

An Isabelle/HOL formalization of Strong Security

Sylvia Grewe, Alexander Lux, Heiko Mantel, Jens Sauer

November 18, 2018

Abstract

Research in information-flow security aims at developing methods to identify undesired information leaks within programs from private sources to public sinks. Noninterference captures this intuition. Strong security from [2] formalizes noninterference for concurrent systems.

We present an Isabelle/HOL formalization of strong security for arbitrary security lattices ([2] uses a two-element security lattice). The formalization includes compositionality proofs for strong security and a soundness proof for a security type system that checks strong security for programs in a simple while language with dynamic thread creation.

Our formalization of the security type system is abstract in the language for expressions and in the semantic side conditions for expressions. It can easily be instantiated with different syntactic approximations for these side conditions. The soundness proof of such an instantiation boils down to showing that these syntactic approximations imply the semantic side conditions.

Contents

1	Preliminary definitions	2
1.1	Type synonyms	2
2	Strong security	4
2.1	Definition of strong security	4
2.2	Proof technique for compositionality results	6
2.3	Proof of parallel compositionality	7
3	Example language and compositionality proofs	8
3.1	Example language with dynamic thread creation	8
3.2	Proofs of atomic compositionality results	9
3.3	Proofs of non-atomic compositionality results	11

4	Security type system	12
4.1	Abstract security type system with soundness proof	12
4.2	Example language for Boolean and arithmetic expressions . . .	13
4.3	Example interpretation of abstract security type system . . .	14

1 Preliminary definitions

1.1 Type synonyms

The formalization is parametric in different aspects. Notably, it is parametric in the security lattice it supports.

For better readability, we use the following type synonyms in our formalization:

```
theory Types
imports Main
begin
```

- type parameters:
- 'exp: expressions (arithmetic, boolean...)
- 'val: values
- 'id: identifier names
- 'com: commands
- 'd: domains

This is a collection of type synonyms. Note that not all of these type synonyms are used within Strong-Security - some are used in WHATandWHERE-Security.

— type for memory states - map ids to values
type-synonym ('id, 'val) *State* = 'id \Rightarrow 'val

— type for evaluation functions mapping expressions to a values depending on a state
type-synonym ('exp, 'id, 'val) *Evalfunction* =
'exp \Rightarrow ('id, 'val) *State* \Rightarrow 'val

— define configurations with threads as pair of commands and states
type-synonym ('id, 'val, 'com) *TConfig* = 'com \times ('id, 'val) *State*

— define configurations with thread pools as pair of command lists (thread pool) and states
type-synonym ('id, 'val, 'com) *TPConfig* =
('com *list*) \times ('id, 'val) *State*

— type for program states (including the set of commands and a symbol for terminating - None)
type-synonym 'com *ProgramState* = 'com *option*

— type for configurations with program states
type-synonym (*'id, 'val, 'com*) *PConfig* =
'com ProgramState × (*'id, 'val*) *State*

— type for labels with a list of spawned threads
type-synonym *'com Label* = *'com list*

— type for step relations from single commands to a program state, with a label
type-synonym (*'exp, 'id, 'val, 'com*) *TLSteps* =
((*'id, 'val, 'com*) *TConfig* × *'com Label*
× (*'id, 'val, 'com*) *PConfig*) *set*

— curried version of previously defined type
type-synonym (*'exp, 'id, 'val, 'com*) *TLSteps-curry* =
'com ⇒ (*'id, 'val*) *State* ⇒ *'com Label* ⇒ *'com ProgramState*
⇒ (*'id, 'val*) *State* ⇒ *bool*

— type for step relations from thread pools to thread pools
type-synonym (*'exp, 'id, 'val, 'com*) *TPSteps* =
((*'id, 'val, 'com*) *TPConfig* × (*'id, 'val, 'com*) *TPConfig*) *set*

— curried version of previously defined type
type-synonym (*'exp, 'id, 'val, 'com*) *TPSteps-curry* =
'com list ⇒ (*'id, 'val*) *State* ⇒ *'com list* ⇒ (*'id, 'val*) *State* ⇒ *bool*

— define type of step relations for single threads to thread pools
type-synonym (*'exp, 'id, 'val, 'com*) *TSteps* =
((*'id, 'val, 'com*) *TConfig* × (*'id, 'val, 'com*) *TPConfig*) *set*

— define the same type as TSteps, but in a curried version (allowing syntax abbreviations)
type-synonym (*'exp, 'id, 'val, 'com*) *TSteps-curry* =
'com ⇒ (*'id, 'val*) *State* ⇒ *'com list* ⇒ (*'id, 'val*) *State* ⇒ *bool*

— type for simple domain assignments; 'd has to be an instance of order (partial order)
type-synonym (*'id, 'd*) *DomainAssignment* = *'id* ⇒ *'d::order*

type-synonym *'com Bisimulation-type* = ((*'com list*) × (*'com list*)) *set*

— type for escape hatches
type-synonym (*'d, 'exp*) *Hatch* = *'d* × *'exp*

— type for sets of escape hatches
type-synonym (*'d, 'exp*) *Hatches* = ((*'d, 'exp*) *Hatch*) *set*

— type for local escape hatches
type-synonym (*'d, 'exp*) *lHatch* = *'d* × *'exp* × *nat*

— type for sets of local escape hatches
type-synonym (*'d*, *'exp*) *lHatches* = ((*'d*, *'exp*) *lHatch*) *set*

end

2 Strong security

2.1 Definition of strong security

We define strong security such that it is parametric in a security lattice (*'d*). The definition of strong security by itself is language-independent, therefore the definition is parametric in a programming language (*'com*) in addition.

theory *Strong-Security*
imports *Types*
begin

locale *Strong-Security* =
fixes *SR* :: (*'exp*, *'id*, *'val*, *'com*) *TSteps*
and *DA* :: (*'id*, *'d::order*) *DomainAssignment*

begin

— define when two states are indistinguishable for an observer on domain *d*

definition *d-equal* :: *'d::order* \Rightarrow (*'id*, *'val*) *State*
 \Rightarrow (*'id*, *'val*) *State* \Rightarrow *bool*

where

d-equal *d m m'* $\equiv \forall x. ((DA\ x) \leq d \longrightarrow (m\ x) = (m'\ x))$

abbreviation *d-equal'* :: (*'id*, *'val*) *State*
 \Rightarrow *'d::order* \Rightarrow (*'id*, *'val*) *State* \Rightarrow *bool*
 ((- =_ -))

where

m =_d m' \equiv *d-equal* *d m m'*

— transitivity of *d*-equality

lemma *d-equal-trans*:

$\llbracket m =_d m'; m' =_d m'' \rrbracket \Longrightarrow m =_d m''$
<proof>

abbreviation *SRabbr* :: (*'exp*, *'id*, *'val*, *'com*) *TSteps-curry*
 ((*1*<-,->) \rightarrow / (*1*<-,->) [*0,0,0,0*] *81*)

where

<c,m> \rightarrow *<c',m'>* $\equiv ((c,m),(c',m')) \in SR$

— predicate for strong d-bisimulation

definition *Strong-d-Bisimulation* $:: 'd \Rightarrow 'com \text{ Bisimulation-type} \Rightarrow bool$

where

Strong-d-Bisimulation $d \ R \equiv$

$$\begin{aligned} & (sym \ R) \wedge \\ & (\forall (V, V') \in R. \ length \ V = \ length \ V') \wedge \\ & (\forall (V, V') \in R. \ \forall i < \ length \ V. \ \forall m1 \ m1' \ m2 \ W. \\ & \langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle \wedge m1 =_d m1' \\ & \rightarrow (\exists W' \ m2'. \ \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle \wedge (W, W') \in R \wedge m2 =_d m2')) \end{aligned}$$

— union of all strong d-bisimulations

definition *USdB* $:: 'd \Rightarrow 'com \text{ Bisimulation-type}$

(\approx_d 65)

where

$$\approx_d \equiv \bigcup \{r. \ (Strong-d-Bisimulation \ d \ r)\}$$

abbreviation *relatedbyUSdB* $:: 'com \ list \Rightarrow 'd \Rightarrow 'com \ list \Rightarrow bool$

(($- \approx_d -$) [66,66] 65)

where $V \approx_d V' \equiv (V, V') \in USdB \ d$

— predicate to define when a program is strongly secure

definition *Strongly-Secure* $:: 'com \ list \Rightarrow bool$

where

$$Strongly-Secure \ V \equiv (\forall d. \ V \approx_d \ V)$$

— auxiliary lemma to obtain central strong d-Bisimulation property as Lemma in meta logic (allows instantiating all the variables manually if necessary)

lemma *strongdB-aux*: $\bigwedge V \ V' \ m1 \ m1' \ m2 \ W \ i. \ \llbracket Strong-d-Bisimulation \ d \ R;$

$i < \ length \ V ; (V, V') \in R; \langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle; \ m1 =_d \ m1' \rrbracket$

$\implies (\exists W' \ m2'. \ \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle \wedge (W, W') \in R \wedge m2 =_d m2')$

<proof>

lemma *trivialpair-in-USdB*:

$\llbracket \rrbracket \approx_d \llbracket \rrbracket$

<proof>

lemma *USdBsym*: $sym \ (\approx_d)$

<proof>

lemma *USdBqlen*:

$V \approx_d \ V' \implies \ length \ V = \ length \ V'$

<proof>

lemma *USdB-Strong-d-Bisimulation*:

Strong-d-Bisimulation $d \ (\approx_d)$

<proof>

lemma *USdBtrans: trans* (\approx_d)
 $\langle proof \rangle$

end

end

2.2 Proof technique for compositionality results

For proving compositionality results for strong security, we formalize the following “up-to technique” and prove it sound:

theory *Up-To-Technique*
imports *Strong-Security*
begin

context *Strong-Security*
begin

— define d-bisimulation ‘up to’ union of strong d-Bisimulations

definition *d-Bisimulation-Up-To-USdB* ::

$'d \Rightarrow 'com$ *Bisimulation-type* $\Rightarrow bool$

where

d-Bisimulation-Up-To-USdB $d R \equiv$

$(sym R) \wedge (\forall (V, V') \in R. length V = length V') \wedge$
 $(\forall (V, V') \in R. \forall i < length V. \forall m1 m1' W m2.$
 $\langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle \wedge (m1 =_d m1') \rightarrow$
 $(\exists W' m2'. \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle$
 $\wedge (W, W') \in (R \cup (\approx_d)) \wedge (m2 =_d m2')))$

lemma *UpTo-aux*: $\bigwedge V V' m1 m1' m2 W i. \llbracket d\text{-Bisimulation-Up-To-USdB } d R;$
 $i < length V; (V, V') \in R; \langle V!i, m1 \rangle \rightarrow \langle W, m2 \rangle; m1 =_d m1' \rrbracket$
 $\implies (\exists W' m2'. \langle V!i, m1' \rangle \rightarrow \langle W', m2' \rangle$
 $\wedge (W, W') \in (R \cup (\approx_d)) \wedge (m2 =_d m2'))$
 $\langle proof \rangle$

lemma *RuUSdBeglen*:

$\llbracket d\text{-Bisimulation-Up-To-USdB } d R;$
 $(V, V') \in (R \cup (\approx_d)) \rrbracket$
 $\implies length V = length V'$
 $\langle proof \rangle$

lemma *Up-To-Technique*:

assumes *upToR*: *d-Bisimulation-Up-To-USdB* $d R$
shows $R \subseteq \approx_d$
 $\langle proof \rangle$

end

end

2.3 Proof of parallel compositionality

We prove that strong security is preserved under composition of strongly secure threads.

theory *Parallel-Composition*
imports *Up-To-Technique*
begin

context *Strong-Security*
begin

theorem *parallel-composition*:
 assumes *eqlen*: $\text{length } V = \text{length } V'$
 assumes *partsrelated*: $\forall i < \text{length } V. [V!i] \approx_d [V'!i]$
 shows $V \approx_d V'$
<proof>

lemma *parallel-decomposition*:
 assumes *related*: $V \approx_d V'$
 shows $\forall i < \text{length } V. [V!i] \approx_d [V'!i]$
<proof>

lemma *USdB-comp-head-tail*:
 assumes *relatedhead*: $[c] \approx_d [c']$
 assumes *relatedtail*: $V \approx_d V'$
 shows $(c\#V) \approx_d (c'\#V')$
<proof>

lemma *USdB-decomp-head-tail*:
 assumes *relatedlist*: $(c\#V) \approx_d (c'\#V')$
 shows $[c] \approx_d [c'] \wedge V \approx_d V'$
<proof>

end

end

3 Example language and compositionality proofs

3.1 Example language with dynamic thread creation

As in [2], we instantiate the language with a simple while language that supports dynamic thread creation via a fork command (Multi-threaded While Language with fork, MWLf). Note that the language is still parametric in the language used for Boolean and arithmetic expressions ($'exp$).

```
theory MWLf
imports Types
begin
```

— SYNTAX

— Commands for the multi-threaded while language with fork (to instantiate $'com$)

```
datatype ('exp, 'id) MWLfCom
  = Skip (skip)
  | Assign 'id 'exp
    (:-= [70,70] 70)

  | Seq ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
    (-;- [61,60] 60)

  | If-Else 'exp ('exp, 'id) MWLfCom ('exp, 'id) MWLfCom
    (if - then - else - fi [80,79,79] 70)

  | While-Do 'exp ('exp, 'id) MWLfCom
    (while - do - od [80,79] 70)

  | Fork ('exp, 'id) MWLfCom (('exp, 'id) MWLfCom) list
    (fork - - [70,70] 70)
```

— SEMANTICS

```
locale MWLf-semantics =
fixes E :: ('exp, 'id, 'val) Evalfunction
and BMap :: 'val  $\Rightarrow$  bool
begin
```

— steps semantics, set of deterministic steps from single threads to either single threads or thread pools

inductive-set

```
MWLFSteps-det :: ('exp, 'id, 'val, ('exp, 'id) MWLfCom) TSteps
and MWLFSteps-det' :: ('exp, 'id, 'val, ('exp, 'id) MWLfCom) TSteps-curry
  ((1<-,->)  $\rightarrow$  / (1<-,->) [0,0,0,0] 81)
where
  <c1,m1>  $\rightarrow$  <c2,m2>  $\equiv$  ((c1,m1),(c2,m2))  $\in$  MWLFSteps-det |
  skip: <skip,m>  $\rightarrow$  <[],m> |
```


assign: $\langle E e m \rangle = v \implies \langle x := e, m \rangle \rightarrow \langle [], m(x := v) \rangle \mid$
seq1: $\langle c1, m \rangle \rightarrow \langle [], m^\wedge \rangle \implies \langle c1; c2, m \rangle \rightarrow \langle [c2], m^\wedge \rangle \mid$
seq2: $\langle c1, m \rangle \rightarrow \langle c1 \# V, m^\wedge \rangle \implies \langle c1; c2, m \rangle \rightarrow \langle (c1'; c2) \# V, m^\wedge \rangle \mid$
iftrue: $BMap (E b m) = True \implies$
 $\langle \text{if } b \text{ then } c1 \text{ else } c2 \text{ fi}, m \rangle \rightarrow \langle [c1], m \rangle \mid$
iffalse: $BMap (E b m) = False \implies$
 $\langle \text{if } b \text{ then } c1 \text{ else } c2 \text{ fi}, m \rangle \rightarrow \langle [c2], m \rangle \mid$
whiletrue: $BMap (E b m) = True \implies$
 $\langle \text{while } b \text{ do } c \text{ od}, m \rangle \rightarrow \langle [c; (\text{while } b \text{ do } c \text{ od})], m \rangle \mid$
whilefalse: $BMap (E b m) = False \implies$
 $\langle \text{while } b \text{ do } c \text{ od}, m \rangle \rightarrow \langle [], m \rangle \mid$
fork: $\langle \text{fork } c \ V, m \rangle \rightarrow \langle c \# V, m \rangle$

inductive-cases *MWLFSteps-det-cases*:

$\langle \text{skip}, m \rangle \rightarrow \langle W, m^\wedge \rangle$
 $\langle x := e, m \rangle \rightarrow \langle W, m^\wedge \rangle$
 $\langle c1; c2, m \rangle \rightarrow \langle W, m^\wedge \rangle$
 $\langle \text{if } b \text{ then } c1 \text{ else } c2 \text{ fi}, m \rangle \rightarrow \langle W, m^\wedge \rangle$
 $\langle \text{while } b \text{ do } c \text{ od}, m \rangle \rightarrow \langle W, m^\wedge \rangle$
 $\langle \text{fork } c \ V, m \rangle \rightarrow \langle W, m^\wedge \rangle$

— non-deterministic, possibilistic system step (added for intuition, not used in the proofs)

inductive-set

MWLFSteps-ndet :: ('exp, 'id, 'val, ('exp, 'id) MWLFCom) TPSteps
and *MWLFSteps-ndet'* :: ('exp, 'id, 'val, ('exp, 'id) MWLFCom) TPSteps-curry
 $((1 \langle -, - \rangle) \Rightarrow / (1 \langle -, - \rangle) [0, 0, 0, 0] \ 81)$

where

$\langle V1, m1 \rangle \Rightarrow \langle V2, m2 \rangle \equiv ((V1, m1), (V2, m2)) \in MWLFSteps-ndet \mid$
 $\langle ci, m \rangle \rightarrow \langle c, m^\wedge \rangle \implies \langle Vf \ @ \ [ci] \ @ \ Va, m \rangle \Rightarrow \langle Vf \ @ \ c \ @ \ Va, m^\wedge \rangle$

end

end

3.2 Proofs of atomic compositionality results

We prove for each atomic command of our example programming language (i.e. a command that is not composed out of other commands) that it is strongly secure if the expressions involved are indistinguishable for an observer on security level d .

theory *Strongly-Secure-Skip-Assign*
imports *MWLF Parallel-Composition*
begin

locale *Strongly-Secure-Programs* =

$L?$: MWLf-*semantics* E $BMap$
 + $SS?$: *Strong-Security* MWLfSteps-det DA
for E :: ('exp, 'id, 'val) *Evalfunction*
and $BMap$:: 'val \Rightarrow bool
and DA :: ('id, 'd::order) *DomainAssignment*
begin

abbreviation $USdBname$:: 'd \Rightarrow ('exp, 'id) MWLfCom *Bisimulation-type*
 (\approx_-)

where $\approx_d \equiv USdB\ d$

abbreviation $relatedbyUSdB$:: ('exp, 'id) MWLfCom list \Rightarrow 'd
 \Rightarrow ('exp, 'id) MWLfCom list \Rightarrow bool (**infixr** \approx_- 65)

where $V \approx_d V' \equiv (V, V') \in USdB\ d$

— define when two expressions are indistinguishable with respect to a domain d

definition d -indistinguishable :: 'd::order \Rightarrow 'exp \Rightarrow 'exp \Rightarrow bool

where

d -indistinguishable $d\ e1\ e2 \equiv$
 $\forall m\ m'. ((m =_d m') \longrightarrow ((E\ e1\ m) = (E\ e2\ m')))$

abbreviation d -indistinguishable' :: 'exp \Rightarrow 'd::order \Rightarrow 'exp \Rightarrow bool
 $((- \equiv_- -))$

where

$e1 \equiv_d e2 \equiv d$ -indistinguishable $d\ e1\ e2$

— symmetry of d -indistinguishable

lemma d -indistinguishable-sym:

$e \equiv_d e' \Longrightarrow e' \equiv_d e$

$\langle proof \rangle$

lemma d -indistinguishable-trans:

$\llbracket e \equiv_d e'; e' \equiv_d e'' \rrbracket \Longrightarrow e \equiv_d e''$

$\langle proof \rangle$

theorem *Strongly-Secure-Skip*:

$[skip] \approx_d [skip]$

$\langle proof \rangle$

theorem *Strongly-Secure-Assign*:

assumes d -indistinguishable-exp: $e \equiv_{DA}\ x\ e'$

shows $[x := e] \approx_d [x := e']$

$\langle proof \rangle$

end

end

3.3 Proofs of non-atomic compositionality results

We prove compositionality results for each non-atomic command of our example programming language (i.e. a command that is composed out of other commands): If the components are strongly secure and the expressions involved indistinguishable for an observer on security level d , then the composed command is also strongly secure.

```
theory Language-Composition
imports Strongly-Secure-Skip-Assign
begin

context Strongly-Secure-Programs
begin

theorem Compositionality-Seq:
  assumes relatedpart1:  $[c1] \approx_d [c1']$ 
  assumes relatedpart2:  $[c2] \approx_d [c2']$ 
  shows  $[c1; c2] \approx_d [c1'; c2']$ 
  <proof>

theorem Compositionality-Fork:
  fixes  $V :: ('exp, 'id) MWLfCom\ list$ 
  assumes relatedmain:  $[c] \approx_d [c']$ 
  assumes relatedthreads:  $V \approx_d V'$ 
  shows  $[fork\ c\ V] \approx_d [fork\ c'\ V']$ 
  <proof>

theorem Compositionality-If:
  assumes dind-or-branchesrelated:
     $b \equiv_d b' \vee [c1] \approx_d [c2] \vee [c1'] \approx_d [c2']$ 
  assumes branch1related:  $[c1] \approx_d [c1']$ 
  assumes branch2related:  $[c2] \approx_d [c2']$ 
  shows  $[if\ b\ then\ c1\ else\ c2\ fi] \approx_d [if\ b'\ then\ c1'\ else\ c2'\ fi]$ 
  <proof>

theorem Compositionality-While:
  assumes dind:  $b \equiv_d b'$ 
  assumes bodyrelated:  $[c] \approx_d [c']$ 
  shows  $[while\ b\ do\ c\ od] \approx_d [while\ b'\ do\ c'\ od]$ 
  <proof>

end

end
```

4 Security type system

4.1 Abstract security type system with soundness proof

We formalize an abstract version of the type system in [2] using locales [1]. Our formalization of the type system is abstract in the sense that the rules specify abstract semantic side conditions on the expressions within a command that satisfy for proving the soundness of the rules. That is, it can be instantiated with different syntactic approximations for these semantic side conditions in order to achieve a type system for a concrete language for Boolean and arithmetic expressions. Obtaining a soundness proof for such a concrete type system then boils down to proving that the concrete type system interprets the abstract type system.

We prove the soundness of the abstract type system by simply applying the compositionality results proven before.

```
theory Type-System
imports Language-Composition
begin

locale Type-System =
  SSP? : Strongly-Secure-Programs E BMap DA
  for E :: ('exp, 'id, 'val) Evalfunction
  and BMap :: 'val  $\Rightarrow$  bool
  and DA :: ('id, 'd::order) DomainAssignment
+
fixes
AssignSideCondition :: 'id  $\Rightarrow$  'exp  $\Rightarrow$  bool
and WhileSideCondition :: 'exp  $\Rightarrow$  bool
and IfSideCondition ::
  'exp  $\Rightarrow$  ('exp, 'id) MWLfCom  $\Rightarrow$  ('exp, 'id) MWLfCom  $\Rightarrow$  bool
assumes semAssignSC: AssignSideCondition x e  $\Longrightarrow$  e  $\equiv_{DA}$  x e
and semWhileSC: WhileSideCondition e  $\Longrightarrow$   $\forall d. e \equiv_d e$ 
and semIfSC: IfSideCondition e c1 c2  $\Longrightarrow$   $\forall d. e \equiv_d e \vee [c1] \approx_d [c2]$ 
begin
```

— Security typing rules for the language commands

```
inductive
ComSecTyping :: ('exp, 'id) MWLfCom  $\Rightarrow$  bool
  ( $\vdash_C$  -)
and ComSecTypingL :: ('exp, 'id) MWLfCom list  $\Rightarrow$  bool
  ( $\vdash_V$  -)
where
skip:  $\vdash_C$  skip |
Assign:  $\llbracket$  AssignSideCondition x e  $\rrbracket \Longrightarrow \vdash_C$  x := e |
Fork:  $\llbracket \vdash_C$  c;  $\vdash_V$  V  $\rrbracket \Longrightarrow \vdash_C$  fork c V |
Seq:  $\llbracket \vdash_C$  c1;  $\vdash_C$  c2  $\rrbracket \Longrightarrow \vdash_C$  c1;c2 |
While:  $\llbracket \vdash_C$  c; WhileSideCondition b  $\rrbracket$ 
```

$\implies \vdash_{\mathcal{C}} \text{while } b \text{ do } c \text{ od} \mid$
If: $\llbracket \vdash_{\mathcal{C}} c1; \vdash_{\mathcal{C}} c2; \text{IfSideCondition } b \text{ } c1 \text{ } c2 \rrbracket$
 $\implies \vdash_{\mathcal{C}} \text{if } b \text{ then } c1 \text{ else } c2 \text{ fi} \mid$
Parallel: $\llbracket \forall i < \text{length } V. \vdash_{\mathcal{C}} V!i \rrbracket \implies \vdash_{\mathcal{V}} V$

inductive-cases *parallel-cases*:

$\vdash_{\mathcal{V}} V$

— soundness proof of abstract type system

theorem *ComSecTyping-single-is-sound*:

$\vdash_{\mathcal{C}} c \implies \text{Strongly-Secure } [c]$

<proof>

theorem *ComSecTyping-list-is-sound*:

$\vdash_{\mathcal{V}} V \implies \text{Strongly-Secure } V$

<proof>

end

end

4.2 Example language for Boolean and arithmetic expressions

As an example, we provide a simple example language for instantiating the parameter *'exp* for the language for Boolean and arithmetic expressions.

theory *Expr*

imports *Types*

begin

— type parameters:

— *'val*: numbers, boolean constants....

— *'id*: identifier names

type-synonym (*'val*) *operation* = *'val list* \Rightarrow *'val*

datatype (*dead 'id*, *dead 'val*) *Expr* =

Const 'val |

Var 'id |

Op 'val operation (('id, 'val) Expr) list

— defining a simple recursive evaluation function on this datatype

primrec *ExprEval* :: ((*'id*, *'val*) *Expr*, *'id*, *'val*) *Evalfunction*

and *ExprEvalL* :: ((*'id*, *'val*) *Expr*) *list* \Rightarrow (*'id*, *'val*) *State* \Rightarrow *'val list*

where

ExprEval (*Const v*) *m* = *v* |

```

ExprEval (Var x) m = (m x) |
ExprEval (Op f arglist) m = (f (ExprEvalL arglist m)) |

ExprEvalL [] m = [] |
ExprEvalL (e#V) m = (ExprEval e m)#(ExprEvalL V m)

```

end

4.3 Example interpretation of abstract security type system

Using the example instantiation of the language for Boolean and arithmetic expressions, we give an example instantiation of our abstract security type system, instantiating the parameter for domains 'd' with a two-level security lattice.

```

theory Domain-example
imports Expr
begin

```

— When interpreting, we have to instantiate the type for domains. As an example, we take a type containing 'low' and 'high' as domains.

```

datatype Dom = low | high

```

```

instantiation Dom :: order
begin

```

```

definition
less-eq-Dom-def: d1 ≤ d2 = (if d1 = d2 then True
  else (if d1 = low then True else False))

```

```

definition
less-Dom-def: d1 < d2 = (if d1 = d2 then False
  else (if d1 = low then True else False))

```

```

instance <proof>

```

end

end

```

theory Type-System-example
imports Type-System Expr Domain-example
begin

```

— When interpreting, we have to instantiate the type for domains.
 — As an example, we take a type containing 'low' and 'high' as domains.

consts $DA :: ('id, Dom) DomainAssignment$

consts $BMap :: 'val \Rightarrow bool$

abbreviation $d\text{-indistinguishable}' :: ('id, 'val) Expr \Rightarrow Dom$

$\Rightarrow ('id, 'val) Expr \Rightarrow bool$

$((- \equiv_d -))$

where

$e1 \equiv_d e2$

$\equiv Strongly\text{-Secure}\text{-Programs}.d\text{-indistinguishable ExprEval DA d e1 e2$

abbreviation $relatedbyUSdB' :: (('id, 'val) Expr, 'id) MWLfCom list$

$\Rightarrow Dom \Rightarrow (('id, 'val) Expr, 'id) MWLfCom list \Rightarrow bool$ (**infixr** \approx_d 65)

where $V \approx_d V' \equiv (V, V') \in Strong\text{-Security}.USdB$

$(MWLf\text{-semantics}.MWLfSteps\text{-det ExprEval BMap) DA d$

— Security typing rules for expressions - will be part of a side condition

inductive

$ExprSecTyping :: ('id, 'val) Expr \Rightarrow Dom set \Rightarrow bool$

$(\vdash_{\mathcal{E}} - : -)$

where

$Consts: \vdash_{\mathcal{E}} (Const v) : \{d\} |$

$Vars: \vdash_{\mathcal{E}} (Var x) : \{DA x\} |$

$Ops: \forall i < length arglist. \vdash_{\mathcal{E}} (arglist!i) : (d!!i)$

$\implies \vdash_{\mathcal{E}} (Op f arglist) : (\bigcup \{d. (\exists i < length arglist. d = (d!!i))\})$

definition $synAssignSC :: 'id \Rightarrow ('id, 'val) Expr \Rightarrow bool$

where

$synAssignSC x e \equiv \exists D. (\vdash_{\mathcal{E}} e : D \wedge (\forall d \in D. (d \leq DA x)))$

definition $synWhileSC :: ('id, 'val) Expr \Rightarrow bool$

where

$synWhileSC e \equiv \exists D. (\vdash_{\mathcal{E}} e : D \wedge (\forall d \in D. \forall d'. d \leq d'))$

definition $synIfSC :: ('id, 'val) Expr \Rightarrow (('id, 'val) Expr, 'id) MWLfCom$

$\Rightarrow (('id, 'val) Expr, 'id) MWLfCom \Rightarrow bool$

where

$synIfSC e c1 c2 \equiv$

$\forall d. (\neg (e \equiv_d e) \longrightarrow [c1] \approx_d [c2])$

lemma $ExprTypable\text{-with-smallerD}\text{-implies-d-indistinguishable}$:

$\llbracket \vdash_{\mathcal{E}} e : D'; \forall d' \in D'. d' \leq d \rrbracket \implies e \equiv_d e$

$\langle proof \rangle$

interpretation $Type\text{-System-example}$: $Type\text{-System ExprEval BMap DA$

$synAssignSC synWhileSC synIfSC$

$\langle proof \rangle$

end

References

- [1] C. Ballarin. Locales and Locale Expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2003.
- [2] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop, 2000. CSFW-13. Proceedings. 13th IEEE*, pages 200–214. IEEE, 2000.