A formalized programming language with speculative execution

Jamie Wright

August 25, 2024

Abstract

We present the formalization of a programming language whose operational semantics allows for the speculative execution of its statements. This type of semantics is relevant for discussing transient execution security vulnerabilities such as Spectre and Meltdown. An instantiation of Relative Security to this language is provided along with proofs of security and insecurity of selected programs from the Spectre benchmark.

Contents

1 A Simple Imperative Language

theory *Language-Syntax* **imports** *Language-Prelims Relative-Security.Trivia* **begin**

A Simple Imperative Language with arrays, inputs and outputs, and speculation fences, based off the syntax for IMP in Concrete Semantics [3]

Scalar variables are defined as strings, and so are the array variables

type-synonym *vname* = *string*
type-synonym *avname* = *string*

Since the Spectre benchmark examples reason about integer variables, we define our set of values to be integers

type-synonym *val* = *int*

We define our set of locations to be integers

type-synonym *loc* = *nat*

1.1 Arithmetic and Boolean Expressions

Arithmetic expressions can either be literals, variables or array variables (array variable name, index), or some operation on these. The arithmetic operators we capture in an expression are addition and multiplication. For boolean expressions we capture negation and conjunction, and the arithmetic comparison operator "less than" where equality of two arithmetic terms is later defined in terms of these constructors

datatype *aexp* = *N int* | *V vname* | *VA avname aexp* | *Plus aexp aexp* | *Times aexp aexp* |
Ite bexp aexp aexp | *Fun aexp aexp*
and *bexp* = *Bc bool* | *Not bexp* | *And bexp bexp* | *Less aexp aexp*

To enable reasoning about more subtle Spectre-like examples require the existence of trusted and untrusted I/O channels

datatype *trustStat* = *Trusted* (*T*)| *Untrusted* (*U*)

consts *func* :: *aexp* × *aexp* ⇒ *val*

A little syntax magic to write larger states compactly:

definition *null-state* (<>) **where**
null-state ≡ λ*x*. *0*
syntax
-State :: *updbinds* => *'a* (<->)
translations
-State ms == *-Update* <> *ms*
-State (*-updbinds b bs*) <= *-Update* (*-State b*) *bs*

1.2 Commmands

The language defined by this grammar capture standard basic mechanisms for manipulating scalar and array variables, and (un)conditional jumps, using Jump and IfJump, as control structures. It is also an I/O interactive language, accepting inputs on various input channels and producing outputs on various output channels. Most of the commands are standard, however there

is an inclusion of Fences and Masking commands which are non-standard. The "Fence" command models the lfence instruction which prevents further speculative execution and is crucial in capturing key Spectre benchmark examples. The Mask command models Speculative Load Hardening (SLH), which masks variable values with respect to a given condition, contextually it can protect against leaks by masking values during misspeculation. It can be read as "M var I b T exp1 E exp2 == IF b THEN var = exp1 ELSE var = exp2"

datatype (*discs-sels*) *com* =
Start
| *Skip*

| *getInput trustStat vname* ((*Input* -/ -) [*0*, *61*] *61*)
| *Output trustStat aexp* ((*Output* -/ -) [*0*, *61*] *61*)
| *Fence*
| *Mask vname bexp aexp aexp* (*M* -/ *I* -/ *T* -/ *E* - [*1000*, *61*, *61*, *61*] *61*)
| *Jump nat*
| *Assign vname aexp* (- ::= - [*1000*, *61*] *61*)
| *ArrAssign avname aexp aexp* (- [-] ::= - [*1000*, *61*] *61*)
| *IfJump bexp nat nat* ((*IfJump* -/ -/ -) [*0*, *0*, *61*] *61*)

A predicate which determines whether or not a memory read occurs in an arithmetic expression

fun *isReadMemory* ::*aexp* ⇒ *bool* **where**
isReadMemory (*N n*) = *False* |
isReadMemory (*V x*) = *False* |
isReadMemory (*VA a i*) = *True* |
isReadMemory (*Plus a1 a2*) = (*isReadMemory a1* ∨ *isReadMemory a2*)|
isReadMemory (*Times a1 a2*) = (*isReadMemory a1* ∨ *isReadMemory a2*)

1.3 Stores, States and Configurations

Defining a variable store, array variable store and a heap. The variable store is as standard, mapping variable names to values. The array variable store maps array name, to a base address in the and the size of the array. The heap maps memory locations to values

datatype *vstore* = *Vstore* (*vstore*:*vname* ⇒ *val*)
datatype *avstore* = *Avstore* (*avstore*:*avname* ⇒ *loc* ∗ *nat*)
datatype *heap* = *Heap* (*hheap*:*loc* ⇒ *val*)

A given value of an element in an array is assigned in the heap at location "array base+index". For example if the array "a1" has array base = 0, then the value a1[3] can be found at memory location 3 in the heap

definition *array-base* :: *avname* ⇒ *avstore* ⇒ *loc* **where**
array-base arr avst ≡ *case avst of* (*Avstore as*) ⇒ *fst* (*as arr*)

definition *array-bound* :: *avname* ⇒ *avstore* ⇒ *nat* **where**
array-bound arr avst ≡ *case avst of* (*Avstore as*) ⇒ *snd* (*as arr*)

definition *array-loc* :: *avname* ⇒ *nat* ⇒ *avstore* ⇒ *loc* **where**
array-loc arr i avst ≡ *array-base arr avst* + *i*

lemma *array-locBase*: *array-base arr avst* = *array-loc arr 0 avst*
⟨*proof*⟩

A state consists of: (command, variable store, heap, next free location in the heap).

datatype *state* = *State* (*getVstore*: *vstore*) (*getAvstore*: *avstore*) (*getHeap*: *heap*) (*getFree*: *nat*)

fun *getHheap* **where** *getHheap* (*State vst avst h p*) = *hheap h*

A configuration for the normal semantics consists of: (command,state,the set of read memory locations so far).

type-synonym *pcounter* = *nat*

datatype *config* = *Config* (*pcOf*: *pcounter*) (*stateOf*: *state*)

fun *vstoreOf* **where** *vstoreOf* (*Config pc s*) = *vstore* (*getVstore s*)
fun *avstoreOf* **where** *avstoreOf* (*Config pc s*) = *avstore* (*getAvstore s*)
fun *heapOf* **where** *heapOf* (*Config pc s*) = *getHeap s*
fun *freeOf* **where** *freeOf* (*Config pc s*) = *getFree s*
fun *hheapOf* **where** *hheapOf* (*Config pc s*) = *getHheap s*

1.4 Evaluation of arithmetic and boolean expressions

A standard recursive function which evaluates a given expression

fun *aval* :: *aexp* ⇒ *state* ⇒ *val*
and *bval* :: *bexp* ⇒ *state* ⇒ *bool* **where**
aval (*N n*) *s* = *n*
|
aval (*V x*) *s* = *vstore* (*getVstore s*) *x*
|
aval (*VA a i*) *s* = *getHheap s* (*array-loc a* (*nat*(*aval i s*)) (*getAvstore s*))
|
aval (*Plus a1 a2*) *s* = *aval a1 s* + *aval a2 s*
|
aval (*Times a1 a2*) *s* = *aval a1 s* ∗ *aval a2 s*
|
aval (*Ite b a1 a2*) *s* = (*if bval b s then aval a1 s else aval a2 s*)
|
aval (*Fun x y*) *s* = *func* (*x*, *y*)

|
bval (*Bc v*) *s* = *v*
|
bval (*Not b*) *s* = (¬ *bval b s*)
|
bval (*And b1 b2*) *s* = (*bval b1 s* ∧ *bval b2 s*)
|
bval (*Less a1 a2*) *s* = (*aval a1 s* < *aval a2 s*)

An arithmetic equivalence of two terms as a boolean expression

definition *Eq* :: *aexp* ⇒ *aexp* ⇒ *bexp* **where**
Eq a1 a2 ≡ *And* (*Not* (*Less a1 a2*)) (*Not* (*Less a2 a1*))

lemma *Eq-verif*:*bval* (*Eq a1 a2*) *s* ⟷ *aval a1 s* = *aval a2 s*
⟨*proof*⟩

fun *outOf* :: *com* ⇒ *state* ⇒ *val* **where**
outOf c s = (*case c of Output T aexp* ⇒ *aval aexp s* |- ⇒ *undefined*)

end

2 Basic Semantics

theory *Step-Basic*
imports *Language-Syntax*
begin

This theory introduces a standard semantics for the commands defined

2.1 Well-formed programs

A well-formed program is a nonempty list of commands where the head of the list is the "Start" command

type-synonym *prog* = *com list*

locale *Prog* =
fixes *prog* :: *prog*
assumes
wf-prog: *prog* ≠ [] ∧ *hd prog* = *Start*
begin

This is the program counter signifying the end of the program:

definition *endPC* ≡ *length prog*

And some sanity checks for a well formed program...

lemma *lenth-prog-gt-0*: *length prog* > *0*
⟨*proof*⟩

lemma *lenth-prog-not-0*: *length prog* ≠ *0*
⟨*proof*⟩

lemma *endPC-gt-0*: *endPC* > *0*
⟨*proof*⟩

lemma *endPC-not-0*: *endPC* ≠ *0*
⟨*proof*⟩

lemma *hd-prog-Start*: *hd prog* = *Start*
⟨*proof*⟩

lemma *prog-0*: *prog* ! *0* = *Start*
⟨*proof*⟩

2.2 Basic Semantics of Commands

The basic small step semantics of the language, parameterised by a fixed program. The semantics operate on input streams and memories which are consumed and updated while the program counter moves through the list of commands. This emulates standard (and expected) execution of the commands defined. Since no speculation is captured in this basic semantics, the Fence command the same as SKIP

inductive
stepB :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* *55*)
where
Seq-Start-Skip-Fence:
pc < *endPC* ⟹ *prog*!*pc* ∈ {*Start*, *Skip*, *Fence*} ⟹
(*Config pc s*, *ibT*, *ibUT*) →*B* (*Config* (*Suc pc*) *s*, *ibT*, *ibUT*)
|
Assign:
pc < *endPC* ⟹ *prog*!*pc* = (*x* ::= *a*) ⟹
s = *State* (*Vstore vs*) *avst h p* ⟹
(*Config pc s*, *ibT*, *ibUT*)
→*B*
(*Config* (*Suc pc*) (*State* (*Vstore* (*vs*(*x* := *aval a s*))) *avst h p*), *ibT*, *ibUT*)
|
ArrAssign:
pc < *endPC* ⟹ *prog*!*pc* = (*arr*[*index*] ::= *a*) ⟹
v = *aval index s* ⟹ *w* = *aval a s* ⟹
0 ≤ *v* ⟹ *v* < *int* (*array-bound arr avst*) ⟹
l = *array-loc arr* (*nat v*) *avst* ⟹
s = *State vst avst* (*Heap h*) *p*

$\Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ (State\ vst\ avst\ (Heap\ (h(l := w)))\ p),\ ibT,\ ibUT)$
|
getTrustedInput:
$pc < endPC \Longrightarrow prog!pc = Input\ T\ x \Longrightarrow$
$(Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ LCons\ i\ ibT,\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := i)))\ avst\ h\ p),\ ibT,\ ibUT)$
|
getUntrustedInput:
$pc < endPC \Longrightarrow prog!pc = Input\ U\ x \Longrightarrow$
$(Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ LCons\ i\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := i)))\ avst\ h\ p),\ ibT,\ ibUT)$
|
Output:
$pc < endPC \Longrightarrow prog!pc = Output\ t\ aexp \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ s,\ ibT,\ ibUT)$
|
Jump:
$pc < endPC \Longrightarrow prog!pc = Jump\ pc1 \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT) \rightarrow B\ (Config\ pc1\ s,\ ibT,\ ibUT)$
|
IfTrue:
$pc < endPC \Longrightarrow prog!pc = IfJump\ b\ pc1\ pc2 \Longrightarrow$
$bval\ b\ s \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT) \rightarrow B\ (Config\ pc1\ s,\ ibT,\ ibUT)$
|
IfFalse:
$pc < endPC \Longrightarrow prog!pc = IfJump\ b\ pc1\ pc2 \Longrightarrow$
$\neg\ bval\ b\ s \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT) \rightarrow B\ (Config\ pc2\ s,\ ibT,\ ibUT)$
|
MaskTrue:
$pc < endPC \Longrightarrow prog!pc = (M\ x\ I\ b\ T\ a1\ E\ a2\ ) \Longrightarrow$
$bval\ b\ s \Longrightarrow$
$s = State\ (Vstore\ vs)\ avst\ h\ p \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := aval\ a1\ s)))\ avst\ h\ p),\ ibT,\ ibUT)$
|
MaskFalse:
$pc < endPC \Longrightarrow prog!pc = (M\ x\ I\ b\ T\ a1\ E\ a2\ ) \Longrightarrow$
$\neg bval\ b\ s \Longrightarrow$
$s = State\ (Vstore\ vs)\ avst\ h\ p \Longrightarrow$

$(Config\ pc\ s,\ ibT,\ ibUT)$
$\rightarrow B$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := aval\ a2\ s)))\ avst\ h\ p),\ ibT,\ ibUT)$

lemmas *stepB-induct* = *stepB.induct*[*split-format*(*complete*)]

abbreviation
stepsB :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** $\rightarrow B*$ 55)
where $x \rightarrow B*\ y == star\ stepB\ x\ y$

declare *stepB.intros*[*simp*,*intro*]

2.3 State Transitions

Useful lemmas regarding valid transitions of the semantics along with conditions for termination (finalB)

definition *finalB* = *final stepB*
lemmas *finalB-defs* = *final-def finalB-def*

lemma *finalB-iff-aux*:
$pc < endPC \wedge$
$(\forall\, x\ i\ a.\ prog!pc = (x[i] ::= a) \longrightarrow aval\ i\ s \geq 0 \wedge$
$\quad aval\ i\ s < int\ (array\text{-}bound\ x\ (getAvstore\ s))) \wedge$
$(\forall\, y.\ prog!pc = Input\ T\ y \longrightarrow ibT \neq LNil) \wedge$
$(\forall\, y.\ prog!pc = Input\ U\ y \longrightarrow ibUT \neq LNil)$
$\longleftrightarrow$
$(\exists\, cfg'.\ (Config\ pc\ s,\ ibT,\ ibUT) \rightarrow B\ cfg')$
⟨*proof*⟩

lemma *finalB-iff*:
$finalB\ (Config\ pc\ s,\ ibT,\ ibUT)$
$\longleftrightarrow$
$(pc \geq endPC \vee$
$(\exists\, x\ i\ a.\ prog!pc = (x[i] ::= a) \wedge$
$\quad (\neg\ aval\ i\ s \geq 0 \vee \neg\ aval\ i\ s < int\ (array\text{-}bound\ x\ (getAvstore\ s))))) \vee$
$(\exists\, y.\ prog!pc = Input\ T\ y \wedge ibT = LNil) \vee$
$(\exists\, y.\ prog!pc = Input\ U\ y \wedge ibUT = LNil)$
⟨*proof*⟩

lemma *stepB-determ*:
$cfg\text{-}ib \rightarrow B\ cfg\text{-}ib' \Longrightarrow cfg\text{-}ib \rightarrow B\ cfg\text{-}ib'' \Longrightarrow cfg\text{-}ib'' = cfg\text{-}ib'$
⟨*proof*⟩

definition *nextB* :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* **where**
nextB cfg-ib ≡ *SOME cfg′-ib′*. *cfg-ib* →*B cfg′-ib′*

lemma *nextB-stepB*: ¬ *finalB cfg-ib* ⟹ *cfg-ib* →*B* (*nextB cfg-ib*)
⟨*proof*⟩

lemma *stepB-nextB*: *cfg-ib* →*B cfg′-ib′* ⟹ *cfg′-ib′* = *nextB cfg-ib*
⟨*proof*⟩

lemma *nextB-iff-stepB*: ¬ *finalB cfg-ib* ⟹ *nextB cfg-ib* = *cfg′-ib′* ⟷ *cfg-ib* →*B cfg′-ib′*
⟨*proof*⟩

lemma *stepB-iff-nextB*: *cfg-ib* →*B cfg′-ib′* ⟷ ¬ *finalB cfg-ib* ∧ *nextB cfg-ib* = *cfg′-ib′*
⟨*proof*⟩

2.3.1 Simplification Rules

Sufficient conditions for a given command to "execute" transit to the next state

lemma *nextB-Start-Skip-Fence*[*simp*]:
pc < *endPC* ⟹ *prog*!*pc* ∈ {*Start*, *Skip*, *Fence*} ⟹
nextB (*Config pc s*, *ibT*, *ibUT*) = (*Config* (*Suc pc*) *s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-Assign*[*simp*]:
pc < *endPC* ⟹ *prog*!*pc* = (*x* ::= *a*) ⟹
s = *State* (*Vstore vs*) *avst h p* ⟹
nextB (*Config pc s*, *ibT*, *ibUT*)
=
(*Config* (*Suc pc*) (*State* (*Vstore* (*vs*(*x* := *aval a s*))) *avst h p*),
ibT, *ibUT*)
⟨*proof*⟩

lemma *nextB-ArrAssign*[*simp*]:
pc < *endPC* ⟹ *prog*!*pc* = (*arr*[*index*] ::= *a*) ⟹
ls′ = *readLocs a vst avst* (*Heap h*) ⟹
v = *aval index s* ⟹ *w* = *aval a s* ⟹
0 ≤ *v* ⟹ *v* < *int* (*array-bound arr avst*) ⟹
l = *array-loc arr* (*nat v*) *avst* ⟹
s = *State vst avst* (*Heap h*) *p*
⟹
nextB (*Config pc s*, *ibT*, *ibUT*)
=
(*Config* (*Suc pc*) (*State vst avst* (*Heap* (*h*(*l* := *w*))) *p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-getTrustedInput*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = (Input\ T\ x) \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ LCons\ i\ ibT,\ ibUT)$
$=$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := i)))\ avst\ h\ p),\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-getUntrustedInput*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = (Input\ U\ x) \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ LCons\ i\ ibUT)$
$=$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := i)))\ avst\ h\ p),\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-getTrustedInput'*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = Input\ T\ x \Longrightarrow$
$ibT \neq LNil \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ ibUT)$
$=$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := lhd\ ibT)))\ avst\ h\ p),\ ltl\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-getUntrustedInput'*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = Input\ U\ x \Longrightarrow$
$ibUT \neq LNil \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ ibUT)$
$=$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := lhd\ ibUT)))\ avst\ h\ p),\ ibT,\ ltl\ ibUT)$
$\langle proof \rangle$

lemma *nextB-Output*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = Output\ t\ aexp \Longrightarrow$
$nextB\ (Config\ pc\ s,\ ibT,\ ibUT)$
$=$
$(Config\ (Suc\ pc)\ s,\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-Jump*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = Jump\ pc1 \Longrightarrow$
$nextB\ (Config\ pc\ s,\ ibT,\ ibUT) = (Config\ pc1\ s,\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-IfTrue*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = IfJump\ b\ pc1\ pc2 \Longrightarrow$
$bval\ b\ s \Longrightarrow$
$nextB\ (Config\ pc\ s,\ ibT,\ ibUT) = (Config\ pc1\ s,\ ibT,\ ibUT)$
$\langle proof \rangle$

lemma *nextB-IfFalse*[*simp*]:

$pc < endPC \Longrightarrow prog!pc = IfJump\ b\ pc1\ pc2 \Longrightarrow$
$\neg\ bval\ b\ s \Longrightarrow$
$nextB\ (Config\ pc\ s,\ ibT,\ ibUT) = (Config\ pc2\ s,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *nextB-MaskTrue*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = (M\ x\ I\ b\ T\ a1\ E\ a2) \Longrightarrow$
$bval\ b\ (State\ (Vstore\ vs)\ avst\ h\ p) \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ ibUT) =$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := aval\ a1\ (State\ (Vstore\ vs)\ avst\ h\ p))))\ avst\ h\ p),\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *nextB-MaskFalse*[*simp*]:
$pc < endPC \Longrightarrow prog!pc = (M\ x\ I\ b\ T\ a1\ E\ a2) \Longrightarrow$
$\neg bval\ b\ (State\ (Vstore\ vs)\ avst\ h\ p) \Longrightarrow$
$nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ ibUT) =$
$(Config\ (Suc\ pc)\ (State\ (Vstore\ (vs(x := aval\ a2\ (State\ (Vstore\ vs)\ avst\ h\ p))))\ avst\ h\ p),\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *finalB-endPC*: $pcOf\ cfg = endPC \Longrightarrow finalB\ (cfg,ibT,\ ibUT)$
⟨*proof*⟩

lemma *stepB-endPC*: $pcOf\ cfg = endPC \Longrightarrow \neg\ (cfg,\ ibT,\ ibUT) \rightarrow B\ (cfg',\ ibT',ibUT')$
⟨*proof*⟩

lemma *stepB-imp-le-endPC*: **assumes** $(cfg,\ ibT,\ ibUT) \rightarrow B\ (cfg',\ ibT',ibUT')$
shows $pcOf\ cfg < endPC$
⟨*proof*⟩

lemma *stebB-0*: $(Config\ 0\ s,\ ibT,\ ibUT) \rightarrow B\ (Config\ 1\ s,\ ibT,\ ibUT)$
⟨*proof*⟩

2.3.2 Elimination Rules

In the unwinding proofs of relative security it is often the case that two traces will progress in lockstep, when doing so we wish to preserve/update invariants of the current state. T he following are some useful elimination rules to help simplify reasoning

lemma *stepB-Seq-Start-Skip-FenceE*:
assumes ‹$(cfg,\ ibT,\ ibUT) \rightarrow B\ (cfg',\ ibT',ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›

and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p')$›
and ‹$prog!pc \in \{Start, Skip, Fence\}$›
shows ‹$vs' = vs \wedge ibT = ibT' \wedge ibUT = ibUT' \wedge$
$pc' = Suc\ pc \wedge avst' = avst \wedge h' = h \wedge$
$p' = p$›
⟨*proof*⟩

lemma *stepB-AssignE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = (x ::= a)$›
shows ‹$vs' = (vs(x := aval\ a\ (stateOf\ cfg))) \wedge$
$ibT = ibT' \wedge ibUT = ibUT' \wedge pc' = Suc\ pc \wedge$
$avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-getTrustedInputE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = Input\ T\ x$›
shows ‹$vs' = (vs(x := lhd\ ibT)) \wedge$
$ibT' = ltl\ ibT \wedge ibUT = ibUT' \wedge pc' = Suc\ pc \wedge$
$avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-getUntrustedInputE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = Input\ U\ x$›
shows ‹$vs' = (vs(x := lhd\ ibUT)) \wedge$
$ibT' = ibT \wedge ibUT' = ltl\ ibUT \wedge pc' = Suc\ pc \wedge$
$avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-OutputE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = Output\ t\ aexp$›
shows ‹$vs' = vs \wedge ibT' = ibT \wedge ibUT' = ibUT \wedge$
$pc' = Suc\ pc \wedge avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-JumpE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›

and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p')$›
and ‹$prog!pc = Jump\ pc1$›
shows ‹$vs' = vs \wedge ibT' = ibT \wedge ibUT' = ibUT \wedge$
$pc' = pc1 \wedge avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-IfTrueE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = IfJump\ b\ pc1\ pc2$› **and** ‹$bval\ b\ (stateOf\ cfg)$›
shows ‹$vs' = vs \wedge ibT' = ibT \wedge ibUT' = ibUT \wedge$
$pc' = pc1 \wedge avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

lemma *stepB-IfFalseE*:
assumes ‹$(cfg, ibT, ibUT) \rightarrow B\ (cfg', ibT', ibUT')$›
and ‹$cfg = (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p))$›
and ‹$cfg' = (Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p'))$›
and ‹$prog!pc = IfJump\ b\ pc1\ pc2$› **and** ‹$\neg bval\ b\ (stateOf\ cfg)$›
shows ‹$vs' = vs \wedge ibT' = ibT \wedge ibUT' = ibUT \wedge$
$pc' = pc2 \wedge avst' = avst \wedge h' = h \wedge p' = p$›
⟨*proof*⟩

end

2.4 Read locations

For modeling Spectre-like vulnerabilities, we record memory reads (as in [1]), i.e., accessed for reading during the execution. We let readLocs(pc,u) be the (possibly empty) set of locations that are read when executing the current command c - computed from all sub-expressions of the form a[e]. i.e. array reads. For example, if c is the assignment "x = a [b[3]]", then readLocs returns two locations: counting from 0, the 3rd location of b and the b[3]'th location of a.

fun *readLocsA* :: *aexp* ⇒ *state* ⇒ *loc set* **and**
readLocsB :: *bexp* ⇒ *state* ⇒ *loc set* **where**
readLocsA $(N\ n)\ s = \{\}$
|
readLocsA $(V\ x)\ s = \{\}$
|
readLocsA $(VA\ arr\ index)\ s =$
$insert\ (array\text{-}loc\ arr\ (nat\ (aval\ index\ s)))\ (getAvstore\ s))$
$(readLocsA\ index\ s)$
|
readLocsA $(Plus\ a1\ a2)\ s = readLocsA\ a1\ s \cup readLocsA\ a2\ s$
|
readLocsA $(Times\ a1\ a2)\ s = readLocsA\ a1\ s \cup readLocsA\ a2\ s$

```
|
readLocsA (Ite b a1 a2) s = readLocsB b s ∪ readLocsA a1 s ∪ readLocsA a2 s
|
readLocsA (Fun a b) s = {}
|
readLocsB (Bc c) s= {}
|
readLocsB (Not b) s = readLocsB b s
|
readLocsB (And b1 b2) s = readLocsB b1 s ∪ readLocsB b2 s
|
readLocsB (Less a1 a2) s = readLocsA a1 s ∪ readLocsA a2 s


fun readLocsC :: com ⇒ state ⇒ loc set where
readLocsC (x ::= a) s = readLocsA a s
|
readLocsC (arr[index] ::= a) s = readLocsA a s
|
readLocsC (Output t a) s = readLocsA a s
|
readLocsC (IfJump b n1 n2) s = readLocsB b s
|
readLocsC (M x I b T a1 E a2) s = readLocsB b s ∪ (if (bval b s) then readLocsA
a1 s
                                  else readLocsA a2 s)
|
readLocsC - - = {}

context Prog
begin


definition readLocs cfg ≡ readLocsC (prog!(pcOf cfg)) (stateOf cfg)

end


end
```

3 Normal Semantics

This theory augments the basic semantics to include a set of read locations which is a simple representation of a cache

The normal semantics is defined by a single rule which involves the basic semantics, extended to accumulate the read locations, which accounts for cache side-channels

theory *Step-Normal*
imports *Step-Basic*
begin

context *Prog*
begin

fun *stepN* :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* 55)
where
(*cfg*,*ibT*,*ibUT*,*ls*) →*N* (*cfg'*,*ibT'*,*ibUT'*,*ls'*) =
((*cfg*,*ibT*,*ibUT*) →*B* (*cfg'*,*ibT'*,*ibUT'*) ∧ *ls'* = *ls* ∪ *readLocs cfg*)

abbreviation
stepsN :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ 55)
where *x* →*N*∗ *y* == *star stepN x y*

definition *finalN* = *final stepN*
lemmas *finalN-defs* = *final-def finalN-def*

lemma *finalN-iff-finalB*[*simp*]:
finalN (*cfg*, *ibT*, *ibUT*, *ls*) ⟷ *finalB* (*cfg*, *ibT*, *ibUT*)
⟨*proof*⟩

3.1 State Transitions

fun *nextN* :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* **where**
nextN (*cfg*, *ibT*, *ibUT*, *ls*) = (*case nextB* (*cfg*,*ibT*,*ibUT*) *of* (*cfg'*,*ibT'*,*ibUT'*) ⇒ (*cfg'*,*ibT'*,*ibUT'*,*ls* ∪ *readLocs cfg*))

lemma *nextN-stepN*: ¬ *finalN cfg-ib-ls* ⟹ *cfg-ib-ls* →*N* (*nextN cfg-ib-ls*)
⟨*proof*⟩

lemma *stepN-nextN*: *cfg-ib-ls* →*N* *cfg'-ib'-ls'* ⟹ *cfg'-ib'-ls'* = *nextN cfg-ib-ls*
⟨*proof*⟩

lemma *nextN-iff-stepN*:
¬ *finalN cfg-ib-ls* ⟹ *nextN cfg-ib-ls* = *cfg'-ib'-ls'* ⟷ *cfg-ib-ls* →*N* *cfg'-ib'-ls'*
⟨*proof*⟩

lemma *stepN-iff-nextN*: *cfg-ib-ls* →*N* *cfg'-ib'-ls'* ⟷ ¬ *finalN cfg-ib-ls* ∧ *nextN cfg-ib-ls* = *cfg'-ib'-ls'*
⟨*proof*⟩

lemma *finalN-endPC*: *pcOf cfg* = *endPC* ⟹ *finalN* (*cfg*,*ibT*,*ibUT*)
⟨*proof*⟩

lemma *stepN-endPC*: *pcOf cfg* = *endPC* ⟹ ¬ (*cfg*,*ibT*,*ibUT*) →*N* (*cfg'*,*ibT'*,*ibUT'*)
⟨*proof*⟩

lemma *stebN-0*: (*Config 0 s*, *ibT*, *ibUT*, *ls*) →*N* (*Config 1 s*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalB-eq-finalN*:*finalB* (*cfg*, *ibT*,*ibUT*) ⟷ (∀ *ls*. *finalN* (*cfg*, *ibT*,*ibUT*, *ls*))
⟨*proof*⟩

3.2 Elimination Rules

lemma *stepN-Assign2E*:
assumes ‹(*cfg1*, *ibT1*,*ibUT1*, *ls1*) →*N* (*cfg1'*, *ibT1'*,*ibUT1'*, *ls1'*)›
and ‹(*cfg2*, *ibT2*,*ibUT2*, *ls2*) →*N* (*cfg2'*, *ibT2'*,*ibUT2'*, *ls2'*)›
and ‹*cfg1* = (*Config pc1* (*State* (*Vstore vs1*) *avst1 h1 p1*))› **and** ‹*cfg1'* = (*Config pc1'* (*State* (*Vstore vs1'*) *avst1' h1' p1'*))›
and ‹*cfg2* = (*Config pc2* (*State* (*Vstore vs2*) *avst2 h2 p2*))› **and** ‹*cfg2'* = (*Config pc2'* (*State* (*Vstore vs2'*) *avst2' h2' p2'*))›
and ‹*prog*!*pc1* = (*x* ::= *a*)› **and** ‹*pcOf cfg1* = *pcOf cfg2*›
shows ‹*vs1'* = (*vs1*(*x* := *aval a* (*stateOf cfg1*))) ∧ *ibT1* = *ibT1'* ∧ *ibUT1* = *ibUT1'* ∧
vs2' = (*vs2*(*x* := *aval a* (*stateOf cfg2*))) ∧ *ibT2* = *ibT2'* ∧ *ibUT2* = *ibUT2'* ∧
pc1' = *Suc pc1* ∧ *pc2'* = *Suc pc2* ∧ *ls2'* = *ls2* ∪ *readLocs cfg2* ∧
avst1' = *avst1* ∧ *avst2'* = *avst2* ∧ *ls1'* = *ls1* ∪ *readLocs cfg1*›
⟨*proof*⟩

lemma *stepN-Seq-Start-Skip-Fence2E*:
assumes ‹(*cfg1*, *ibT1*,*ibUT1*, *ls1*) →*N* (*cfg1'*, *ibT1'*,*ibUT1'*, *ls1'*)›
and ‹(*cfg2*, *ibT2*,*ibUT2*, *ls2*) →*N* (*cfg2'*, *ibT2'*,*ibUT2'*, *ls2'*)›
and ‹*cfg1* = (*Config pc1* (*State* (*Vstore vs1*) *avst1 h1 p1*))› **and** ‹*cfg1'* = (*Config pc1'* (*State* (*Vstore vs1'*) *avst1' h1' p1'*))›
and ‹*cfg2* = (*Config pc2* (*State* (*Vstore vs2*) *avst2 h2 p2*))› **and** ‹*cfg2'* = (*Config pc2'* (*State* (*Vstore vs2'*) *avst2' h2' p2'*))›
and ‹*prog*!*pc1* ∈ {*Start*, *Skip*, *Fence*}› **and** ‹*pcOf cfg1* = *pcOf cfg2*›
shows ‹*vs1'* = *vs1* ∧ *vs2'* = *vs2* ∧
pc1' = *Suc pc1* ∧ *pc2'* = *Suc pc2* ∧
avst1' = *avst1* ∧ *avst2'* = *avst2* ∧
ls2' = *ls2* ∧ *ls1'* = *ls1*›
⟨*proof*⟩

end

end

4 Misprediction and Speculative Semantics

This theory formalizes an optimized speculative semantics, which allows for a characterization of the Spectre vulnerability, this work is inspired and based off the speculative semantics introduced by Cheang et al. [1]

theory *Step-Spec*
imports *Step-Basic*
begin

4.1 Misprediction Oracle

The speculative semantics is parameterised by a misprediction oracle. This consists of a predictor state:

typedecl *predState*

Along with predicates "mispred" (which decides when a misprediction occurs), "resolve" (which decides for when a speculation is resolved)

Both depend on the predictor state (which evolves via the update function) and the program counters of nested speculation

locale *Prog-Mispred* =
Prog prog
for *prog* :: *com list*
+
fixes *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
and *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool*
and *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
begin

4.2 Mispredicting Step

stepM simply goes the other way than stepB at branches

inductive
stepM :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*M* *55*)
where
IfTrue[*intro*]:
pc < *endPC* ⟹ *prog*!*pc* = *IfJump b pc1 pc2* ⟹
bval b s ⟹
(*Config pc s*, *ibT*, *ibUT*) →*M* (*Config pc2 s*, *ibT*, *ibUT*)

|
IfFalse[*intro*]:
$pc < endPC \Longrightarrow prog!pc = IfJump\ b\ pc1\ pc2 \Longrightarrow$
$\neg\ bval\ b\ s \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT) \rightarrow M\ (Config\ pc1\ s,\ ibT,\ ibUT)$

4.2.1 State Transitions

definition $finalM = final\ stepM$

lemma *finalM-iff-aux*:
$pc < endPC \wedge is\text{-}IfJump\ (prog!pc)$
$\longleftrightarrow$
$(\exists\ cfg'.\ (Config\ pc\ s,\ ibT,\ ibUT) \rightarrow M\ cfg')$
⟨*proof*⟩

lemma *finalM-iff*:
$finalM\ (Config\ pc\ (State\ vst\ avst\ h\ p),\ ibT,\ ibUT)$
$\longleftrightarrow$
$(pc \geq endPC \vee \neg\ is\text{-}IfJump\ (prog!pc))$
⟨*proof*⟩

lemma *finalB-imp-finalM*:
$finalB\ (cfg,\ ibT,\ ibUT) \Longrightarrow finalM\ (cfg,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *not-finalM-imp-not-finalB*:
$\neg\ finalM\ (cfg,\ ibT,\ ibUT) \Longrightarrow \neg\ finalB\ (cfg,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *stepM-determ*:
$cfg\text{-}ib \rightarrow M\ cfg\text{-}ib' \Longrightarrow cfg\text{-}ib \rightarrow M\ cfg\text{-}ib'' \Longrightarrow cfg\text{-}ib'' = cfg\text{-}ib'$
⟨*proof*⟩

definition $nextM :: config \times val\ llist \times val\ llist \Rightarrow config \times val\ llist \times val\ llist$
where
$nextM\ cfg\text{-}ib \equiv SOME\ cfg'\text{-}ib'.\ cfg\text{-}ib \rightarrow M\ cfg'\text{-}ib'$

lemma *nextM-stepM*: $\neg\ finalM\ cfg\text{-}ib \Longrightarrow cfg\text{-}ib \rightarrow M\ (nextM\ cfg\text{-}ib)$
⟨*proof*⟩

lemma *stepM-nextM*: $cfg\text{-}ib \rightarrow M\ cfg'\text{-}ib' \Longrightarrow cfg'\text{-}ib' = nextM\ cfg\text{-}ib$
⟨*proof*⟩

lemma *nextM-iff-stepM*: $\neg\ finalM\ cfg\text{-}ib \Longrightarrow nextM\ cfg\text{-}ib\ = cfg'\text{-}ib' \longleftrightarrow cfg\text{-}ib \rightarrow M\ cfg'\text{-}ib'$
⟨*proof*⟩

lemma *stepM-iff-nextM*: *cfg-ib* $\rightarrow$*M cfg'-ib'* $\longleftrightarrow$ $\neg$ *finalM cfg-ib* $\wedge$ *nextM cfg-ib* = *cfg'-ib'*
⟨*proof*⟩

lemma *nextM-IfTrue*[*simp*]:
pc < *endPC* $\Longrightarrow$ *prog*!*pc* = *IfJump b pc1 pc2* $\Longrightarrow$
$\neg$ *bval b s* $\Longrightarrow$
nextM (*Config pc s*, *ibT*, *ibUT*) = (*Config pc1 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-IfFalse*[*simp*]:
pc < *endPC* $\Longrightarrow$ *prog*!*pc* = *IfJump b pc1 pc2* $\Longrightarrow$
bval b s $\Longrightarrow$
nextM (*Config pc s*, *ibT*, *ibUT*) = (*Config pc2 s*, *ibT*, *ibUT*)
⟨*proof*⟩

end

4.3 Speculative Semantics

A "speculative" configuration is a quadruple consisting of:

- The predictor's state
- The nonspeculative configuration (at level 0 so to speak)
- The list of speculative configurations (modelling nested speculation, levels 1 to n, from left to right: so the last in this list is at the current speculaton level, n)
- The list of inputs in the input buffer

We think of cfgs as a stack of configurations, one for each speculation level in a nested speculative execution. At level 0 (empty list) we have the configuration for normal, non-speculative execution. At each moment, only the top of the configuration stack, "hd cfgs" is active.

type-synonym *configS* = *predState* $\times$ *config* $\times$ *config list* $\times$ *val llist* $\times$ *val llist* $\times$ *loc set*

context *Prog-Mispred*
begin

The speculative semantics is more involved than both the normal and basic semantics, so a short description of each rule is provided:

- Non_spec_ normal: when we are either not mispredicting or not at a branch and there is no current speculation, i.e. normal execution
- Nonspec_mispred: when we are mispredicting and at a branch, speculation occurs down the wrong branch, i.e. branch misprediction
- Spec_normal: when we are either not mispredicting or not at a branch BUT there is speculation, i.e. standard speculative execution
- Spec_mispred: when we are mispredicting and at a branch, AND also speculating... speculation occurs down the wrong branch, and we go to another speculation level i.e. nested speculative execution
- Spec_Fence: when there is current speculation and a Fence is hit, all speculation resolves
- Spec Resolve: If the resolve predicate is true, resolution occurs for one speculation level. In contrast to Fences, resolve does not necessarily kill all speculation levels, but allows resolution one level at a time

inductive
stepS :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* 55)
where
nonspec-normal:
cfgs = [] ⟹
¬ *is-IfJump* (*prog*!(*pcOf cfg*)) ∨ ¬ *mispred pstate* [*pcOf cfg*] ⟹
pstate′ = *pstate* ⟹
¬ *finalB* (*cfg*, *ibT*, *ibUT*) ⟹ (*cfg*′, *ibT*′, *ibUT*′) = *nextB* (*cfg*, *ibT*, *ibUT*) ⟹
cfgs′ = [] ⟹
ls′ = *ls* ∪ *readLocs cfg*
⟹
(*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate*′, *cfg*′, *cfgs*′, *ibT*′, *ibUT*′, *ls*′)
|
nonspec-mispred:
cfgs = [] ⟹
is-IfJump (*prog*!(*pcOf cfg*)) ⟹ *mispred pstate* [*pcOf cfg*] ⟹
pstate′ = *update pstate* [*pcOf cfg*] ⟹
¬ *finalM* (*cfg*, *ibT*, *ibUT*) ⟹ (*cfg*′, *ibT*′, *ibUT*′) = *nextB* (*cfg*, *ibT*, *ibUT*) ⟹
(*cfg1*′, *ibT1*′, *ibUT1*′) = *nextM* (*cfg*, *ibT*, *ibUT*) ⟹
cfgs′ = [*cfg1*′] ⟹
ls′ = *ls* ∪ *readLocs cfg*
⟹
(*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate*′, *cfg*′, *cfgs*′, *ibT*′, *ibUT*′, *ls*′)
|
spec-normal:
cfgs ≠ [] ⟹
¬ *resolve pstate* (*pcOf cfg* # *map pcOf cfgs*) ⟹
¬ *is-IfJump* (*prog*!(*pcOf* (*last cfgs*))) ∨ ¬ *mispred pstate* (*pcOf cfg* # *map pcOf cfgs*) ⟹

$prog!(pcOf\ (last\ cfgs)) \neq Fence \Longrightarrow$
$pstate' = pstate \Longrightarrow$
$\neg\ is\text{-}getInput\ (prog!(pcOf\ (last\ cfgs))) \Longrightarrow$
$\neg\ is\text{-}Output\ (prog!(pcOf\ (last\ cfgs))) \Longrightarrow$
$\neg\ finalB\ (last\ cfgs,\ ibT,\ ibUT) \Longrightarrow (cfg1',ibT',\ ibUT') = nextB\ (last\ cfgs,\ ibT,\ ibUT) \Longrightarrow$
$cfg' = cfg \Longrightarrow cfgs' = butlast\ cfgs\ @\ [cfg1'] \Longrightarrow$
$ls' = ls \cup readLocs\ (last\ cfgs)$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
|
spec-mispred:
$cfgs \neq [] \Longrightarrow$
$\neg\ resolve\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$is\text{-}IfJump\ (prog!(pcOf\ (last\ cfgs))) \Longrightarrow mispred\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$pstate' = update\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$\neg\ finalM\ (last\ cfgs,\ ibT,\ ibUT) \Longrightarrow$
$(lcfg',\ ibT',\ ibUT') = nextB\ (last\ cfgs,\ ibT,\ ibUT) \Longrightarrow (cfg1',\ ibT1',\ ibUT1') = nextM\ (last\ cfgs,\ ibT,\ ibUT) \Longrightarrow$
$cfg' = cfg \Longrightarrow cfgs' = butlast\ cfgs\ @\ [lcfg']\ @\ [cfg1'] \Longrightarrow$
$ls' = ls \cup readLocs\ (last\ cfgs)$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
|
spec-Fence:
$cfgs \neq [] \Longrightarrow$
$\neg\ resolve\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$prog!(pcOf\ (last\ cfgs)) = Fence \Longrightarrow$
$pstate' = pstate \Longrightarrow cfg' = cfg \Longrightarrow cfgs' = [] \Longrightarrow$
$ibT = ibT' \Longrightarrow ibUT = ibUT' \Longrightarrow ls' = ls$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
|
spec-resolve:
$cfgs \neq [] \Longrightarrow$
$resolve\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$pstate' = update\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$cfg' = cfg \Longrightarrow cfgs' = butlast\ cfgs \Longrightarrow$
$ibT = ibT' \Longrightarrow ibUT = ibUT' \Longrightarrow ls' = ls$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$

lemmas *stepS-induct* = *stepS.induct*[*split-format*(*complete*)]

4.3.1 State Transitions

lemma *stepS-nonspec-normal-iff*[*simp*]:
$cfgs = [] \Longrightarrow \neg\ is\text{-}IfJump\ (prog!(pcOf\ cfg)) \vee \neg\ mispred\ pstate\ [pcOf\ cfg]$

$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
$\longleftrightarrow$
$(pstate' = pstate \wedge \neg\ finalB\ (cfg,\ ibT,\ ibUT) \wedge$
$(cfg',\ ibT',\ ibUT') = nextB\ (cfg,\ ibT,\ ibUT) \wedge$
$cfgs' = [] \wedge ls' = ls \cup readLocs\ cfg)$
$\langle proof \rangle$

lemma *stepS-nonspec-normal-iff1*[*simp*]:
$cfgs = [] \Longrightarrow \neg\ is\text{-}IfJump\ (prog!pc) \vee \neg\ mispred\ pstate\ [pc]$
$\Longrightarrow$
$(pstate,\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p)),\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',$
$(Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p')),\ cfgs',\ ibT',\ ibUT',\ ls')$
$\longleftrightarrow$
$(pstate' = pstate \wedge \neg\ finalB\ ((Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p)),\ ibT,\ ibUT)$
$\wedge$
$((Config\ pc'\ (State\ (Vstore\ vs')\ avst'\ h'\ p')),\ ibT',\ ibUT') = nextB\ ((Config\ pc$
$(State\ (Vstore\ vs)\ avst\ h\ p)),\ ibT,\ ibUT) \wedge$
$cfgs' = [] \wedge ls' = ls \cup readLocs\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ h\ p)))$
$\langle proof \rangle$

lemma *stepS-nonspec-mispred-iff*[*simp*]:
$cfgs = [] \Longrightarrow is\text{-}IfJump\ (prog!(pcOf\ cfg)) \Longrightarrow mispred\ pstate\ [pcOf\ cfg]$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
$\longleftrightarrow$
$(\exists\ cfg1'\ ibT1'\ ibUT1'.\ pstate' = update\ pstate\ [pcOf\ cfg] \wedge$
$\neg\ finalM\ (cfg,\ ibT,\ ibUT) \wedge (cfg',\ ibT',\ ibUT') = nextB\ (cfg,\ ibT,\ ibUT) \wedge$
$(cfg1',\ ibT1',\ ibUT1') = nextM\ (cfg,\ ibT,\ ibUT) \wedge$
$cfgs' = [cfg1'] \wedge ls' = ls \cup readLocs\ cfg)$
$\langle proof \rangle$

lemma *stepS-spec-normal-iff*[*simp*]:
$cfgs \neq [] \Longrightarrow$
$\neg\ resolve\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs) \Longrightarrow$
$\neg\ is\text{-}IfJump\ (prog!(pcOf\ (last\ cfgs))) \vee \neg\ mispred\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf$
$cfgs) \Longrightarrow$
$prog!(pcOf\ (last\ cfgs)) \neq Fence$
$\Longrightarrow$
$(pstate,\ cfg,\ cfgs,\ ibT,\ ibUT,\ ls) \rightarrow S\ (pstate',\ cfg',\ cfgs',\ ibT',\ ibUT',\ ls')$
$\longleftrightarrow$
$(\exists\ cfg1'.\ pstate' = pstate \wedge$
$\neg\ is\text{-}getInput\ (prog!(pcOf\ (last\ cfgs))) \wedge$
$\neg\ is\text{-}getInput\ (prog!(pcOf\ (last\ cfgs))) \wedge \neg\ is\text{-}Output\ (prog!(pcOf\ (last\ cfgs)))$
$\wedge$
$\neg\ finalB\ (last\ cfgs,\ ibT,\ ibUT) \wedge (cfg1',ibT',ibUT') = nextB\ (last\ cfgs,\ ibT,$
$ibUT) \wedge$
$cfg' = cfg \wedge cfgs' = butlast\ cfgs\ @\ [cfg1'] \wedge ls' = ls \cup readLocs\ (last\ cfgs))$

⟨*proof*⟩

lemma *stepS-spec-mispred-iff*[*simp*]:
cfgs ≠ [] ⟹
¬ *resolve pstate* (*pcOf cfg* # *map pcOf cfgs*) ⟹
is-IfJump (*prog*!(*pcOf* (*last cfgs*))) ⟹ *mispred pstate* (*pcOf cfg* # *map pcOf cfgs*)
⟹
(*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate'*, *cfg'*, *cfgs'*, *ibT'*, *ibUT'*, *ls'*)
⟷
(∃ *cfg1' ibT1' ibUT1' lcfg'*. *pstate'* = *update pstate* (*pcOf cfg* # *map pcOf cfgs*) ∧

¬ *finalM* (*last cfgs*, *ibT*, *ibUT*) ∧
(*lcfg'*, *ibT'*, *ibUT'*) = *nextB* (*last cfgs*, *ibT*, *ibUT*) ∧
(*cfg1'*, *ibT1'*, *ibUT1'*) = *nextM* (*last cfgs*, *ibT*, *ibUT*) ∧
cfg' = *cfg* ∧ *cfgs'* = *butlast cfgs* @ [*lcfg'*] @ [*cfg1'*] ∧ *ls'* = *ls* ∪ *readLocs* (*last cfgs*))
⟨*proof*⟩

lemma *stepS-spec-Fence-iff*[*simp*]:
cfgs ≠ [] ⟹
¬ *resolve pstate* (*pcOf cfg* # *map pcOf cfgs*) ⟹
prog!(*pcOf* (*last cfgs*)) = *Fence*
⟹
(*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate'*, *cfg'*, *cfgs'*, *ibT'*, *ibUT'*, *ls'*)
⟷
(*pstate'* = *pstate* ∧ *cfg* = *cfg'* ∧ *cfgs'* = [] ∧ *ibT'* = *ibT* ∧ *ibUT'* = *ibUT* ∧ *ls'* = *ls*)
⟨*proof*⟩

lemma *stepS-spec-resolve-iff*[*simp*]:
cfgs ≠ [] ⟹
resolve pstate (*pcOf cfg* # *map pcOf cfgs*)
⟹
(*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate'*, *cfg'*, *cfgs'*, *ibT'*, *ibUT'*, *ls'*)
⟷
(*pstate'* = *update pstate* (*pcOf cfg* # *map pcOf cfgs*) ∧
cfg' = *cfg* ∧ *cfgs'* = *butlast cfgs* ∧ *ibT'* = *ibT* ∧ *ibUT'* = *ibUT* ∧ *ls'* = *ls*)
⟨*proof*⟩

lemma *stepS-cases*[*cases pred*: *stepS*,
consumes 1,
case-names nonspec-normal nonspec-mispred
spec-normal spec-mispred spec-Fence spec-resolve]:
assumes (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*) →*S* (*pstate'*, *cfg'*, *cfgs'*, *ibT'*, *ibUT'*, *ls'*)
obtains

$cfgs = []$
  $\neg$ *is-IfJump* (*prog*!(*pcOf cfg*)) $\vee \neg$ *mispred pstate* [*pcOf cfg*]
  $pstate' = pstate$
  $\neg$ *finalB* (*cfg*, *ibT*, *ibUT*)
  (*cfg'*, *ibT'*, *ibUT'*) = *nextB* (*cfg*, *ibT*, *ibUT*)
  $cfgs' = []$
  $ls' = ls \cup$ *readLocs cfg*
|

$cfgs = []$
  is-IfJump (*prog*!(*pcOf cfg*)) *mispred pstate* [*pcOf cfg*]
  $pstate' =$ *update pstate* [*pcOf cfg*]
  $\neg$ *finalM* (*cfg*, *ibT*, *ibUT*)
  (*cfg'*, *ibT'*, *ibUT'*) = *nextB* (*cfg*, *ibT*, *ibUT*)
  $\exists$ *cfg1' ibT1' ibUT1'*. (*cfg1'*, *ibT1'*, *ibUT1'*) = *nextM* (*cfg*, *ibT*, *ibUT*)
    $\wedge$ *cfgs'* = [*cfg1'*]
  $ls' = ls \cup$ *readLocs cfg*
|

$cfgs \neq []$
  $\neg$ *resolve pstate* (*pcOf cfg # map pcOf cfgs*)
    $\neg$ *is-IfJump* (*prog*!(*pcOf* (*last cfgs*))) $\vee \neg$ *mispred pstate* (*pcOf cfg # map pcOf cfgs*)
    prog!(*pcOf* (*last cfgs*)) $\neq$ *Fence*
    $pstate' = pstate$
    $\neg$ *is-getInput* (*prog*!(*pcOf* (*last cfgs*)))
    $\neg$ *is-Output* (*prog*!(*pcOf* (*last cfgs*)))
    $cfg' = cfg$
    $ls' = ls \cup$ *readLocs* (*last cfgs*)
    $\exists$ *cfg1'*. *nextB* (*last cfgs*, *ibT*, *ibUT*) = (*cfg1'*,*ibT'*,*ibUT'*)
      $\wedge$ *cfgs'* = *butlast cfgs* @ [*cfg1'*]
|

$cfgs \neq []$
  $\neg$ *resolve pstate* (*pcOf cfg # map pcOf cfgs*)
    is-IfJump (*prog*!(*pcOf* (*last cfgs*))) *mispred pstate* (*pcOf cfg # map pcOf cfgs*)
    $pstate' =$ *update pstate* (*pcOf cfg # map pcOf cfgs*)
    $\neg$ *finalM* (*last cfgs*, *ibT*, *ibUT*)
    $cfg' = cfg$
    $\exists$ *lcfg' cfg1' ibT1' ibUT1'*.
     nextB (*last cfgs*, *ibT*, *ibUT*) = (*lcfg'*,*ibT'*,*ibUT'*) $\wedge$
     (*cfg1'*, *ibT1'*, *ibUT1'*) = *nextM* (*last cfgs*, *ibT*, *ibUT*) $\wedge$
     cfgs' = *butlast cfgs* @ [*lcfg'*] @ [*cfg1'*]
    $ls' = ls \cup$ *readLocs* (*last cfgs*)
|

$cfgs \neq []$
  $\neg$ *resolve pstate* (*pcOf cfg # map pcOf cfgs*)
    prog!(*pcOf* (*last cfgs*)) = *Fence*

$pstate' = pstate$
$cfg' = cfg$
$cfgs' = []$
$ibT' = ibT$
$ibUT' = ibUT$
$ls' = ls$
|

$cfgs \neq []$
resolve pstate (pcOf cfg # map pcOf cfgs)
$pstate' = update\ pstate\ (pcOf\ cfg\ \#\ map\ pcOf\ cfgs)$
$cfg' = cfg$
$cfgs' = butlast\ cfgs$
$ls' = ls$
$ibT' = ibT$
$ibUT' = ibUT$
⟨*proof*⟩

lemma *stepS-endPC*: $pcOf\ cfg = endPC \Longrightarrow \neg\ (pstate, cfg, [], ibT, ibUT, ls) \rightarrow S\ ss'$
⟨*proof*⟩

abbreviation
stepsS :: *configS* ⇒ *configS* ⇒ *bool* (**infix** $\rightarrow S*$ 55)
where $x \rightarrow S*\ y \equiv star\ stepS\ x\ y$

definition *finalS* = *final stepS*
lemmas *finalS-defs* = *final-def finalS-def*

lemma *stepS-0*: $(pstate, Config\ 0\ s, [], ibT, ibUT, ls) \rightarrow S\ (pstate, Config\ 1\ s, [], ibT, ibUT, ls)$
⟨*proof*⟩

lemma *stepS-imp-stepB*:$(pstate, cfg, [], ibT,ibUT, ls) \rightarrow S\ (pstate', cfg', cfgs', ibT',ibUT', ls') \Longrightarrow (cfg, ibT,ibUT) \rightarrow B\ (cfg', ibT',ibUT')$
⟨*proof*⟩

4.3.2 Elimination Rules

lemma *stepS-Assign2E*:
assumes ‹$(ps3, cfg3, cfgs3, ibT3,ibUT3, ls3) \rightarrow S\ (ps3', cfg3', cfgs3', ibT3',ibUT3', ls3')$›
and ‹$(ps4, cfg4, cfgs4, ibT4,ibUT4, ls4) \rightarrow S\ (ps4', cfg4', cfgs4', ibT4',ibUT4', ls4')$›
and ‹$cfg3 = (Config\ pc3\ (State\ (Vstore\ vs3)\ avst3\ h3\ p3))$› **and** ‹$cfg3' = (Config\ pc3'\ (State\ (Vstore\ vs3')\ avst3'\ h3'\ p3'))$›
and ‹$cfg4 = (Config\ pc4\ (State\ (Vstore\ vs4)\ avst4\ h4\ p4))$› **and** ‹$cfg4' = (Config\ pc4'\ (State\ (Vstore\ vs4')\ avst4'\ h4'\ p4'))$›
and ‹$cfgs3 = []$› **and** ‹$cfgs4 = []$›

and ‹*prog*!*pc3* = (*x* ::= *a*)› **and** ‹*pcOf cfg3* = *pcOf cfg4*›
shows ‹*cfgs3'* = [] ∧ *cfgs4'* = [] ∧
vs3' = (*vs3*(*x* := *aval a* (*stateOf cfg3*))) ∧
vs4' = (*vs4*(*x* := *aval a* (*stateOf cfg4*))) ∧
pc3' = *Suc pc3* ∧ *pc4'* = *Suc pc4* ∧ *ls4'* = *ls4* ∪ *readLocs cfg4* ∧
avst3' = *avst3* ∧ *avst4'* = *avst4* ∧ *ls3'* = *ls3* ∪ *readLocs cfg3* ∧
p3 = *p3'* ∧ *p4* = *p4'*›
⟨*proof*⟩

end

end

5 Relative Security instantiation - Common Aspects

This theory sets up a generic instantiaton infrastructure for all our running examples. For a detailed explanation of each example and it's (dis)proof of Relative Security see the work by Dongol et al. [2]

theory *Instance-Common*
imports *../IMP/Step-Normal ../IMP/Step-Spec*
begin

no-notation *bot* (⊥)

abbreviation *noninform* (⊥) **where** ⊥ ≡ *undefined*

declare *split-paired-All*[*simp del*]
declare *split-paired-Ex*[*simp del*]

definition *noMisSpec* **where** *noMisSpec* (*cfgs*::*config list*) ≡ (*cfgs* = [])
lemma *noMisSpec-ext*[*simp*]:*map x cfgs* = *map x cfgs'* ⟹ *noMisSpec cfgs* ⟷ *noMisSpec cfgs'*
⟨*proof*⟩

definition *misSpecL1* **where** *misSpecL1* (*cfgs*::*config list*) ≡ (*length cfgs* = *Suc 0*)
lemma *misSpecL1-len*[*simp*]:*misSpecL1 cfgs* ⟷ *length cfgs* = *1* ⟨*proof*⟩

definition *misSpecL2* **where** *misSpecL2* (*cfgs*::*config list*) ≡ (*length cfgs* = *2*)

fun *tuple*::$'a \times 'b \times 'c \Rightarrow 'a \times 'b$
where *tuple* $(a,b,c) = (a,b)$

fun *tuple-sel*::$'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b \times 'd$
where *tuple-sel* $(a,b,c,d,e) = (b,d)$

fun *cfgsOf*::$'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'c$
where *cfgsOf* $(a,b,c,d,e) = c$

fun *pstateOf*::$'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'a$
where *pstateOf* $(a,b,c,d,e) = a$

fun *stateOfs*::$'a \times 'b \times 'c \times 'd \times 'e \Rightarrow 'b$
where *stateOfs* $(a,b,c,d,e) = b$

context *Prog-Mispred*
begin

The "vanilla-semantics" transitions are the normal executions (featuring no speculation):

Vanilla-semantics system model: given by the normal semantics

type-synonym *stateV* = *config* × *val llist* × *val llist* × *loc set*
fun *validTransV* **where** *validTransV* (*cfg-ib-ls*,*cfg-ib-ls'*) = *cfg-ib-ls* →*N* *cfg-ib-ls'*

Vanilla-semantics observation infrastructure (part of the vanilla-semantics state-wise attacker model):

The attacker observes the output value, the program counter history and the set of accessed locations so far:

type-synonym *obsV* = *val* × *loc set*

The attacker-action is just a value (used as input to the function):

type-synonym *actV* = *val*

The attacker's interaction

fun *isIntV* :: *stateV* ⇒ *bool* **where**
isIntV ss = (¬ *finalN ss*)

The attacker interacts with the system by passing input to the function and reading the outputs (standard channel) and the accessed locations (side channel)

fun *getIntV* :: *stateV* ⇒ *actV* × *obsV* **where**
getIntV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*case prog*!(*pcOf cfg*) *of*
  Input T - ⇒ (*lhd ibT*, ⊥)
 |*Input U* - ⇒ (*lhd ibUT*, ⊥)
 |*Output U* - ⇒ (⊥, (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*))
 |- ⇒ (⊥,⊥)
)

lemma *validTransV-iff-nextN*: *validTransV* (*s1*, *s2*) = (¬ *finalN s1* ∧ *nextN s1* = *s2*)
 ⟨*proof*⟩

The optimization-enhanced semantics system model: given by the speculative semantics

type-synonym *stateO* = *configS*
fun *validTransO* **where** *validTransO* (*cfgS*,*cfgS'*) = *cfgS* →*S cfgS'*

Optimization-enhanced semantics observation infrastructure (part of the optimization-enhanced semantics state-wise attacker model): similar to that of the vanilla semantics, in that the standard-channel inputs and outputs are those produced by the normal execution. However, the side-channel outputs (the sets of read locations) are also collected.

type-synonym *obsO* = *val* × *loc set*
type-synonym *actO* = *val*
fun *isIntO* :: *stateO* ⇒ *bool* **where**
isIntO ss = (¬ *finalS ss*)
fun *getIntO* :: *stateO* ⇒ *actO* × *obsO* **where**
getIntO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*case* (*cfgs*, *prog*!(*pcOf cfg*)) *of*
  ([],*Input T* -) ⇒ (*lhd ibT*, ⊥)
 |([],*Input U* -) ⇒ (*lhd ibUT*, ⊥)
 |([],*Output U* -) ⇒
    (⊥, (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*))
 |- ⇒ (⊥,⊥)
)

end

locale *Prog-Mispred-Init* =
Prog-Mispred prog mispred resolve update
for *prog* :: *com list*
and *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
and *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool*
and *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
+
fixes *initPstate* :: *predState*

and *istate* :: *state* ⇒ *bool*
and *input* :: *nat*
begin

fun *istateV* :: *stateV* ⇒ *bool* **where**
istateV (*cfg*, *ibT*, *ibUT*, *ls*) ⟷
pcOf cfg = *0* ∧ *istate* (*stateOf cfg*) ∧
llength ibT = ∞ ∧ *llength ibUT* = ∞ ∧
ls = {}

fun *istateO* :: *stateO* ⇒ *bool* **where**
istateO (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*) ⟷
pstate = *initPstate* ∧
pcOf cfg = *0* ∧ *ls* = {} ∧
istate (*stateOf cfg*) ∧
cfgs = [] ∧ *llength ibT* = ∞ ∧ *llength ibUT* = ∞

lemma *istateV-config-imp*:
istateV (*cfg*, *ibT*,*ibUT*, *ls*) ⟹ *pcOf cfg* = *0* ∧ *ls* = {} ∧ *ibT* ≠ *LNil*
⟨*proof*⟩

lemma *istateO-config-imp*:
istateO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) ⟹
cfgs = [] ∧ *pcOf cfg* = *0* ∧ *ls* = {} ∧ *ibT* ≠ *LNil*
⟨*proof*⟩

definition *same-var-all x cfg1 cfg2 cfg3 cfgs3 cfg4 cfgs4* ≡
vstore (*getVstore* (*stateOf cfg1*)) *x* = *vstore* (*getVstore* (*stateOf cfg4*)) *x* ∧
vstore (*getVstore* (*stateOf cfg2*)) *x* = *vstore* (*getVstore* (*stateOf cfg4*)) *x* ∧
vstore (*getVstore* (*stateOf cfg3*)) *x* = *vstore* (*getVstore* (*stateOf cfg4*)) *x* ∧
(∀ *cfg3′*∈*set cfgs3*. *vstore* (*getVstore* (*stateOf cfg3′*)) *x* = *vstore* (*getVstore* (*stateOf cfg3*)) *x*) ∧
(∀ *cfg4′*∈*set cfgs4*. *vstore* (*getVstore* (*stateOf cfg4′*)) *x* = *vstore* (*getVstore* (*stateOf cfg4*)) *x*)

definition *same-var x cfg cfg′* ≡
vstore (*getVstore* (*stateOf cfg*)) *x* = *vstore* (*getVstore* (*stateOf cfg′*)) *x*

definition *same-var-val x* (*val*::*int*) *cfg cfg′* ≡
vstore (*getVstore* (*stateOf cfg*)) *x* = *vstore* (*getVstore* (*stateOf cfg′*)) *x* ∧
vstore (*getVstore* (*stateOf cfg*)) *x* = *val*

definition *same-var-o ii cfg3 cfgs3 cfg4 cfgs4* ≡
vstore (*getVstore* (*stateOf cfg3*)) *ii* = *vstore* (*getVstore* (*stateOf cfg4*)) *ii* ∧
(∀ *cfg3′*∈*set cfgs3*. *vstore* (*getVstore* (*stateOf cfg3′*)) *ii* = *vstore* (*getVstore* (*stateOf cfg3*)) *ii*) ∧
(∀ *cfg4′*∈*set cfgs4*. *vstore* (*getVstore* (*stateOf cfg4′*)) *ii* = *vstore* (*getVstore* (*stateOf cfg4*)) *ii*)

lemma *set-var-shrink*:∀ *cfg3′*∈*set cfgs*.
vstore (*getVstore* (*stateOf cfg3′*)) *var* =
vstore (*getVstore* (*stateOf cfg*)) *var*
⟹
∀ *cfg3′*∈*set* (*butlast cfgs*).
vstore (*getVstore* (*stateOf cfg3′*)) *var* =
vstore (*getVstore* (*stateOf cfg*)) *var*
⟨*proof*⟩

lemma *heapSimp*:(∀ *cfg′′*∈*set cfgs′′*. *getHheap* (*stateOf cfg′*) = *getHheap* (*stateOf cfg′′*)) ∧ *cfgs′′* ≠ []
⟹ *getHheap* (*stateOf cfg′*) = *getHheap* (*stateOf* (*last cfgs′′*))
⟨*proof*⟩

lemma *heapSimp2*:(∀ *cfg′′*∈*set cfgs′′*. *getHheap* (*stateOf cfg′*) = *getHheap* (*stateOf cfg′′*)) ∧ *cfgs′′* ≠ []

⟹ *getHheap* (*stateOf cfg′*) = *getHheap* (*stateOf* (*hd cfgs′′*))
⟨*proof*⟩

lemma *array-baseSimp*:*array-base aa1* (*getAvstore* (*stateOf cfg*)) =
array-base aa1 (*getAvstore* (*stateOf cfg′*)) ∧
(∀ *cfg′*∈*set cfgs*. *array-base aa1* (*getAvstore* (*stateOf cfg′*)) =
array-base aa1 (*getAvstore* (*stateOf cfg*)))
∧ *cfgs* ≠ []
⟹
array-base aa1 (*getAvstore* (*stateOf cfg*)) =
array-base aa1 (*getAvstore* (*stateOf* (*last cfgs*)))

⟨*proof*⟩

lemma *finalB-imp-finalS*:*finalB* (*cfg*, *ibT*,*ibUT*) ⟹ (∀ *pstate cfgs ls*. *finalS* (*pstate*, *cfg*, [], *ibT*,*ibUT*, *ls*))
⟨*proof*⟩

lemma *cfgs-Suc-zero*[*simp*]:*length cfgs* = *Suc 0* ⟹ *cfgs* = [*last cfgs*]
⟨*proof*⟩

lemma *cfgs-map*[*simp*]:*length cfgs* = *Suc 0* ⟹ *map pcOf cfgs* = [*pcOf* (*last cfgs*)]
⟨*proof*⟩

end

end

6 Relative Security Instance: Secret Memory

This theory sets up an instance of Relative Security with the secrets as the initial memories

theory *Instance-Secret-IMem*
imports *Instance-Common Relative-Security.Relative-Security*
begin

no-notation *bot* (⊥)
type-synonym *secret* = *state*

context *Prog-Mispred*
begin

fun *corrState* :: *stateV* ⇒ *stateO* ⇒ *bool* **where**
corrState cfgO cfgA = *True*

Since all our programs will have "Start" followed by the rest, with the rest not containing "Start". The secret will be "uploaded" at this Start moment.

definition *isSecV* :: *stateV* ⇒ *bool* **where**
isSecV ss ≡ *case ss of* (*cfg*,*ibT*,*ibUT*) ⇒ (*pcOf cfg* = *0*)

We consider the entire initial state as a secret:

fun *getSecV* :: *stateV* ⇒ *secret* **where**
getSecV (*cfg*,*ibT*,*ibUT*) = *stateOf cfg*

The secrecy infrastructure is similar to that of the "original" semantics:

definition *isSecO* :: *stateO* ⇒ *bool* **where**
isSecO ss ≡ *case ss of* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) ⇒ (*pcOf cfg* = *0* ∧ *cfgs* = [])
fun *getSecO* :: *stateO* ⇒ *secret* **where**
getSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) = *stateOf cfg*
lemma *isSecV-iff*:*isSecV ss* ⟷ *pcOf* (*fst ss*) = *0*
⟨*proof*⟩

lemma *validTransO-iff-nextS*: *validTransO* (*s1*, *s2*) = (¬ *finalS s1* ∧ (*stepS s1 s2*))
⟨*proof*⟩

end

sublocale *Prog-Mispred-Init* < *Rel-Sec* **where**
validTransV = *validTransV* **and** *istateV* = *istateV*
and *finalV* = *finalN*
and *isSecV* = *isSecV* **and** *getSecV* = *getSecV*
and *isIntV* = *isIntV* **and** *getIntV* = *getIntV*

and *validTransO* = *validTransO* **and** *istateO* = *istateO*
and *finalO* = *finalS*
and *isSecO* = *isSecO* **and** *getSecO* = *getSecO*
and *isIntO* = *isIntO* **and** *getIntO* = *getIntO*
and *corrState* = *corrState*
⟨*proof*⟩

context *Prog-Mispred-Init*
begin

lemmas *reachV-induct* = *Van.reach.induct*[*split-format*(*complete*)]
lemmas *reachO-induct* = *Opt.reach.induct*[*split-format*(*complete*)]

lemma *is-getTrustedInput-getActV*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Input T s* ⟹ *getActV* (*cfg*,*ibT*,*ibUT*,*ls*) = *lhd ibT*
⟨*proof*⟩

lemma *not-is-getTrustedInput-getActV*[*simp*]:
¬ *is-getInput* (*prog*!(*pcOf cfg*)) ⟹ *getActV* (*cfg*,*ibT*,*ibUT*,*ls*) = *noninform*
⟨*proof*⟩

lemma *is-Output-getObsV*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Output U out* ⟹ *getObsV* (*cfg*,*ibT*,*ibUT*,*ls*) =
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
⟨*proof*⟩

lemma *not-is-Output-getObsV*[*simp*]:
¬ *is-Output* (*prog*!(*pcOf cfg*)) ⟹ *getObsV* (*cfg*,*ibT*,*ibUT*,*ls*) = ⊥
⟨*proof*⟩

lemma *is-getTrustedInput-Nil-getActO*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Input T s* ⟹ *getActO* (*pstate*,*cfg*,[],*ibT*,*ibUT*,*ls*) = *lhd ibT*
⟨*proof*⟩

lemma *not-is-getTrustedInput-Nil-getActO*[*simp*]:
¬ *is-getInput* (*prog*!(*pcOf cfg*))

$\vee$ *cfgs* $\neq$ [] $\Longrightarrow$ *getActO* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) = $\bot$
⟨*proof*⟩

lemma *is-Output-Nil-getObsO*[*simp*]:
prog!(*pcOf cfg*) = *Output U s* $\Longrightarrow$
getObsO (*pstate*,*cfg*,[],*ibT*,*ibUT*,*ls*) = (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
⟨*proof*⟩

lemma *not-is-Output-Nil-getObsO*[*simp*]:
$\neg$ *is-Output* (*prog*!(*pcOf cfg*)) $\vee$ *cfgs* $\neq$ [] $\Longrightarrow$ *getObsO* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*)
= $\bot$
⟨*proof*⟩

lemma *getActV-simps*:
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*case prog*!(*pcOf cfg*) *of*
Input T - ⇒ *lhd ibT*
|*Input U* - ⇒ *lhd ibUT*
|- ⇒ $\bot$
)
⟨*proof*⟩

lemma *getObsV-simps*:
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*case prog*!(*pcOf cfg*) *of*
Output U - ⇒ (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
|- ⇒ $\bot$
)
⟨*proof*⟩

lemma *getActO-simps*:
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*case* (*cfgs*,*prog*!(*pcOf cfg*)) *of*
([],*Input T* -) ⇒ *lhd ibT*
|([],*Input U* -) ⇒ *lhd ibUT*
|- ⇒ $\bot$
)
⟨*proof*⟩

lemma *getObsO-simps*:
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*case* (*cfgs*,*prog*!(*pcOf cfg*)) *of*
([],*Output U* -) ⇒ (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
|- ⇒ $\bot$
)
⟨*proof*⟩

end

end

7 Relative Security Instance: Secret Memory Input

This theory sets up an instance of Relative Security used to prove an Security of a potentially infinite program

theory *Instance-Secret-IMem-Inp*
imports *Instance-Common Relative-Security.Relative-Security*
begin

Using the following notation to denote an undefined element

no-notation *bot* ($\bot$)

definition *ffile* :: *vname* **where** *ffile* = *''ffile''*
definition *xx* :: *vname* **where** *xx* = *''x''*
definition *yy* :: *vname* **where** *yy* = *''yy''*
type-synonym *secret* = *state* × *val* × *val*

abbreviation *writeSecretOnFile* **where** *writeSecretOnFile* ≡ (*Output T* (*Fun* (*V xx*) (*V yy*)))
lemma *writeOnFile-not-Jump*[*simp*]:¬*is-IfJump writeSecretOnFile* ⟨*proof*⟩
lemma *writeOnFile-not-Inp*[*simp*]:¬*is-getInput writeSecretOnFile* ⟨*proof*⟩
lemma *writeOnFile-not-Fence*[*simp*]:*writeSecretOnFile* ≠ *Fence* ⟨*proof*⟩

definition *ffileVal* **where** *ffileVal cfg* = *vstoreOf*(*cfg*) *ffile*
lemma *ffileVal-vstore*[*simp*]:*ffileVal cfg* = *vstoreOf*(*cfg*) *ffile* ⟨*proof*⟩

context *Prog-Mispred*
begin

The following functions and definitions make up the required components of the Relative Security locale

fun *corrState* :: *stateV* ⇒ *stateO* ⇒ *bool* **where**
corrState cfgO cfgA = *True*

definition *isSecV* :: *stateV* ⇒ *bool* **where**
isSecV ss ≡ *case ss of* (*cfg*,*ibT*,*ibUT*,*ls*) ⇒ ¬*finalN ss*

fun *getSecV* :: *stateV* ⇒ *secret* **where**
getSecV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*case prog*!(*pcOf cfg*) *of*

```
   Start ⇒ (stateOf cfg, ⊥, ⊥)
  |Input T - ⇒ (⊥, lhd ibT, ⊥)
  |Output T - ⇒ (⊥,⊥,outOf (prog!(pcOf cfg)) (stateOf cfg))
  |- ⇒ (⊥,⊥,⊥))

lemma isSecV-iff:isSecV ss ⟷ ¬finalN ss
  ⟨proof⟩

definition isSecO :: stateO ⇒ bool where
isSecO ss ≡ case ss of (pstate,cfg,cfgs,ibT,ibUT,ls) ⇒ ¬finalS ss ∧ cfgs = []
fun getSecO :: stateO ⇒ secret where
getSecO (pstate,cfg,cfgs,ibT,ibUT,ls) =
 (case prog!(pcOf cfg) of
    Start ⇒ (stateOf cfg, ⊥, ⊥)
   |Input T - ⇒ (⊥, lhd ibT, ⊥)
   |Output T - ⇒ (⊥,⊥,outOf (prog!(pcOf cfg)) (stateOf cfg))
   |- ⇒ (⊥,⊥,⊥))
end

sublocale Prog-Mispred-Init < Rel-Sec where
  validTransV = validTransV and istateV = istateV
  and finalV = finalN
  and isSecV = isSecV and getSecV = getSecV
  and isIntV = isIntV and getIntV = getIntV

  and validTransO = validTransO and istateO = istateO
  and finalO = finalS
  and isSecO = isSecO  and getSecO = getSecO
  and isIntO = isIntO and getIntO = getIntO
  and corrState = corrState
  ⟨proof⟩

context Prog-Mispred-Init
begin

lemmas reachV-induct = Van.reach.induct[split-format(complete)]
lemmas reachO-induct = Opt.reach.induct[split-format(complete)]

lemma is-getInputT-getActV[simp]:
(prog!(pcOf cfg)) = Input U inp ⟹ getActV (cfg,ibT,ibUT,ls) = lhd ibUT
```

⟨*proof*⟩

lemma *is-getInputU-getActV*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Input T inp* ⟹ *getActV* (*cfg*,*ibT*,*ibUT*,*ls*) = *lhd ibT*
⟨*proof*⟩

lemma *not-is-getInput-getActV*[*simp*]:
¬ *is-getInput* (*prog*!(*pcOf cfg*)) ⟹ *getActV* (*cfg*,*ibT*,*ibUT*,*ls*) = ⊥
⟨*proof*⟩

lemma *is-Output-getObsV*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Output U out* ⟹ *getObsV* (*cfg*,*ibT*,*ibUT*,*ls*) =
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
⟨*proof*⟩

lemma *not-is-Output-getObsV*[*simp*]:
¬ *is-Output* (*prog*!(*pcOf cfg*)) ⟹ *getObsV* (*cfg*,*ibT*,*ibUT*,*ls*) = ⊥
⟨*proof*⟩

lemma *is-getInputT-Nil-getActO*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Input T inp* ⟹ *getActO* (*pstate*,*cfg*,[],*ibT*,*ibUT*,*ls*) = *lhd ibT*
⟨*proof*⟩

lemma *is-getInputU-Nil-getActO*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Input U inp* ⟹ *getActO* (*pstate*,*cfg*,[],*ibT*,*ibUT*,*ls*) = *lhd ibUT*
⟨*proof*⟩

lemma *not-is-getInput-Nil-getActO*[*simp*]:
(¬ *is-getInput* (*prog*!(*pcOf cfg*)))
∨ *cfgs* ≠ [] ⟹ *getActO* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) = ⊥
⟨*proof*⟩

lemma *is-Output-Nil-getObsO*[*simp*]:
(*prog*!(*pcOf cfg*)) = *Output U out* ⟹
getObsO (*pstate*,*cfg*,[],*ibT*,*ibUT*,*ls*) = (*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
⟨*proof*⟩

lemma *not-is-Output-Nil-getObsO*[*simp*]:
¬ *is-Output* (*prog*!(*pcOf cfg*)) ∨ *cfgs* ≠ [] ⟹ *getObsO* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*)
= ⊥
⟨*proof*⟩

lemma *getActV-simps*:
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =

```
 (case prog!(pcOf cfg) of
    Input T - ⇒ lhd ibT
   |Input U - ⇒ lhd ibUT
   |- ⇒ ⊥
 )
  ⟨proof⟩

lemma getObsV-simps:
getObsV (cfg,ibT,ibUT,ls) =
 (case prog!(pcOf cfg) of
    Output U - ⇒ (outOf (prog!(pcOf cfg)) (stateOf cfg), ls)
   |- ⇒ ⊥
 )
  ⟨proof⟩

lemma getActO-simps:
getActO (pstate,cfg,cfgs,ibT,ibUT,ls) =
 (case (cfgs,prog!(pcOf cfg)) of
    ([],Input T -)  ⇒ lhd ibT
   |([],Input U -) ⇒ lhd ibUT
   |- ⇒ ⊥
 )
  ⟨proof⟩

lemma getObsO-simps:
getObsO (pstate,cfg,cfgs,ibT,ibUT,ls) =
 (case (cfgs,prog!(pcOf cfg)) of
    ([],Output U -) ⇒ (outOf (prog!(pcOf cfg)) (stateOf cfg), ls)
   |- ⇒ ⊥
 )
  ⟨proof⟩


end


end
```

8 Disproof of Relative Security for fun1

```
theory Fun1
imports ../Instance-IMP/Instance-Secret-IMem
  Secret-Directed-Unwinding.SD-Unwinding-fin
begin
```

8.1 Function definition and Boilerplate

```
no-notation bot (⊥)
consts NN :: nat
```

consts *input* :: *int*
definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''
definition *vv* :: *avname* **where** *vv* = ''*v*''
definition *xx* :: *avname* **where** *xx* = ''*i*''
definition *tt* :: *avname* **where** *tt* = ''*tt*''

lemma *NN-suc*[*simp*]:*nat* (*NN* + *1*) = *Suc* (*nat NN*)
⟨*proof*⟩

lemma *NN*:*NN*≥*0* ⟨*proof*⟩

lemmas *vvars-defs* = *aa1-def aa2-def vv-def xx-def tt-def*

lemma *vvars-dff*[*simp*]:
aa1 ≠ *aa2 aa1* ≠ *vv aa1* ≠ *xx aa1* ≠ *tt*
aa2 ≠ *aa1 aa2* ≠ *vv aa2* ≠ *xx aa2* ≠ *tt*
vv ≠ *aa1 vv* ≠ *aa2 vv* ≠ *xx vv* ≠ *tt*
xx ≠ *aa1 xx* ≠ *aa2 xx* ≠ *vv xx* ≠ *tt*
tt ≠ *aa1 tt* ≠ *aa2 tt* ≠ *vv tt* ≠ *xx*
⟨*proof*⟩

consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*

definition *s-add* = {*a*. *a* ≠ *nat NN*+*1*}
fun vs_0::*char list* ⇒ *int* **where**
vs_0 *x* = *0*

lemma *vs0*[*simp*]:(λ*x*. *0*) = vs_0 ⟨*proof*⟩

fun *as*:: *char list* ⇒ *nat* × *nat* **where**
as a = (*if a* = *aa1 then* (*0*, *nat NN*)
else (*if a* = *aa2 then* (*nat NN*, *nat size-aa2*)
else (*nat size-aa2*,*0*)))

definition *avst*′ ≡ (*Avstore as*)

lemmas *avst-defs* = *avst*′*-def as.simps*

lemma *avstore-loc*[*simp*]:*Avstore* (λ*a*. *if a* = *aa1 then* (*0*, *nat NN*) *else if a* = *aa2 then* (*nat NN*, *nat size-aa2*) *else* (*nat size-aa2*, *0*)) =
avst′
⟨*proof*⟩

abbreviation *read-add* ≡ {*a*. *a* ≠ (*nat NN* + *1*)}

fun *initVstore* :: *vstore* ⇒ *bool* **where**
initVstore (*Vstore vst*) = (*vst* = vs_0)

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore avst = (*avst* = *avst'*)
fun *initHeap*::(*nat* ⇒ *int*) ⇒ *bool* **where**
initHeap h = (∀ *x*∈*read-add*. *h x* = *0*)

lemma *initAvstore-0*[*intro*]:*initAvstore avst'* ⟹ *array-base aa1 avst'* = *0*
⟨*proof*⟩

fun *istate* ::*state* ⇒ *bool* **where**
istate s =
(*initVstore* (*getVstore s*) ∧
initAvstore (*getAvstore s*) ∧
initHeap (*getHheap s*))

definition *prog* ≡
[
0// *Start* ,
1// *Input U xx* ,
2// *tt* ::= (*N 0*),
3// *IfJump* (*Less* (*V xx*) (*N NN*)) *4 5*,
4// *tt* ::= (*VA aa2* (*Times* (*VA aa1* (*V xx*)) (*N 512*))) ,
5// *Output U* (*V tt*)
]

lemma *cases-5*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨ *i* > *5*
⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < (*int NN*) ∨ *vs xx* ≥ (*int NN*) ⟨*proof*⟩

lemma *is-If-pcOf*[*simp*]:
pcOf cfg < *6* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3*
⟨*proof*⟩

lemma *is-If-pc*[*simp*]:
pc < *6* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3*
⟨*proof*⟩

lemma *eq-Fence-pc*[*simp*]:
pc < *6* ⟹ *prog* ! *pc* ≠ *Fence*
⟨*proof*⟩

fun *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
mispred p pc = (*if pc* = [*3*] *then True else False*)

fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc = (*if pc* = [*5*,*5*] *then True else False*)

consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts $pstate_0$ :: *predState*

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = $pstate_0$ **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B 55*)
where *x* →*B y* == *stepB x y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ *55*)
where *x* →*B*∗ *y* == *star stepB x y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*M 55*)
where *x* →*M y* == *stepM x y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N 55*)
where *x* →*N y* == *stepN x y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ *55*)
where *x* →*N*∗ *y* == *star stepN x y*

abbreviation

stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* *55*)
where *x* →*S* *y* == *stepS x y*

abbreviation
stepsS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S*∗ *55*)
where *x* →*S*∗ *y* == *star stepS x y*

lemma *endPC*[*simp*]: *endPC* = *6*
⟨*proof*⟩

lemma *is-getTrustedInput-pcOf*[*simp*]: *pcOf cfg* < *6* ⟹ *is-getInput* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = *1*
⟨*proof*⟩

lemma *getTrustedInput-pcOf*[*simp*]: (*prog*!*1*) = *Input U xx*
⟨*proof*⟩

lemma *is-Output-pcOf*[*simp*]: *pcOf cfg* < *6* ⟹ *is-Output* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = *5* ∨ *pcOf cfg* = *6*
⟨*proof*⟩

lemma *is-Fence-pcOf*[*simp*]: *pcOf cfg* < *6* ⟹(*prog*!(*pcOf cfg*)) ≠ *Fence*
⟨*proof*⟩

lemma *prog0*[*simp*]:*prog* ! *0* = *Start*
⟨*proof*⟩

lemma *prog1*[*simp*]:*prog* ! (*Suc 0*) = *Input U xx*
⟨*proof*⟩

lemma *prog2*[*simp*]:*prog* ! *2* = *tt* ::= (*N 0*)
⟨*proof*⟩

lemma *prog3*[*simp*]:*prog* ! *3* = *IfJump* (*Less* (*V xx*) (*N NN*)) *4 5*
⟨*proof*⟩

lemma *prog4*[*simp*]:*prog* ! *4* = *tt* ::= (*VA aa2* (*Times* (*VA aa1* (*V xx*)) (*N 512*)))
⟨*proof*⟩

lemma *prog5*[*simp*]:*prog* ! *5* = *Output U* (*V tt*)
⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*, *ibUT*) ⟷ *pcOf cfg* = *0*
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) ⟷ (*pcOf cfg* = *0* ∧ *cfgs* = [])
⟨*proof*⟩

lemma *getInputT-not*[*simp*]: *pcOf cfg* < *6* ⟹
(*prog* ! *pcOf cfg*) ≠ *Input T x*
⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < *6* ⟹
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *1 then lhd ibUT else* ⊥)
⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < *6* ⟹
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *5 then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < *6* ⟹
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *1* ∧ *cfgs* = [] *then lhd ibUT else* ⊥)
⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < *6* ⟹
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if* (*pcOf cfg* = *5* ∧ *cfgs* = []) *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *nextB-pc0*[*simp*]:

nextB (*Config 0 s*, *ibT*, *ibUT*) =
(*Config 1 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc0*[*simp*]:
readLocs (*Config 0 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config 1* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1*[*simp*]:
readLocs (*Config 1 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1′*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config* (*Suc 0*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
=
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1′*[*simp*]:
readLocs (*Config* (*Suc 0*) *s*) = {}
⟨*proof*⟩

lemma *nextB-pc2*[*simp*]:
nextB (*Config 2* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
((*Config 3* (*State* (*Vstore* (*vs*(*tt* := *0*))) *avst h p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc2*[*simp*]:
readLocs (*Config 2* (*State* (*Vstore vs*) *avst h p*)) = {}
⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx < *NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx ≥ *NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =

(*Config 5* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *NN then 4 else 5*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx ≥ *NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx < *NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 5* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *NN then 5 else 4*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:
readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
nextB (*Config 4* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*, *ibUT*) =
(*let i* = *array-loc aa1* (*nat* (*vs xx*)) *avst*; *j* = (*array-loc aa2* (*nat* ((*h i*) ∗ *512*)) *avst*)
in (*Config 5* (*State* (*Vstore* (*vs*(*tt* := *h j*))) *avst* (*Heap h*) *p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:
readLocs (*Config 4* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*)) =
(*let i* = *array-loc aa1* (*nat* (*vs xx*)) *avst*;
j = (*array-loc aa2* (*nat* ((*h i*) ∗ *512*)) *avst*)
in {*i*, *j*})
⟨*proof*⟩

lemma *nextB-pc5*[*simp*]:

$nextB\ (Config\ 5\ s,\ ibT,\ ibUT) = (Config\ 6\ s,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
$readLocs\ (Config\ 5\ (State\ (Vstore\ vs)\ avst\ (Heap\ h)\ p)) =$
$\{\}$
⟨*proof*⟩

lemma *nextB-stepB-pc*:
$pc < 6 \Longrightarrow (pc = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow$
$(Config\ pc\ s,\ ibT,\ ibUT) \rightarrow B\ nextB\ (Config\ pc\ s,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *not-finalB*:
$pc < 6 \Longrightarrow (pc = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow$
$\neg\ finalB\ (Config\ pc\ s,\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *finalB-pc-iff'*:
$pc < 6 \Longrightarrow$
$finalB\ (Config\ pc\ s,\ ibT,\ ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil)$
⟨*proof*⟩

lemma *finalB-pc-iff*:
$pc \leq 6 \Longrightarrow$
$finalB\ (Config\ pc\ s,\ ibT,\ ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil \vee pc = 6)$
⟨*proof*⟩

lemma *finalB-pcOf-iff*[*simp*]:
$pcOf\ cfg \leq 6 \Longrightarrow$
$finalB\ (cfg,\ ibT,\ ibUT) \longleftrightarrow (pcOf\ cfg = 1 \wedge ibUT = LNil \vee pcOf\ cfg = 6)$
⟨*proof*⟩

definition $vs_i\text{-}t\ cfg \equiv (vstore\ (getVstore\ (stateOf\ cfg))\ xx) < NN$
definition $vs_i\text{-}f\ cfg \equiv (vstore\ (getVstore\ (stateOf\ cfg))\ xx) \geq NN$
lemma *vs-xx-cases*:$vs_i\text{-}t\ cfg \vee vs_i\text{-}f\ cfg$ ⟨*proof*⟩

lemmas $vs_i\text{-}defs = vs_i\text{-}t\text{-}def\ vs_i\text{-}f\text{-}def$

lemma *bool-invar*[*simp*]:$\neg vs_i\text{-}t\ (Config\ 6\ s) \Longrightarrow vs_i\text{-}t\ (Config\ 6\ s) \Longrightarrow (Config\ 6\ s,\ ib1) \rightarrow B\ (Config\ 6\ s,\ ib1) \Longrightarrow False$
⟨*proof*⟩
lemma *nextB-vs-consistent-aux*:

$2 \leq pc \wedge pc < 6 \Longrightarrow$
$(nextB\ (Config\ pc\ (State\ (Vstore\ vs)\ avst\ (Heap\ h)\ p), ibT, ibUT)) = (Config\ pc'\ (State\ (Vstore\ vs')\ avst''\ (Heap\ h')\ p'), ibT', ibUT') \Longrightarrow$
$avst = avst'' \wedge$
$vs\ xx = vs'\ xx \wedge$
$h = h' \wedge$
$pc < pc'$
⟨*proof*⟩

lemma *nextB-vs-consistent*:
$2 \leq pcOf\ cfg \wedge pcOf\ cfg < 6 \Longrightarrow$
$(nextB\ (cfg, ibT, ibUT)) = (cfg', ibT', ibUT') \Longrightarrow$
$(getAvstore\ (stateOf\ cfg)) = (getAvstore\ (stateOf\ cfg')) \wedge$
$(getHheap\ (stateOf\ cfg)) = (getHheap\ (stateOf\ cfg')) \wedge$
$vstore\ (getVstore\ (stateOf\ cfg))\ xx = vstore\ (getVstore\ (stateOf\ cfg'))\ xx$
⟨*proof*⟩

lemma *nextB-vs$_i$-t-consistent*:
$2 \leq pcOf\ cfg \wedge pcOf\ cfg < 6 \Longrightarrow$
$(nextB\ (cfg, ibT, ibUT)) = (cfg', ibT', ibUT') \Longrightarrow$
$vs_i\text{-}t\ cfg \longleftrightarrow vs_i\text{-}t\ cfg'$
⟨*proof*⟩

lemma *nextB-vs$_i$-f-consistent*:
$2 \leq pcOf\ cfg \wedge pcOf\ cfg < 6 \Longrightarrow$
$(nextB\ (cfg, ibT, ibUT)) = (cfg', ibT', ibUT') \Longrightarrow$
$vs_i\text{-}f\ cfg \longleftrightarrow vs_i\text{-}f\ cfg'$
⟨*proof*⟩

end

8.2 Proof

theory *Fun1-insecure*
imports *Fun1*
begin

8.2.1 Concrete leak

definition $PC \equiv \{0..6\}$

definition *same-xx cfg3 cfgs3 cfg4 cfgs4* $\equiv$
$vstore\ (getVstore\ (stateOf\ cfg3))\ xx = vstore\ (getVstore\ (stateOf\ cfg4))\ xx \wedge$
$(\forall cfg3' \in set\ cfgs3.\ vstore\ (getVstore\ (stateOf\ cfg3'))\ xx = vstore\ (getVstore\ (stateOf\ cfg3))\ xx) \wedge$
$(\forall cfg4' \in set\ cfgs4.\ vstore\ (getVstore\ (stateOf\ cfg4'))\ xx = vstore\ (getVstore\ (stateOf\ cfg4))\ xx)$

definition *trueProg* = {*2*,*3*,*4*,*5*,*6*}
definition *falseProg* = {*2*,*3*,*5*,*6*}

definition $pstate_1 \equiv update\ pstate_0\ [3]$
definition $pstate_2 \equiv update\ pstate_1\ [5,5]$

lemmas *pstate-def* = $pstate_1$*-def* $pstate_2$*-def*

fun hh_3:: *nat* ⇒ *int* **where**
$hh_3\ x = (if\ x = (nat\ NN + 1)\ then\ 5\ else\ 0)$

definition $h_3 \equiv (Heap\ hh_3)$

fun hh_4:: *nat* ⇒ *int* **where**
$hh_4\ x = (if\ x = (nat\ NN + 1)\ then\ 6\ else\ 0)$

definition $h_4 \equiv (Heap\ hh_4)$

lemmas *h-def* = h_3*-def* h_4*-def* hh_3*.simps* hh_4*.simps*

lemma *ss-neq-aux1*:*nat*(*5* ∗ *512*) ≠ *nat* (*6* ∗ *512*) ⟨*proof*⟩
lemma *ss-neq-aux2*:*nat*(*3* ∗ *512*) ≠ *nat* (*5* ∗ *512*) ⟨*proof*⟩
lemmas *ss-neq* = *ss-neq-aux1 ss-neq-aux2*

definition *p* ≡ *nat size-aa1* + *nat size-aa2*

definition $vs_1 \equiv (vs_0(xx := NN + 1))$

definition $vs_2 \equiv (vs_1(tt := 0))$

definition $aa1_i \equiv array\text{-}loc\ aa1\ (nat\ (vs_2\ xx))\ avst'$

definition $aa2_{vs3} \equiv array\text{-}loc\ aa2\ (nat\ (hh_3\ aa1_i * 512))\ avst'$

definition $vs_{33} = vs_2(tt := hh_3\ aa2_{vs3})$

definition $aa2_{vs4} \equiv array\text{-}loc\ aa2\ (nat\ (hh_4\ aa1_i * 512))\ avst'$

definition $vs_{34} = vs_2(tt := hh_4\ aa2_{vs4})$

lemmas $reads_m$*-def* $= aa1_i$*-def* $aa2_{vs3}$*-def* $aa2_{vs4}$*-def*
lemmas *vs-def* $= vs_0$*.simps* vs_1*-def* vs_2*-def* vs_{33}*-def* vs_{34}*-def*

definition $s_{03} \equiv (State\ (Vstore\ vs_0)\ avst'\ h_3\ p)$
definition $s_{13} \equiv (State\ (Vstore\ vs_1)\ avst'\ h_3\ p)$
definition $s_{23} \equiv (State\ (Vstore\ vs_2)\ avst'\ h_3\ p)$
definition $s_{33} \equiv (State\ (Vstore\ vs_{33})\ avst'\ h_3\ p)$

definition $s_{04} \equiv (State\ (Vstore\ vs_0)\ avst'\ h_4\ p)$
definition $s_{14} \equiv (State\ (Vstore\ vs_1)\ avst'\ h_4\ p)$
definition $s_{24} \equiv (State\ (Vstore\ vs_2)\ avst'\ h_4\ p)$
definition $s_{34} \equiv (State\ (Vstore\ vs_{34})\ avst'\ h_4\ p)$

lemmas *s-def* $= s_{03}$*-def* s_{13}*-def* s_{23}*-def* s_{33}*-def*
s_{04}*-def* s_{14}*-def* s_{24}*-def* s_{34}*-def*

definition $(s3_0{::}\ stateO) \equiv (pstate_0,\ (Config\ 0\ s_{03}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{\})$
definition $(s3_1{::}\ stateO) \equiv (pstate_0,\ (Config\ 1\ s_{03}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{\})$
definition $(s3_2{::}\ stateO) \equiv (pstate_0,\ (Config\ 2\ s_{13}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{\})$
definition $(s3_3{::}\ stateO) \equiv (pstate_0,\ (Config\ 3\ s_{23}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{\})$
definition $(s3_4{::}\ stateO) \equiv (pstate_1,\ (Config\ 5\ s_{23}),\ [Config\ 4\ s_{23}],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{\})$
definition $(s3_5{::}\ stateO) \equiv (pstate_1,\ (Config\ 5\ s_{23}),\ [Config\ 5\ s_{33}],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{aa2_{vs3},\ aa1_i\})$
definition $(s3_6{::}\ stateO) \equiv (pstate_2,\ (Config\ 5\ s_{23}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{aa2_{vs3},\ aa1_i\})$
definition $(s3_7{::}\ stateO) \equiv (pstate_2,\ (Config\ 6\ s_{23}),\ [],\ repeat\ (NN{+}1),\ repeat\ (NN{+}1),\ \{aa2_{vs3},\ aa1_i\})$

lemmas *s3-def* $= s3_0$*-def* $s3_1$*-def* $s3_2$*-def* $s3_3$*-def* $s3_4$*-def* $s3_5$*-def* $s3_6$*-def* $s3_7$*-def*

lemmas *state-def* $=$ *s-def h-def vs-def* $reads_m$*-def pstate-def avst-defs*

definition *s3-trans* $\equiv [s3_0,\ s3_1,\ s3_2,\ s3_3,\ s3_4,\ s3_5,\ s3_6,\ s3_7]$
lemmas *s3-trans-defs* $=$ *s3-trans-def s3-def*

lemma *hd-s3-trans*[*simp*]: *hd s3-trans* $= s3_0$ ⟨*proof*⟩

lemma *s3-trans-nemp*[*simp*]: *s3-trans* ≠ [] ⟨*proof*⟩

lemma $s3_{01}$[*simp*]:$s3_0$ →*S* $s3_1$
⟨*proof*⟩

lemma $s3_{12}$[*simp*]:$s3_1$ →*S* $s3_2$
⟨*proof*⟩

lemma $s3_{23}$[*simp*]:$s3_2$ →*S* $s3_3$
⟨*proof*⟩

lemma $s3_{34}$[*simp*]:$s3_3$ →*S* $s3_4$
⟨*proof*⟩

lemma $s3_{45}$[*simp*]:$s3_4$ →*S* $s3_5$
⟨*proof*⟩

lemma $s3_{56}$[*simp*]:$s3_5$ →*S* $s3_6$
⟨*proof*⟩

lemma $s3_{67}$[*simp*]:$s3_6$ →*S* $s3_7$
⟨*proof*⟩

lemma *finalS-*$s3_7$[*simp*]:*finalS* $s3_7$
⟨*proof*⟩

lemmas *s3-trans-simps* = $s3_{01}$ $s3_{12}$ $s3_{23}$ $s3_{34}$ $s3_{45}$ $s3_{56}$ $s3_{67}$

definition ($s4_0$:: *stateO*) ≡ ($pstate_0$, (*Config 0* s_{04}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {})
definition ($s4_1$:: *stateO*) ≡ ($pstate_0$, (*Config 1* s_{04}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {})
definition ($s4_2$:: *stateO*) ≡ ($pstate_0$, (*Config 2* s_{14}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {})
definition ($s4_3$:: *stateO*) ≡ ($pstate_0$, (*Config 3* s_{24}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {})
definition ($s4_4$:: *stateO*) ≡ ($pstate_1$, (*Config 5* s_{24}), [*Config 4* s_{24}], *repeat* (*NN+1*), *repeat* (*NN+1*), {})
definition ($s4_5$:: *stateO*) ≡ ($pstate_1$, (*Config 5* s_{24}), [*Config 5* s_{34}], *repeat* (*NN+1*), *repeat* (*NN+1*), {$aa2_{vs4}$, $aa1_i$})
definition ($s4_6$:: *stateO*) ≡ ($pstate_2$, (*Config 5* s_{24}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {$aa2_{vs4}$, $aa1_i$})
definition ($s4_7$:: *stateO*) ≡ ($pstate_2$, (*Config 6* s_{24}), [], *repeat* (*NN+1*), *repeat* (*NN+1*), {$aa2_{vs4}$, $aa1_i$})

lemmas *s4-def* = $s4_0$-*def* $s4_1$-*def* $s4_2$-*def* $s4_3$-*def* $s4_4$-*def* $s4_5$-*def* $s4_6$-*def* $s4_7$-*def*

definition *s4-trans* ≡ [$s4_0$, $s4_1$, $s4_2$, $s4_3$, $s4_4$, $s4_5$, $s4_6$, $s4_7$]
lemmas *s4-trans-defs* = *s4-trans-def s4-def*

lemma *hd-s4-trans*[*simp*]: *hd s4-trans* = $s4_0$ ⟨*proof*⟩
lemma *s4-trans-nemp*[*simp*]: *s4-trans* ≠ [] ⟨*proof*⟩

lemma $s4_{01}$[*simp*]:$s4_0$ →*S* $s4_1$
⟨*proof*⟩

lemma $s4_{12}$[*simp*]:$s4_1$ →*S* $s4_2$
⟨*proof*⟩

lemma $s4_{24}$[*simp*]:$s4_2$ →*S* $s4_3$
⟨*proof*⟩

lemma $s4_{34}$[*simp*]:$s4_3$ →*S* $s4_4$
⟨*proof*⟩

lemma $s4_{45}$[*simp*]:$s4_4$ →*S* $s4_5$
⟨*proof*⟩

lemma $s4_{56}$[*simp*]:$s4_5$ →*S* $s4_6$
⟨*proof*⟩

lemma $s4_{67}$[*simp*]:$s4_6$ →*S* $s4_7$
⟨*proof*⟩

lemma *finalS-*$s4_7$[*simp*]:*finalS* $s4_7$
⟨*proof*⟩

lemmas *s4-trans-simps* = $s4_{01}$ $s4_{12}$ $s4_{24}$ $s4_{34}$ $s4_{45}$ $s4_{56}$ $s4_{67}$

8.2.2 Auxillary lemmas for disproof

lemma *validS-s3-trans*[*simp*]:*Opt.validS s3-trans*
⟨*proof*⟩

lemma *validS-s4-trans*[*simp*]:*Opt.validS s4-trans*
⟨*proof*⟩

lemma *finalS-s3*[*simp*]:*finalS* (*last s3-trans*) ⟨*proof*⟩

lemma *finalS-s4*[*simp*]:*finalS* (*last s4-trans*) ⟨*proof*⟩

lemma *filter-s3*[*simp*]:(*filter isIntO* (*butlast s3-trans*)) = (*butlast s3-trans*)
⟨*proof*⟩

lemma *filter-s4*[*simp*]:(*filter isIntO* (*butlast s4-trans*)) = (*butlast s4-trans*)
⟨*proof*⟩

lemma *S-s3-trans*[*simp*]:*Opt.S s3-trans* = [s_{03}]
⟨*proof*⟩

lemma *S-s4-trans*[*simp*]:*Opt.S s4-trans* = [s_{04}]
⟨*proof*⟩

lemma *finalB-noStep*[*simp*]:⋀*s1'*. *finalB* (*cfg1*, *ibT1*, *ibUT1*) ⟹ (*cfg1*, *ibT1*, *ibUT1*, *ls1*) →*N s1'* ⟹ *False*
⟨*proof*⟩

8.2.3 Disproof of fun1

fun *common-memory*::*config* ⇒ *config* ⇒ *bool* **where**
common-memory cfg1 cfg2 =
(*let h1* = (*getHheap* (*stateOf cfg1*));
h2 = (*getHheap* (*stateOf cfg2*)) *in*
((∀ *x*∈*read-add*. *h1 x* = *h2 x* ∧ *h1 x* = *0*) ∧
(*getAvstore* (*stateOf cfg1*)) = *avst'* ∧
(*getAvstore* (*stateOf cfg2*)) = *avst'*))

lemma *heap-eq0*[*simp*]: ∀ *x*. *x* ≠ *Suc NN* ⟶ *hh1' x* = *hh2' x* ∧ *hh1' x* = *0* ⟹ *hh2' NN* = *0*
⟨*proof*⟩
lemma *heap1-eq0*[*simp*]:∀ *x*. *x* ≠ *Suc NN* ⟶ *hh1' x* = *hh2' x* ∧ *hh1' x* = *0* ⟹
vs2 xx < *NN* ⟹
hh2' (*nat* (*vs2 xx*)) = *0*
⟨*proof*⟩

fun Γ-*inv*::*stateV* ⇒ *state list* ⇒ *stateV* ⇒ *state list* ⇒ *bool* **where**
Γ-*inv* (*cfg1*,*ibT1*,*ibUT1*,*ls1*) *sl1* (*cfg2*,*ibT2*,*ibUT2*,*ls2*) *sl2* =
(
(*pcOf cfg1* = *pcOf cfg2*) ∧

(*pcOf cfg1* < *2* ⟶ *ibUT1* ≠ *LNil* ∧ *ibUT2* ≠ *LNil*) ∧

(*pcOf cfg1* > *2* ⟶ *same-var-val tt 0 cfg1 cfg2*) ∧

(*pcOf cfg1* > *1* ⟶ (*same-var xx cfg1 cfg2*) ∧

(*vs*$_i$*-t cfg1* ⟶ *pcOf cfg1* ∈ *trueProg*) ∧

$$(vs_i\text{-}f\ cfg1 \longrightarrow pcOf\ cfg1 \in falseProg))$$
$\wedge$
$ls1 = ls2 \wedge$

$pcOf\ cfg1 \in PC \wedge$
$common\text{-}memory\ cfg1\ cfg2$
)

declare Γ-*inv.simps*[*simp del*]
lemmas Γ-*def* = Γ-*inv.simps*
lemmas Γ-*defs* = Γ-*def common-memory.simps PC-def* $aa1_i$*-def*
trueProg-def falseProg-def same-var-val-def same-var-def

lemma Γ-*implies*:Γ-*inv* (*cfg1*,*ibT1*, *ibUT1*,*ls1*) *sl1* (*cfg2*,*ibT2*,*ibUT2*,*ls2*) *sl2* $\Longrightarrow$

$pcOf\ cfg1 \leq 6 \wedge pcOf\ cfg2 \leq 6 \wedge$

$(pcOf\ cfg1 = 4 \longrightarrow vs_i\text{-}t\ cfg1) \wedge$
$(pcOf\ cfg2 = 4 \longrightarrow vs_i\text{-}t\ cfg2) \wedge$
$(pcOf\ cfg1 > 1 \longrightarrow vs_i\text{-}t\ cfg1 \longleftrightarrow vs_i\text{-}t\ cfg2) \wedge$

$(finalB\ (cfg1,ibT1,ibUT1) \longleftrightarrow pcOf\ cfg1 = 6) \wedge$
$(finalB\ (cfg2,ibT2,ibUT2) \longleftrightarrow pcOf\ cfg2 = 6)$

⟨*proof*⟩

lemma *istateO-s3*[*simp*]:*istateO* $s3_0$ ⟨*proof*⟩
lemma *istateO-s4*[*simp*]:*istateO* $s4_0$ ⟨*proof*⟩

lemma *validFromS-s3*[*simp*]:*Opt.validFromS* $s3_0$ *s3-trans*
⟨*proof*⟩

lemma *validFromS-s4*[*simp*]:*Opt.validFromS* $s4_0$ *s4-trans*
⟨*proof*⟩

lemma *completedFromO-s3*[*simp*]:*completedFromO* $s3_0$ *s3-trans*
⟨*proof*⟩

lemma *completedFromO-s4*[*simp*]:*completedFromO* $s4_0$ *s4-trans*
⟨*proof*⟩

lemma *Act-eq*[*simp*]:*Opt.A s3-trans* = *Opt.A s4-trans*
⟨*proof*⟩

lemma *aa2-neq*:$aa2_{vs3} \neq aa2_{vs4}$
⟨*proof*⟩

lemma *aa1-neq*:$aa2_{vs3} \neq aa1_i$
⟨*proof*⟩

lemma *aa1-neq2*:$aa2_{vs4} \neq aa1_i$
⟨*proof*⟩

lemma *Obs-neq*[*simp*]:*Opt.O s3-trans* ≠ *Opt.O s4-trans*
⟨*proof*⟩

lemma Γ-*init*[*simp*]:⋀*s1 s2*. *istateV s1* ⟹ *corrState s1* $s3_0$ ⟹ *istateV s2* ⟹ *corrState s2* $s4_0$ ⟹ Γ-*inv s1* [s_{03}] *s2* [s_{04}]
⟨*proof*⟩

lemma *val-neq-1*:*nat* (*hh2'* (*nat* (*vs2 xx*)) ∗ *512*) ≠ *1*
⟨*proof*⟩

lemma *unwindSD*[*simp*]:*Rel-Sec.unwindSDCond validTransV istateV isSecV getSecV isIntV getIntV* Γ-*inv*
⟨*proof*⟩

theorem ¬*rsecure*
⟨*proof*⟩

end

9 Proof of Relative Security for fun2

theory *Fun2*
imports
../Instance-IMP/Instance-Secret-IMem
Relative-Security.Unwinding-fin
begin

9.1 Function definition and Boilerplate

no-notation *bot* (⊥)

consts *NN* :: *nat*
lemma *NN*: *NN* ≥ *0* ⟨*proof*⟩

definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''
definition *xx* :: *avname* **where** *xx* = ''*xx*''
definition *tt* :: *avname* **where** *tt* = ''*tt*''

lemmas *vvars-defs* = *aa1-def aa2-def xx-def tt-def*

lemma *vvars-dff*[*simp*]:
aa1 ≠ *aa2 aa1* ≠ *xx aa1* ≠ *tt*
aa2 ≠ *aa1 aa2* ≠ *xx aa2* ≠ *tt*
xx ≠ *aa1 xx* ≠ *aa2 xx* ≠ *tt*
tt ≠ *aa1 tt* ≠ *aa2 tt* ≠ *xx*
⟨*proof*⟩

consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*

lemma *aa1*: *size-aa1* ≥ *0* **and** *aa2*:*size-aa2* ≥ *0* ⟨*proof*⟩

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore (*Avstore as*) = (*as aa1* = (*0*, *nat size-aa1*) ∧ *as aa2* = (*nat size-aa1*, *nat size-aa2*))

fun *istate* ::*state* ⇒ *bool* **where**
istate s = (*initAvstore* (*getAvstore s*))

definition *prog* ≡
[
0// *Start* ,
1// *Input U xx* ,
2// *tt* ::= (*N 0*) ,
3// *IfJump* (*Less* (*V xx*) (*N NN*)) *4 6* ,
4// *Fence* ,
5// *tt* ::= (*VA aa2* (*Times* (*VA aa1* (*V xx*)) (*N 512*))),
6// *Output U* (*V tt*)
]

lemma *cases-6*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨ *i* = *6* ∨ *i* > *6*
⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < *int*(*NN*) ∨ *vs xx* ≥ *int*(*NN*) ⟨*proof*⟩

lemma *is-If-pcOf*[*simp*]:
pcOf cfg < *6* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3*
⟨*proof*⟩

lemma *is-If-pc*[*simp*]:
pc < *6* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3*
⟨*proof*⟩

lemma *eq-Fence-pc*[*simp*]:
pc < *6* ⟹ *prog* ! *pc* = *Fence* ⟷ *pc* = *4*
⟨*proof*⟩

consts *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc = (*if* (*set pc* = {*6*,*4*}) *then True else False*)

consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts *initPstate* :: *predState*

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = *initPstate* **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* *55*)
where *x* →*B* *y* == *stepB x y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ *55*)
where *x* →*B*∗ *y* == *star stepB x y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*M* *55*)
where *x* →*M* *y* == *stepM x y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* 55)
where *x* →*N y* == *stepN x y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ 55)
where *x* →*N*∗ *y* == *star stepN x y*

abbreviation
stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* 55)
where *x* →*S y* == *stepS x y*

abbreviation
stepsS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S*∗ 55)
where *x* →*S*∗ *y* == *star stepS x y*

lemma *endPC*[*simp*]: *endPC* = 7
⟨*proof*⟩

lemma *is-getUntrustedInput-pcOf*[*simp*]: *pcOf cfg* < 6 ⟹ *is-getInput* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = 1
⟨*proof*⟩

lemma *start*[*simp*]:*prog* ! 0 = *Start*
⟨*proof*⟩

lemma *getUntrustedInput-pcOf*[*simp*]: *prog*!1 = *Input U xx*
⟨*proof*⟩

lemma *if-stat*[*simp*]: *prog*! 3 = (*IfJump* (*Less* (*V xx*) (*N NN*)) 4 6)
⟨*proof*⟩

lemma *isOutput1*[*simp*]:*prog* ! 6 = *Output U* (*V tt*)
⟨*proof*⟩

lemma *is-Output-pcOf*[*simp*]: *pcOf cfg* < 6 ⟹ *is-Output* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = 6
⟨*proof*⟩

lemma *is-Fence-pcOf*[*simp*]: *pcOf cfg* < 6 ⟹(*prog*!(*pcOf cfg*)) = *Fence* ⟷ *pcOf*

$cfg = 4$
⟨*proof*⟩

lemma *is-Output*[*simp*]: *is-Output* (*prog* ! 6)
⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*, *ibUT*) ⟷ *pcOf cfg* = 0
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*, *ibUT*,*ls*) ⟷ (*pcOf cfg* = 0 ∧ *cfgs* = [])
⟨*proof*⟩

lemma *getInputT-not*[*simp*]: *pcOf cfg* < 7 ⟹
(*prog* ! *pcOf cfg*) ≠ *Input T inp*
⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < 7 ⟹
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = 1 *then lhd ibUT else* ⊥)
⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < 7 ⟹
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = 6 *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < 7 ⟹
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = 1 ∧ *cfgs* = [] *then lhd ibUT else* ⊥)
⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < 7 ⟹
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*, *ibUT*,*ls*) =
(*if* (*pcOf cfg* = 6 ∧ *cfgs* = []) *then*

(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *eqSec-pcOf*[*simp*]:
eqSec (*cfg1*, *ibT*, *ibUT1*, *ls1*) (*pstate3*, *cfg3*, *cfgs3*, *ibT*, *ibUT3*, *ls3*) ⟷
(*pcOf cfg1* = *0* ⟷ *pcOf cfg3* = *0* ∧ *cfgs3* = []) ∧
(*pcOf cfg1* = *0* ⟶ *stateOf cfg1* = *stateOf cfg3*)
⟨*proof*⟩

lemma *nextB-pc0*[*simp*]:
nextB (*Config 0 s*, *ibT*, *ibUT*) =
(*Config 1 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc0'*[*simp*]:*nextB* (*Config 0* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
=
(*Config* (*Suc 0*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc0*[*simp*]:
readLocs (*Config 0 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config 1* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1*[*simp*]:
readLocs (*Config 1 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1'*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config* (*Suc 0*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
=
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1'*[*simp*]:
readLocs (*Config* (*Suc 0*) *s*) = {}
⟨*proof*⟩

lemma *nextB-pc2*[*simp*]:
nextB (*Config 2* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 3* (*State* (*Vstore* (*vs*(*tt* := *0*))) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc2*[*simp*]:
readLocs (*Config 2 s*) = {}
⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx < *NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx ≥ *NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *NN then 4 else 6*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx ≥ *NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx < *NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =

(*Config* (*if vs xx* < *NN then 6 else 4*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:
readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
nextB (*Config 4 s*, *ibT*, *ibUT*) = (*Config 5 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:
readLocs (*Config 4 s*) = {}
⟨*proof*⟩

lemma *nextB-pc5*[*simp*]:
nextB (*Config 5* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*, *ibUT*) =
(*let l* = (*array-loc aa2* (*nat* (*h* (*array-loc aa1* (*nat* (*vs xx*)) *avst*) ∗ *512*)) *avst*)
in (*Config 6* (*State* (*Vstore* (*vs*(*tt* := *h l*))) *avst* (*Heap h*) *p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
readLocs (*Config 5* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*)) =
{*array-loc aa2* (*nat* (*h* (*array-loc aa1* (*nat* (*vs xx*)) *avst*) ∗ *512*)) *avst*, *array-loc aa1* (*nat* (*vs xx*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc6*[*simp*]:
nextB (*Config 6 s*, *ibT*, *ibUT*) = (*Config 7 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc6*[*simp*]:
readLocs (*Config 6* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*)) =
{}
⟨*proof*⟩

lemma *nextB-stepB-pc*:
pc < *7* ⟹ (*pc* = *1* ⟶ *ibUT* ≠ *LNil*) ⟹
(*Config pc s*, *ibT*, *ibUT*) →*B nextB* (*Config pc s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *not-finalB*:
$pc < 7 \Longrightarrow (pc = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow$
$\neg$ *finalB* (*Config pc s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *finalB-pc-iff'*:
$pc < 7 \Longrightarrow$
finalB (*Config pc s*, *ibT*, *ibUT*) $\longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil)$
⟨*proof*⟩

lemma *finalB-pc-iff*:
$pc \leq 7 \Longrightarrow$
finalB (*Config pc s*, *ibT*, *ibUT*) $\longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil \vee pc = 7)$
⟨*proof*⟩

lemma *finalB-pcOf-iff*[*simp*]:
$pcOf\ cfg \leq 7 \Longrightarrow$
finalB (*cfg*, *ibT*, *ibUT*) $\longleftrightarrow$ $(pcOf\ cfg = 1 \wedge ibUT = LNil \vee pcOf\ cfg = 7)$
⟨*proof*⟩

lemma *finalS-cond*:$pcOf\ cfg < 7 \Longrightarrow cfgs = [] \Longrightarrow (pcOf\ cfg = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow \neg$ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-cond-spec*:
$pcOf\ cfg < 7 \Longrightarrow$
$(pcOf\ (last\ cfgs) = 4 \wedge pcOf\ cfg = 6) \vee (pcOf\ (last\ cfgs) = 6 \wedge pcOf\ cfg = 4) \Longrightarrow$
$length\ cfgs = Suc\ 0 \Longrightarrow$
$\neg$ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

end

9.2 Proof

theory *Fun2-secure*
imports *Fun2*
begin

definition $PC \equiv \{0..6\}$

definition *same-xx cfg3 cfgs3 cfg4 cfgs4* ≡
vstore (*getVstore* (*stateOf cfg3*)) *xx* = *vstore* (*getVstore* (*stateOf cfg4*)) *xx* ∧
(∀ *cfg3'*∈*set cfgs3*. *vstore* (*getVstore* (*stateOf cfg3'*)) *xx* = *vstore* (*getVstore* (*stateOf cfg3*)) *xx*) ∧
(∀ *cfg4'*∈*set cfgs4*. *vstore* (*getVstore* (*stateOf cfg4'*)) *xx* = *vstore* (*getVstore* (*stateOf cfg4*)) *xx*)

definition *beforeInput* = {*0*,*1*}
definition *afterInput* = {*2*,*3*,*4*,*5*,*6*}
definition *inThenBranch* = {*4*,*5*,*6*}
definition *startOfThenBranch* = *4*
definition *elseBranch* = *6*

definition *common* :: *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common = (λ
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧
cfg1 = *cfg3* ∧ *cfg2* = *cfg4* ∧
pcOf cfg3 = *pcOf cfg4* ∧ *map pcOf cfgs3* = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ' (*set cfgs3*) ⊆ *PC* ∧
///
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧
///
(*statA* = *Diff* ⟶ *statO* = *Diff*)))

lemma *common-implies*: *common* (*pstate3*,*cfg3*,*cfgs3*,*ibT*, *ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT*, *ibUT4*,*ls4*)

statA
(*cfg1*,*ibT*, *ibUT1*,*ls1*)
(*cfg2*,*ibT*, *ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg1 $<$ *8* $\wedge$ *pcOf cfg2* $=$ *pcOf cfg1*
⟨*proof*⟩

definition $\Delta 0$ :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
$\Delta 0$ = (λ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*, *ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*, *ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*, *ibUT1*,*ls1*)
(*cfg2*,*ibT2*, *ibUT2*,*ls2*)
statO.
(*common* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*, *ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*, *ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*, *ibUT1*,*ls1*)
(*cfg2*,*ibT2*, *ibUT2*,*ls2*)
statO $\wedge$
ibUT1 $=$ *ibUT3* $\wedge$ *ibUT2* $=$ *ibUT4* $\wedge$
(*pcOf cfg3* $>$ *1* $\longrightarrow$ *same-xx cfg3 cfgs3 cfg4 cfgs4*) $\wedge$
(*pcOf cfg3* $<$ *2* $\longrightarrow$ *ibUT1*$\neq$*LNil* $\wedge$ *ibUT2*$\neq$*LNil* $\wedge$ *ibUT3*$\neq$*LNil* $\wedge$ *ibUT4*$\neq$*LNil*)
$\wedge$
ls1 $=$ *ls3* $\wedge$ *ls2* $=$ *ls4* $\wedge$
pcOf cfg3 $\in$ *beforeInput* $\wedge$
noMisSpec cfgs3
))

lemmas $\Delta 0$-*defs* $=$ $\Delta 0$-*def common-def PC-def*
beforeInput-def
same-xx-def noMisSpec-def

lemma $\Delta 0$-*implies*: $\Delta 0$ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
(*pcOf cfg3* $=$ *1* $\longrightarrow$ *ibUT3* $\neq$ *LNil*) $\wedge$
(*pcOf cfg4* $=$ *1* $\longrightarrow$ *ibUT4* $\neq$ *LNil*) $\wedge$
pcOf cfg1 $<$ *7* $\wedge$ *pcOf cfg2* $=$ *pcOf cfg1* $\wedge$
cfgs3 $=$ [] $\wedge$ *pcOf cfg3* $<$ *7* $\wedge$
cfgs4 $=$ [] $\wedge$ *pcOf cfg4* $<$ *7*
⟨*proof*⟩

definition $\Delta 1$ :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status*
$\Rightarrow$ *bool* **where**
$\Delta 1$ = (λ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*, *ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*, *ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*, *ibUT1*,*ls1*)
(*cfg2*,*ibT2*, *ibUT2*,*ls2*)
statO.
(*common* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
ls1 = *ls3* $\wedge$ *ls2* = *ls4* $\wedge$
same-xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$
pcOf cfg3 $\in$ *afterInput* $\wedge$
noMisSpec cfgs3
))

lemmas Δ*1-defs* = Δ*1-def common-def PC-def afterInput-def noMisSpec-def same-xx-def*

lemma Δ*1-implies*: $\Delta 1$ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg1 < 7 $\wedge$
cfgs3 = [] $\wedge$ *pcOf cfg3* $\neq$ 1 $\wedge$ *pcOf cfg3* < 7 $\wedge$
cfgs4 = [] $\wedge$ *pcOf cfg4* $\neq$ 1 $\wedge$ *pcOf cfg4* < 7
$\langle$*proof*$\rangle$

definition $\Delta 2$ :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status*
$\Rightarrow$ *bool* **where**
$\Delta 2$ = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)

statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧
ls1 = *ls3* ∧ *ls2* = *ls4* ∧
same-xx cfg3 cfgs3 cfg4 cfgs4 ∧
pcOf cfg3 = *startOfThenBranch* ∧
pcOf (*last cfgs3*) = *elseBranch* ∧
misSpecL1 cfgs3
))

lemmas Δ*2-defs* = Δ*2-def common-def PC-def same-xx-def inThenBranch-def elseBranch-def startOfThenBranch-def misSpecL1-def same-xx-def*

lemma Δ*2-implies*: Δ*2 num* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf (*last cfgs3*) = *6* ∧ *pcOf cfg3* = *4* ∧
pcOf (*last cfgs4*) = *pcOf* (*last cfgs3*) ∧
pcOf cfg3 = *pcOf cfg4* ∧
length cfgs3 = *Suc 0* ∧
length cfgs3 = *length cfgs4*
⟨*proof*⟩

definition Δ*3* :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
Δ*3* = (λ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧
ls1 = *ls3* ∧ *ls2* = *ls4* ∧
pcOf cfg3 = *elseBranch* ∧
pcOf (*last cfgs3*) = *startOfThenBranch* ∧
same-xx cfg3 cfgs3 cfg4 cfgs4 ∧
misSpecL1 cfgs3

))

lemmas $\Delta 3\text{-}defs = \Delta 3\text{-}def\ common\text{-}def\ PC\text{-}def\ same\text{-}xx\text{-}def\ elseBranch\text{-}def\ startOfThen\text{-}Branch\text{-}def$
$misSpecL1\text{-}def\ same\text{-}xx\text{-}def$

lemma $\Delta 3\text{-}implies$: $\Delta 3\ num$
$(pstate3,cfg3,cfgs3,ibT3,\ ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,\ ibUT4,ls4)$
$statA$
$(cfg1,ibT1,\ ibUT1,ls1)$
$(cfg2,ibT2,\ ibUT2,ls2)$
$statO \Longrightarrow$
$pcOf\ (last\ cfgs3) = 4 \wedge pcOf\ cfg3 = 6 \wedge$
$pcOf\ (last\ cfgs4) = pcOf\ (last\ cfgs3) \wedge$
$pcOf\ cfg3 = pcOf\ cfg4 \wedge$
$array\text{-}base\ aa1\ (getAvstore\ (stateOf\ (last\ cfgs3))) = array\text{-}base\ aa1\ (getAvstore\ (stateOf\ cfg3)) \wedge$
$array\text{-}base\ aa1\ (getAvstore\ (stateOf\ (last\ cfgs4))) = array\text{-}base\ aa1\ (getAvstore\ (stateOf\ cfg4)) \wedge$
$length\ cfgs3 = Suc\ 0 \wedge$
$length\ cfgs3 = length\ cfgs4$
⟨*proof*⟩

definition $\Delta 4 :: enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV \Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 4 = (\lambda num$
$(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO.$
$(pcOf\ cfg3 = endPC \wedge pcOf\ cfg4 = endPC \wedge cfgs3 = [] \wedge cfgs4 = [] \wedge$
$pcOf\ cfg1 = endPC \wedge pcOf\ cfg2 = endPC))$

lemmas $\Delta 4\text{-}defs = \Delta 4\text{-}def\ common\text{-}def\ endPC\text{-}def$

lemma $init$: $initCond\ \Delta 0$
⟨*proof*⟩

lemma $step0$: $unwindIntoCond\ \Delta 0\ (oor\ \Delta 0\ \Delta 1)$
⟨*proof*⟩

lemma *step1*: *unwindIntoCond* $\Delta 1$ (*oor4* $\Delta 1$ $\Delta 2$ $\Delta 3$ $\Delta 4$)
⟨*proof*⟩

lemma *step2*: *unwindIntoCond* $\Delta 2$ $\Delta 1$
⟨*proof*⟩

lemma *step3*: *unwindIntoCond* $\Delta 3$ (*oor* $\Delta 3$ $\Delta 1$)
⟨*proof*⟩

lemma *stepe*: *unwindIntoCond* $\Delta 4$ $\Delta 4$
⟨*proof*⟩

lemmas *theConds* = *step0 step1 step2 step3 stepe*

proposition *rsecure*
⟨*proof*⟩

end

10 Proof of Relative Security for fun3

theory *Fun3*
imports *../Instance-IMP/Instance-Secret-IMem*
Relative-Security.Unwinding-fin
begin

10.1 Function definition and Boilerplate

no-notation *bot* (⊥)

consts *NN*::*nat*

lemma *NN*:*int* *NN* ≥ *0* ⟨*proof*⟩
consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*
consts *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts *initPstate* :: *predState*

definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''

definition *vv* :: *avname* **where** *vv* = ''*v*''
definition *xx* :: *avname* **where** *xx* = ''*x*''
definition *tt* :: *avname* **where** *tt* = ''*t*''

lemmas *vvars-defs* = *aa1-def aa2-def vv-def xx-def tt-def*

lemma *vvars-dff*[*simp*]:
aa1 ≠ *aa2 aa1* ≠ *vv aa1* ≠ *xx aa1* ≠ *tt*
aa2 ≠ *aa1 aa2* ≠ *vv aa2* ≠ *xx aa2* ≠ *tt*
vv ≠ *aa1 vv* ≠ *aa2 vv* ≠ *xx vv* ≠ *tt*
xx ≠ *aa1 xx* ≠ *aa2 xx* ≠ *vv xx* ≠ *tt*
tt ≠ *aa1 tt* ≠ *aa2 tt* ≠ *vv tt* ≠ *xx*
⟨*proof*⟩

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore (*Avstore as*) = (*as aa1* = (*0*, *size-aa1*) ∧ *as aa2* = (*size-aa1*, *size-aa2*))
fun *istate* ::*state* ⇒ *bool* **where**
istate s = (*initAvstore* (*getAvstore s*))

definition *prog* ≡
[
0// *Start* ,
1// *Input U xx* ,
2// *tt* ::= (*N 0*) ,
3// *IfJump* (*Less* (*V xx*) (*N NN*)) *4 7* ,
4// *vv* ::= *VA aa1* (*V xx*) ,
5// *Fence* ,
6// *tt* ::= (*VA aa2* (*Times* (*V vv*) (*N 512*))) ,
7// *Output U* (*V tt*)
]

lemma *cases-7*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨
i = *6* ∨ *i* = *7* ∨ *i* > *7*
⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < *int NN* ∨ *vs xx* ≥ *int NN* ⟨*proof*⟩

lemma *is-If-pcOf*[*simp*]:
pcOf cfg < *8* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3*
⟨*proof*⟩

lemma *is-If-pc*[*simp*]:
pc < *8* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3*
⟨*proof*⟩

lemma *eq-Fence-pc*[*simp*]:
$pc < 8 \Longrightarrow prog\ !\ pc = Fence \longleftrightarrow pc = 5$
⟨*proof*⟩

fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc = (*if* (*pc* = [4,7]) *then True else False*)

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = *initPstate* **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* 55)
where *x* →*B y* == *stepB x y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ 55)
where *x* →*B*∗ *y* == *star stepB x y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*M* 55)
where *x* →*M y* == *stepM x y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* 55)
where *x* →*N y* == *stepN x y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ 55)
where *x* →*N*∗ *y* == *star stepN x y*

abbreviation
stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* 55)

where $x \rightarrow S\ y == stepS\ x\ y$

abbreviation
$stepsS\text{-}abbrev :: configS \Rightarrow configS \Rightarrow bool$ (**infix** $\rightarrow S*\ 55$)
where $x \rightarrow S*\ y == star\ stepS\ x\ y$

lemma $endPC[simp]: endPC = 8$
⟨*proof*⟩

lemma $is\text{-}getTrustedInput\text{-}pcOf[simp]: pcOf\ cfg < 8 \Longrightarrow is\text{-}getInput\ (prog!(pcOf\ cfg)) \longleftrightarrow pcOf\ cfg = 1$
⟨*proof*⟩
lemma $getUntrustedInput\text{-}pcOf[simp]: prog!1 = Input\ U\ xx$
⟨*proof*⟩

lemma $getInput\text{-}not3[simp]: \neg is\text{-}getInput\ (prog\ !\ 3)$
⟨*proof*⟩

lemma $getInput\text{-}not4[simp]: \neg is\text{-}getInput\ (prog\ !\ 4)$
⟨*proof*⟩
lemma $Output\text{-}not4[simp]: \neg is\text{-}Output\ (prog\ !\ 4)$
⟨*proof*⟩

lemma $is\text{-}Output\text{-}pcOf[simp]: pcOf\ cfg < 8 \Longrightarrow is\text{-}Output\ (prog!(pcOf\ cfg)) \longleftrightarrow pcOf\ cfg = 7$
⟨*proof*⟩

lemma $is\text{-}Output: is\text{-}Output\ (prog\ !\ 7)$
⟨*proof*⟩

lemma $is\text{-}Fence[simp]: (prog\ !\ 5) = Fence$
⟨*proof*⟩

lemma $not\text{-}is\text{-}getTrustedInput[simp]: cfg = Config\ 3\ (State\ (Vstore\ vs)\ (Avstore\ as)\ (Heap\ h)\ p) \Longrightarrow \neg\ is\text{-}getInput\ (prog\ !\ pcOf\ cfg)$
⟨*proof*⟩

lemma $not\text{-}is\text{-}Output[simp]: cfg = Config\ pc\ (State\ (Vstore\ vs)\ (Avstore\ as)\ (Heap\ h)\ p) \Longrightarrow$
$pc = 3 \Longrightarrow \neg\ is\text{-}Output\ (prog\ !\ pcOf\ cfg)$
⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*,*ibUT*) $\longleftrightarrow$ *pcOf cfg* = *0*
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) $\longleftrightarrow$ (*pcOf cfg* = *0* $\wedge$ *cfgs* = [])
⟨*proof*⟩

lemma *getInputT-not*[*simp*]: *pcOf cfg* < *8* $\Longrightarrow$
(*prog* ! *pcOf cfg*) $\neq$ *Input T inp*
⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < *8* $\Longrightarrow$
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *1 then lhd ibUT else* $\perp$)
⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < *8* $\Longrightarrow$
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *7 then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else $\perp$
)
⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < *8* $\Longrightarrow$
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *1* $\wedge$ *cfgs* = [] *then lhd ibUT else* $\perp$)
⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < *8* $\Longrightarrow$
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if* (*pcOf cfg* = *7* $\wedge$ *cfgs* = []) *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else $\perp$
)
⟨*proof*⟩

lemma *eqSec-pcOf*[*simp*]:
eqSec (*cfg1*, *ibT*,*ibUT1*, *ls1*) (*pstate3*, *cfg3*, *cfgs3*, *ibT*,*ibUT3*, *ls3*) $\longleftrightarrow$

$(pcOf\ cfg1 = 0 \longleftrightarrow pcOf\ cfg3 = 0 \wedge cfgs3 = []) \wedge$
$(pcOf\ cfg1 = 0 \longrightarrow stateOf\ cfg1 = stateOf\ cfg3)$
⟨*proof*⟩

lemma *nextB-pc0*[*simp*]:
$nextB\ (Config\ 0\ s,\ ibT,ibUT) =$
$(Config\ 1\ s,\ ibT,ibUT)$
⟨*proof*⟩

lemma *readLocs-pc0*[*simp*]:
$readLocs\ (Config\ 0\ s) = \{\}$
⟨*proof*⟩

lemma *nextB-pc1*[*simp*]:
$ibUT \neq LNil \Longrightarrow nextB\ (Config\ 1\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,ibUT) =$
$(Config\ 2\ (State\ (Vstore\ (vs(xx := lhd\ ibUT)))\ avst\ h\ p),\ ibT,\ ltl\ ibUT)$
⟨*proof*⟩

lemma *readLocs-pc1*[*simp*]:
$readLocs\ (Config\ 1\ s) = \{\}$
⟨*proof*⟩

lemma *nextB-pc1'*[*simp*]:
$ibUT \neq LNil \Longrightarrow nextB\ (Config\ (Suc\ 0)\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,ibUT)$
$=$
$(Config\ 2\ (State\ (Vstore\ (vs(xx := lhd\ ibUT)))\ avst\ h\ p),\ ibT,\ ltl\ ibUT)$
⟨*proof*⟩

lemma *readLocs-pc1'*[*simp*]:
$readLocs\ (Config\ (Suc\ 0)\ s) = \{\}$
⟨*proof*⟩

lemma *nextB-pc2*[*simp*]:
$nextB\ (Config\ 2\ (State\ (Vstore\ vs)\ avst\ h\ p),\ ibT,\ ibUT) =$
$(Config\ 3\ (State\ (Vstore\ (vs(tt := 0)))\ avst\ h\ p),\ ibT,\ ibUT)$
⟨*proof*⟩

lemma *readLocs-pc2*[*simp*]:
$readLocs\ (Config\ 2\ s) = \{\}$
⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx < *int NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx ≥ *int NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 7* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *NN then 4 else 7*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx ≥ *int NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx < *int NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 7* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *NN then 7 else 4*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:
readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
nextB (*Config 4* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa1* (*nat* (*vs xx*)) *avst*
in (*Config 5* (*State* (*Vstore* (*vs*(*vv* := *h l*))) *avst* (*Heap h*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:

readLocs (*Config 4* (*State* (*Vstore vs*) *avst h p*)) = {*array-loc aa1* (*nat* (*vs xx*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc5*[*simp*]:
nextB (*Config 5 s*, *ibT*,*ibUT*) = (*Config 6 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
readLocs (*Config 5 s*) = {}
⟨*proof*⟩

lemma *nextB-pc6*[*simp*]:
nextB (*Config 6* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*
in (*Config 7* (*State* (*Vstore* (*vs*(*tt* := *h l*))) *avst* (*Heap h*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc6*[*simp*]:
readLocs (*Config 6* (*State* (*Vstore vs*) *avst h p*)) = {*array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc7*[*simp*]:
nextB (*Config 7 s*, *ibT*,*ibUT*) = (*Config 8 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc7*[*simp*]:
readLocs (*Config 7 s*) = {}
⟨*proof*⟩

lemma *nextB-stepB-pc*:
pc < *8* ⟹ (*pc* = *1* ⟶ *ibUT* ≠ *LNil*) ⟹
(*Config pc s*, *ibT*,*ibUT*) →*B nextB* (*Config pc s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *not-finalB*:
pc < *8* ⟹ (*pc* = *1* ⟶ *ibUT* ≠ *LNil*) ⟹
¬ *finalB* (*Config pc s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *finalB-pc-iff'*:
$pc < 8 \Longrightarrow$
$finalB\ (Config\ pc\ s,\ ibT,ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil)$
$\langle proof \rangle$

lemma *finalB-pc-iff*:
$pc \leq 8 \Longrightarrow$
$finalB\ (Config\ pc\ s,\ ibT,ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil \vee pc = 8)$
$\langle proof \rangle$

lemma *finalB-pcOf-iff*[*simp*]:
$pcOf\ cfg \leq 8 \Longrightarrow$
$finalB\ (cfg,\ ibT,ibUT) \longleftrightarrow (pcOf\ cfg = 1 \wedge ibUT = LNil \vee pcOf\ cfg = 8)$
$\langle proof \rangle$

lemma *finalS-cond*:$pcOf\ cfg < 8 \Longrightarrow cfgs = [] \Longrightarrow (pcOf\ cfg = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow \neg\ finalS\ (pstate,\ cfg,\ cfgs,\ ibT,ibUT,\ ls)$
$\langle proof \rangle$

lemma *finalS-cond-spec*:
$pcOf\ cfg < 8 \Longrightarrow$
$(((pcOf\ (last\ cfgs) = 4 \vee pcOf\ (last\ cfgs) = 5) \wedge pcOf\ cfg = 7) \vee$
$(pcOf\ (last\ cfgs) = 7 \wedge pcOf\ cfg = 4)) \Longrightarrow$
$length\ cfgs = Suc\ 0 \Longrightarrow$
$\neg\ finalS\ (pstate,\ cfg,\ cfgs,\ ibT,ibUT,\ ls)$
$\langle proof \rangle$
end

10.2 Proof

theory *Fun3-secure*
imports *Fun3*
begin

type-synonym $stateO = configS$
type-synonym $stateV = config \times val\ llist \times val\ llist \times loc\ set$

definition $PC \equiv \{0..7\}$

definition $beforeInput = \{0,1\}$

definition *afterInput* = {*2*,*3*,*4*,*5*,*6*,*7*}
definition *startOfThenBranch* = *4*
definition *inThenBranchBeforeFence* = {*4*,*5*}
definition *elseBranch* = *7*
definition *beforeFence* = {*2*..*4*}
definition *beforeAssign-vv* = {*0*..*4*}

definition *common* :: *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common = (λ
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧
cfg1 = *cfg3* ∧ *cfg2* = *cfg4* ∧
pcOf cfg3 = *pcOf cfg4* ∧ *map pcOf cfgs3* = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ' (*set cfgs3*) ⊆ *PC* ∧
///
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧
///
(*statA* = *Diff* ⟶ *statO* = *Diff*)))

lemma *common-implies*: *common*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < *9* ∧ *pcOf cfg2* = *pcOf cfg1*
⟨*proof*⟩

definition $\Delta 0$ :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status*
$\Rightarrow$ *bool* **where**
$\Delta 0$ = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
ibUT1 = *ibUT3* $\wedge$ *ibUT2* = *ibUT4* $\wedge$
(*pcOf cfg3* > 1 $\longrightarrow$ *same-var-o xx cfg3 cfgs3 cfg4 cfgs4*) $\wedge$
(*pcOf cfg3* < 2 $\longrightarrow$ *ibUT1*$\neq$*LNil* $\wedge$ *ibUT2*$\neq$*LNil* $\wedge$ *ibUT3*$\neq$*LNil* $\wedge$ *ibUT4*$\neq$*LNil*)
$\wedge$
pcOf cfg3 $\in$ *beforeInput* $\wedge$
ls1 = *ls3* $\wedge$ *ls2* = *ls4* $\wedge$
noMisSpec cfgs3
))

lemmas Δ*0-defs* = Δ*0-def common-def PC-def beforeInput-def noMisSpec-def same-var-o-def*

lemma Δ*0-implies*: Δ*0 num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
(*pcOf cfg3* = 1 $\longrightarrow$ *ibUT3* $\neq$ *LNil*) $\wedge$
(*pcOf cfg4* = 1 $\longrightarrow$ *ibUT4* $\neq$ *LNil*) $\wedge$
pcOf cfg1 < 8 $\wedge$ *pcOf cfg2* = *pcOf cfg1* $\wedge$
cfgs3 = [] $\wedge$ *pcOf cfg3* < 8 $\wedge$
cfgs4 = [] $\wedge$ *pcOf cfg4* < 8
$\langle$*proof*$\rangle$

definition Δ*1* :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status*
$\Rightarrow$ *bool* **where**
Δ*1* = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA

(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧
pcOf cfg3 ∈ *afterInput* ∧
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
ls1 = *ls3* ∧ *ls2* = *ls4* ∧
noMisSpec cfgs3
))

lemmas Δ*1-defs* = Δ*1-def common-def PC-def afterInput-def same-var-o-def noMis-Spec-def*

lemma Δ*1-implies*: Δ*1 num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < *8* ∧
cfgs3 = [] ∧ *pcOf cfg3* ≠ *1* ∧ *pcOf cfg3* < *8* ∧
cfgs4 = [] ∧ *pcOf cfg4* ≠ *1* ∧ *pcOf cfg4* < *8*
⟨*proof*⟩

definition Δ*2* :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
Δ*2* = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧

pcOf cfg3 = startOfThenBranch ∧
pcOf (last cfgs3) = elseBranch ∧
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
ls1 = ls3 ∧ ls2 = ls4 ∧
misSpecL1 cfgs3
))

lemmas *Δ2-defs = Δ2-def common-def PC-def same-var-def startOfThenBranch-def*

misSpecL1-def elseBranch-def

lemma *Δ2-implies*: *Δ2 num*
(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO ⟹
pcOf (last cfgs3) = 7 ∧ pcOf cfg3 = 4 ∧
pcOf (last cfgs4) = pcOf (last cfgs3) ∧
pcOf cfg3 = pcOf cfg4 ∧
length cfgs3 = Suc 0 ∧
length cfgs3 = length cfgs4
⟨*proof*⟩

definition *Δ3* :: *enat ⇒ stateO ⇒ stateO ⇒ status ⇒ stateV ⇒ stateV ⇒ status ⇒ bool* **where**
Δ3 = (λnum
(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO.
(common (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO ∧
pcOf cfg3 = elseBranch ∧
pcOf (last cfgs3) ∈ inThenBranchBeforeFence ∧
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
Language-Prelims.dist ls3 ls4 ⊆ Language-Prelims.dist ls1 ls2 ∧
(pcOf (last cfgs3) = 4 ⟶ ls1 = ls3 ∧ ls2 = ls4) ∧
misSpecL1 cfgs3
))

lemmas $\Delta 3\text{-}defs = \Delta 3\text{-}def\ common\text{-}def\ PC\text{-}def\ inThenBranchBeforeFence\text{-}def$
$beforeAssign\text{-}vv\text{-}def\ misSpecL1\text{-}def\ elseBranch\text{-}def$
$same\text{-}var\text{-}o\text{-}def$

lemma $\Delta 3\text{-}implies$: $\Delta 3\ num$
$(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \Longrightarrow$
$(pcOf\ (last\ cfgs3) = 4 \vee pcOf\ (last\ cfgs3) = 5) \wedge pcOf\ cfg3 = 7 \wedge$
$pcOf\ (last\ cfgs4) = pcOf\ (last\ cfgs3) \wedge$
$pcOf\ cfg3 = pcOf\ cfg4 \wedge$
$array\text{-}base\ aa1\ (getAvstore\ (stateOf\ (last\ cfgs3))) = array\text{-}base\ aa1\ (getAvstore\ (stateOf\ cfg3)) \wedge$
$array\text{-}base\ aa1\ (getAvstore\ (stateOf\ (last\ cfgs4))) = array\text{-}base\ aa1\ (getAvstore\ (stateOf\ cfg4)) \wedge$
$length\ cfgs3 = Suc\ 0 \wedge$
$length\ cfgs3 = length\ cfgs4 \wedge$
$vstore\ (getVstore\ (stateOf\ (last\ cfgs3)))\ xx = vstore\ (getVstore\ (stateOf\ (last\ cfgs4)))\ xx$
⟨*proof*⟩

definition $\Delta 1' :: enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV \Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 1' = (\lambda num$
$(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO.$
$(common$
$(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \wedge$
$pcOf\ cfg3 = elseBranch \wedge$
$same\text{-}var\text{-}o\ xx\ cfg3\ cfgs3\ cfg4\ cfgs4 \wedge$
$Language\text{-}Prelims.dist\ ls3\ ls4 \subseteq Language\text{-}Prelims.dist\ ls1\ ls2 \wedge$
$noMisSpec\ cfgs3$
$))$

lemmas $\Delta 1'$*-defs* = $\Delta 1'$*-def common-def PC-def afterInput-def same-var-o-def noMisSpec-def elseBranch-def*

lemma $\Delta 1'$*-implies*: $\Delta 1'$ *num*
(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO $\Longrightarrow$
pcOf cfg1 $<$ *8* $\wedge$
cfgs3 $=$ [] $\wedge$ *pcOf cfg3* $\neq$ *1* $\wedge$ *pcOf cfg3* $<$ *8* $\wedge$
cfgs4 $=$ [] $\wedge$ *pcOf cfg4* $\neq$ *1* $\wedge$ *pcOf cfg4* $<$ *8*
⟨*proof*⟩

definition $\Delta 4$ :: *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
$\Delta 4$ = (λ*num*
(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO.
(*pcOf cfg3* $=$ *endPC* $\wedge$ *pcOf cfg4* $=$ *endPC* $\wedge$ *cfgs3* $=$ [] $\wedge$ *cfgs4* $=$ [] $\wedge$
pcOf cfg1 $=$ *endPC* $\wedge$ *pcOf cfg2* $=$ *endPC*))

lemmas $\Delta 4$*-defs* = $\Delta 4$*-def common-def endPC-def*

lemma *init*: *initCond* $\Delta 0$
⟨*proof*⟩

lemma *step0*: *unwindIntoCond* $\Delta 0$ (*oor* $\Delta 0$ $\Delta 1$)
⟨*proof*⟩

lemma *step1*: *unwindIntoCond* $\Delta 1$ (*oor4* $\Delta 1$ $\Delta 2$ $\Delta 3$ $\Delta 4$)
⟨*proof*⟩

lemma *step2*: *unwindIntoCond* $\Delta 2$ $\Delta 1$
⟨*proof*⟩

lemma *step3*: *unwindIntoCond* $\Delta 3$ (*oor* $\Delta 3$ $\Delta 1'$)
⟨*proof*⟩

lemma *step1'*: *unwindIntoCond* $\Delta 1'$ $\Delta 4$
⟨*proof*⟩

lemma *stepe*: *unwindIntoCond* $\Delta 4$ $\Delta 4$
⟨*proof*⟩

lemmas *theConds* = *step0 step1 step2 step3 step1' stepe*

proposition *rsecure*
⟨*proof*⟩
end

11 Proof of Relative Security for fun4

theory *Fun4*
imports *../Instance-IMP/Instance-Secret-IMem*
Relative-Security.Unwinding-fin
begin

11.1 Function definition and Boilerplate

no-notation *bot* (⊥)

consts *NN* :: *nat*
consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*
lemma *NN*: *int NN* ≥ *0* ⟨*proof*⟩

locale *array-nempty* = **assumes** *aa1*:*size-aa1* > *0* **and** *NN*: *int NN* > *0*

definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''
definition *vv* :: *avname* **where** *vv* = ''*v*''
definition *xx* :: *avname* **where** *xx* = ''*i*''
definition *tt* :: *avname* **where** *tt* = ''*w*''

lemmas *vvars-defs = aa1-def aa2-def vv-def xx-def tt-def*

lemma *vvars-dff*[*simp*]:
aa1 ≠ aa2 aa1 ≠ vv aa1 ≠ xx aa1 ≠ tt
aa2 ≠ aa1 aa2 ≠ vv aa2 ≠ xx aa1 ≠ tt
vv ≠ aa1 vv ≠ aa2 vv ≠ xx vv ≠ tt
xx ≠ aa1 xx ≠ aa2 xx ≠ vv xx ≠ tt
tt ≠ aa1 tt ≠ aa2 tt ≠ vv tt ≠ xx
⟨*proof*⟩

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore (*Avstore as*) = (*as aa1* = (*0*, *size-aa1*) ∧ *as aa2* = (*size-aa1*, *size-aa2*))

fun *istate* ::*state* ⇒ *bool* **where**
istate s = (*initAvstore* (*getAvstore s*))

definition *prog* ≡
[
0// *Start* ,
1// *Input U xx* ,
2// *tt* ::= (*N 0*) ,
3// *IfJump* (*Less* (*V xx*) (*N NN*)) *4 6* ,
4// *vv* ::= *VA aa1* (*N 0*) ,
5// *tt* ::= *Plus* (*VA aa2* (*Times* (*V vv*) (*N 512*))) (*V xx*) ,
6// *Output U* (*V tt*)
]

lemma *cases-6*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨ *i* = *6* ∨ *i* > *6*
⟨*proof*⟩

lemma *cases-thenBranch*: (*i*::*pcounter*) < *4* ∨ *i* = *4* ∨ *i* = *5* ∨ *i* = *6* ∨ *i* > *6*
⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < *int NN* ∨ *vs xx* ≥ *int NN* ⟨*proof*⟩

lemma *is-If-pcOf*[*simp*]:
pcOf cfg < *7* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3*
⟨*proof*⟩

lemma *is-If-pc*[*simp*]:

pc < *7* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3*
⟨*proof*⟩

lemma *is-If-pcThen*[*simp*]: *pcOf cfg* ∈ {*4*..*6*} ⟹ ¬*is-IfJump* (*prog* ! *pcOf cfg*)
⟨*proof*⟩

consts *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc = (*if* (*pc* = [*6*,*6*] ∨ *pc* = [*4*,*6*]) *then True else False*)

consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts *initPstate* :: *predState*

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = *initPstate* **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* *55*)
where *x* →*B* *y* == *stepB x y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ *55*)
where *x* →*B*∗ *y* == *star stepB x y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*M* *55*)
where *x* →*M* *y* == *stepM x y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* *55*)
where *x* →*N* *y* == *stepN x y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ *55*)
where *x* →*N*∗ *y* == *star stepN x y*

abbreviation
stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* *55*)
where *x* →*S* *y* == *stepS x y*

abbreviation
stepsS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S*∗ *55*)
where *x* →*S*∗ *y* == *star stepS x y*

lemma *endPC*[*simp*]: *endPC* = *7*
⟨*proof*⟩

lemma *is-getUntrustedInput-pcOf*[*simp*]: *pcOf cfg* < *7* ⟹ *is-getInput* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = *1*
⟨*proof*⟩

lemma *getUntrustedInput-pcOf*[*simp*]: *prog*!*1* = *Input U xx*
⟨*proof*⟩

lemma *is-getTrustedInput*[*simp*]: *is-getInput* (*prog* ! *1*)
⟨*proof*⟩

lemma *getInput-not4*[*simp*]:¬*is-getInput* (*prog* ! *4*)
⟨*proof*⟩
lemma *getInput-not5*[*simp*]:¬*is-getInput* (*prog* ! *5*)
⟨*proof*⟩
lemma *OutputT-not6*[*simp*]:(*prog* ! *6*) = *Output U* (*V tt*)
⟨*proof*⟩

lemma *is-Output-pcOf*[*simp*]: *pcOf cfg* < *7* ⟹ *is-Output* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = *6*
⟨*proof*⟩

lemma *is-Fence-pcOf*[*simp*]:*pcOf cfg* < *7* ⟹ *prog* ! (*pcOf cfg*) ≠ *Fence*
⟨*proof*⟩

lemma *is-Fence-pcThen*[*simp*]: *3* ≤ *pcOf cfg* ∧ *pcOf cfg* ≤ *5* ⟹ (*prog* ! *pcOf cfg*)≠*Fence*
⟨*proof*⟩

lemma *is-Output*[*simp*]: *is-Output* (*prog* ! *6*)
⟨*proof*⟩

lemma *getInput-not*[*intro*]:*is-getInput* (*prog* ! *4*) ⟹ *False* ⟨*proof*⟩

lemma *Output-not4*[*intro*]:*is-Output* (*prog* ! *4*) $\Longrightarrow$ *False* ⟨*proof*⟩
lemma *Fence-not4*[*intro*]:*prog* ! *4* = *Fence* $\Longrightarrow$ *False* ⟨*proof*⟩

lemma *getInput-not55*[*intro*]:*is-getInput* (*prog* ! *5*) $\Longrightarrow$ *False* ⟨*proof*⟩
lemma *Output-not5*[*intro*]:*is-Output* (*prog* ! *5*) $\Longrightarrow$ *False* ⟨*proof*⟩
lemma *Fence-not5*[*intro*]:*prog* ! *5* = *Fence* $\Longrightarrow$ *False* ⟨*proof*⟩

lemma *Jump-not6*:¬ *is-IfJump* (*prog* ! *6*)⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*,*ibUT*) $\longleftrightarrow$ *pcOf cfg* = *0*
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) $\longleftrightarrow$ (*pcOf cfg* = *0* ∧ *cfgs* = []⟩
⟨*proof*⟩

lemma *inputT-not*[*simp*]: *pcOf cfg* < *7* $\Longrightarrow$
(*prog* ! *pcOf cfg*) ≠ *Input T inp*
⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < *7* $\Longrightarrow$
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *1 then lhd ibUT else* ⊥)
⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < *7* $\Longrightarrow$
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *6 then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *getObsV-pcOf6*[*simp*]:
pcOf cfg = *6* $\Longrightarrow$
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)

⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < *7* $\Longrightarrow$
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =

(*if pcOf cfg = 1 ∧ cfgs = [] then lhd ibUT else ⊥*)
⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < 7 ⟹
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if* (*pcOf cfg = 6 ∧ cfgs = []*) *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *eqSec-pcOf*[*simp*]:
eqSec (*cfg1*, *ibT*, *ibUT1*, *ls1*) (*pstate3*, *cfg3*, *cfgs3*, *ibT3*, *ibUT3*, *ls3*) ⟷
(*pcOf cfg1 = 0* ⟷ *pcOf cfg3 = 0 ∧ cfgs3 = []*) ∧
(*pcOf cfg1 = 0* ⟶ *stateOf cfg1 = stateOf cfg3*)
⟨*proof*⟩

lemma *nextB-pc0*[*simp*]:
nextB (*Config 0 s*, *ibT*,*ibUT*) =
(*Config 1 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc0*[*simp*]:
readLocs (*Config 0 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1*[*simp*]:
ibUT ≠ LNil ⟹ nextB (*Config 1* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1*[*simp*]:
readLocs (*Config 1 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1′*[*simp*]:
ibUT ≠ LNil ⟹ nextB (*Config* (*Suc 0*) (*State* (*Vstore vs*) *avst h p*), *ibT*,*ibUT*)
=
(*Config 2* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst h p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1′*[*simp*]:

readLocs (*Config* (*Suc 0*) *s*) = {}
⟨*proof*⟩

lemma *nextB-pc2*[*simp*]:
nextB (*Config 2* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 3* (*State* (*Vstore* (*vs*(*tt* := *0*))) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc2*[*simp*]:
readLocs (*Config 2 s*) = {}
⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx < *int NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx ≥ *int NN* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *int NN then 4 else 6*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx ≥ *int NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx < *int NN* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *int NN then 6 else 4*) (*State* (*Vstore vs*) *avst h p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:

readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
nextB (*Config 4* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa1 0 avst*
in (*Config 5* (*State* (*Vstore* (*vs*(*vv* := *h l*))) *avst* (*Heap h*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:
readLocs (*Config 4* (*State* (*Vstore vs*) *avst h p*)) = {*array-loc aa1 0 avst*}
⟨*proof*⟩

lemma *nextB-pc5*[*simp*]:
nextB (*Config 5* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*
in (*Config 6* (*State* (*Vstore* (*vs*(*tt* := *h l* + *vs xx*))) *avst* (*Heap h*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
readLocs (*Config 5* (*State* (*Vstore vs*) *avst h p*)) = {*array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc6*[*simp*]:
nextB (*Config 6 s*, *ibT*,*ibUT*) = (*Config 7 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc6*[*simp*]:
readLocs (*Config 6* (*State* (*Vstore vs*) *avst h p*)) = {}
⟨*proof*⟩

lemma *nextB-stepB-pc*:
pc < *7* ⟹ (*pc* = *1* ⟶ *ibUT* ≠ *LNil*) ⟹
(*Config pc s*, *ibT*,*ibUT*) →*B nextB* (*Config pc s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-avst-consistent-aux*:
4 ≤ *pc* ∧ *pc* ≤ *6* ⟹
(*nextB* (*Config pc* (*State* (*Vstore vs*) *avst* (*Heap h*) *p*),*ibT*,*ibUT*)) = (*Config pc′*

$(State\ (Vstore\ vs')\ avst'\ (Heap\ h')\ p', ibT, ibUT') \Longrightarrow$
$avst = avst' \wedge$
$vs\ xx = vs'\ xx \wedge$
$h = h'$
⟨*proof*⟩

lemma *nextB-avst-consistent*:
$4 \leq pcOf\ cfg \wedge pcOf\ cfg \leq 6 \Longrightarrow$
$(nextB\ (cfg, ibT, ibUT)) = (cfg', ibT, ibUT') \Longrightarrow$
$(getAvstore\ (stateOf\ cfg)) = (getAvstore\ (stateOf\ cfg')) \wedge$
$(getHheap\ (stateOf\ cfg)) = (getHheap\ (stateOf\ cfg')) \wedge$
$vstore\ (getVstore\ (stateOf\ cfg))\ xx = vstore\ (getVstore\ (stateOf\ cfg'))\ xx$
⟨*proof*⟩

lemma *nextB-pcs-consistent*:
$4 \leq pcOf\ cfg1 \wedge pcOf\ cfg1 \leq 6 \Longrightarrow pcOf\ cfg1 = pcOf\ cfg2 \Longrightarrow$
$(nextB\ (cfg1, ibT1, ibUT1)) = (cfg1', ibT1', ibUT1') \Longrightarrow$
$(nextB\ (cfg2, ibT2, ibUT2)) = (cfg2', ibT2', ibUT2') \Longrightarrow$
$pcOf\ cfg1' = pcOf\ cfg2'$
⟨*proof*⟩

lemma *not-finalB*:
$pc < 7 \Longrightarrow (pc = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow$
$\neg\ finalB\ (Config\ pc\ s, ibT, ibUT)$
⟨*proof*⟩

lemma *finalB-pc-iff'*:
$pc < 7 \Longrightarrow$
$finalB\ (Config\ pc\ s, ibT, ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil)$
⟨*proof*⟩

lemma *finalB-pc-iff*:
$pc \leq 7 \Longrightarrow$
$finalB\ (Config\ pc\ s, ibT, ibUT) \longleftrightarrow$
$(pc = 1 \wedge ibUT = LNil \vee pc = 7)$
⟨*proof*⟩

lemma *finalB-pcOf-iff*[*simp*]:
$pcOf\ cfg \leq 7 \Longrightarrow$
$finalB\ (cfg, ibT, ibUT) \longleftrightarrow (pcOf\ cfg = 1 \wedge ibUT = LNil \vee pcOf\ cfg = 7)$
⟨*proof*⟩

lemma *finalS-cond*:$pcOf\ cfg < 7 \Longrightarrow cfgs = [] \Longrightarrow (pcOf\ cfg = 1 \longrightarrow ibUT \neq LNil) \Longrightarrow \neg\ finalS\ (pstate, cfg, cfgs, ibT, ibUT, ls)$

⟨*proof*⟩

lemma *finalS-cond-spec*:
pcOf cfg < *7* ⟹
((*pcOf* (*last cfgs*) = *4* ∨ *pcOf* (*last cfgs*) = *5* ∨ *pcOf* (*last cfgs*) = *6*) ∧ *pcOf cfg* = *6*) ∨ (*pcOf* (*last cfgs*) = *6* ∧ *pcOf cfg* = *4*) ⟹
length cfgs = *Suc 0* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

end

11.2 Proof

theory *Fun4-secure*
imports *Fun4*
begin

definition *PC* ≡ {*0..6*}

definition *same-xx-cp cfg1 cfg2* ≡
vstore (*getVstore* (*stateOf cfg1*)) *xx* = *vstore* (*getVstore* (*stateOf cfg2*)) *xx*
∧ *vstore* (*getVstore* (*stateOf cfg1*)) *xx* = *0*

definition *common-memory cfg cfg′ cfgs′* ≡
array-base aa1 (*getAvstore* (*stateOf cfg*)) = *array-base aa1* (*getAvstore* (*stateOf cfg′*)) ∧
(∀ *cfg′′*∈*set cfgs′*. *array-base aa1* (*getAvstore* (*stateOf cfg′′*)) = *array-base aa1* (*getAvstore* (*stateOf cfg*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg*)) = *array-base aa2* (*getAvstore* (*stateOf cfg′*)) ∧
(∀ *cfg′′*∈*set cfgs′*. *array-base aa2* (*getAvstore* (*stateOf cfg′′*)) = *array-base aa2* (*getAvstore* (*stateOf cfg*))) ∧
(*getHheap* (*stateOf cfg*)) = (*getHheap* (*stateOf cfg′*)) ∧
(∀ *cfg′′*∈*set cfgs′*. *getHheap* (*stateOf cfg*) = (*getHheap* (*stateOf cfg′′*))) ∧
(*getAvstore* (*stateOf cfg*)) = (*getAvstore* (*stateOf cfg′*))

definition *beforeInput* = {*0*,*1*}
definition *afterInput* = {*2..6*}
definition *elseBranch* = *6*
definition *startOfThenBranch* = *4*
definition *inThenBranch* = {*4..6*}

definition *afterInputNotInElse* = {*2*,*3*,*4*,*5*,*6*,*8*}
definition *inThenBranchBeforeOutput* = {*3*,*4*,*5*}
definition *atCond* = *3*
definition *atThenOutput* = *5*
definition *atJump* = *6*

definition *common-strat1* :: *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common-strat1 =
(λ (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧
cfg1 = *cfg3* ∧ *cfg2* = *cfg4* ∧
pcOf cfg3 = *pcOf cfg4* ∧ *map pcOf cfgs3* = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ‘ (*set cfgs3*) ⊆ *PC* ∧

try and try to have common memory
common-memory cfg1 cfg3 cfgs3 ∧

likewise try and try
common-memory cfg2 cfg4 cfgs4 ∧

(∀ *n*≥*0*. *array-loc aa1 0* (*getAvstore* (*stateOf cfg2*)) ≠ *array-loc aa2 n* (*getAvstore* (*stateOf cfg2*)) ∧
array-loc aa1 0 (*getAvstore* (*stateOf cfg1*)) ≠ *array-loc aa2 n* (*getAvstore* (*stateOf cfg1*))) ∧
///
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3*′∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3*′)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4*′∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4*′)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3*′∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3*′)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4*′∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4*′)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧
///
(*statA* = *Diff* ⟶ *statO* = *Diff*)))

lemmas *common-strat1-defs* = *common-strat1-def common-memory-def*

definition *common* :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common = (λ(*num*::*enat*)
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧

(*num* = (*endPC* − *pcOf cfg1*) ∨ *num* = ∞) ∧

//tr1 and tr2 stay on the same path//
pcOf cfg1 = *pcOf cfg2* ∧

//likewise tr3 and tr4//
pcOf cfg3 = *pcOf cfg4* ∧
map pcOf cfgs3 = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ' (*set cfgs3*) ⊆ *PC* ∧
pcOf cfg1 ∈ *PC* ∧

//tr1 and tr3 have common memory//
common-memory cfg1 cfg3 cfgs3 ∧

//likewise tr2 and tr4//
common-memory cfg2 cfg4 cfgs4 ∧

(∀ *n*≥*0*. *array-loc aa1 0* (*getAvstore* (*stateOf cfg2*)) ≠ *array-loc aa2 n* (*getAvstore* (*stateOf cfg2*)) ∧
array-loc aa1 0 (*getAvstore* (*stateOf cfg1*)) ≠ *array-loc aa2 n* (*getAvstore* (*stateOf cfg1*))) ∧
//All traces have same base addresses Avo1: Maybe this is also worth being extracted as a predicate//
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧

$(statA = Diff \longrightarrow statO = Diff)$
))

lemmas *common-defs = common-def common-memory-def*

lemma *common-implies*: *common num*
$(pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
statA
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \Longrightarrow$
$pcOf\ cfg1 < 9 \wedge pcOf\ cfg3 < 9 \wedge$

$(n \geq 0 \longrightarrow array\text{-}loc\ aa1\ 0\ (getAvstore\ (stateOf\ cfg2)) \neq array\text{-}loc\ aa2\ n\ (getAvstore\ (stateOf\ cfg2)) \wedge$
$array\text{-}loc\ aa1\ 0\ (getAvstore\ (stateOf\ cfg1)) \neq array\text{-}loc\ aa2\ n\ (getAvstore\ (stateOf\ cfg1)))$
⟨*proof*⟩

definition $\Delta 0$ :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
$\Delta 0 = (\lambda num\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
statA
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
statO.
$(common\ num\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
statA
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
statO ∧

head of the buffers are equal component-wise
$(llength\ ibUT1 = \infty \wedge llength\ ibUT2 = \infty \wedge$
$llength\ ibUT3 = \infty \wedge llength\ ibUT4 = \infty) \wedge$
$(lhd\ ibUT3 \geq NN \wedge (lhd\ ibUT1 = 0) \wedge ibUT1 = ibUT2$
$\vee lhd\ ibUT3 < NN \wedge ibUT1 = ibUT3 \wedge ibUT2 = ibUT4) \wedge$
$pcOf\ cfg3 \in beforeInput \wedge$

since memory is equal, cfg1 = cfg3 and cfg2 = cfg4 in "loading" state
$cfg1 = cfg3 \wedge cfg2 = cfg4 \wedge$
$ls1 = ls3 \wedge ls2 = ls4 \wedge$
$ls1 = \{\} \wedge ls2 = \{\} \wedge$

noMisSpec cfgs3
))
lemmas *Δ0-defs′ = Δ0-def common-defs PC-def beforeInput-def noMisSpec-def*

lemma *Δ0-def2*:
Δ0 num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO
=
(common num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO ∧

//and if the buffers are equal componentwise//
(llength ibUT1 = ∞ ∧ llength ibUT2 = ∞ ∧
llength ibUT3 = ∞ ∧ llength ibUT4 = ∞) ∧
(ibUT1 ≠ [[]] ∧ ibUT2 ≠ [[]] ∧ ibUT3 ≠ [[]] ∧ ibUT4 ≠ [[]]) ∧
(lhd ibUT3 ≥ NN ∧ (lhd ibUT1 = 0) ∧ ibUT1 = ibUT2
∨lhd ibUT3 < NN ∧ ibUT1 = ibUT3 ∧ ibUT2 = ibUT4) ∧
pcOf cfg3 ∈ beforeInput ∧

//if memory is equal, cfg1#cfg3 and cfg2#cfg4 in Loading//state//
cfg1 = cfg3 ∧ cfg2 = cfg4 ∧
ls1 = ls3 ∧ ls2 = ls4 ∧
ls1 ={} ∧ ls2={} ∧
noMisSpec cfgs3
)
⟨*proof*⟩

lemmas *Δ0-defs = Δ0-def2 common-defs PC-def beforeInput-def noMisSpec-def*

lemma *Δ0-implies*: *Δ0 num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)*
(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
statA
(cfg1,ibT1,ibUT1,ls1)
(cfg2,ibT2,ibUT2,ls2)
statO ⟹
(pcOf cfg3 = 1 ⟶ ibUT3 ≠ LNil) ∧
(pcOf cfg4 = 1 ⟶ ibUT4 ≠ LNil) ∧
pcOf cfg1 < 7 ∧ pcOf cfg2 = pcOf cfg1 ∧
cfgs3 = [] ∧ pcOf cfg3 < 7 ∧
cfgs4 = [] ∧ pcOf cfg4 < 7
⟨*proof*⟩

definition $\Delta 1$ :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
$\Delta 1$ = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common-strat1* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧
pcOf cfg3 ∈ *afterInput* ∧
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
vstore (*getVstore* (*stateOf cfg3*)) *xx* < *NN* ∧

ls1 = *ls3* ∧ *ls2* = *ls4* ∧
noMisSpec cfgs3
))

lemmas $\Delta 1$-*defs* = $\Delta 1$-*def common-strat1-defs PC-def afterInput-def same-var-o-def noMisSpec-def*

lemma $\Delta 1$-*implies*: $\Delta 1$ *num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < 7 ∧
cfgs3 = [] ∧ *pcOf cfg3* ≠ 1 ∧ *pcOf cfg3* < 7 ∧
cfgs4 = [] ∧ *pcOf cfg4* ≠ 1 ∧ *pcOf cfg4* < 7
⟨*proof*⟩

definition $\Delta 2$ :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
$\Delta 2$ = (λ*num*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)

```
   (cfg2,ibT2,ibUT2,ls2)
     statO.
 (common-strat1
   (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
   (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
   statA
   (cfg1,ibT1,ibUT1,ls1)
   (cfg2,ibT2,ibUT2,ls2)
     statO ∧
  pcOf cfg3 = startOfThenBranch ∧
  pcOf cfg1 = pcOf cfg3 ∧

  pcOf (last cfgs3) = elseBranch ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
  vstore (getVstore (stateOf cfg3)) xx < NN ∧
  ls1 = ls3 ∧ ls2 = ls4 ∧
  misSpecL1 cfgs3
))
```

lemmas Δ*2-defs* = Δ*2-def common-strat1-defs PC-def same-var-def startOfThen-Branch-def*
misSpecL1-def elseBranch-def

lemma Δ*2-implies*: Δ*2 num*
```
   (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
   (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
   statA
   (cfg1,ibT1,ibUT1,ls1)
   (cfg2,ibT2,ibUT2,ls2)
   statO ⟹
   pcOf (last cfgs3) = 6 ∧ pcOf cfg3 = 4 ∧
   pcOf (last cfgs4) = pcOf (last cfgs3) ∧
   pcOf cfg3 = pcOf cfg4 ∧
   length cfgs3 = Suc 0 ∧
   length cfgs3 = length cfgs4
  ⟨proof⟩
```

definition Δ*1′* :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
```
  Δ1′ = (λnum (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
     (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
     statA
     (cfg1,ibT1,ibUT1,ls1)
     (cfg2,ibT2,ibUT2,ls2)
     statO.
```

```
 (common num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO ∧
 ///
  pcOf cfg3 ∈ afterInput ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧

  (pcOf cfg1 > 2 ⟶ vstore (getVstore (stateOf cfg3)) tt = vstore (getVstore
(stateOf cfg4)) tt) ∧

  vstore (getVstore (stateOf cfg3)) xx ≥ NN ∧

  (pcOf cfg1 < 4 ⟶ pcOf cfg1 = pcOf cfg3 ∧
                   ls1 = {} ∧ ls2 = {} ∧
                   ls1 = ls3 ∧ ls2 = ls4) ∧
  (pcOf cfg1 ≤ 5 ⟶ ls1 ⊆ {array-loc aa1 0 (getAvstore (stateOf cfg1))}
                ∧ ls1 = ls2 ∧ ls3 = ls4) ∧

  (Language-Prelims.dist ls3 ls4 ⊆ Language-Prelims.dist ls1 ls2) ∧

  (pcOf cfg1 ≥ 4 ⟶ pcOf cfg1 ∈ inThenBranch ∧ pcOf cfg3 = elseBranch) ∧
  same-xx-cp cfg1 cfg2 ∧
  vstore (getVstore (stateOf cfg1)) xx = 0 ∧

  ls3 ⊆ ls1 ∧ ls4 ⊆ ls2 ∧
  noMisSpec cfgs3
))
lemmas Δ1'-defs = Δ1'-def common-defs PC-def afterInput-def
  same-var-o-def same-xx-cp-def noMisSpec-def inThenBranch-def elseBranch-def
lemma Δ1'-implies: Δ1' num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
   statO ⟹
   pcOf cfg1 < 7 ∧ pcOf cfg1 ≠ Suc 0 ∧
   pcOf cfg2 = pcOf cfg1 ∧
   cfgs3 = [] ∧ pcOf cfg3 < 7 ∧
   cfgs4 = [] ∧ pcOf cfg4 < 7
  ⟨proof⟩


definition Δ3' :: enat ⇒ stateO ⇒ stateO ⇒ status ⇒ stateV ⇒ stateV ⇒ status
⇒ bool where
  Δ3' = (λ num (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
```

statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common num* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ∧
///
pcOf cfg3 = *elseBranch* ∧ *cfgs3* ≠ [] ∧
pcOf (*last cfgs3*) ∈ *inThenBranch* ∧
pcOf (*last cfgs4*) = *pcOf*(*last cfgs3*) ∧
we want the cp traces with the speculation cfgs3 cfgs4
pcOf cfg1 = *pcOf*(*last cfgs3*) ∧

same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
(*getAvstore* (*stateOf cfg3*)) = (*getAvstore* (*stateOf* (*last cfgs3*))) ∧
(*getAvstore* (*stateOf cfg4*)) = (*getAvstore* (*stateOf* (*last cfgs4*))) ∧

same-xx-cp cfg1 cfg2 ∧
ls1 = *ls3* ∧ *ls2* = *ls4* ∧

vstore (*getVstore* (*stateOf cfg3*)) *tt* = *vstore* (*getVstore* (*stateOf cfg4*)) *tt* ∧

vstore (*getVstore* (*stateOf cfg3*)) *xx* ≥ *NN* ∧

(*pcOf cfg1* = *4* ⟶ *ls1* = {} ∧ *ls2* = {}) ∧
(*pcOf cfg1* ≤ *5* ⟶ *ls1* ⊆ {*array-loc aa1 0* (*getAvstore* (*stateOf cfg1*))}
∧ *ls2* ⊆ {*array-loc aa1 0* (*getAvstore* (*stateOf cfg2*))}
∧ *ls3* = *ls4*) ∧

(*pcOf cfg1* > *4* ⟶*same-var vv cfg1* (*last cfgs3*) ∧ *same-var vv cfg2* (*last cfgs4*))
∧
misSpecL1 cfgs3
))
lemmas Δ*3′-defs* = Δ*3′-def common-defs PC-def elseBranch-def*
inThenBranch-def startOfThenBranch-def
same-var-o-def same-xx-cp-def misSpecL1-def same-var-def

lemma Δ*3′-implies*: Δ*3′ num* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < *7* ∧ *pcOf cfg1* ≠ *Suc 0* ∧

pcOf cfg2 = pcOf cfg1 ∧
pcOf cfg3 < 7 ∧ pcOf cfg4 < 7 ∧
(pcOf (last cfgs3) = 4 ∨ pcOf (last cfgs3) = 5 ∨ pcOf (last cfgs3) = 6) ∧ pcOf cfg3 = 6
⟨*proof*⟩

definition Δ*e* :: *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**
Δ*e* = (λ(*num*::*enat*) (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
((*num* = (*endPC* − *pcOf cfg1*) ∨ *num* = ∞) ∧
pcOf cfg3 = *endPC* ∧ *pcOf cfg4* = *endPC* ∧ *cfgs3* = [] ∧ *cfgs4* = [] ∧
pcOf cfg1 = *endPC* ∧ *pcOf cfg2* = *endPC*))

lemmas Δ*e-defs* = Δ*e-def common-def endPC*

context *array-nempty*
begin
lemma *init*: *initCond* Δ*0*
⟨*proof*⟩

lemma *step0*: *unwindIntoCond* Δ*0* (*oor3* Δ*0* Δ*1* Δ*1'*)
⟨*proof*⟩

lemma *step1*: *unwindIntoCond* Δ*1* (*oor3* Δ*1* Δ*2* Δ*e*)
⟨*proof*⟩

lemma *step2*: *unwindIntoCond* Δ*2* Δ*1*
⟨*proof*⟩

lemma *xx-le-NN*[*simp*]:*cfg* = *Config pc* (*State* (*Vstore vs*) *avst h p*) ⟹ *vs xx* = *0* ⟹ *vs xx* < *int NN*
⟨*proof*⟩

lemma *match12I*:*match12* (*oor3* Δ*1'* Δ*3'* Δ*e*) *ss3 ss4 statA ss1 ss2 statO* ⟹
(∃ *v*<*n*. *proact* (*oor3* Δ*1'* Δ*3'* Δ*e*) *v ss3 ss4 statA ss1 ss2 statO*) ∨

match (*oor3* $\Delta 1'$ $\Delta 3'$ Δe) *ss3 ss4 statA ss1 ss2 statO*
⟨*proof*⟩

lemma *step1'*: *unwindIntoCond* $\Delta 1'$ (*oor3* $\Delta 1'$ $\Delta 3'$ Δe)
⟨*proof*⟩

lemma *step3'*: *unwindIntoCond* $\Delta 3'$ (*oor* $\Delta 3'$ $\Delta 1'$)
⟨*proof*⟩

lemma *stepe*: *unwindIntoCond* Δe Δe
⟨*proof*⟩

lemmas *theConds* = *step0 step1 step2*
step1' step3' stepe

proposition *rsecure*
⟨*proof*⟩
end
end

12 Proof of Relative Security for fun5

theory *Fun5*
imports *../Instance-IMP/Instance-Secret-IMem*
Relative-Security.Unwinding
begin

12.1 Function definition and Boilerplate

no-notation *bot* (⊥)
consts *NN* :: *nat*
consts *SS* :: *nat*
lemma *NN*: *int NN* ≥ *0* **and** *SS*: *int SS*≥*0* ⟨*proof*⟩

definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''
definition *vv* :: *avname* **where** *vv* = ''*v*''
definition *xx* :: *avname* **where** *xx* = ''*x*''
definition *tt* :: *avname* **where** *tt* = ''*y*''
definition *temp* :: *avname* **where** *temp* = ''*temp*''

lemmas *vvars-defs* = *aa1-def aa2-def vv-def xx-def tt-def temp-def*

lemma *vvars-dff*[*simp*]:
aa1 ≠ *aa2 aa1* ≠ *vv aa1* ≠ *xx aa1* ≠ *temp aa1* ≠ *tt*

$aa2 \neq aa1\ aa2 \neq vv\ aa2 \neq xx\ aa2 \neq temp\ aa2 \neq tt$
$vv \neq aa1\ vv \neq aa2\ \ vv \neq xx\ vv \neq temp\ vv \neq tt$
$xx \neq aa1\ xx \neq aa2\ xx \neq vv\ xx \neq temp\ xx \neq tt$
$tt \neq aa1\ tt \neq aa2\ tt \neq vv\ tt \neq temp\ tt \neq xx$
$temp \neq aa1\ temp \neq aa2\ temp \neq vv\ temp \neq xx\ temp \neq tt$
$\langle proof \rangle$

consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore (*Avstore as*) = (*as aa1* = (*0*, *size-aa1*) ∧ *as aa2* = (*size-aa1*, *size-aa2*))

fun *istate* ::*state* ⇒ *bool* **where**
istate s = (*initAvstore* (*getAvstore s*))

definition *prog* ≡
[
0// *Start* ,
1// *tt* ::= (*N 0*),
2// *xx* ::= (*N 1*),
3// *IfJump* (*Not* (*Eq* (*V xx*) (*N 0*))) *4 11* ,
4// *Input U xx* ,
5// *IfJump* (*Less* (*V xx*) (*N NN*)) *6 10* ,
6// *vv* ::= *VA aa1* (*V xx*) ,
7// *Fence* ,
8// *tt* ::= (*VA aa2* (*Times* (*V vv*) (*N SS*))) ,
9// *Output U* (*V tt*) ,
10// *Jump 3*,
11// *Output U* (*N 0*)
]

definition *PC* ≡ {*0..11*}

definition *beforeWhile* = {*0*,*1*,*2*}
definition *inWhile* = {*3..11*}
definition *startOfWhileThen* = *4*
definition *startOfIfThen* = *6*
definition *inThenIfBeforeFence* = {*6*,*7*}
definition *startOfElseBranch* = *10*
definition *inElseIf* = {*10*,*3*,*4*,*11*}
definition *whileElse* = *11*

fun *leftWhileSpec* **where**
leftWhileSpec cfg cfg′ =
(*pcOf cfg* = *whileElse* ∧
pcOf cfg′ = *startOfWhileThen*)

fun *rightWhileSpec* **where**
rightWhileSpec cfg cfg′ =
(*pcOf cfg* = *startOfWhileThen* ∧
pcOf cfg′ = *whileElse*)

fun *whileSpeculation* **where**
whileSpeculation cfg cfg′ =
(*leftWhileSpec cfg cfg′* ∨
rightWhileSpec cfg cfg′)
lemmas *whileSpec-def* = *whileSpeculation.simps*
startOfWhileThen-def
whileElse-def

lemmas *whileSpec-defs* = *whileSpec-def*
leftWhileSpec.simps
rightWhileSpec.simps

lemma *cases-12*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨
i = *6* ∨ *i* = *7* ∨ *i* = *8* ∨ *i* = *9* ∨ *i* = *10* ∨ *i* = *11* ∨ *i* = *12* ∨ *i* > *12*
⟨*proof*⟩

lemma *xx-0-cases*: *vs xx* = *0* ∨ *vs xx* ≠ *0* ⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < *int NN* ∨ *vs xx* ≥ *int NN* ⟨*proof*⟩

lemma *is-IfJump-pcOf*[*simp*]:
pcOf cfg < *12* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3* ∨ *pcOf cfg* = *5*
⟨*proof*⟩

lemma *is-IfJump-pc*[*simp*]:
pc < *12* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3* ∨ *pc* = *5*
⟨*proof*⟩

lemma *eq-Fence-pc*[*simp*]:
pc < *12* ⟹ *prog* ! *pc* = *Fence* ⟷ *pc* = *7*
⟨*proof*⟩

lemma *output1*[*simp*]:
prog ! *9* = *Output U* (*V tt*) ⟨*proof*⟩
lemma *output2*[*simp*]:
prog ! *11* = *Output U* (*N 0*) ⟨*proof*⟩
lemma *is-if*[*simp*]:*is-IfJump* (*prog* ! *3*) ⟨*proof*⟩

lemma *is-nif1*[*simp*]:¬*is-IfJump* (*prog* ! *6*) ⟨*proof*⟩
lemma *is-nif2*[*simp*]:¬*is-IfJump* (*prog* ! *7*) ⟨*proof*⟩

lemma *is-nin1*[*simp*]:¬ *is-getInput* (*prog* ! *6*) ⟨*proof*⟩
lemma *is-nout1*[*simp*]:¬ *is-Output* (*prog* ! *6*) ⟨*proof*⟩
lemma *is-nin2*[*simp*]:¬ *is-getInput* (*prog* ! *10*) ⟨*proof*⟩
lemma *is-nout2*[*simp*]:¬ *is-Output* (*prog* ! *10*) ⟨*proof*⟩

lemma *fence*[*simp*]:*prog* ! *7* = *Fence* ⟨*proof*⟩

lemma *nfence*[*simp*]:*prog* ! *6* ≠ *Fence* ⟨*proof*⟩

consts *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc =
(*if* (*set pc* = {*4*,*11*} ∨ (*6* ∈ *set pc* ∧ (*4* ∈ *set pc* ∨ *11* ∈ *set pc*)))
then True else False)

lemma *resolve-63*: ¬*resolve p* [*6*,*3*] ⟨*proof*⟩
lemma *resolve-64*: *resolve p* [*6*,*4*] ⟨*proof*⟩
lemma *resolve-611*: *resolve p* [*6*,*11*] ⟨*proof*⟩
lemma *resolve-106*: ¬*resolve p* [*10*,*6*] ⟨*proof*⟩

consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts *initPstate* :: *predState*

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = *initPstate* **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* *55*)
where *x* →*B* *y* == *stepB x y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ *55*)
where *x* →*B*∗ *y* == *star stepB x y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*MM* *55*)
where *x* →*MM* *y* == *stepM x y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* 55)
where *x* →*N y* == *stepN x y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ 55)
where *x* →*N*∗ *y* == *star stepN x y*

abbreviation
stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* 55)
where *x* →*S y* == *stepS x y*

abbreviation
stepsS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S*∗ 55)
where *x* →*S*∗ *y* == *star stepS x y*

lemma *endPC*[*simp*]: *endPC* = 12
⟨*proof*⟩

lemma *is-getInput-pcOf*[*simp*]: *pcOf cfg* < 12 ⟹ *is-getInput* (*prog*!(*pcOf cfg*)) ⟷ *pcOf cfg* = 4
⟨*proof*⟩

lemma *getUntrustedInput-pcOf*[*simp*]: *prog*!4 = *Input U xx*
⟨*proof*⟩

lemma *getInput-not6*[*simp*]: ¬*is-getInput* (*prog* ! 6) ⟨*proof*⟩
lemma *getInput-not7*[*simp*]: ¬*is-getInput* (*prog* ! 7) ⟨*proof*⟩
lemma *getInput-not10*[*simp*]: ¬*is-getInput* (*prog* ! 10) ⟨*proof*⟩

lemma *is-Output-pcOf*[*simp*]: *pcOf cfg* < 12 ⟹ *is-Output* (*prog*!(*pcOf cfg*)) ⟷ (*pcOf cfg* = 9 ∨ *pcOf cfg* = 11)
⟨*proof*⟩

lemma *is-Output*: *is-Output* (*prog* ! 9)
⟨*proof*⟩
lemma *is-Output-1*: *is-Output* (*prog* ! 11)
⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*,*ibUT*) ⟷ *pcOf cfg* = 0
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) $\longleftrightarrow$ (*pcOf cfg* = *0* $\wedge$ *cfgs* = []) ⟨*proof*⟩

lemma *getInputT-not*[*simp*]: *pcOf cfg* < *12* $\Longrightarrow$
(*prog* ! *pcOf cfg*) $\neq$ *Input T inp* ⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < *12* $\Longrightarrow$
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *4 then lhd ibUT else* $\bot$) ⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < *12* $\Longrightarrow$
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *9* $\vee$ *pcOf cfg* = *11 then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else $\bot$
) ⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < *12* $\Longrightarrow$
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *4* $\wedge$ *cfgs* = [] *then lhd ibUT else* $\bot$) ⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < *12* $\Longrightarrow$
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if* (*pcOf cfg* = *9* $\vee$ *pcOf cfg* = *11*) $\wedge$ *cfgs* = [] *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else $\bot$
) ⟨*proof*⟩

lemma *eqSec-pcOf*[*simp*]:
eqSec (*cfg1*, *ibT1*,*ibUT1*, *ls1*) (*pstate3*, *cfg3*, *cfgs3*, *ibT3*,*ibUT3*, *ls3*) $\longleftrightarrow$
(*pcOf cfg1* = *0* $\longleftrightarrow$ *pcOf cfg3* = *0* $\wedge$ *cfgs3* = []) $\wedge$
(*pcOf cfg1* = *0* $\longrightarrow$ *stateOf cfg1* = *stateOf cfg3*) ⟨*proof*⟩

lemma *getActInput:pc4 = 4 ⟹ pc3 = 4 ⟹ cfgs3 = [] ⟹ cfgs4 = [] ⟹*
getActO (pstate3, Config pc3 (State (Vstore vs3) avst3 h3 p3), [], ibT3,ibUT3, ls3) =
getActO (pstate4, Config pc4 (State (Vstore vs4) avst4 h4 p4), [], ibT4,ibUT4, ls4)
⟹ lhd ibUT3 = lhd ibUT4
⟨*proof*⟩

lemma *nextB-pc0[simp]*:
nextB (Config 0 s, ibT,ibUT) =
(Config 1 s, ibT,ibUT)
⟨*proof*⟩

lemma *readLocs-pc0[simp]*:
readLocs (Config 0 s) = {}
⟨*proof*⟩

lemma *nextB-pc1[simp]*:
nextB (Config 1 (State (Vstore vs) avst hh p), ibT,ibUT) =
((Config 2 (State (Vstore (vs(tt := 0))) avst hh p)), ibT,ibUT)
⟨*proof*⟩

lemma *nextB-pc1'[simp]*:
nextB (Config (Suc 0) (State (Vstore vs) avst hh p), ibT,ibUT) =
((Config 2 (State (Vstore (vs(tt := 0))) avst hh p)), ibT,ibUT)
⟨*proof*⟩

lemma *readLocs-pc1[simp]*:
readLocs (Config 1 s) = {}
⟨*proof*⟩

lemma *readLocs-pc1'[simp]*:
readLocs (Config (Suc 0) s) = {}
⟨*proof*⟩

lemma *nextB-pc2[simp]*:
nextB (Config 2 (State (Vstore vs) avst hh p), ibT,ibUT) =
((Config 3 (State (Vstore (vs(xx := 1))) avst hh p)), ibT,ibUT)
⟨*proof*⟩

lemma *readLocs-pc2[simp]*:
readLocs (Config 2 s) = {}

⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx ≠ *0* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx = *0* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 11* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config* (*if vs xx* ≠ *0 then 4 else 11*) (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:
readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx = *0* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx ≠ *0* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 11* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config* (*if vs xx* ≠ *0 then 11 else 4*) (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 5* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst hh p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:
readLocs (*Config 4 s*) = {}
⟨*proof*⟩

lemma *nextB-pc5-then*[*simp*]:
vs xx < *int NN* ⟹
nextB (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc5-else*[*simp*]:
vs xx ≥ *int NN* ⟹
nextB (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 10* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc5*:
nextB (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config* (*if vs xx* < *NN then 6 else 10*) (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
readLocs (*Config 5 s*) = {}
⟨*proof*⟩

lemma *nextM-pc5-then*[*simp*]:
vs xx ≥ *int NN* ⟹
nextM (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextM-pc5-else*[*simp*]:
vs xx < *int NN* ⟹
nextM (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config 10* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextM-pc5*:
nextM (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*) =
(*Config* (*if vs xx* < *NN then 10 else 6*) (*State* (*Vstore vs*) *avst hh p*), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *nextB-pc6*[*simp*]:
nextB (*Config 6* (*State* (*Vstore vs*) *avst* (*Heap hh*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa1* (*nat* (*vs xx*)) *avst*

in (*Config 7* (*State* (*Vstore* (*vs*(*vv* := *hh l*))) *avst* (*Heap hh*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc6*[*simp*]:
readLocs (*Config 6* (*State* (*Vstore vs*) *avst hh p*)) = {*array-loc aa1* (*nat* (*vs xx*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc7*[*simp*]:
nextB (*Config 7 s*, *ibT*,*ibUT*) = (*Config 8 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc7*[*simp*]:
readLocs (*Config 7 s*) = {}
⟨*proof*⟩

lemma *nextB-pc8*[*simp*]:
nextB (*Config 8* (*State* (*Vstore vs*) *avst* (*Heap hh*) *p*), *ibT*,*ibUT*) =
(*let l* = *array-loc aa2* (*nat* (*vs vv* ∗ *SS*)) *avst*
in (*Config 9* (*State* (*Vstore* (*vs*(*tt* := *hh l*))) *avst* (*Heap hh*) *p*)), *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc8*[*simp*]:
readLocs (*Config 8* (*State* (*Vstore vs*) *avst hh p*)) = {*array-loc aa2* (*nat* (*vs vv* ∗ *SS*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc9*[*simp*]:
nextB (*Config 9 s*, *ibT*,*ibUT*) = (*Config 10 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc9*[*simp*]:
readLocs (*Config 9 s*) = {}
⟨*proof*⟩

lemma *nextB-pc10*[*simp*]:
nextB (*Config 10 s*, *ibT*,*ibUT*) = (*Config 3 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc10*[*simp*]:

readLocs (*Config 10 s*) = {}
⟨*proof*⟩

lemma *nextB-pc11*[*simp*]:
nextB (*Config 11 s*, *ibT*,*ibUT*) =
(*Config 12 s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *readLocs-pc11*[*simp*]:
readLocs (*Config 11 s*) = {}
⟨*proof*⟩

lemma *map-L1*:*length cfgs* = *Suc 0* ⟹
pcOf (*last cfgs*) = *y* ⟹ *map pcOf cfgs* = [*y*]
⟨*proof*⟩

lemma *map-L2*:*length cfgs* = *2* ⟹
pcOf (*cfgs* ! *0*) = *x* ⟹
pcOf (*last cfgs*) = *y* ⟹ *map pcOf cfgs* = [*x*,*y*]
⟨*proof*⟩

lemma *length cfgs* = *2* ⟹ (*cfgs* ! *0*) = *last* (*butlast cfgs*)
⟨*proof*⟩

lemma *nextB-stepB-pc*:
pc < *12* ⟹ (*pc* = *4* ⟶ *ibUT* ≠ *LNil*) ⟹
(*Config pc s*, *ibT*,*ibUT*) →*B nextB* (*Config pc s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *not-finalB*:
pc < *12* ⟹ (*pc* = *4* ⟶ *ibUT* ≠ *LNil*) ⟹
¬ *finalB* (*Config pc s*, *ibT*,*ibUT*)
⟨*proof*⟩

lemma *finalB-pc-iff'*:
pc < *12* ⟹
finalB (*Config pc s*, *ibT*,*ibUT*) ⟷
(*pc* = *4* ∧ *ibUT* = *LNil*)
⟨*proof*⟩

lemma *finalB-pc-iff*:
pc ≤ *12* ⟹
finalB (*Config pc s*, *ibT*,*ibUT*) ⟷
(*pc* = *12* ∨ *pc* = *4* ∧ *ibUT* = *LNil*)

⟨*proof*⟩

lemma *finalB-pcOf-iff*[*simp*]:
pcOf cfg ≤ *12* ⟹
finalB (*cfg*, *ibT*,*ibUT*) ⟷ (*pcOf cfg* = *12* ∨ *pcOf cfg* = *4* ∧ *ibUT* = *LNil*)
⟨*proof*⟩

lemma *finalS-cond*:*pcOf cfg* < *12* ⟹ *noMisSpec cfgs* ⟹ *ibUT* ≠ *LNil* ⟹ ¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-cond'*:*pcOf cfg* < *12* ⟹ *cfgs* = [] ⟹ *ibUT* ≠ *LNil* ⟹ ¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-while-spec*:
whileSpeculation cfg (*last cfgs*) ⟹
length cfgs = *Suc 0* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-while-spec-L2*:
pcOf cfg = *6* ⟹
whileSpeculation (*cfgs*!*0*) (*last cfgs*) ⟹
length cfgs = *2* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-if-spec*:
(*pcOf* (*last cfgs*) ∈ *inThenIfBeforeFence* ∧ *pcOf cfg* = *10*) ∨
(*pcOf* (*last cfgs*) ∈ *inElseIf* ∧ *pcOf cfg* = *6*) ⟹
length cfgs = *Suc 0* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*,*ibUT*, *ls*)
⟨*proof*⟩

end

12.2 Proof

theory *Fun5-secure*
imports *Fun5*
begin

definition *common* :: *enat* ⇒ *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common = (λ*w1 w2*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧
cfg1 = *cfg3* ∧ *cfg2* = *cfg4* ∧
pcOf cfg3 = *pcOf cfg4* ∧ *map pcOf cfgs3* = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ' (*set cfgs3*) ⊆ *PC* ∧
llength ibUT1 = ∞ ∧ *llength ibUT2* = ∞ ∧
ibUT1 = *ibUT3* ∧ *ibUT2* = *ibUT4* ∧

w1 = *w2* ∧
///
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧
///
(*statA* = *Diff* ⟶ *statO* = *Diff*) ∧
Dist ls1 ls2 ls3 ls4))

lemma *common-implies*: *common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < *12* ∧ *pcOf cfg2* = *pcOf cfg1* ∧
ibUT1 ≠ [[]] ∧ *ibUT2* ≠ [[]] ∧ *w1* = *w2*
⟨*proof*⟩

definition Δ*0* :: *enat* ⇒ *enat* ⇒ *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool* **where**

$\Delta 0$ = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
pcOf cfg3 $\in$ *beforeWhile* $\wedge$
(*pcOf cfg3* $>$ *1* $\longrightarrow$ *same-var-o tt cfg3 cfgs3 cfg4 cfgs4*) $\wedge$
(*pcOf cfg3* $>$ *2* $\longrightarrow$ *same-var-o xx cfg3 cfgs3 cfg4 cfgs4*) $\wedge$
(*pcOf cfg3* $>$ *4* $\longrightarrow$ *same-var-o xx cfg3 cfgs3 cfg4 cfgs4*) $\wedge$
noMisSpec cfgs3
))

lemmas Δ*0-defs* = Δ*0-def common-def PC-def same-var-o-def*
beforeWhile-def noMisSpec-def

lemma Δ*0-implies*: Δ*0 num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg1 $<$ *12* $\wedge$ *pcOf cfg2* = *pcOf cfg1* $\wedge$
ibUT1 $\neq$ [[]] $\wedge$ *ibUT2* $\neq$ [[]] $\wedge$ *cfgs4* = []
$\langle$*proof*$\rangle$

definition Δ*1* :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV*
$\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
Δ*1* = (λ *num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
pcOf cfg3 $\in$ *inWhile* $\wedge$
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$

noMisSpec cfgs3
))
lemmas *Δ1-defs = Δ1-def common-def PC-def noMisSpec-def inWhile-def same-var-o-def*
lemma *Δ1-implies*: *Δ1 n w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg3 $<$ *12* $\wedge$ *cfgs3* $=$ [] $\wedge$ *ibUT3* $\neq$ [[]] $\wedge$
pcOf cfg4 $<$ *12* $\wedge$ *cfgs4* $=$ [] $\wedge$ *ibUT4* $\neq$ [[]]
⟨*proof*⟩

definition *Δ1′* :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV*
$\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
Δ1′ = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$
whileSpeculation cfg3 (*last cfgs3*) $\wedge$
misSpecL1 cfgs3 $\wedge$ *misSpecL1 cfgs4* $\wedge$
w1 $=$ ∞
))
lemmas *Δ1′-defs = Δ1′-def common-def PC-def same-var-def*
startOfIfThen-def startOfElseBranch-def
misSpecL1-def whileSpec-defs

lemma *Δ1′-implies*: *Δ1′ num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg3 $<$ *12* $\wedge$ *pcOf cfg4* $<$ *12* $\wedge$
whileSpeculation cfg3 (*last cfgs3*) $\wedge$
whileSpeculation cfg4 (*last cfgs4*) $\wedge$
length cfgs3 $=$ *Suc 0* $\wedge$ *length cfgs4* $=$ *Suc 0*
⟨*proof*⟩

definition $\Delta 2 :: enat \Rightarrow enat \Rightarrow enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV \Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 2 = (\lambda num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO.$
$(common\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \wedge$

$same\text{-}var\text{-}o\ xx\ cfg3\ cfgs3\ cfg4\ cfgs4 \wedge$
$pcOf\ cfg3 = startOfIfThen \wedge pcOf\ (last\ cfgs3) \in inElseIf \wedge$
$misSpecL1\ cfgs3 \wedge misSpecL1\ cfgs4 \wedge$

$(pcOf\ (last\ cfgs3) = startOfElseBranch \longrightarrow w1 = \infty) \wedge$
$(pcOf\ (last\ cfgs3) = 3 \longrightarrow w1 = 3) \wedge$

$(pcOf\ (last\ cfgs3) = startOfWhileThen \vee$
$pcOf\ (last\ cfgs3) = whileElse \longrightarrow w1 = 1)$
))

lemmas $\Delta 2\text{-}defs = \Delta 2\text{-}def\ common\text{-}def\ PC\text{-}def\ same\text{-}var\text{-}o\text{-}def\ misSpecL1\text{-}def$
$startOfIfThen\text{-}def\ inElseIf\text{-}def\ same\text{-}var\text{-}def$
$startOfWhileThen\text{-}def\ whileElse\text{-}def\ startOfElseBranch\text{-}def$

lemma $\Delta 2\text{-}implies$: $\Delta 2\ num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \Longrightarrow$
$pcOf\ (last\ cfgs3) \in inElseIf \wedge pcOf\ cfg3 = 6 \wedge$
$pcOf\ (last\ cfgs4) = pcOf\ (last\ cfgs3) \wedge$
$pcOf\ cfg4 = pcOf\ cfg3 \wedge length\ cfgs3 = Suc\ 0 \wedge$
$length\ cfgs4 = Suc\ 0 \wedge same\text{-}var\ xx\ (last\ cfgs3)\ (last\ cfgs4)$
$\langle proof \rangle$

definition $\Delta 2' :: enat \Rightarrow enat \Rightarrow enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV \Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 2' = (\lambda num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$

```
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO.
 (common w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
  pcOf cfg3 = startOfIfThen ∧
  whileSpeculation (cfgs3!0) (last cfgs3) ∧
  misSpecL2 cfgs3 ∧ misSpecL2 cfgs4 ∧
  w1 = 2
))

lemmas Δ2'-defs = Δ2'-def common-def PC-def same-var-def
                  startOfElseBranch-def startOfIfThen-def
                  whileSpec-defs misSpecL2-def

lemma Δ2'-implies: Δ2' num w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
   (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
   statA
   (cfg1,ibT1,ibUT1,ls1)
   (cfg2,ibT2,ibUT2,ls2)
   statO ⟹
  pcOf cfg3 = 6 ∧ pcOf cfg4 = 6 ∧
  whileSpeculation (cfgs3!0) (last cfgs3) ∧
  whileSpeculation (cfgs4!0) (last cfgs4) ∧
  length cfgs3 = 2 ∧ length cfgs4 = 2
  ⟨proof⟩


definition Δ3 :: enat ⇒ enat ⇒ enat ⇒ stateO ⇒ stateO  ⇒ status ⇒ stateV
⇒ stateV ⇒ status ⇒ bool where
Δ3 = (λnum w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO.
 (common w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
```

$pcOf\ cfg3 = startOfElseBranch \wedge pcOf\ (last\ cfgs3) \in inThenIfBeforeFence \wedge$
$misSpecL1\ cfgs3 \wedge$
$(pcOf\ (last\ cfgs3) = 6 \longrightarrow w1 = \infty) \wedge$
$(pcOf\ (last\ cfgs3) = 7 \longrightarrow w1 = 1)$
))

lemmas $\Delta 3\text{-}defs = \Delta 3\text{-}def\ common\text{-}def\ PC\text{-}def\ same\text{-}var\text{-}o\text{-}def$
$startOfElseBranch\text{-}def\ inThenIfBeforeFence\text{-}def$

lemma $\Delta 3\text{-}implies$: $\Delta 3\ num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \Longrightarrow$
$pcOf\ (last\ cfgs3) \in inThenIfBeforeFence \wedge$
$pcOf\ (last\ cfgs4) = pcOf\ (last\ cfgs3) \wedge$
$pcOf\ cfg3 = 10 \wedge pcOf\ cfg3 = pcOf\ cfg4 \wedge$
$length\ cfgs3 = Suc\ 0 \wedge length\ cfgs4 = Suc\ 0$
⟨*proof*⟩

definition $\Delta e :: enat \Rightarrow enat \Rightarrow enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV \Rightarrow$
$stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta e = (\lambda num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO.$
$(pcOf\ cfg3 = endPC \wedge pcOf\ cfg4 = endPC \wedge cfgs3 = [] \wedge cfgs4 = [] \wedge$
$pcOf\ cfg1 = endPC \wedge pcOf\ cfg2 = endPC))$

lemmas $\Delta e\text{-}defs = \Delta e\text{-}def\ common\text{-}def\ endPC\text{-}def$

lemma $init$: $initCond\ \Delta 0$
⟨*proof*⟩

lemma $step0$: $unwindIntoCond\ \Delta 0\ (oor\ \Delta 0\ \Delta 1)$
⟨*proof*⟩

lemma *step1*: *unwindIntoCond* $\Delta 1$ (*oor5* $\Delta 1$ $\Delta 1'$ $\Delta 2$ $\Delta 3$ Δe)
⟨*proof*⟩

lemma *step2*: *unwindIntoCond* $\Delta 2$ (*oor3* $\Delta 2$ $\Delta 2'$ $\Delta 1$)
⟨*proof*⟩

lemma *step3*: *unwindIntoCond* $\Delta 3$ (*oor* $\Delta 3$ $\Delta 1$)
⟨*proof*⟩

lemma *step4*: *unwindIntoCond* $\Delta 1'$ $\Delta 1$
⟨*proof*⟩

lemma *step5*: *unwindIntoCond* $\Delta 2'$ $\Delta 2$
⟨*proof*⟩

lemma *stepe*: *unwindIntoCond* Δe Δe
⟨*proof*⟩

lemmas *theConds* = *step0 step1 step2 step3 step4 step5 stepe*

proposition *lrsecure*
⟨*proof*⟩

end

13 Proof of Relative Security for fun6

theory *Fun6*
imports *../Instance-IMP/Instance-Secret-IMem-Inp*
Relative-Security.Unwinding
begin

13.1 Function definition and Boilerplate

no-notation *bot* ($\bot$)
consts *NN* :: *nat*
lemma *NN*: $NN \geq 0$ ⟨*proof*⟩

definition *aa1* :: *avname* **where** *aa1* = ''*a1*''
definition *aa2* :: *avname* **where** *aa2* = ''*a2*''
definition *vv* :: *vname* **where** *vv* = ''*v*''
definition *tt* :: *vname* **where** *tt* = ''*y*''

lemmas *vvars-defs* = *aa1-def aa2-def vv-def xx-def tt-def yy-def ffile-def*

lemma *vvars-dff*[*simp*]:
$aa1 \neq aa2\ aa1 \neq vv\ aa1 \neq xx\ aa1 \neq yy\ aa1 \neq tt\ aa1 \neq ffile$
$aa2 \neq aa1\ aa2 \neq vv\ aa2 \neq xx\ aa2 \neq yy\ aa2 \neq tt\ aa2 \neq ffile$
$vv \neq aa1\ vv \neq aa2\ \ vv \neq xx\ vv \neq yy\ vv \neq tt\ vv \neq ffile$
$xx \neq aa1\ xx \neq aa2\ xx \neq vv\ xx \neq yy\ xx \neq tt\ xx \neq ffile$
$tt \neq aa1\ tt \neq aa2\ tt \neq vv\ tt \neq yy\ tt \neq xx\ tt \neq ffile$
$yy \neq aa1\ yy \neq aa2\ yy \neq vv\ yy \neq xx\ yy \neq tt\ yy \neq ffile$
$ffile \neq aa1\ ffile \neq aa2\ ffile \neq vv\ ffile \neq xx\ ffile \neq tt\ ffile \neq yy$
⟨*proof*⟩

consts *size-aa1* :: *nat*
consts *size-aa2* :: *nat*

fun *initAvstore* :: *avstore* ⇒ *bool* **where**
initAvstore (*Avstore as*) = (*as aa1* = (*0*, *size-aa1*) ∧ *as aa2* = (*size-aa1*, *size-aa2*))

fun *istate* ::*state* ⇒ *bool* **where**
istate s = (*initAvstore* (*getAvstore s*))

definition *prog* ≡
[
0// *Start* ,
1// *tt* ::= (*N 0*),
2// *xx* ::= (*N 1*),
3// *IfJump* (*Not* (*Eq* (*V xx*) (*N 0*))) *4 13* ,
4// *Input U xx* ,
5// *Input T yy* ,
6// *IfJump* (*Less* (*V xx*) (*N NN*)) *7 12* ,
7// *vv* ::= *VA aa1* (*V xx*) ,
8// *writeSecretOnFile*,
9// *Fence* ,
10// *tt* ::= (*VA aa2* (*Times* (*V vv*) (*N 512*))) ,
11// *Output U* (*V tt*) ,
12// *Jump 3*,
13// *Output U* (*N 0*)

]

definition *PC* ≡ {*0..13*}

definition *beforeWhile* = {*0*,*1*,*2*}
definition *afterWhile* = {*3..13*}
definition *startOfWhileThen* = *4*
definition *startOfIfThen* = *7*
definition *inThenIfBeforeOutput* = {*7*,*8*}
definition *startOfElseBranch* = *12*
definition *inElseIf* = {*12*,*3*,*4*,*13*}
definition *whileElse* = *13*

fun *leftWhileSpec* **where**
leftWhileSpec cfg cfg′ =
(*pcOf cfg* = *whileElse* ∧
pcOf cfg′ = *startOfWhileThen*)

fun *rightWhileSpec* **where**
rightWhileSpec cfg cfg′ =
(*pcOf cfg* = *startOfWhileThen* ∧
pcOf cfg′ = *whileElse*)

fun *whileSpeculation* **where**
whileSpeculation cfg cfg′ =
(*leftWhileSpec cfg cfg′* ∨
rightWhileSpec cfg cfg′)
lemmas *whileSpec-def* = *whileSpeculation.simps*
startOfWhileThen-def
whileElse-def

lemmas *whileSpec-defs* = *whileSpec-def*
leftWhileSpec.simps
rightWhileSpec.simps

lemma *cases-14*: (*i*::*pcounter*) = *0* ∨ *i* = *1* ∨ *i* = *2* ∨ *i* = *3* ∨ *i* = *4* ∨ *i* = *5* ∨
i = *6* ∨ *i* = *7* ∨ *i* = *8* ∨ *i* = *9* ∨ *i* = *10* ∨ *i* = *11* ∨ *i* = *12* ∨ *i* = *13* ∨ *i* = *14*
∨ *i* > *14*
⟨*proof*⟩

lemma *xx-0-cases*: *vs xx* = *0* ∨ *vs xx* ≠ *0* ⟨*proof*⟩

lemma *xx-NN-cases*: *vs xx* < *int NN* ∨ *vs xx* ≥ *int NN* ⟨*proof*⟩

lemma *is-If-pcOf*[*simp*]:
pcOf cfg < *14* ⟹ *is-IfJump* (*prog* ! (*pcOf cfg*)) ⟷ *pcOf cfg* = *3* ∨ *pcOf cfg* = *6*

⟨*proof*⟩

lemma *is-If-pc*[*simp*]:
pc < *14* ⟹ *is-IfJump* (*prog* ! *pc*) ⟷ *pc* = *3* ∨ *pc* = *6*
⟨*proof*⟩

lemma *eq-Fence-pc*[*simp*]:
pc < *14* ⟹ *prog* ! *pc* = *Fence* ⟷ *pc* = *9*
⟨*proof*⟩

lemma *output1*[*simp*]:*prog* ! *11* = *Output U* (*V tt*) ⟨*proof*⟩
lemma *output2*[*simp*]:*prog* ! *13* = *Output U* (*N 0*) ⟨*proof*⟩
lemma *is-if*[*simp*]:*is-IfJump* (*prog* ! *3*) ⟨*proof*⟩

lemma *is-nif1*[*simp*]:¬*is-IfJump* (*prog* ! *7*) ⟨*proof*⟩
lemma *is-nif2*[*simp*]:¬*is-IfJump* (*prog* ! *8*) ⟨*proof*⟩

lemma *getInput-not6*[*simp*]:¬ *is-getInput* (*prog* ! *6*) ⟨*proof*⟩
lemma *Output-not6*[*simp*]:¬ *is-Output* (*prog* ! *6*) ⟨*proof*⟩

lemma *getInput-not7*[*simp*]:¬ *is-getInput* (*prog* ! *7*) ⟨*proof*⟩
lemma *Output-not7*[*simp*]:¬ *is-Output* (*prog* ! *7*) ⟨*proof*⟩

lemma *getInput-not8*[*simp*]:¬ *is-getInput* (*prog* ! *8*) ⟨*proof*⟩
lemma *Output-not8*[*simp*]:*is-Output* (*prog* ! *8*) ⟨*proof*⟩

lemma *is-nif*[*simp*]:¬ *is-IfJump* (*prog* ! *9*) ⟨*proof*⟩
lemma *getInput-not10*[*simp*]:¬ *is-getInput* (*prog* ! *10*) ⟨*proof*⟩
lemma *Output-not10*[*simp*]:¬ *is-Output* (*prog* ! *10*) ⟨*proof*⟩

lemma *getInput-not12*[*simp*]:¬ *is-getInput* (*prog* ! *12*) ⟨*proof*⟩
lemma *Output-not12*[*simp*]:¬ *is-Output* (*prog* ! *12*) ⟨*proof*⟩

lemma *fence*[*simp*]:*prog* ! *9* = *Fence* ⟨*proof*⟩

lemma *nfence*[*simp*]:*prog* ! *7* ≠ *Fence* ⟨*proof*⟩

consts *mispred* :: *predState* ⇒ *pcounter list* ⇒ *bool*
fun *resolve* :: *predState* ⇒ *pcounter list* ⇒ *bool* **where**
resolve p pc =
(*if* (*set pc* = {*4*,*13*} ∨ (*7* ∈ *set pc* ∧ (*4* ∈ *set pc* ∨ *13* ∈ *set pc*)) ∨ *pc* = [*12*,*8*])
then True else False)

lemma *resolve-73*: ¬*resolve p* [*7*,*3*] ⟨*proof*⟩
lemma *resolve-74*: *resolve p* [*7*,*4*] ⟨*proof*⟩
lemma *resolve-713*: *resolve p* [*7*,*13*] ⟨*proof*⟩
lemma *resolve-127*: ¬*resolve p* [*12*,*7*] ⟨*proof*⟩
lemma *resolve-129*: ¬*resolve p* [*12*,*9*] ⟨*proof*⟩

consts *update* :: *predState* ⇒ *pcounter list* ⇒ *predState*
consts *initPstate* :: *predState*

interpretation *Prog-Mispred-Init* **where**
prog = *prog* **and** *initPstate* = *initPstate* **and**
mispred = *mispred* **and** *resolve* = *resolve* **and** *update* = *update* **and**
istate = *istate*
⟨*proof*⟩

abbreviation
stepB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B* 55)
where *x* →*B* *y* == *stepB* *x* *y*

abbreviation
stepsB-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*B*∗ 55)
where *x* →*B*∗ *y* == *star* *stepB* *x* *y*

abbreviation
stepM-abbrev :: *config* × *val llist* × *val llist* ⇒ *config* × *val llist* × *val llist* ⇒ *bool* (**infix** →*MM* 55)
where *x* →*MM* *y* == *stepM* *x* *y*

abbreviation
stepN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N* 55)
where *x* →*N* *y* == *stepN* *x* *y*

abbreviation
stepsN-abbrev :: *config* × *val llist* × *val llist* × *loc set* ⇒ *config* × *val llist* × *val llist* × *loc set* ⇒ *bool* (**infix** →*N*∗ 55)
where *x* →*N*∗ *y* == *star* *stepN* *x* *y*

abbreviation
stepS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S* 55)
where *x* →*S* *y* == *stepS* *x* *y*

abbreviation
stepsS-abbrev :: *configS* ⇒ *configS* ⇒ *bool* (**infix** →*S*∗ 55)
where *x* →*S*∗ *y* == *star* *stepS* *x* *y*

lemma *endPC*[*simp*]: *endPC* = *14*
⟨*proof*⟩

lemma *is-getInput-pcOf*[*simp*]: *pcOf cfg* < *14* ⟹ *is-getInput* (*prog*!(*pcOf cfg*))
⟷ *pcOf cfg* = *4* ∨ *pcOf cfg* = *5*
⟨*proof*⟩

lemma *is-Output-pcOf*[*simp*]: *pcOf cfg* < *14* ⟹ *is-Output* (*prog*!(*pcOf cfg*)) ⟷
(*pcOf cfg* = *8* ∨ *pcOf cfg* = *11* ∨ *pcOf cfg* = *13*)
⟨*proof*⟩

lemma *is-Output-T*: *is-Output* (*prog* ! *8*)
⟨*proof*⟩
lemma *is-Output*: *is-Output* (*prog* ! *11*)
⟨*proof*⟩
lemma *is-Output-1*: *is-Output* (*prog* ! *13*)
⟨*proof*⟩

lemma *isSecV-pcOf*[*simp*]:
isSecV (*cfg*,*ibT*,*ibUT*,*ls*) ⟷ ¬*finalB* (*cfg*,*ibT*,*ibUT*)
⟨*proof*⟩

lemma *isSecO-pcOf*[*simp*]:
isSecO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) ⟷
¬*finalS* (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) ∧ *cfgs* = []
⟨*proof*⟩

lemma *getActV-pcOf*[*simp*]:
pcOf cfg < *14* ⟹
getActV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *4 then lhd ibUT*
else if pcOf cfg = *5 then lhd ibT*
else ⊥)
⟨*proof*⟩

lemma *getObsV-pcOf*[*simp*]:
pcOf cfg < *14* ⟹
getObsV (*cfg*,*ibT*,*ibUT*,*ls*) =
(*if pcOf cfg* = *11* ∨ *pcOf cfg* = *13 then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *getActO-pcOf*[*simp*]:
pcOf cfg < *12* ⟹
getActO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if cfgs* = [] *then*
(*if pcOf cfg* = *4 then lhd ibUT*
else if pcOf cfg = *5 then lhd ibT*
else ⊥) *else* ⊥)
⟨*proof*⟩

lemma *getObsO-pcOf*[*simp*]:
pcOf cfg < *14* ⟹
getObsO (*pstate*,*cfg*,*cfgs*,*ibT*,*ibUT*,*ls*) =
(*if* (*pcOf cfg* = *11* ∨ *pcOf cfg* = *13*) ∧ *cfgs* = [] *then*
(*outOf* (*prog*!(*pcOf cfg*)) (*stateOf cfg*), *ls*)
else ⊥
)
⟨*proof*⟩

lemma *getActTrustedInput*:*pc4* = *4* ⟹ *pc3* = *4* ⟹ *cfgs3* = [] ⟹ *cfgs4* = []
⟹
getActO (*pstate3*, *Config pc3* (*State* (*Vstore vs3*) *avst3 h3 p3*), [], *ib3T*, *ib3UT*, *ls3*) =
getActO (*pstate4*, *Config pc4* (*State* (*Vstore vs4*) *avst4 h4 p4*), [], *ib4T*, *ib4UT*, *ls4*)
⟹ *lhd ib3UT* = *lhd ib4UT*
⟨*proof*⟩

lemma *getActUntrustedInput*:*pc4* = *5* ⟹ *pc3* = *5* ⟹ *cfgs3* = [] ⟹ *cfgs4* = []
⟹
getActO (*pstate3*, *Config pc3* (*State* (*Vstore vs3*) *avst3 h3 p3*), [], *ib3T*, *ib3UT*, *ls3*) =
getActO (*pstate4*, *Config pc4* (*State* (*Vstore vs4*) *avst4 h4 p4*), [], *ib4T*, *ib4UT*, *ls4*)
⟹ *lhd ib3T* = *lhd ib4T*
⟨*proof*⟩

lemma *nextB-pc0*[*simp*]:
nextB (*Config 0 s*, *ibT*, *ibUT*) = (*Config 1 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc0*[*simp*]:
readLocs (*Config 0 s*) = {}
⟨*proof*⟩

lemma *nextB-pc1*[*simp*]:
nextB (*Config 1* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
((*Config 2* (*State* (*Vstore* (*vs*(*tt* := *0*))) *avst hh p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc1'*[*simp*]:
nextB (*Config* (*Suc 0*) (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
((*Config 2* (*State* (*Vstore* (*vs*(*tt* := *0*))) *avst hh p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc1*[*simp*]:
readLocs (*Config 1 s*) = {}
⟨*proof*⟩

lemma *readLocs-pc1'*[*simp*]:
readLocs (*Config* (*Suc 0*) *s*) = {}
⟨*proof*⟩

lemma *nextB-pc2*[*simp*]:
nextB (*Config 2* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
((*Config 3* (*State* (*Vstore* (*vs*(*xx* := *1*))) *avst hh p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc2*[*simp*]:
readLocs (*Config 2 s*) = {}
⟨*proof*⟩

lemma *nextB-pc3-then*[*simp*]:
vs xx ≠ *0* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3-else*[*simp*]:
vs xx = *0* ⟹
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 13* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc3*:
nextB (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* ≠ *0 then 4 else 13*) (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc3*[*simp*]:
readLocs (*Config 3 s*) = {}
⟨*proof*⟩

lemma *nextM-pc3-then*[*simp*]:
vs xx = *0* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3-else*[*simp*]:
vs xx ≠ *0* ⟹
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 13* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc3*:
nextM (*Config 3* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* ≠ *0 then 13 else 4*) (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc4*[*simp*]:
ibUT ≠ *LNil* ⟹ *nextB* (*Config 4* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 5* (*State* (*Vstore* (*vs*(*xx* := *lhd ibUT*))) *avst hh p*), *ibT*, *ltl ibUT*)
⟨*proof*⟩

lemma *readLocs-pc4*[*simp*]:
readLocs (*Config 4 s*) = {}
⟨*proof*⟩

lemma *nextB-pc5*[*simp*]:
ibT ≠ *LNil* ⟹ *nextB* (*Config 5* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 6* (*State* (*Vstore* (*vs*(*yy* := *lhd ibT*))) *avst hh p*), *ltl ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc5*[*simp*]:
readLocs (*Config 5 s*) = {}
⟨*proof*⟩

lemma *nextB-pc6-then*[*simp*]:
vs xx < *int NN* ⟹
nextB (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =

(*Config 7* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc6-else*[*simp*]:
vs xx ≥ *int NN* ⟹
nextB (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 12* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc6*:
nextB (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *int NN then 7 else 12*) (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc6*[*simp*]:
readLocs (*Config 6 s*) = {}
⟨*proof*⟩

lemma *nextM-pc6-then*[*simp*]:
vs xx ≥ *int NN* ⟹
nextM (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 7* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc6-else*[*simp*]:
vs xx < *int NN* ⟹
nextM (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config 12* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextM-pc6*:
nextM (*Config 6* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
(*Config* (*if vs xx* < *int NN then 12 else 7*) (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *nextB-pc7*[*simp*]:
nextB (*Config 7* (*State* (*Vstore vs*) *avst* (*Heap hh*) *p*), *ibT*, *ibUT*) =
(*let l* = *array-loc aa1* (*nat* (*vs xx*)) *avst*
in (*Config 8* (*State* (*Vstore* (*vs*(*vv* := *hh l*))) *avst* (*Heap hh*) *p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc7*[*simp*]:
readLocs (*Config 7* (*State* (*Vstore vs*) *avst hh p*)) = {*array-loc aa1* (*nat* (*vs xx*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc8*[*simp*]:
nextB (*Config 8* (*State* (*Vstore vs*) *avst hh p*), *ibT*, *ibUT*) =
((*Config 9* (*State* (*Vstore vs*) *avst hh p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc8*[*simp*]:
readLocs (*Config 8 s*) = {}
⟨*proof*⟩

lemma *nextB-pc9*[*simp*]:
nextB (*Config 9 s*, *ibT*, *ibUT*) = (*Config 10 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc9*[*simp*]:
readLocs (*Config 9 s*) = {}
⟨*proof*⟩

lemma *nextB-pc10*[*simp*]:
nextB (*Config 10* (*State* (*Vstore vs*) *avst* (*Heap hh*) *p*), *ibT*, *ibUT*) =
(*let l* = *array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*
in (*Config 11* (*State* (*Vstore* (*vs*(*tt* := *hh l*))) *avst* (*Heap hh*) *p*)), *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc10*[*simp*]:
readLocs (*Config 10* (*State* (*Vstore vs*) *avst hh p*)) = {*array-loc aa2* (*nat* (*vs vv* ∗ *512*)) *avst*}
⟨*proof*⟩

lemma *nextB-pc11*[*simp*]:
nextB (*Config 11 s*, *ibT*, *ibUT*) = (*Config 12 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc11*[*simp*]:
readLocs (*Config 11 s*) = {}
⟨*proof*⟩

lemma *nextB-pc12*[*simp*]:
nextB (*Config 12 s*, *ibT*, *ibUT*) = (*Config 3 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc12*[*simp*]:
readLocs (*Config 12 s*) = {}
⟨*proof*⟩

lemma *nextB-pc13*[*simp*]:
nextB (*Config 13 s*, *ibT*, *ibUT*) =
(*Config 14 s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *readLocs-pc13*[*simp*]:
readLocs (*Config 13 s*) = {}
⟨*proof*⟩

lemma *map-L1*:*length cfgs* = *Suc 0* ⟹
pcOf (*last cfgs*) = *y* ⟹ *map pcOf cfgs* = [*y*]
⟨*proof*⟩

lemma *map-L2*:*length cfgs* = *2* ⟹
pcOf (*cfgs* ! *0*) = *x* ⟹
pcOf (*last cfgs*) = *y* ⟹ *map pcOf cfgs* = [*x*,*y*]
⟨*proof*⟩

lemma *length cfgs* = *2* ⟹ (*cfgs* ! *0*) = *last* (*butlast cfgs*)
⟨*proof*⟩

lemma *nextB-stepB-pc*:
pc < *14* ⟹ (*pc* = *4* ⟶ *ibUT* ≠ *LNil*) ⟹ (*pc* = *5* ⟶ *ibT* ≠ *LNil*) ⟹
(*Config pc s*, *ibT*, *ibUT*) →*B nextB* (*Config pc s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *not-finalB*:
pc < *14* ⟹ (*pc* = *4* ⟶ *ibUT* ≠ *LNil*) ⟹ (*pc* = *5* ⟶ *ibT* ≠ *LNil*) ⟹
¬ *finalB* (*Config pc s*, *ibT*, *ibUT*)
⟨*proof*⟩

lemma *finalB-pc-iff'*:
pc < *14* ⟹
finalB (*Config pc s*, *ibT*, *ibUT*) ⟷
(*pc* = *4* ∧ *ibUT* = *LNil*) ∨ (*pc* = *5* ∧ *ibT* = *LNil*)
⟨*proof*⟩

lemma *finalB-pc-iff*:
pc ≤ *14* ⟹

finalB (*Config pc s*, *ibT*, *ibUT*) ⟷
(*pc* = *14* ∨ (*pc* = *4* ∧ *ibUT* = *LNil*) ∨ (*pc* = *5* ∧ *ibT* = *LNil*))
⟨*proof*⟩

lemma *finalB-pcOf-iff*[*simp*]:
pcOf cfg ≤ *14* ⟹
finalB (*cfg*, *ibT*, *ibUT*) ⟷ (*pcOf cfg* = *14* ∨ (*pcOf cfg* = *4* ∧ *ibUT* = *LNil*) ∨
(*pcOf cfg* = *5* ∧ *ibT* = *LNil*))
⟨*proof*⟩

lemma *finalS-cond*:*pcOf cfg* < *14* ⟹ *noMisSpec cfgs* ⟹ *ibT* ≠ *LNil* ⟹ *ibUT*
≠ *LNil* ⟹ ¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-cond'*:*pcOf cfg* < *14* ⟹ *cfgs* = [] ⟹ *ibT* ≠ *LNil* ⟹ *ibUT* ≠
LNil ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-while-spec*:
whileSpeculation cfg (*last cfgs*) ⟹
length cfgs = *Suc 0* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-while-spec-L2*:
pcOf cfg = *7* ⟹
whileSpeculation (*cfgs*!*0*) (*last cfgs*) ⟹
length cfgs = *2* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

lemma *finalS-if-spec*:
(*pcOf* (*last cfgs*) ∈ *inThenIfBeforeOutput* ∧ *pcOf cfg* = *12*) ∨
(*pcOf* (*last cfgs*) ∈ *inElseIf* ∧ *pcOf cfg* = *7*) ⟹
length cfgs = *Suc 0* ⟹
¬ *finalS* (*pstate*, *cfg*, *cfgs*, *ibT*, *ibUT*, *ls*)
⟨*proof*⟩

end

13.2 Proof

theory *Fun6-secure*
imports *Fun6*

begin

definition *common* :: *enat* ⇒ *enat* ⇒ *stateO* ⇒ *stateO* ⇒ *status* ⇒ *stateV* ⇒ *stateV* ⇒ *status* ⇒ *bool*
where
common = (λ*w1 w2*
(*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*pstate3* = *pstate4* ∧
cfg1 = *cfg3* ∧ *cfg2* = *cfg4* ∧
pcOf cfg3 = *pcOf cfg4* ∧ *map pcOf cfgs3* = *map pcOf cfgs4* ∧
pcOf cfg3 ∈ *PC* ∧ *pcOf* ' (*set cfgs3*) ⊆ *PC* ∧
llength ibT1 = ∞ ∧ *llength ibT2* = ∞ ∧
llength ibUT1 = ∞ ∧ *llength ibUT2* = ∞ ∧

ibT1 = *ibT3* ∧ *ibT2* = *ibT4* ∧
ibUT1 = *ibUT3* ∧ *ibUT2* = *ibUT4* ∧

w1 = *w2* ∧
///
array-base aa1 (*getAvstore* (*stateOf cfg3*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa1* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa1* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa1* (*getAvstore* (*stateOf cfg4*))) ∧
array-base aa2 (*getAvstore* (*stateOf cfg3*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*)) ∧
(∀ *cfg3'*∈*set cfgs3*. *array-base aa2* (*getAvstore* (*stateOf cfg3'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg3*))) ∧
(∀ *cfg4'*∈*set cfgs4*. *array-base aa2* (*getAvstore* (*stateOf cfg4'*)) = *array-base aa2* (*getAvstore* (*stateOf cfg4*))) ∧
///
(*statA* = *Diff* ⟶ *statO* = *Diff*) ∧
Dist ls1 ls2 ls3 ls4))

lemma *common-implies*: *common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO ⟹
pcOf cfg1 < *14* ∧ *pcOf cfg2* = *pcOf cfg1* ∧
ibT1 ≠ [[]] ∧ *ibT2* ≠ [[]] ∧

$ibUT1 \neq [[]] \wedge ibUT2 \neq [[]] \wedge$
$w1 = w2$
$\langle proof \rangle$

definition $\Delta 0 :: enat \Rightarrow enat \Rightarrow enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV$
$\Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 0 = (\lambda num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO.$
$(common\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \wedge$
$pcOf\ cfg3 \in beforeWhile \wedge$
$(pcOf\ cfg3 > 1 \longrightarrow same\text{-}var\text{-}o\ tt\ cfg3\ cfgs3\ cfg4\ cfgs4) \wedge$
$(pcOf\ cfg3 > 2 \longrightarrow same\text{-}var\text{-}o\ xx\ cfg3\ cfgs3\ cfg4\ cfgs4) \wedge$
$(pcOf\ cfg3 > 4 \longrightarrow same\text{-}var\text{-}o\ xx\ cfg3\ cfgs3\ cfg4\ cfgs4) \wedge$
$noMisSpec\ cfgs3$
))

lemmas $\Delta 0\text{-}defs = \Delta 0\text{-}def\ common\text{-}def\ PC\text{-}def\ same\text{-}var\text{-}o\text{-}def$
$beforeWhile\text{-}def\ noMisSpec\text{-}def$

lemma $\Delta 0\text{-}implies$: $\Delta 0\ num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$
$(cfg2,ibT2,ibUT2,ls2)$
$statO \Longrightarrow$
$pcOf\ cfg1 < 14 \wedge pcOf\ cfg2 = pcOf\ cfg1 \wedge$
$ibT1 \neq [[]] \wedge ibT2 \neq [[]] \wedge$
$ibUT1 \neq [[]] \wedge ibUT2 \neq [[]] \wedge$
$cfgs4 = []$
$\langle proof \rangle$

definition $\Delta 1 :: enat \Rightarrow enat \Rightarrow enat \Rightarrow stateO \Rightarrow stateO \Rightarrow status \Rightarrow stateV$
$\Rightarrow stateV \Rightarrow status \Rightarrow bool$ **where**
$\Delta 1 = (\lambda num\ w1\ w2\ (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)$
$(pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)$
$statA$
$(cfg1,ibT1,ibUT1,ls1)$

```
    (cfg2,ibT2,ibUT2,ls2)
    statO.
 (common w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO ∧
  pcOf cfg3 ∈ afterWhile ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
  noMisSpec cfgs3
))
lemmas Δ1-defs = Δ1-def common-def noMisSpec-def PC-def afterWhile-def same-var-o-def
lemma Δ1-implies: Δ1 n w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
   (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
   statA
   (cfg1,ibT1,ibUT1,ls1)
   (cfg2,ibT2,ibUT2,ls2)
   statO ⟹
   pcOf cfg3 < 14 ∧ cfgs3 = [] ∧ ibT3 ≠ [[]] ∧
   pcOf cfg4 < 14 ∧ cfgs4 = [] ∧ ibT4 ≠ [[]] ∧
   ibUT3 ≠ [[]] ∧ ibUT4 ≠ [[]]
  ⟨proof⟩



definition Δ1' :: enat ⇒ enat ⇒ enat ⇒ stateO ⇒ stateO  ⇒ status ⇒ stateV
⇒ stateV ⇒ status ⇒ bool where
Δ1' = (λnum w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO.
 (common w1 w2 (pstate3,cfg3,cfgs3,ibT3,ibUT3,ls3)
    (pstate4,cfg4,cfgs4,ibT4,ibUT4,ls4)
    statA
    (cfg1,ibT1,ibUT1,ls1)
    (cfg2,ibT2,ibUT2,ls2)
    statO  ∧
  same-var-o xx cfg3 cfgs3 cfg4 cfgs4 ∧
  whileSpeculation cfg3 (last cfgs3) ∧
  misSpecL1 cfgs3 ∧ misSpecL1 cfgs4 ∧
  w1 = ∞
))

lemmas Δ1'-defs = Δ1'-def common-def PC-def same-var-def
                  startOfIfThen-def startOfElseBranch-def
                  misSpecL1-def whileSpec-defs
```

lemma $\Delta 1'$*-implies*: $\Delta 1'$ *num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg3 $<$ *14* $\wedge$ *pcOf cfg4* $<$ *14* $\wedge$
whileSpeculation cfg3 (*last cfgs3*) $\wedge$
whileSpeculation cfg4 (*last cfgs4*) $\wedge$
length cfgs3 $=$ *Suc 0* $\wedge$ *length cfgs4* $=$ *Suc 0*
⟨*proof*⟩

definition $\Delta 2$ :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
$\Delta 2$ = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$

same-var-o xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$
pcOf cfg3 $=$ *startOfIfThen* $\wedge$ *pcOf* (*last cfgs3*) $\in$ *inElseIf* $\wedge$
misSpecL1 cfgs3 $\wedge$ *misSpecL1 cfgs4* $\wedge$

(*pcOf* (*last cfgs3*) $=$ *startOfElseBranch* $\longrightarrow$ *w1* $=$ ∞) $\wedge$
(*pcOf* (*last cfgs3*) $=$ *3* $\longrightarrow$ *w1* $=$ *3*) $\wedge$

(*pcOf* (*last cfgs3*) $=$ *startOfWhileThen* $\vee$
pcOf (*last cfgs3*) $=$ *whileElse* $\longrightarrow$ *w1* $=$ *1*)

))

lemmas $\Delta 2$*-defs* = $\Delta 2$*-def common-def PC-def same-var-o-def misSpecL1-def*
startOfIfThen-def inElseIf-def same-var-def
startOfWhileThen-def whileElse-def startOfElseBranch-def

lemma $\Delta 2$*-implies*: $\Delta 2$ *num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA

(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf (*last cfgs3*) $\in$ *inElseIf* $\wedge$ *pcOf cfg3* = *7* $\wedge$
pcOf (*last cfgs4*) = *pcOf* (*last cfgs3*) $\wedge$
pcOf cfg4 = *pcOf cfg3* $\wedge$ *length cfgs3* = *Suc 0* $\wedge$
length cfgs4 = *Suc 0* $\wedge$ *same-var xx* (*last cfgs3*) (*last cfgs4*)
⟨*proof*⟩

definition $\Delta 2'$ :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
$\Delta 2'$ = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$
pcOf cfg3 = *startOfIfThen* $\wedge$
whileSpeculation (*cfgs3*!*0*) (*last cfgs3*) $\wedge$
misSpecL2 cfgs3 $\wedge$ *misSpecL2 cfgs4* $\wedge$
w1 = *2*
))

lemmas $\Delta 2'$*-defs* = $\Delta 2'$*-def common-def PC-def same-var-def*
startOfElseBranch-def startOfIfThen-def
whileSpec-defs misSpecL2-def

lemma $\Delta 2'$*-implies*: $\Delta 2'$ *num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf cfg3 = *7* $\wedge$ *pcOf cfg4* = *7* $\wedge$
whileSpeculation (*cfgs3*!*0*) (*last cfgs3*) $\wedge$
whileSpeculation (*cfgs4*!*0*) (*last cfgs4*) $\wedge$
length cfgs3 = *2* $\wedge$ *length cfgs4* = *2*
⟨*proof*⟩

definition $\Delta 3$ :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV*

$\Rightarrow$ *stateV* $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
$\Delta 3$ = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO.
(*common w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\wedge$
same-var-o xx cfg3 cfgs3 cfg4 cfgs4 $\wedge$
pcOf cfg3 = *startOfElseBranch* $\wedge$ *pcOf* (*last cfgs3*) $\in$ *inThenIfBeforeOutput* $\wedge$
misSpecL1 cfgs3 $\wedge$
(*pcOf* (*last cfgs3*) = *7* $\longrightarrow$ *w1* = ∞) $\wedge$
(*pcOf* (*last cfgs3*) = *8* $\longrightarrow$ *w1* = *2*) $\wedge$
(*pcOf* (*last cfgs3*) = *9* $\longrightarrow$ *w1* = *1*)
))

lemmas Δ*3-defs* = Δ*3-def common-def PC-def same-var-o-def*
startOfElseBranch-def inThenIfBeforeOutput-def

lemma Δ*3-implies*: Δ*3 num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ibT3*,*ibUT3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ibT4*,*ibUT4*,*ls4*)
statA
(*cfg1*,*ibT1*,*ibUT1*,*ls1*)
(*cfg2*,*ibT2*,*ibUT2*,*ls2*)
statO $\Longrightarrow$
pcOf (*last cfgs3*) $\in$ *inThenIfBeforeOutput* $\wedge$
pcOf (*last cfgs4*) = *pcOf* (*last cfgs3*) $\wedge$
pcOf cfg3 = *12* $\wedge$ *pcOf cfg3* = *pcOf cfg4* $\wedge$
length cfgs3 = *Suc 0* $\wedge$ *length cfgs4* = *Suc 0*
$\langle$*proof*$\rangle$

definition Δe :: *enat* $\Rightarrow$ *enat* $\Rightarrow$ *enat* $\Rightarrow$ *stateO* $\Rightarrow$ *stateO* $\Rightarrow$ *status* $\Rightarrow$ *stateV* $\Rightarrow$
stateV $\Rightarrow$ *status* $\Rightarrow$ *bool* **where**
Δe = (λ*num w1 w2* (*pstate3*,*cfg3*,*cfgs3*,*ib3*,*ls3*)
(*pstate4*,*cfg4*,*cfgs4*,*ib4*,*ls4*)
statA
(*cfg1*,*ib1*,*ls1*)
(*cfg2*,*ib2*,*ls2*)
statO.
(*pcOf cfg3* = *endPC* $\wedge$ *pcOf cfg4* = *endPC* $\wedge$ *cfgs3* = [] $\wedge$ *cfgs4* = [] $\wedge$
pcOf cfg1 = *endPC* $\wedge$ *pcOf cfg2* = *endPC*))

lemmas $\Delta e\text{-}defs = \Delta e\text{-}def\ common\text{-}def\ endPC\text{-}def$

lemma *init*: $initCond\ \Delta 0$
⟨*proof*⟩

lemma *step0*: $unwindIntoCond\ \Delta 0\ (oor\ \Delta 0\ \Delta 1)$
⟨*proof*⟩

lemma *step1*: $unwindIntoCond\ \Delta 1\ (oor5\ \Delta 1\ \Delta 1'\ \Delta 2\ \Delta 3\ \Delta e)$
⟨*proof*⟩

lemma *step2*: $unwindIntoCond\ \Delta 2\ (oor3\ \Delta 2\ \Delta 2'\ \Delta 1)$
⟨*proof*⟩

lemma *step3*: $unwindIntoCond\ \Delta 3\ (oor\ \Delta 3\ \Delta 1)$
⟨*proof*⟩

lemma *step4*: $unwindIntoCond\ \Delta 1'\ \Delta 1$
⟨*proof*⟩

lemma *step5*: $unwindIntoCond\ \Delta 2'\ \Delta 2$
⟨*proof*⟩

lemma *stepe*: $unwindIntoCond\ \Delta e\ \Delta e$
⟨*proof*⟩

lemmas $theConds = step0\ step1\ step2\ step3\ step4\ step5\ stepe$

proposition *lrsecure*
⟨*proof*⟩

end
[3] [1]

References

[1] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *CSF*. IEEE, 2019, pp. 288–303. [Online]. Available: https://doi.org/10.1109/CSF.2019.00027

[2] B. Dongol, M. Griffin, A. Popescu, and J. Wright, "Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities," in *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2024, pp. 409–424. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSF61375.2024.00027

[3] T. Nipkow and G. Klein, *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.