} [\text{free}, R^{-1}([a_{st}, e_{st}]) \mapsto \text{revoked}] [\text{priv}, T_{priv_{stk}} \mapsto \{t_{\text{sta}_s}\}, g_{c}, a + call_len] & \\
R_W(r)[\text{free}, R^{-1}([a_{st}, e_{st}]) \mapsto \text{revoked}] [\text{priv}, T_{priv_{stk}} \mapsto \{t_{\text{sta}_s}\}, g_{c}, a + call_len] & \text{for } r = r_{priv_{stk}} \\
R_W(r) & \text{otherwise}
\end{cases}
\]

where \( W_{M,free_{stack}}|_{\a_{st},e_{st}} \) is the world that the free stack assumption uses to satisfy that range of addresses.

\(^5\)Thanks to the register-file safety assumption we know for sure that the memory we remove from the free stack is actually there.

\(^6\)Thanks to the register-file safety assumption we know for sure that the memory we remove from the free stack is actually there.
Most of the condition trivially holds. The only one that requires some argumentation is

\((n', (ms_{stk, priv, S}, ms_{T, free_stack}|_{a_{stk}, e_{stk}}) | a_{stk} \mapsto 42)) \in t_{\text{std}, \text{so}, \square}^{(ms_{stk, priv, S}, \Phi_T^{\text{mem}}|_{a_{stk}, e_{stk}}); gc, H \xi^{-1}(R'_{W}(r_{priv, stk}))}

for all \(n' < n - 1\). Where \(R'_{W}(r_{priv, stk})\) is the part of \(W_{R, stack}\) with the ownership used for addresses \([a_{stk}, e_{stk}]\) in the memory satisfaction assumption.

As we have the safe register-file assumption, we know that the stack capability is safe. This means that addresses \([a_{stk}, e_{stk}]\) must be part of the free stack. Further, \(\Phi_T^{\text{mem}} = \Phi_T^{\text{mem}}[a_{stk} \mapsto 42]\) and \(ms_{T, free_stack} \subseteq \Phi_T^{\text{mem}}\) and \(\text{dom}(ms_{T, free_stack}) \supseteq [a_{stk}, e_{stk}]\) from which it follows that the memories are equal to the one of the static region.

It remains to show

\[\forall a \in [a_{stk}, e_{stk}], (n', (ms_{stk, priv, S}(a), ms_{T, free_stack}|_{a_{stk}, e_{stk}}) | a_{stk} \mapsto 42(a)) \in V_{\text{untrusted}}^{\square, gc}(W'_{R,a})\]

for \(n' < n - 1\). For \(a = a_{stk}\) it is trivial as we have to show

\[(n - 1, (42, 42)) \in V_{\text{untrusted}}^{\square, gc}(W'_{R,a})\]

for \(a \in [a_{stk} + 1, e_{stk}]\) we need to show

\[(n', (ms_{stk, priv, S}(a), ms_{T, free_stack}|_{a_{stk}, e_{stk}}) | a_{stk} \mapsto 42(a)) \in V_{\text{untrusted}}^{\square, gc}(W'_{R,a})\]

for \(n' < n - 1\). If we can show

\[(n', (\Phi_S, ms_{stk}(a), ms_{T, free_stack}(a))) \in V_{\text{untrusted}}^{\square, gc}(W_{R,a})\]

for \(n' < n - 1\), then we are done by monotonicity of \(V_{\text{untrusted}}^{\square, gc}\).

By assumption we know \((n, (\Phi_S, reg, \Phi_T, reg)) \in R_{\text{untrusted}}^{\square, gc}(W_{R})\) which entails \((n, (\Phi_S, reg(r_{stk}), \Phi_T, reg(r_{stk}))) \in R_{\text{untrusted}}^{\square, gc}(W_{R, stk})\). We know \(\Phi_S, reg(r_{stk}) = \text{stack-ptr}(rw, stk_base, e_{stk}, a_{stk})\), so by the definition of \(V_{\text{untrusted}}^{\square, gc}\) we get

\[(n, [stk_base, e_{stk}]) \in \text{stackReadCondition}^{\square, gc}(W_{R, stk})\]

which in turn gives us \(S_{stk} \subseteq \text{addressable}(\text{linear}, W_{free})\) and \(R_{stk} : S_{stk} \rightarrow \mathcal{P}(\mathbb{N})\) for which

\[\forall r \in S_{stk}, \vert R_{stk}(r) \vert = 1\]

\[\forall r \in S_{stk}, R_{stk}(r) \supseteq [\text{stk_base}, e_{stk}]\]

\[\forall r \in S_{stk}, W_{R, stk}(r).H \xi \subseteq t_{\text{std}, \text{so}, \square}^{(r_{stk})(r), gc}\]

Further, we know \((n, (\Phi_S, ms_{stk}, ms_{T, free_stack})) \in F^{gc}(W_{M, free_stack})\), which means that we have \(R_{ms} : \text{dom}(active(W_{M, free_stack})) \rightarrow \text{MemSeg} \times \text{MemorySegment} \) and \(R_{W} : \text{dom}(W_{M, free_stack}) \rightarrow \text{World}_{\text{private stack}}\).

\(R_{W}\) distributes the ownership of \(W_{M, free_stack}\) and \(R_{ms}\) partitions the memories.

We know that all regions in \(S_{stk}\) govern singleton memory segments, so \(R_{ms}\) must map to singleton memory segment pairs for \(r \in S_{stk}\). Further, by definition of the free stack satisfaction for \(r \in S_{stk}\) we have

\[(n', R_{ms}(r)) \in W_{R, stk}(r).H \xi^{-1}(W_{R}(r))\]

for \(n' < n\) which entails

\[(n - 1, R_{ms}(r)) \in t_{\text{std}, \text{so}, \square}^{(r_{stk})(r), gc, H \xi^{-1}(W_{R}(r))}\]

which entails

\[(n - 1, R_{ms}(r)(a)) \in V_{\text{untrusted}}^{\square, gc}(W_{R}(r))\]

for \(a \in R_{stk}(r)\).

Now, using Lemma 44, this is exactly what we wanted to show because \(W_{R}(r)\) is what we picked as \(W_{R,a}\).

• \((n - 1, (\Phi_S, ms_{stk} - \Phi_S, ms_{stk}|_{[a_{stk}, e_{stk}]}, ms_{T, free_stack} \setminus ms_{T, free_stack}|_{[a_{stk}, e_{stk}]}) \in F^{gc}(W_{M, free_stack})\):

From the safety assumption on the stack capability, we can deduce a number of things:

- \([a_{stk}, e_{stk}]\) must have been part of the free stack
- for every address in \([a_{stk}, e_{stk}]\) there is a region for that singleton memory segment.

The first part means that we do indeed remove all of the memory we try to subtract. The latter means that we can reuse the same split of the remaining memory and the world ownership as we get from assumption \((n, (\Phi_S, ms_{stk}, ms_{T, free_stack})) \in F^{gc}(W_{M, free_stack})\). Using this, the result follows from monotonicity of the \(H\) function.
\( (n - 1, (\overline{\sigma}, ms_S, ms_{T,heap})) \in \mathcal{H}(W_{M,heap})(W_{M,heap}') \):

Follows from Lemma 69, Lemma 44, the fact that the heap part of the world remains unchanged and 
\( W_{M,heap}' \supseteq W_{M,heap} \).

Case 5.

First show that
\[ (n - 1, \{(\text{RX, normal}), b, e, a\}, \{(\text{RX, normal}), b, e, a\})) \in \mathcal{V}_{\text{tst}}^{\sqsupset,gc}(\text{purePart}(W_R)) \]

First observe that by Lemma 8, purePart\( (W_R) = \text{purePart}(W_{pc}) \). Further, by assumption, Lemma 44, Lemma 8, and Lemma 29, we have
\[ (n - 1, [b, e]) \in \text{readXCondition}^{\sqsupset,gc}(\text{purePart}(W_R)) \]

If \( \text{tst} = \text{trusted} \), then by assumption we have \([b, e] \subseteq T_A \) which means that all the conditions for are met for the capability pair to be in the trusted part of the value relation.

If \( \text{tst} = \text{untrusted} \), then we need to show that the capability pair is in the untrusted part of the value relation which means that we need to show:

- \( (n - 1, [b, e]) \in \text{readCondition}^{\sqsupset,gc}(\text{normal, purePart}(W_R)) \)
  This follows by assumption, Lemma 29, Lemma 8, and Lemma 14.
- \( (n - 1, [b, e]) \in \text{readXCondition}^{\sqsupset,gc}(\text{purePart}(W_R)) \)
  We already showed this.
- \( (n - 1, [b, e]) \in \text{executeCondition}^{\sqsupset,gc}(\text{purePart}(W_R)) \)
  To this end let \( W' \supseteq \text{purePart}(W_R) \) and \( n' < n - 1 \) and \( a' \in [b', e'] \) be given, and show
  \[ (n', \{(\text{RX, normal}), b', e', a'\}, \{(\text{RX, normal}), b', e', a'\})) \in \mathcal{E}^{\sqsupset,gc}(W') \]
  This follows immediately from the FTLR: we know that \([b, e] \neq T_A \) since \( \text{tst} = \text{untrusted} \) and we know that \( n' < n - 1 < n \).

Now show
\[ (n - 1, (\Phi_{S, reg}, \Phi_{T, reg})) \in \mathcal{R}^{\sqsupset,gc}(W_R) \]

Note that we know that \( \Phi_{S, reg}(\text{pc}) \) is not linear (which will sometimes help to eliminate some cases).

By assumption we have
\[ (n, (\Phi_{S, reg}, \Phi_{T, reg})) \in \mathcal{R}^{\sqsupset,gc}(W_R) \]
which gives us \( R_R : (\text{RegisterName} \setminus \{\text{pc}\}) \rightarrow \text{World} \) such that \( W_R = \bigoplus_{r \in \text{RegisterName} \setminus \{\text{pc}\}} R_R(r) \) and for all \( r \) in \( \text{RegisterName} \setminus \{\text{pc}\} \) we have \( (n, (\Phi_{S, reg}(r), \Phi_{T, reg}(r))) \in \mathcal{R}^{\sqsupset,gc}(R_R(r)) \). To this end, we need to consider each of cases 5.1, 5.2, 5.3, 5.4.

- **Case 5.1.**
  Pick \( R_R \) as the ownership distribution. For \( r \neq r_1 \) it follows by assumption and Lemma 44. For \( r = r_1 \) it also follows by assumption and Lemma 44, 51, and 54.

- **Case 5.2.**
  Pick \( R_R \) as the ownership distribution. For \( r \neq r_1 \) it follows by assumption and Lemma 44. For \( r = r_1 \) it also follows by assumption and Lemma 44, 52, and 59.

- **Case 5.3.**
  Pick \( R_R \) as the ownership distribution. For \( r \neq r_1 \) it follows by assumption and Lemma 44. For \( r = r_1 \) it also follows by assumption and Lemma 44 and 53.

- **Case 5.4.** Pick the ownership distribution based on the linearity of \( w_2 \): If \( \text{isLinear}(w_2) \), then pick
  \[
  R'_R(r) = \begin{cases} 
  R_R(r_2) \oplus R_R(r_1) & r = r_1 \\
  \text{purePart}(W_R) & r = r_2 \\
  R_R(r) & \text{otherwise}
  \end{cases}
  \]
  if \( \neg \text{isLinear}(w_2) \), then pick
  \[ R'_R(r) = R_R(r) \]
In the case where \textit{isLinear}(w_2), we may assume \( r_2 \neq \text{pc} \). We need to show (assuming \( r_1 \neq \text{pc} \))

\[
(n - 1, (\Phi_S(r_1), \Phi_T(r_1))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r_2) \oplus R_R(r_1))
\]

which is

\[
(n - 1, (\Phi_S(r_2), \Phi_T(r_2))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r_2) \oplus R_R(r_1))
\]

this follows by assumption and Lemma \[44\] and \[13\].

We also need to show

\[
(n - 1, (\Phi'_S(r_2), \Phi'_T(r_2))) \in \mathcal{V}_{\text{tst}}^{gc}(\text{purePart}(W_R))
\]

which is trivial as \( \Phi'_S(r_2) = \Phi'_T(r_2) = 0 \).

Finally for \( r \neq r_1, r_2, \text{pc} \)

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r))
\]

Follows by assumption and Lemma \[44\].

In the case where \( \neg \text{isLinear}(w_2) \)

If \( r_2 \neq \text{pc}, \) then for \( r \neq r_1 \)

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r))
\]

Follows by assumption and Lemma \[44\].

For \( r \neq r_1, r_2 \)

\[
(n - 1, (\Phi'_S(r_1), \Phi'_T(r_1))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r_1))
\]

amounts to

\[
(n - 1, (\Phi_S(r_2), \Phi_T(r_2))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r_1))
\]

By assumption and Lemma \[8\] and \[29\].

If \( r_2 = \text{pc}, \) show

\[
(n - 1, (\Phi_S(\text{pc}), \Phi_T(\text{pc}))) \in \mathcal{V}_{\text{tst}}^{gc}(R_R(r_1))
\]

which follows from Lemma \[47\] and \[12\] and what we have proven about the pc.

Case 5.5.

Pick \( R'_R = R_R \). For \( r \neq r_1, r_2, \) we have

\[
(n - 1, (\Phi_S(r), \Phi_T(r))) \in \mathcal{V}_{\text{untrusted}}^{gc}(R_R(r))
\]

by assumption and Lemma \[44\].

For \( r_1 \) use Lemma \[72\] and consider the following 2 cases:

- \( T_A \#[b, e] \): by assumption this entails \textit{tst} = untrusted.

By assumption we have \((n, (\Phi_S(r_2), \Phi_T(r_2))) \in \mathcal{V}_{\text{untrusted}}^{gc}(R_R(r_2))\) which gives us the following facts:

* \([\sigma_b, \sigma_e] \# (\overline{\sigma}_{\text{glob,ret}} \oplus \overline{\sigma}_{\text{glob,v}})\)
* \(\forall \sigma' \in [\sigma_b, \sigma_e], \exists r \in R_R(r_2).\text{heap}(r) = (\text{pure, } H_\sigma) \land H_\sigma \sigma' \triangleq (\mathcal{V}_{\text{untrusted}}^{gc} \circ \xi)\)

This means that for \( \sigma \) there is a region \( r \) for which \( R_R(r_2).\text{heap}(r) = (\text{pure, } H_\sigma) \) and \( H_\sigma \sigma' \triangleq (\mathcal{V}_{\text{untrusted}}^{gc} \circ \xi) \). Pick this as the region in the sealed case and \( \overline{\sigma}_{\text{ret}} = \emptyset, \overline{\sigma}_{\text{v}} = [\sigma_b, \sigma_e], \) and \( ms_{\text{code}} = [b, e] \rightarrow \emptyset \). We now need to show the following:

* \( H_\sigma \sigma W \triangleq H_\sigma^{\text{code,} \square} \overline{\sigma}_{\text{ret}} \overline{\sigma}_{\text{v}} ms_{\text{code}} gc \sigma W \)

To this end let \( \hat{W} \) be given and show

\[
H_\sigma \sigma \hat{W} \triangleq H_\sigma^{\text{code,} \square} \overline{\sigma}_{\text{ret}} \overline{\sigma}_{\text{v}} ms_{\text{code}} gc \sigma \hat{W}
\]

By transitivity of \( n \)-equality it suffices to show \( H_\sigma \sigma' \hat{W} \triangleq H_\sigma^{\text{code,} \square} \overline{\sigma}_{\text{ret}} \overline{\sigma}_{\text{v}} ms_{\text{code}} gc \sigma \hat{W} \triangleq \mathcal{V}_{\text{untrusted}}^{gc} \circ \xi(\hat{W}) \), which follows by assumption and \( H_\sigma^{\text{code,} \square} \overline{\sigma}_{\text{ret}} \overline{\sigma}_{\text{v}} ms_{\text{code}} gc \sigma \hat{W} \triangleq \mathcal{V}_{\text{untrusted}}^{gc} \circ \xi(\hat{W}) \) which follows by definition of \( H_\sigma^{\text{code,} \square} \) and the fact that \( \sigma \in \overline{\sigma}_{\text{v}} \) and \( T_A \#[b, e] \).

* \((n', (\Phi_S(r_1), \Phi_T(r_1))) \in H_\sigma \sigma^{-1}(R_R(r_1))\) for all \( n' < n \)

which corresponds to showing \((n', (\Phi_S(r_1), \Phi_T(r_1))) \in \mathcal{V}_{\text{untrusted}}^{gc}(R_R(r_1))\) which is true by assumption and Lemma \[44\].
\* (isLinear(sc_S) ⇒ ∀ W′ ⊇ R(R(r_1)), W_o, n′ < n−1, (n′, (sc'_S, sc'_T)) ∈ H_σ ξ^{-1}(W_o). (n′, sc_S, sc'_S, sc_T, sc'_T) ∈ E^{gc}_{\text{close}}(W′ ⊕ W_o))

Let W′ ⊇ R(R(r_1)), W_o, n′ < n, (n′, (sc'_S, sc'_T)) ∈ E^{gc}_{\text{close}}. Further let n'' ≤ n′ be given and assume (n'', (reg_S, reg_T)) ∈ \mathcal{R}_{\text{untrusted}}(\{\text{data}\})(W_R), ms_S, stk, ms stk, ms_T \cdot _n^{gc} W_M.

Now consider the following cases:

\* executable(sc'_S):
   In this case, pick Φ'_S = Φ'_T = failed (as xjump fails). It is trivial to show (n, Φ'_S, Φ'_T) ∈ Ω^{\text{gc}}.

\* nonExecutable(sc'_S):
   In this case consider what Φ_S(r_1) is.
   If Φ_S(r_1) = (\{perm, l\}, b, e, a) and perm ∈ \{\text{RWX, RX}\}:
   If perm = RWX, then we have a contradiction with (n, (Φ_S(r_1), Φ_T(r_1))) ∈ \mathcal{R}_{\text{untrusted}}(R(R(r_1))).
   If perm = RX, then by Lemma 77 we have a contradiction with isLinear(Φ_S(r_1)).

Otherwise (not Φ_S(r_1) = (\{perm, l\}, b, e, a) and perm ∈ \{\text{RWX, RX}\}):

In this case, pick Φ'_S = (ms_S, reg_S[pc ↦ Φ_S(r_1)][\text{data} ↦ sc'_S], stk, ms stk) and Φ'_T = (ms_T, reg_T[pc ↦ Φ_T(r_1)][\text{data} ↦ sc'_T]). In this case, the next step of execution fails which makes it trivial to show (n, Φ'_S, Φ'_T) ∈ \Omega^{\text{gc}}.

\* (nonLinear(sc_S) ⇒ ∀ W′ ⊇ purePart(R(R(r_2))), W_o, n′ < n−1, (n′, (sc'_S, sc'_T)) ∈ H_σ ξ^{-1}(W_o). (n′, sc_S, sc'_S, sc_T, sc'_T) ∈ E^{gc}_{\text{close}}(W′ ⊕ W_o))

Let W′ ⊇ purePart(R(R(r_1))), W_o, n′ < n, (n′, (sc'_S, sc'_T)) ∈ E^{gc}_{\text{close}}. Further let n'' ≤ n′ be given and assume (n'', (reg_S, reg_T)) ∈ \mathcal{R}_{\text{untrusted}}(\{\text{data}\})(W_R), ms_S, stk, ms stk, ms_T \cdot _n^{gc} W_M.

Now consider the following cases:

\* executable(sc'_S):
   In this case, pick Φ'_S = Φ'_T = failed (as xjump fails). It is trivial to show (n, Φ'_S, Φ'_T) ∈ Ω^{\text{gc}}.

\* nonExecutable(sc'_S):
   In this case consider what Φ_S(r_1) is.
   If Φ_S(r_1) = (\{perm', l\}, b', e', a') and perm' ∈ \{\text{RWX, RX}\}:
   If perm' = RWX, then we have a contradiction with (n, (Φ_S(r_1), Φ_T(r_1))) ∈ \mathcal{R}_{\text{untrusted}}(R(R(r_1))).
   If perm' = RX, then the result follows from 77

- T_A ⊆ \{b, c\}: by assumption this entails \text{tst} = \text{trusted}.

Further, we know Φ_S points to casual r_1 r_2 in T_A, so by the rationality assumption on Φ_S, we know σ ∈ \sigma_{\text{glob, close}} and one of the following holds:

* executable(Φ(r_1)) and Φ(r_1) behaves reasonably up to n−1 steps.

* nonExecutable(Φ(r_1)) and Φ(r_1) is reasonable up to n−1 steps in memory Φ.ms and free stack Φ.ms stk

By σ ∈ \sigma_{\text{glob, close}} we can conclude that (n, (Φ_S(r_2), Φ_T(r_2)) /∈ \mathcal{R}_{\text{untrusted}}(W) which means that the assumption (n, (Φ_S(r_2), Φ_T(r_2)) ∈ \mathcal{R}_{\text{trusted}}(W) gives us r ∈ dom(W.heap) such that W.heap(r) = E^{lc}_{\text{close}} ms code \cdot gc; [\sigma b, σ c] ⊆ (\sigma_{\text{ret}} \cup \sigma_{\text{close}}) and \sigma_{\text{ret}} ⊆ \sigma_{\text{glob, close}} and \sigma_{\text{close}} ⊆ \sigma_{\text{glob, close}}, dom(ms code) ⊆ T_A

Now pick r and show

* (n′, (Φ_S(r_1), Φ_T(r_1))) ∈ E^{gc}_{\text{close}}. Further let n'' ≤ n′ be given and assume (n'', (reg_S, reg_T)) ∈ \mathcal{R}_{\text{untrusted}}(\{\text{data}\})(W_R), ms_S, stk, ms stk, ms_T \cdot _n^{gc} W_M.

Now consider the following cases:

\* executable(sc'_S):
   In this case, pick Φ'_S = Φ'_T = failed (as xjump fails). It is trivial to show (n, Φ'_S, Φ'_T) ∈ Ω^{\text{gc}}.

72
**Case 5.7.** In this case, we can take

\[ \Phi_s(r_1) = (\sqcup_{r \in R} l, b, e, a) \text{ and } \text{perm} \in \{\text{RXW}, \text{RX}\}. \]

If \( \text{perm} = \text{RXW} \), then we have a contradiction with \((n, (\Phi_s(r_1), \Phi_T(r_1))) \in V_{\text{trusted}}(R_R(r_1)) \).

If \( \text{perm} = \text{RX} \), then by Lemma 74 we have a contradiction with \( \text{isLinear}(\Phi_s(r_1)) \).

Otherwise (not \( \Phi_s(r_1) = (\sqcup_{r \in R} l, b, e, a) \) and \( \text{perm} \in \{\text{RXW}, \text{RX}\} \):

In this case, pick \( \Phi'_s = (ms_s, \text{reg}_S[pc \mapsto \Phi_s(r_1)][\text{data} \mapsto sc'_S], \text{stk}, ms_s.stk) \) and \( \Phi'_T = (ms_T, \text{reg}_T[pc \mapsto \Phi_T(r_1)][\text{data} \mapsto sc'_T]) \). In this case, the next step of execution fails which makes it trivial to show \((n, \Phi'_s, \Phi'_T) \in O_{\text{trusted}}. \)

* \( \text{nonLinear}(\Phi_s(r_1)) \Rightarrow \forall W' \ni \text{purePart}(R_R(r_1)), W_o, n' < n, (n', (sc'_s, sc'_T)) \in O_{\text{trusted}}(W_o), \)

\[ (n', \Phi_s(r_1), sc'_s, \Phi_T(r_1), sc'_T) \in E_{\text{jmp}}^{\text{code}}(W' \ni W_o) \]

Let \( W' \ni \text{purePart}(R_R(r_1)), W_o, n' < n, (n', (sc'_s, sc'_T)) \in O_{\text{trusted}}(W_o) \). Further let \( n'' \leq n' \) be given and assume \((n'', (\text{reg}_S, \text{reg}_T)) \in R_{\text{untrusted}}(W_R), ms_s, stk, ms_s.stk, ms_T : \text{untrusted}\) \( W_M \).

Now consider the following cases:

- **executeable(sc'_S):**
  - In this case, pick \( \Phi'_s = \Phi'_T = \text{failed} \) (as \( \text{jump} \) fails). It is trivial to show \((n, \Phi'_s, \Phi'_T) \in O_{\text{trusted}}. \)

- **nonExecuteable(sc'_S):**
  - In this case consider what \( \Phi_s(r_1) \) is.

**Case 5.6.** From \((n, (\Phi_s(r_3), \Phi_s(r_3))) \in V_{\text{trust}}(R_R(r_3)), \) Lemma 55 gives us \( W_1, W_2 \) and \( W_3 \) such that \( R_R(r_3) = W_1 \oplus W_2 \oplus W_3 \) and \((n, (w_1, w_1)) \in V_{\text{trust}}(W_1), (n, (w_2, w_2)) \in V_{\text{trust}}(W_2) \) and \((n, (w_3, w_3)) \in V_{\text{trust}}(W_3) \).

We take \( R'_R(r_1) = W_1, R'_R(r_2) = W_2 \) and \( R'_R(r_3) = W_3 \oplus R_R(r_1) \oplus R_R(r_2) \) and \( R'_R = R_R(r) \) elsewhere.

By the above points, by assumption and using Lemma 46 we then have for all \( r \) that:

\[ (n - 1, (\Phi'_s(r), \Phi'_T(r))) \in V_{\text{trust}}(R'_R(r)) \]

**Case 5.7.** In this case, we can take \( R'_R = R_R \) and use Lemma 8 to give us that all \( \text{purePart}(R_R(r)) \) are equal. For \( r = r_1, r_2, r_3 \), we then get easily by definition that

\[ (n - 1, (\Phi'_s(r), \Phi'_T(r))) \in V_{\text{trust}}(R'_R(r)) \]

and for other registers, it follows by Lemma 44

**Case 5.8.** First we only consider registers \( r_1, r_2, r_3 \).

From \((n, (\Phi_s(r_3), \Phi_T(r_3))) \in V_{\text{trust}}(R_R(r_3)), \) Lemma 66 gives us \( W_1, W_2 \) and \( W_3 \) such that \( R_R(r_3) = W_1 \oplus W_2 \oplus W_3 \) and \((n, (w_1, w_1)) \in V_{\text{trust}}(W_1), (n, (w_2, w_2)) \in V_{\text{trust}}(W_2) \) and \((n, (w_3, w_3)) \in V_{\text{trust}}(W_3) \).

We take \( R'_R(r_1) = W_1, R'_R(r_2) = W_2 \) and \( R'_R(r_3) = W_3 \oplus R_R(r_1) \oplus R_R(r_2) \) and \( R'_R = R_R(r) \) elsewhere.
By the above points, by assumption and using Lemma \[44\] we then have for all \(r\) that:

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R'_R(r))
\]

- **Case 5.9.**

From \((n, (\Phi_S(r_2), \Phi_T(r_2))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_2))\) and \((n, (\Phi_S(r_3), \Phi_T(r_3))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_3))\), Lemma \[58\] tells us that \((n, (w_1, w'_1) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_2) \circledast R_R(r_3)))\). Since \(w_2 = w'_2 = w_3 = w'_3 = 0\), it’s clear that \((n, (w_2, w'_2) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_1)))\) and \((n, (w_3, w'_3) \in \mathcal{V}_{\text{tst}}^{\square, gc}(\text{purePart}(R_R(r_3))))\). We take \(R'_R(r_1) = R_R(r_2) \circledast R_R(r_3)\) and \(R'_R(r_3) = \text{purePart}(R_R(r_3))\) and \(R'_R = R_R(r)\) elsewhere.

By the above points, by assumption and using Lemma \[44\] we then have for all \(r\) that:

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R'_R(r))
\]

- **Case 5.10.**

We start by arguing the safety of \(r_1\):

From \((n, (\Phi_S(r_2), \Phi_T(r_2))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_2))\) and \((n, (\Phi_S(r_3), \Phi_T(r_3))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R_R(r_3))\), Lemma \[57\] gives us \(W'_1, W'_2, W'_3\) such that \(R_R(r_2) \circledast R_R(r_3) = W'_1 \circledast W'_2 \circledast W'_3\) and \((n, (w_1, w_1) \in \mathcal{V}_{\text{tst}}^{\square, gc}(W'_i))\) for \(i = 1, 2, 3\). We take \(R'_R(r_1) = W'_1 \circledast R_R(r_1)\) (which is defined because \(R_R(r_1) \circledast (R_R(r_2) \circledast R_R(r_3))\) is defined), \(R'_R(r_2) = W'_2\) and \(R'_R(r_3) = W'_3\) and \(R'_R = R_R(r)\) elsewhere.

By the above points, by assumption and using Lemma \[44\] we then have for all \(r\) that:

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R'_R(r))
\]

- **Case 5.11.**

In this case, we can take \(R'_R = R_R\) and use Lemma \[8\] to give us that all \(\text{purePart}(R_R(r))\) are equal. For \(r = r_1, r_2, r_3\), we then get easily by definition that

\[
(n - 1, (\Phi'_S(r), \Phi'_T(r))) \in \mathcal{V}_{\text{tst}}^{\square, gc}(R'_R(r))
\]

and for other registers, it follows by Lemma \[44\]

- **Case 5.12.**

In this case, we can take \(R'_R = R_R\) and for all registers, the result follows by Lemma \[44\]

By Lemma \[56\] it suffices to show

\[
(n - 1, \Phi'_S, \Phi'_T) \in \mathcal{O}_{\square, gc}
\]

which follows from the induction hypothesis. In this case, the IH is applicable because we have the following:

- One of the following sets of requirements holds:
  \[
  - \text{tst} = \text{trusted}, \Phi_S \text{ is reasonable up to } n - 1 \text{ steps and } [b, e] \subseteq \text{dom}(ms_{\text{code}}) = T_A
  - \text{tst} = \text{untrusted}, [b, e] \not\subseteq T_A \text{ and } (n - 1, [b, e]) \in \text{readCondition}_{\square, gc}(\text{normal}, W_{pc})
  \]

  We know one of the following holds:

  \[
  - \text{tst} = \text{trusted}, \Phi_S \text{ is reasonable up to } n \text{ steps and } [b, e] \subseteq \text{dom}(ms_{\text{code}}) = T_A
  - \text{tst} = \text{untrusted}, [b, e] \not\subseteq T_A \text{ and } (n, [b, e]) \in \text{readCondition}_{\square, gc}(\text{normal}, W_{pc})
  \]

  If the latter is the case, then the result follows by Lemma \[44\]

  If the former holds, then it follows by definition of execution configuration reasonability, using the fact that \(\Phi_S\) does not point to call or jmp \(r_1 r_2\) or \(r_1\) \(r_2\).

- \(\Phi'_S(pc) = \Phi'_T(pc) = (\text{RX}, \text{normal}), b, e, \_\): Follows by definition of updatePc and using the fact that \(r_i \neq pc\) for all \(i\) and the corresponding assumption of this lemma.

- \((n - 1, [b, e]) \in \text{readXCondition}_{\square, gc}(W_{pc})\): Follows by Lemma \[44\] from the corresponding assumption of this lemma.

- \((n - 1, (\Phi'_S, \Phi'_T)) \in \mathcal{R}_{\text{tst}}^{\square, gc}(W_{pc})\): See above.
• \( \Phi'_S, \text{mem}, \Phi'_S, \text{stk}, \Phi'_S, \text{ms}_{\text{stk}}, \Phi'_T, \text{mem} \): These components are all unchanged from \( \Phi_S \) and \( \Phi_T \), so the result follows by Lemma 34 from the corresponding assumption of this lemma.
• \( W_{pc} \oplus W_R \oplus W_M \) is defined: Follows from the corresponding assumption of this lemma.
• Theorem 2 holds for all \( n' < n - 1 \): follows from the corresponding assumption of this lemma.

Case 2
By Lemma 50 it suffices to show
\[
(n - 1, \Phi'_S, \Phi'_T) \in \mathcal{O}^\square, gc
\]
First, we show that for some \( W''_R \) and \( W''_M \) such that \( W_R \oplus W_M = W''_R \oplus W''_M \), we have that \( (n - 1, (\Phi'_S, \text{reg}, \Phi'_T, \text{reg})) \in \mathcal{R}^\square, gc(W''_R) \) and \( \Phi'_S, \text{mem}, \Phi'_S, \text{stk}, \Phi'_S, \text{ms}_{\text{stk}}, \Phi'_T, \text{mem} \oplus_{n - 1} W''_M \). We have that \( \Phi'_S = \text{updatePc}(\Phi_S[\text{reg}, r_1, r_2 \mapsto w_1, w_2][\text{mem}, a \mapsto w]) \) and \( \Phi'_T = \text{updatePc}(\Phi_T[\text{reg}, r_1, r_2 \mapsto w_1', w_2'][\text{mem}, a \mapsto w']) \) and we distinguish the following two cases:

1.\( \text{(store)} \): \( w_1 = w_1' = \Phi_S(r_1) = \Phi_T(r_1) = ((\text{perm}, l), b, e, a), \) and \( \text{perm} \in \text{writeAllowed}, \) and \( \text{withinBounds}(w_1) \), and \( w = \Phi_S(r_2) \), and \( w' = \Phi_T(r_2) \), and \( w_2 = \text{linearityConstraint}(\Phi_S(r_2)), w_2' = \text{linearityConstraint}(\Phi_T(r_2)) \), \( r_2 \neq pc \):
   From \( (n, (\Phi_S, \text{reg}, \Phi_T, \text{reg})) \in \mathcal{R}^\square, gc(W_R) \), we know that \( (n, (w, w')) \in \mathcal{Y}^\square, gc(W_R, 2) \) and \( (n, (\Phi_S, \text{reg}[r_2 \mapsto r_2][w_2], \Phi_T, \text{reg}[r_2 \mapsto w_2']) \in \mathcal{R}^\square, gc(W_R') \) with \( W_R = W''_R \oplus W_R' \) (using Lemma 18 and 6).
   From reasonability of \( \Phi_S \), we know that \( \Phi, \text{reg}(r_2) \) is reasonable in memory \( \Phi, \text{mem} \) up to \( n - 1 \) steps.
   Lemma 20 then tells us that \( (n, (w, w')) \in \mathcal{Y}^\square, gc(\text{untrusted}(W_R, 2)) \).
   The result then follows from Lemma 64, redistributing ownership in the obvious way.

2.\( \text{(load)} \): \( w_2 = w_2' = \Phi_T(r_2) = \Phi_S(r_2) = ((\text{perm}, l), b, e, a), \) and \( \text{perm} \in \text{readAllowed}, \) \( \text{withinBounds}((((\text{perm}, l), b, e, a)), \) and \( w_1 = \Phi_S, \text{mem}(a) \), and \( w_1' = \Phi_T, \text{mem}(a) \), and \( w = \text{linearityConstraint}(w_1) \), \( w' = \text{linearityConstraint}(w'') \), \( \text{linearityConstraintPerm}(\text{perm}, w_1) \), \( \text{linearityConstraintPerm}(\text{perm}, w_1') \) and \( r_1 \neq pc \).
   Follows from Lemma 62, redistributing ownership in the obvious way.

By the induction hypothesis, it suffices to prove that:

• One of the following sets of requirements holds:
  \( - \text{tst} = \text{trusted} \) and \( \Phi'_S \) is reasonable up to \( n \) steps and \( [b, e] \subseteq \text{dom}(\text{ms}_{\text{code}}) = T_A \)
  \( - \text{tst} = \text{untrusted} \) and \( [b, e] \neq T_A \) and \( (n, [b, e]) \in \text{readCondition}^\square, gc(\text{normal}, W_{pc}) \)

Follows by the same assumption of this lemma, in the first case using the definition of execution configuration reasonability, using the fact that \( \Phi_S \) does not point to \( \text{call}^{\text{off}-\text{off}} \ast r_1 r_2 \) or \( \text{xjmp} r_1 r_2 \).

• \( \Phi'_S, \text{pc} = (\Phi'_T, \text{pc}) = ((\text{RX}, \text{normal}), b, e, . . . ) \):
   Follows by definition of \( \text{updatePc} \) and the fact that stores from pc into memory and loads into pc are not allowed.

\( (n - 1, [b, e]) \in \text{readXCondition}^\square, gc(W_{pc}) \):
Follows by the same assumption of this lemma, using Lemma 44.

\( (n - 1, (\Phi'_S, \text{reg}, \Phi'_T, \text{reg})) \in \mathcal{R}^\square, gc(W'_R) \):
See above.

\( \Phi'_S, \text{mem}, \Phi'_S, \text{stk}, \Phi'_S, \text{ms}_{\text{stk}}, \Phi'_T, \text{mem} \oplus_{n - 1} W'_M \):
See above.

\( W_{pc} \oplus W'_R \oplus W'_M \) is defined.

• Theorem 2 holds for all \( n' < n - 1 \):
   Follows by the same assumption of this lemma.

Case 3
By Lemma 50 it suffices to show
\[
(n - 1, \Phi'_S, \Phi'_T) \in \mathcal{O}^\square, gc
\]
First, we show that for some \( W''_R \) and \( W''_M \) such that \( W_R \oplus W_M = W''_R \oplus W''_M \), we have that \( (n - 1, (\Phi'_S, \text{reg}, \Phi'_T, \text{reg})) \in \mathcal{R}^\square, gc(W''_R) \) and \( \Phi'_S, \text{mem}, \Phi'_S, \text{stk}, \Phi'_S, \text{ms}_{\text{stk}}, \Phi'_T, \text{mem} \oplus_{n - 1} W''_M \). We have that \( \Phi'_S = \text{updatePc}(\Phi_S[\text{reg}, r_1, r_2 \mapsto w_1, w_2][\text{ms}_{\text{stk}}, a \mapsto w]) \), \( \Phi'_T = \text{updatePc}(\Phi_T[\text{reg}, r_1, r_2 \mapsto w_1', w_2'][\text{ms}_{\text{stk}}, a \mapsto w']) \) and we distinguish the following two cases:
• (store) \( w_1 = \Phi_T(r_1) = ((\text{perm, linear}), b, e, a), \) \( w_1' = \Phi_S(r_1) = \text{stack-ptr}(\text{perm}, b, e, a), \) and \( \text{perm} \in \text{writeAllowed}, \) and \( \text{withinBounds}(w_1), \) and \( w = \Phi_S(r_2), \) and \( w' = \Phi_T(r_2), \) and \( w_2 = \text{linearityConstraint}(\Phi_S(r_2)), \) \( w_2' = \text{linearityConstraint}(\Phi_T(r_2)): \)

Follows from Lemma 65 redistributing ownership in the obvious way.

• (load) \( w_2' = \Phi_T(r_2) = ((\text{perm, linear}), b, e, a), \) and \( w_2 = \Phi_S(r_2) = \text{stack-ptr}(\text{perm}, b, e, a), \) and \( \text{perm} \in \text{readAllowed}, \) \( \text{withinBounds}((\text{perm, l}), b, e, a)), \) and \( a \in \text{dom}(\Phi, ms_{stk}), \) and \( a \in \text{dom}(\Phi, ms_{stk}), \) and \( w_1 = \Phi_S.ms_{stk}(a), \) and \( w_1' = \Phi_T.ms_{stk}(a), \) and \( w = \text{linearityConstraint}(w_1), \) \( w' = \text{linearityConstraint}(w_1'), \) \( \text{linearityConstraintPerm}(\text{perm}, w_1), \) and \( \text{linearityConstraintPerm}(\text{perm}, w_1'), \) and \( r_1 \neq \text{pc}: \)

Follows from Lemma 63 redistributing ownership in the obvious way.

By the induction hypothesis, it suffices to prove that:

• One of the following sets of requirements holds:

  - \( \text{tst} = \text{trusted}, \Phi'_S \) is reasonable up to \( n \) steps and \( [b, e] \subseteq \text{dom}(ms_{code}) = T_A \)
  - \( \text{tst} = \text{untrusted} \) and \( [b, e] \# T_A \) and \( (n, [b, e]) \in \text{readCondition}^{\odot}.gc \)(normal, \( W_{pc} \))

Follows by the same assumption of this lemma, in the first case using the definition of execution configuration reasonability, using the fact that \( \Phi_S \) does not point to \( \text{call}^{\odot}_{\text{pc-off}} r_1 r_2 \) or \( \text{xjmp} r_1 r_2. \)

• \( \Phi'_S(pc) = \Phi'_T(pc) = ((\text{RX, normal}), b, e, \ldots): \)

Follows by definition of \( \text{updatePc} \) and the fact that stores from \( pc \) into memory and loads into \( pc \) are not allowed.

• \( (n - 1, [b, e]) \in \text{readXCondition}^{\odot}.gc(W_{pc}) \):

Follows by the same assumption of this lemma, using Lemma 44.

• \( (n - 1, (\Phi'_S.reg, \Phi'_T.reg)) \in R^{\odot}.gc(W'_R) \):

See above.

• \( \Phi'_S.mem, \Phi'_S.stk, \Phi'_S.ms_{stk}, \Phi'_T.mem \in W_{M'}^{\odot}.gc \):

See above.

• \( W_{pc} \oplus W'_R \oplus W_{M'} \) is defined.

• Theorem 2 holds for all \( n' < n - 1: \)

Follows by the same assumption of this lemma.

Case S

We have that

• \( \Phi_S \rightarrow^{gc} \Phi'_S \)

• \( \Phi_T \rightarrow \Phi'_T \)

• \( \Phi_S \) does not point to \( \text{call}^{\odot}_{\text{pc-off}} r_1 r_2 \) or \( \text{xjmp} r_1 r_2 \)

• One of the following holds

  1. (\( \text{jmp,jnz} \)) \( \Phi'_S = \Phi_S[\text{reg,pc}, r_1 \mapsto \Phi_S(r_1), w_1] \) and \( \Phi'_T = \Phi_T[\text{reg,pc}, r_1' \mapsto \Phi_T(r_1), w_1'] \) and \( \Phi_S(r_1) = \Phi_T(r_1) = ((\text{perm}_1, l_1), b_1, c_1, a_1), \) executable(\( \Phi_S(r_1) \)), \( \text{withinBounds}(\Phi_S(r_1)), w_1 = \text{linearityConstraint}(\Phi_S(r_1)) \) and \( w_1' = \text{linearityConstraint}(\Phi_T(r_1)) \)

  2. (\( \text{xjmp} \)) \( \Phi_S(r_1) = \text{sealed}(\sigma, c_1) \) and \( \Phi_S(r_2) = \text{sealed}(\sigma, c_2) \) and \( \Phi_T(r_1) = \text{sealed}(\sigma, c_1') \) and \( \Phi_T(r_2) = \text{sealed}(\sigma, c_2') \) and \( c_1' \neq \text{ret-prt-code}(\cdot) \) and \( c_2' \neq \text{ret-prt-data}(\cdot) \) and \( \text{nonExecutable}(\Phi_S(r_2)) \) and \( \text{nonExecutable}(\Phi_T(r_2)) \) and \( \Phi'_S = \Phi_S[\text{reg,pc}, r_1 \mapsto \text{linearityConstraint}(c_1), \text{linearityConstraint}(c_2)] \) and \( \Phi'_S = \text{xjumpResult}(c_1, c_2, \Phi'_S) \) and \( \Phi'_T = \Phi_T[\text{reg,pc}, r_1 \mapsto \text{linearityConstraint}(c_1'), \text{linearityConstraint}(c_2')] \) and \( \Phi'_T = \text{xjumpResult}(c_1', c_2', \Phi'_T) \)

According to Lemma 50, it suffices to show:

\[ (n - 1, (\Phi'_S, \Phi'_T)) \in \text{O}^{\odot}.gc \]

If \( n - 1 = 0, \) then this holds vacuously (by definition of \( \text{O}^{\odot}(T_A, \text{stk-base}, \text{glock,ret,ms}_{\text{code}}) \) and \( \text{O}^{\odot}(T_A, \text{stk-base}, \text{glock,ret,ms}_{\text{code}}) \)),

so we assume that \( n - 1 > 0. \)

In the first case (\( \text{jmp,jnz} \)), the fact that \( \Phi_S \) is reasonable, with \( \Phi_S \rightarrow^{gc} \Phi'_S \) and \( \Phi_S \) does not point to \( \text{call}^{\odot}_{\text{pc-off}} r_1 r_2 \) or \( \text{xjmp} r_1 r_2, \) gives us that \( \text{perm}_1 = \text{perm}, l_1 = \text{normal}, b_1 = b, e_1 = e. \) The fact that \( l_1 = \text{normal} \) implies that \( w_1 = \Phi_S(r_1) \) and \( w_1' = \Phi_T(r_1). \)

The induction hypothesis tells us that it suffices to prove the following:

76
• One of the following sets of requirements holds:
  - $\text{tst} = \text{trusted}$, $\Phi'_S$ is reasonable up to $n - 1$ steps and $[b, e] \subseteq T_A$
  - $\text{tst} = \text{untrusted}$ and $[b, e] \not\subseteq T_A$ and $(n - 1, [b, e]) \in \text{readCondition}^{\square, gc}(\text{normal}, W_{pc})$

This follows from the corresponding assumption of this lemma, the fact that $\Phi_S \to^{gc} \Phi'_S$ and $\Phi_S$ does not point to $\text{call}^{off_{FS}}, \text{off}_{FS} r_1 r_2$ or $\text{xjmp} r_1 r_2$ and \[44\]

• $\Phi'_S(pc) = \Phi'_T(pc) = ((\text{rx}, \text{normal}), b, e, \cdot)$: See above.

• $(n - 1, [b, e]) \in \text{readXCondition}^{\square, gc}(W_{pc})$: Follows by Lemma \[44\] from the fact that $(n, [b, e]) \in \text{readXCondition}^{\square, gc}(W_{pc})$.

• $(n - 1, (\Phi'_S, \text{reg}, \Phi'_T, \text{reg})) \in R_{\text{int}, gc}^{\square}(W_R)$:

  We have that $\Phi'_S, \text{reg}(r) = \Phi_S, \text{reg}(r)$ for all $r \neq pc$, so the result follows by Lemma \[44\] from the corresponding assumption of this lemma.

• $\Phi'_S, \text{mem}, \Phi'_S, \text{stk}, \Phi'_S, \text{ms}, \text{stk}, \Phi'_T, \text{mem} \upharpoonright_{\text{n - 1}} W_M$:

  These components of $\Phi'_S$ and $\Phi'_T$ are all unmodified from $\Phi_S$ and $\Phi_T$, so the result follows by Lemma \[44\] from the corresponding assumption of this lemma.

• $W_{pc} \oplus W_R \oplus W_M$ is defined: by assumption.

Theorem \[2\] holds for all $n' < n - 1$: Follows by the same assumption of this lemma.

In the second case (\text{xjmp}), we have that $(n, (\Phi_S(r_1), \Phi_T(r_1))) \in V_{\text{tst}, gc}^{\square}(W_{R,1})$, $(n, (\Phi_S(r_2), \Phi_T(r_2))) \in V_{\text{tst}, gc}^{\square}(W_{R,2})$ and $(n, (\Phi'_S(r_1), \Phi'_T(r_1))) \in R_{\text{int}, gc}^{\square}((r_1, r_2))(W'_R)$ for some $W_{R,1}, W_{R,2}, W'_R$ with $W_R = W_{R,1} \oplus W_{R,2} \oplus W'_R$.

Using the facts that $\Phi_S(r_1) = \text{sealed}(\sigma, c_1)$ and $\Phi_S(r_2) = \text{sealed}(\sigma, c_2)$ and $\Phi_T(r_1) = \text{sealed}(\sigma, c'_1)$ and $\Phi_T(r_2) = \text{sealed}(\sigma, c'_2)$ and $\text{nonExecutable}(\Phi_S(r_1))$ and $\text{nonExecutable}(\Phi_T(r_2))$, and the above points, Lemma \[66\] tells us that $(n - 1, (c_1, c_2, c'_1, c'_2)) \in E_{\text{xjmp}}^{\square}(W_{R,1} \oplus W_{R,2})$.

By definition of $E_{\text{xjmp}}^{\square, gc}$, it suffices to prove that

$$
(n - 1, \left(\Phi_S, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c_1), \text{linearityConstraint}(c_2), \Phi_T, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c'_1), \text{linearityConstraint}(c'_2)\right)) \in R_{\text{untrusted}}^{\square, gc}(\{r_{\text{data}}\})(W'_R)
$$

It is easy to show that $(n, \text{linearityConstraint}(c_1), \text{linearityConstraint}(c'_1)) \in V_{\text{tst}, gc}^{\square}(\text{purePart}(W_{R,1}))$ and $(n, \text{linearityConstraint}(c_2), \text{linearityConstraint}(c'_2)) \in V_{\text{tst}, gc}^{\square}(\text{purePart}(W_{R,2}))$, by using Lemma \[29\] in the non-linear case and the fact that 0 is always related to itself by definition of $V_{\text{tst}, gc}^{\square}$ in the linear case. We also have that $\text{purePart}(W'_R) = \text{purePart}(W_{R,1}) = \text{purePart}(W_{R,2})$ by Lemma \[4\] and the fact that $W_R = W_{R,1} \oplus W_{R,2} \oplus W'_R$, so it follows that $(n, \text{linearityConstraint}(c_1), \text{linearityConstraint}(c'_1)) \in V_{\text{tst}, gc}^{\square}(\text{purePart}(W'_R))$ and $(n, \text{linearityConstraint}(c_2), \text{linearityConstraint}(c'_2)) \in V_{\text{tst}, gc}^{\square}(\text{purePart}(W'_R))$. From this and the fact that $(n, (\Phi'_S, \Phi'_T)) \in R_{\text{int}, gc}^{\square}((r_1, r_2))(W'_R)$, it follows easily that

$$
(n - 1, \left(\Phi_S, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c_1), \text{linearityConstraint}(c_2), \Phi_T, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c'_1), \text{linearityConstraint}(c'_2)\right)) \in R_{\text{untrusted}}^{\square, gc}(\{r_{\text{data}}\})(W'_R)
$$

From the fact that $\Phi_S$ is reasonable up to $n$ steps, tells us that $\Phi, \text{reg}(r)$ is reasonable in memory $\Phi, \text{mem}$ and free stack $\Phi, \text{ms}, \text{stk}$ up to $n - 1$ steps for all $r \neq pc$. Lemma \[20\] then tells us that

$$
(n - 1, \left(\Phi_S, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c_1), \text{linearityConstraint}(c_2), \Phi_T, \text{reg}(r_1, r_2) \mapsto \text{linearityConstraint}(c'_1), \text{linearityConstraint}(c'_2)\right)) \in R_{\text{untrusted}}^{\square, gc}(\{r_{\text{data}}\})(W'_R)
$$

using the fact that $n - 1 > 0$ (by assumption above), Theorem \[2\] holds up to $n - 1$ steps (by assumption).

• $W_M \oplus W'_R \oplus W_{R,1} \oplus W_{R,2}$ is defined.

This follows easily from the facts that $W_R = W_{R,1} \oplus W_{R,2} \oplus W'_R$ (see above) and the assumption that $W_{pc} \oplus W_R \oplus W'_M$ is defined.

\[\square\]

Proof of Theorem \[2\]: By complete induction over $n'$ Assume

\[\text{if } n = 0, \text{ then we have a contradiction with } \Phi_S \neq \Phi T, \text{ when we get to } O^{\square, gc}.\]

77
• (n, [b, e]) ∈ readXCondition\(\Box_{gc}(W)\)

and one of the following sets of requirements holds:

i)  
• [b, e] ⊆ T_A
• ([tx, normal], b, e, a) behaves reasonably up to n steps.

ii)  
• [b, e] ≠ T_A

and show

\[(n, (c, c)) \in E^{\Box_{gc}(W)}\]

For c = ([tx, normal], b, e, a).

Let \(n' \leq n\) be given and assume

1. \((n', (reg_S, reg_T)) \in R^{\Box_{untrusted}(W_R)}\)
2. \(ms_S, stk, ms_{stk}, ms_{T, ^{gc}W_M}\)
3. \(W \oplus W_R \oplus W_M\) is defined

Further let

• \(\Phi_S = (ms_S, reg_S[pc \mapsto c], stk, ms_{stk})\)
• \(\Phi_T = (ms_{T, reg_T}[pc \mapsto c])\)

and show

\[(n', (\Phi_S, \Phi_T)) \in O^{\Box_{gc}}\]

By Lemma 67, taking \(tst = trusted\) if [b, e] ⊆ T_A and \(tst = untrusted\) otherwise, it suffices to show that:

• If \(tst = trusted\) then \(\Phi_S\) is reasonable up to \(n'\) steps.

We know by assumption that \(c\) behaves reasonably up to \(n\) steps.

By definition, it suffices to show that \(reg_S(r)\) is reasonable up to \(n\) steps in memory \(ms_S\) and free stack \(ms_{stk}\) for \(r = pc\) and that \(ms_S, ms_{stk}\) and \(stk\) are all disjoint.

Take an \(r ≠ pc\) and \(gc = (T_A, stk_{base}, σ_{glob, reg}, σ_{glob, loc})\). By Lemma 18, it suffices to prove the following:

- \((n, (w, _)) \in Y^{\Box_{untrusted}(W_w)}\): follows from \((n', (reg_S, reg_T)) \in R^{\Box_{untrusted}(W_R)}\).
- \(ms_S, stk, ms_{stk, ^{gc}W_M}\): by assumption.
- \(purePart(W_w) \oplus purePart(W_M)\) is defined: By Lemma 8

• \(tst = trusted \lor (n', [b, e]) \in readCondition^{\Box_{gc}(l, W)}\): If \(tst = untrusted\) then we know that [b, e] ≠ T_A, so that \((n, [b, e]) \in readCondition^{\Box_{gc}(l, W)}\) follows by Lemma 27.

• \(\Phi_S(pc) = \Phi_T(pc) = ((*, l, b, e, a))\): We know that \(\Phi_S(pc) = \Phi_T(pc) = c = ((*, l, b, e, a))\).

• \((n', [b, e]) \in readXCondition^{\Box_{gc}(W)}\): Follows from \((n, [b, e]) \in readXCondition^{\Box_{gc}(W)}\) using Lemma 44.

• \((n', (\Phi_S, reg_S, \Phi_T, reg_T)) \in R^{\Box_{tst}(W_R)}\): Follows directly from \((n', (reg_S, reg_T)) \in R^{\Box_{untrusted}(W_R)}\) since \(R^{\Box_{untrusted}(W_R)} \subseteq R^{\Box_{tst}(W_R)}\).

• \(\Phi_S.mem, \Phi_S.ms_{stk}, \Phi_S.stk, \Phi_T.mem ^{\Box_{n'}}W_M\): By assumption.

• \(W \oplus W_R \oplus W_M\) is defined: By assumption.

Theorem 2 holds for all \(n'' < n'\): Follows from our induction hypothesis since \(n'' ≤ n\).
Proof. Follows by inspecting the definition of $\mathcal{H}$ and purePart and using the monotonicity of memory relations in the world. 

\textbf{Lemma 70.} $\iota_{\{*,\text{gc}\}}^\text{std, pure}$ is address-stratified.

Proof. Trivial.

\textbf{Lemma 71 (Unique return seals).} If 
\begin{itemize}
  \item $m_{\text{code}}([a..a + \text{call\_len} - 1]) = \text{call}^\text{off\_pc, off\_r} r_1 r_2$
  \item $m_{\text{code}}([a'..a' + \text{call\_len} - 1]) = \text{call}^\text{off\_pc, off\_r'} r'_1 r'_2$
  \item $m_{\text{code}}(a + \text{off\_pc}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b)$ and $\sigma = \sigma_b + \text{off}\_\sigma$
  \item $m_{\text{code}}(a' + \text{off}\_r') = \text{seal}(\sigma'_b, \sigma'_e, \sigma'_b)$ and $\sigma = \sigma'_b + \text{off}\_\sigma'$
  \item $\sigma_{\text{ret}}, \sigma_{\text{cl}} \vdash \text{comp\_code} \text{ms\_code}$
\end{itemize}
then
\begin{align*}
a &= a'
\end{align*}
and $\text{off}\_\text{pc} = \text{off}\_\text{r'}$ and $\text{off}\_\sigma = \text{off}\_\sigma'$ and $r_1 = r'_1$ and $r_2 = r'_2$ and $\sigma_b = \sigma'_b$ and $\sigma_e = \sigma'_e$.

Proof. From $\sigma_{\text{ret}}, \sigma_{\text{cl}} \vdash \text{comp\_code} \text{ms\_code}$, we get a $d_\sigma : \text{dom}(m_{\text{code}}) \to \mathcal{P}(\text{Seal})$ such that the following holds:
\begin{itemize}
  \item $m_{\text{code}}$ has no hidden calls
  \item $\sigma_{\text{ret}} \# \sigma_{\text{cl}}$
  \item $\sigma_{\text{ret}} = \bigcup_{a \in \text{dom}(m_{\text{code}})} d_\sigma(a)$
  \item $\forall a \in \text{dom}(m_{\text{code}}). \sigma_{\text{ret}}, d_\sigma(a), \sigma_{\text{cl}} \vdash \text{comp\_code} \text{ms\_code}, a$
  \item $\exists a. m_{\text{code}}(a) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \land [\sigma_b, \sigma_e] \neq \emptyset$
\end{itemize}
Particularly, we have $\sigma_{\text{ret}}, d_\sigma(a), \sigma_{\text{cl}} \vdash \text{comp\_code} \text{ms\_code}, a$ and $\sigma_{\text{ret}}, d_\sigma(a'), \sigma_{\text{cl}} \vdash \text{comp\_code} \text{ms\_code}, a'$. From these, and the facts that $m_{\text{code}}([a..a + \text{call\_len} - 1]) = \text{call}^\text{off\_pc, off\_r} r_1 r_2$ and $m_{\text{code}}([a'..a' + \text{call\_len} - 1]) = \text{call}^\text{off\_pc, off\_r'} r'_1 r'_2$, it follows that
\begin{itemize}
  \item $m_{\text{code}}(a + \text{off\_pc}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b)$ and $\sigma_b + \text{off}\_\sigma \in d_\sigma(a)$
  \item $m_{\text{code}}(a' + \text{off}\_r') = \text{seal}(\sigma'_b, \sigma'_e, \sigma'_b)$ and $\sigma'_b + \text{off}\_\sigma' \in d_\sigma(a')$
\end{itemize}
From the facts that $\sigma = \sigma_b + \text{off}\_\sigma = \sigma'_b + \text{off}\_\sigma'$, $\sigma_b + \text{off}\_\sigma \in d_\sigma(a)$, $\sigma'_b + \text{off}\_\sigma' \in d_\sigma(a')$ and the disjointness of the $d_\sigma(a)$, we get that $a = a'$. With this fact, the rest of the proof obligations follow directly from the other assumed equations.

\textbf{Lemma 72.} For all $W$, $n$, $w_1$, $w_2$ if 
\begin{itemize}
  \item $(n, (w_1, w_2)) \in V_{\text{std, gc}}^\text{untrusted}(W)$
\end{itemize}
then 
\begin{align*}
isLinear(w_1) \iff isLinear(w_2)
\end{align*}

Proof. (Trivial) We consider the possible cases for $w_1$ and $w_2$
\begin{itemize}
  \item Case $w_1, w_2 \in \mathbb{Z}$: By definition of isLinear.
  \item Case $w_1 = \text{seal}(\sigma, s_c)$ and $w_2 = \text{seal}(\sigma, s_c)$: From the assumption $(n, (w_1, w_2)) \in V_{\text{untrusted}}(W)$ we get the desired result.
  \item Case $w_1 = \text{seal}(n, w)$ and $w_2 = \text{seal}(n, w)$: By definition of isLinear.
  \item Case $w_1 = \text{stack-\_ptr}(n, w)$ and $w_2 = ((n, \text{linear}), n, w)$: By definition of isLinear, stack pointers are linear.
  \item Case $w_1 = ((n, l), n, w)$ and $w_2 = ((n, l), n, w)$: By definition of isLinear.
\end{itemize}
If $\text{tst} = \text{trusted}$, then there are two more cases to consider, but like the above cases, they are trivial.

\textbf{Lemma 73.} For all $W' \supseteq W$, $n$, $w_1$, $w_2$ if
• isLinear($w_1$) or isLinear($w_2$)

or

• $(n, (w_1, w_2)) \in V_{untrusted}^{\square, gc}(W)$

• nonLinear($w_1$) or nonLinear($w_2$)

then

$$(n, (linearityConstraint(w_1), linearityConstraint(w_2))) \in V_{untrusted}^{\square, gc}(purePart(W'))$$

Proof. For the first set of assumptions, we can conclude isLinear($w_1$) and isLinear($w_2$) by Lemma 72 which means that we need to argue

$$(n, (0, 0)) \in V_{untrusted}^{\square, gc}(purePart(W'))$$

which is trivially true.

For the second set of assumptions, we can conclude nonLinear($w_1$) and nonLinear($w_2$) by Lemma 72 which means that we need to show $(n, (w_1, w_2)) \in V_{untrusted}^{\square, gc}(purePart(W'))$ which is true by assumption and Lemma 17

Lemma 74. If

• $c = ((perm, l), b, e, a)$

• $perm = \{RX, RWX\}$

• $(n, (c, \_)) \in \gamma^{\square, gc}_{untrusted}^{\text{tst}}(W)$

then

$$l = \text{normal}$$

Proof. Follows from the definition.

Lemma 75. If

• $c_c = ((RX, \text{normal}), b, e, a)$

• $a \in \{b, e\}$

• $(n, [b, e]) \in \text{executeCondition}^{\square, gc}(W)$

• nonExecutable($c_d$)

• $(n, (c_d, c'_d)) \in V_{untrusted}^{\square, gc}(W_o)$

• $(n, (\text{reg}_S, \text{reg}_T)) \in R_{untrusted}^{\square, gc}([\{r_{\text{data}}\}](W_R))$

• $ms_S, stk, ms_{stk}, ms_T : gc W_M$

• $\Phi_S = (ms_S, \text{reg}_S[p_{\text{pc}} \mapsto c_c][r_{\text{data}} \mapsto c_d], stk, ms_{stk})$

• $\Phi_T = (ms_T, \text{reg}_T[p_{\text{pc}} \mapsto c'_c][r_{\text{data}} \mapsto c'_d])$

• $W' \nsubseteq purePart(W)$

• $W' \oplus W_o \oplus W_R \oplus W_M$ is defined

then

$$(n - 1, (\Phi_S, \Phi_T)) \in O^{\square, gc}$$

Proof. From Lemma 17 and Lemma 29 we get

$$(n, [b, e]) \in \text{executeCondition}^{\square, gc}(W)$$

from which we get

$$(n - 1, (c_c, c'_c)) \in E^{\square, gc}(W')$$

From $(n, (\text{reg}_S, \text{reg}_T)) \in R_{untrusted}^{\square, gc}([\{r_{\text{data}}\}](W_R))$, $(n, (c_d, c'_d)) \in V_{untrusted}^{\square, gc}(W_o)$, and $W' \oplus W_o \oplus W_R \oplus W_M$ is defined, we conclude

$$(n, (\text{reg}_S[r_{\text{data}} \mapsto c_d], \text{reg}_T[r_{\text{data}} \mapsto c_d])) \in V_{untrusted}^{\square, gc}(W_R \oplus W_o)$$

along with $ms_S, stk, ms_{stk}, ms_T : gc W_M$ and Lemma 16 we can now conclude

$$(n - 1, (\Phi_S, \Phi_T)) \in O^{\square, gc}$$
7 Notes

7.1 Notes on linear capabilities

It seems reasonable to have enough instructions to let any program be able to make sufficient checks that it can verify that its execution won’t fail. With our current instruction set, a load may fail if a linear capability happens to be located at a memory address that on attempts to load from with a capability without write permission. To make up for this, one could make an instruction that checks the linearity of a capability in memory without loading it. It may not be practical to make such an instruction if linearity is kept track as a field on each capability, but it may be tractable if linearity tags are kept track of in a table.

7.2 Calling convention design decisions

7.2.1 Returning the full stack

When a callee return from a call, they must return all of the stack they were passed. If we omit this requirement, then we cannot guarantee well-bracketedness. The following is an example of circumventing well-bracketedness by keeping part of the stack:

- An adversary calls our trusted code with a call-back.
- Our code uses part of the stack and calls the callback with the rest of the stack.
- The adversary splits that stack in two and saves the part of the stack adjacent to our stack in some persistent memory. The adversary calls us anew with a callback and the part of the stack they did not save.
- We use part of the stack we receive and call the callback with the rest of the stack. The adversary can now use the saved stack to return from the first call to the callback breaking well-bracketedness. The reason this is possible is because we do not check the size of the stack.

Figure 2 illustrates the above example. In the figure, \( f \) is the function to “is” and \( g \) is the callback passed to “us”. In the end, the part of the stack marked kept by adv can be used to return from the first call-back (the call with the *). The stack that is used to return lines up with Us\textsubscript{priv}, so the return is successful.

7.2.2 Restriction on stack allocation

We need to somehow make sure that it is always the same stack that is used. If we don’t, then an adversary can simply split the stack in two, use one part for one call and the other for another call. At this point, they can return to either of the two calls - in other words, well-bracketedness is not enforced.

- An adversary starts the execution. They split the stack in two and call us with one part (say the top part) along with a callback.
- We use part of the stack and call the adversary with the rest of the stack.
- The adversary calls us again this time using the other part of the stack (here the bottom part of the stack).
- Again, we use part of the stack and call the adversary with the rest of this part of the stack.

At this point, the adversary can return from either of the two calls. Swapping around the order in which the adversary uses the two parts of the stack changes nothing.

The example is illustrated in Figure 3.

One way to solve this problem is to make the ”top address” of the stack known. There are many ways to do this, but we have chosen to do the following:

- The stack grows downwards, so the “last address” of the stack is the base address of the initial stack capability.
- The base address of the stack capability is a fixed address, so in the semantics, it will be expressed as a constant that is publicly known.
- At some point before a call, it must be checked whether the stack we are using actually has the globally known base address (if not we must fail because we cannot trust this stack).
- To ensure that the check is made, we require it in the semantic condition (at this moment of time not defined, so we are yet to see what it looks like).
Figure 2: Illustration of the stack in the example that illustrates why the entire stack must be returned.

Figure 3: Illustration of the potential stack allocation issue.
8 Related Work

8.1 Conditional Full-Abstraction

The idea of conditional full-abstraction was used by Juglaret et al. [2016] to define full abstraction for unsafe languages. Their definition requires both the programs and the context to be fully defined (i.e. not cause undefined behavior). If the programs are not required to be fully-defined, then anything can happen which makes it impossible to reason about. In our work, undefined behavior marks cases that we do not want to consider because they should be excluded further up in the compilation chain. Further, if we have to take these cases into account, then we need to add checks which protects the trusted code against itself, but properly compiled code should not have to protect itself against itself.


References