

CoSMed: A confidentiality-verified social media platform

Thomas Bauereiss Andrei Popescu

February 6, 2026

Abstract

This entry contains the confidentiality verification of the (functional kernel of) the CoSMed social media platform. The confidentiality properties are formalized as instances of BD Security [4, 5]. An innovation in the deployment of BD Security compared to previous work is the use of dynamic declassification triggers, incorporated as part of inductive bounds, for providing stronger guarantees that account for the repeated opening and closing of access windows. To further strengthen the confidentiality guarantees, we also prove “traceback” properties about the accessibility decisions affecting the information managed by the system.

Contents

1	Introduction	2
2	Preliminaries	3
2.1	The basic types	3
2.2	Identifiers	5
3	System specification	6
3.1	The state	6
3.2	The actions	6
3.2.1	Initialization of the system	6
3.2.2	Starting action	7
3.2.3	Creation actions	7
3.2.4	Updating actions	9
3.2.5	Deletion (removal) actions	10
3.2.6	Reading actions	10
3.2.7	Listing actions	12
3.3	The step function	13
3.4	Code generation	18

4	Safety properties	19
5	The observation setup	21
6	Post confidentiality	21
6.1	Preliminaries	22
6.2	Value Setup	25
6.3	Declassification bound	27
6.4	Unwinding proof	28
7	Friendship status confidentiality	30
7.1	Preliminaries	31
7.2	Value Setup	35
7.3	Declassification bound	39
7.4	Unwinding proof	40
8	Friendship request confidentiality	42
8.1	Preliminaries	43
8.2	Value Setup	47
8.3	Declassification bound	53
8.4	Unwinding proof	55
9	Traceback Properties	59
9.1	Tracing Back Post Visibility Status	59
9.2	Tracing Back Friendship Status	63

1 Introduction

CoSMed [1, 2] is a minimalistic social media platform where users can register, create posts and establish friendship relationships. This document presents the formulation and proof of confidentiality properties about posts, friendship relationships, and friendship requests.

After this introduction and a section on technical preliminaries, this document presents the specification of the CoSMed system, as an input/output (I/O) automaton. Next is a section on proved safety properties about the system (invariants) that are needed in the proofs of confidentiality.

The confidentiality properties of CoSMed are expressed as instances of BD Security [4], a general confidentiality verification framework that has been formalized in the AFP entry [5]. They cover confidentiality aspects about:

- posts
- friendship status (whether or not two users are friends)

- friendship request status (whether or not a user has submitted a friendship request to another user)

Each of these types of confidentiality properties have dedicated sections (and corresponding folders in the formalization) with self-explanatory names. BD Security is defined in terms of an observation infrastructure, a secrecy infrastructure, a declassification trigger and a declassification bound. The observations are always given by an arbitrary set of users (which is fixed in the “Observation Setup” section). In each case, the declassification trigger is vacuously false, since we use dynamic triggers which are made part of the inductive definition of bounds. [1, Section 3.3] explains dynamic triggers in detail. The secrets (called “values” in this formalization) and the declassification bounds (which relate indistinguishable secrets) are specific to each property.

The proofs proceed using the method of BD Security unwinding, which is part of the AFP entry on BD Security [5] and is described in detail in [6, Section 4.1] and [4, Section 2.6]. For managing proof complexity, we take a modular approach, building several unwinding relations that are connected in a sequence and also have an exit point into error components. This approach is presented in [6] as Corollary 6 (Sequential Unwinding Theorem) and in [4] as Theorem 4 (Sequential Multiplex Unwinding Theorem).

The last section formalizes what we call *traceback properties*.¹ These are natural “supplements” that strengthen the confidentiality guarantees. Indeed, confidentiality (in its BD security formulation) states: Unless a user acquires such role or a document becomes public, that user cannot learn such information. But can a user not forge the acquisition of that role or maliciously determine the publication of the document? Traceback properties show that this is not possible, except by identity theft. [1, Section 5.2] explains traceback properties (called there “accountability properties”) in detail.

2 Preliminaries

```
theory Prelim
  imports
    Bounded-Deducibility-Security.Compositional-Reasoning
    Fresh-Identifiers.Fresh-String
begin
```

2.1 The basic types

```
definition emptyStr = STR ""
```

¹In previous work, we called these types of properties *accountability properties* [1, 2] or *forensic properties* [3]. The *traceback properties* terminology is used in [6].

```

datatype name = Nam String.literal
definition emptyName ≡ Nam emptyStr
datatype inform = Info String.literal
definition emptyInfo ≡ Info emptyStr

datatype user = Usr (nameUser : name) (infoUser : inform)
definition emptyUser ≡ Usr emptyName emptyInfo
fun niUser where niUser (Usr name info) = (name,info)

```

```

typedecl raw-data
code-printing type-constructor raw-data → (Scala) java.io.File

```

```

datatype img = emptyImg | Imag raw-data

```

```

datatype vis = Vsb String.literal

```

```

abbreviation FriendV ≡ Vsb (STR "friend")
abbreviation PublicV ≡ Vsb (STR "public")
fun stringOfVis where stringOfVis (Vsb str) = str

```

```

datatype title = Tit String.literal
definition emptyTitle ≡ Tit emptyStr
datatype text = Txt String.literal
definition emptyText ≡ Txt emptyStr

```

```

datatype post = Ntc (titlePost : title) (textPost : text) (imgPost : img)

```

```

fun setTitlePost where setTitlePost (Ntc title text img) title' = Ntc title' text img
fun setTextPost where setTextPost (Ntc title text img) text' = Ntc title text' img
fun setImgPost where setImgPost (Ntc title text img) img' = Ntc title text img'

```

```

definition emptyPost :: post where
emptyPost ≡ Ntc emptyTitle emptyText emptyImg

```

```

lemma set-get-post[simp]:
titlePost (setTitlePost ntc title) = title
titlePost (setTextPost ntc text) = titlePost ntc
titlePost (setImgPost ntc img) = titlePost ntc

```

```

textPost (setTitlePost ntc title) = textPost ntc
textPost (setTextPost ntc text) = text

```

textPost (*setImgPost ntc img*) = *textPost ntc*

imgPost (*setTitlePost ntc title*) = *imgPost ntc*
imgPost (*setTextPost ntc text*) = *imgPost ntc*
imgPost (*setImgPost ntc img*) = *img*
{proof}

datatype *password* = *Psw String.literal*
definition *emptyPass* \equiv *Psw emptyStr*

datatype *req* = *ReqInfo String.literal*
definition *emptyReq* \equiv *ReqInfo emptyStr*

2.2 Identifiers

datatype *userID* = *Uid String.literal*
datatype *postID* = *Nid String.literal*

definition *emptyUserID* \equiv *Uid emptyStr*
definition *emptyPostID* \equiv *Nid emptyStr*

fun *userIDAsStr* **where** *userIDAsStr* (*Uid str*) = *str*

definition *getFreshUserID* *userIDs* \equiv *Uid (fresh (set (map userIDAsStr userIDs)))*
(*STR "2"*)

lemma *UserID-userIDAsStr[simp]*: *Uid (userIDAsStr userID)* = *userID*
{proof}

lemma *member-userIDAsStr-iff[simp]*: *str* \in *userIDAsStr* ‘ (*set userIDs*) \longleftrightarrow *Uid*
str \in *userIDs*
{proof}

lemma *getFreshUserID*: \neg *getFreshUserID userIDs* \in *userIDs*
{proof}

fun *postIDAsStr* **where** *postIDAsStr* (*Nid str*) = *str*

definition *getFreshPostID* *postIDs* \equiv *Nid (fresh (set (map postIDAsStr postIDs)))*
(*STR "3"*)

lemma *PostID-postIDAsStr[simp]*: *Nid (postIDAsStr postID)* = *postID*
{proof}

lemma *member-postIDAsStr-iff[simp]*: *str* \in *postIDAsStr* ‘ (*set postIDs*) \longleftrightarrow *Nid*

str $\in\in$ *postIDs*
<proof>

lemma *getFreshPostID*: \neg *getFreshPostID* *postIDs* $\in\in$ *postIDs*
<proof>

end

3 System specification

theory *System-Specification*
imports *Prelim*
begin

declare *List.insert*[*simp*]

3.1 The state

record *state* =
 admin :: *userID*

pendingUReqs :: *userID list*
 userReq :: *userID* \Rightarrow *req*
 userIDs :: *userID list*
 user :: *userID* \Rightarrow *user*
 pass :: *userID* \Rightarrow *password*

pendingFReqs :: *userID* \Rightarrow *userID list*
 friendReq :: *userID* \Rightarrow *userID* \Rightarrow *req*
 friendIDs :: *userID* \Rightarrow *userID list*

postIDs :: *postID list*
 post :: *postID* \Rightarrow *post*
 owner :: *postID* \Rightarrow *userID*
 vis :: *postID* \Rightarrow *vis*

definition *IDsOK* :: *state* \Rightarrow *userID list* \Rightarrow *postID list* \Rightarrow *bool*

where

IDsOK *s* *uIDs* *pIDs* \equiv
 list-all (λ *uID*. *uID* $\in\in$ *userIDs* *s*) *uIDs* \wedge
 list-all (λ *pID*. *pID* $\in\in$ *postIDs* *s*) *pIDs*

3.2 The actions

3.2.1 Initialization of the system

definition *istate* :: *state*

where

istate \equiv

(
 admin = *emptyUserID*,

 pendingUReqs = [],
 userReq = (λ *uID*. *emptyReq*),
 userIDs = [],
 user = (λ *uID*. *emptyUser*),
 pass = (λ *uID*. *emptyPass*),

 pendingFReqs = (λ *uID*. []),
 friendReq = (λ *uID* *uID'*. *emptyReq*),
 friendIDs = (λ *uID*. []),

 postIDs = [],
 post = (λ *papID*. *emptyPost*),
 owner = (λ *pID*. *emptyUserID*),
 vis = (λ *pID*. *FriendV*)
)

3.2.2 Starting action

definition *startSys* ::

state \Rightarrow *userID* \Rightarrow *password* \Rightarrow *state*

where

startSys *s* *uID* *p* \equiv
 s (*admin* := *uID*,
 userIDs := [*uID*],
 user := (*user* *s*) (*uID* := *emptyUser*),
 pass := (*pass* *s*) (*uID* := *p*))

definition *e-startSys* :: *state* \Rightarrow *userID* \Rightarrow *password* \Rightarrow *bool*

where

e-startSys *s* *uID* *p* \equiv *userIDs* *s* = []

3.2.3 Creation actions

definition *createNUReq* :: *state* \Rightarrow *userID* \Rightarrow *req* \Rightarrow *state*

where

createNUReq *s* *uID* *reqInfo* \equiv
 s (*pendingUReqs* := *pendingUReqs* *s* @ [*uID*],
 userReq := (*userReq* *s*) (*uID* := *reqInfo*)
)

definition *e-createNUReq* :: *state* \Rightarrow *userID* \Rightarrow *req* \Rightarrow *bool*

where

e-createNUReq *s* *uID* *req* \equiv
 admin *s* $\in \in$ *userIDs* *s* \wedge \neg *uID* $\in \in$ *userIDs* *s* \wedge \neg *uID* $\in \in$ *pendingUReqs* *s*

definition $createUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow password \Rightarrow state$

where

$createUser\ s\ uID\ p\ uID'\ p' \equiv$
 $s\ (\!|userIDs := uID' \# (userIDs\ s),$
 $user := (user\ s)\ (uID' := emptyUser),$
 $pass := (pass\ s)\ (uID' := p'),$
 $pendingUReqs := remove1\ uID'\ (pendingUReqs\ s),$
 $userReq := (userReq\ s)\ (uID := emptyReq)\!|)$

definition $e-createUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow password \Rightarrow bool$

where

$e-createUser\ s\ uID\ p\ uID'\ p' \equiv$
 $IDsOK\ s\ [uID] \ [] \wedge pass\ s\ uID = p \wedge uID = admin\ s \wedge uID' \in \in pendingUReqs\ s$

definition $createPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow title \Rightarrow state$

where

$createPost\ s\ uID\ p\ pID\ title \equiv$
 $s\ (\!|postIDs := pID \# postIDs\ s,$
 $post := (post\ s)\ (pID := Ntc\ title\ emptyText\ emptyImg),$
 $owner := (owner\ s)\ (pID := uID)\!|)$

definition $e-createPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow title \Rightarrow bool$

where

$e-createPost\ s\ uID\ p\ pID\ title \equiv$
 $IDsOK\ s\ [uID] \ [] \wedge pass\ s\ uID = p \wedge$
 $\neg pID \in \in postIDs\ s$

definition $createFriendReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow req \Rightarrow state$

where

$createFriendReq\ s\ uID\ p\ uID'\ req \equiv$
 $let\ pfr = pendingFReqs\ s\ in$
 $s\ (\!|pendingFReqs := pfr\ (uID' := pfr\ uID' \ @\ [uID]),$
 $friendReq := fun-upd2\ (friendReq\ s)\ uID\ uID'\ req\!|)$

definition $e-createFriendReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow req \Rightarrow bool$

where

$e-createFriendReq\ s\ uID\ p\ uID'\ req \equiv$
 $IDsOK\ s\ [uID, uID'] \ [] \wedge pass\ s\ uID = p \wedge$

$\neg uID \in \text{pendingFReqs } s \ uID' \wedge \neg uID \in \text{friendIDs } s \ uID'$

definition $\text{createFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{state}$

where

$\text{createFriend } s \ uID \ p \ uID' \equiv$

let $fr = \text{friendIDs } s$; $pfr = \text{pendingFReqs } s$ in

$s \ (\text{friendIDs} := fr \ (uID := fr \ uID \ @ \ [uID']), \ uID' := fr \ uID' \ @ \ [uID]),$

$\text{pendingFReqs} := pfr \ (uID := \text{remove1 } uID' \ (pfr \ uID), \ uID' := \text{remove1 } uID$

$(pfr \ uID')),$
 $\text{friendReq} := \text{fun-upd2} \ (\text{fun-upd2} \ (\text{friendReq } s) \ uID' \ uID \ \text{emptyReq}) \ uID \ uID'$
 $\text{emptyReq})$

definition $e\text{-createFriend} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{bool}$

where

$e\text{-createFriend } s \ uID \ p \ uID' \equiv$

$\text{IDsOK } s \ [uID, uID'] \ [] \wedge \text{pass } s \ uID = p \wedge$

$uID' \in \text{pendingFReqs } s \ uID$

3.2.4 Updating actions

definition $\text{updateUser} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{password} \Rightarrow \text{name} \Rightarrow$
 $\text{inform} \Rightarrow \text{state}$

where

$\text{updateUser } s \ uID \ p \ p' \ \text{name} \ \text{info} \equiv$

$s \ (\text{user} := (\text{user } s) \ (uID := \text{Usr } \text{name} \ \text{info}),$

$\text{pass} := (\text{pass } s) \ (uID := p'))$

definition $e\text{-updateUser} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{password} \Rightarrow \text{name} \Rightarrow$
 $\text{inform} \Rightarrow \text{bool}$

where

$e\text{-updateUser } s \ uID \ p \ p' \ \text{name} \ \text{info} \equiv$

$\text{IDsOK } s \ [uID] \ [] \wedge \text{pass } s \ uID = p$

definition $\text{updatePost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{post} \Rightarrow \text{state}$

where

$\text{updatePost } s \ uID \ p \ pID \ \text{pst} \equiv$

$s \ (\text{post} := (\text{post } s) \ (pID := \text{pst}))$

definition $e\text{-updatePost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{post} \Rightarrow \text{bool}$

where

$e\text{-updatePost } s \ uID \ p \ pID \ \text{pst} \equiv$

$\text{IDsOK } s \ [uID] \ [pID] \wedge \text{pass } s \ uID = p \wedge$

$\text{owner } s \ pID = uID$

definition $\text{updateVisPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{vis} \Rightarrow \text{state}$

where

$updateVisPost\ s\ uID\ p\ pID\ vs \equiv$
 $s\ (\!|vis := (vis\ s)\ (pID := vs)|\!)$

definition $e-updateVisPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis \Rightarrow bool$
where

$e-updateVisPost\ s\ uID\ p\ pID\ vs \equiv$
 $IDsOK\ s\ [uID]\ [pID] \wedge pass\ s\ uID = p \wedge$
 $owner\ s\ pID = uID \wedge vs \in \{FriendV, PublicV\}$

3.2.5 Deletion (removal) actions

definition $deleteFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow state$
where

$deleteFriend\ s\ uID\ p\ uID' \equiv$
 $let\ fr = friendIDs\ s\ in$
 $s\ (\!|friendIDs := fr\ (uID := removeAll\ uID'\ (fr\ uID),\ uID' := removeAll\ uID\ (fr\ uID'))|\!)$

definition $e-deleteFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where

$e-deleteFriend\ s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ [] \wedge pass\ s\ uID = p \wedge$
 $uID' \in \in friendIDs\ s\ uID$

3.2.6 Reading actions

definition $readNUReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow req$
where

$readNUReq\ s\ uID\ p\ uID' \equiv userReq\ s\ uID'$

definition $e-readNUReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where

$e-readNUReq\ s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID]\ [] \wedge pass\ s\ uID = p \wedge$
 $uID = admin\ s \wedge uID' \in \in pendingUReqs\ s$

definition $readUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow name \times inform$
where

$readUser\ s\ uID\ p\ uID' \equiv niUser\ (user\ s\ uID')$

definition $e-readUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where

$e-readUser\ s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ [] \wedge pass\ s\ uID = p$

definition $readAmIAdmin :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$
where

$readAmIAdmin\ s\ uID\ p \equiv uID = admin\ s$

definition $e\text{-readAmIAdmin} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{bool}$

where

$e\text{-readAmIAdmin } s \text{ uID } p \equiv$
 $IDsOK\ s\ [uID]\ [] \wedge \text{pass } s\ \text{uID} = p$

definition $\text{readPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{post}$

where

$\text{readPost } s \text{ uID } p \text{ pID} \equiv \text{post } s \text{ pID}$

definition $e\text{-readPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readPost } s \text{ uID } p \text{ pID} \equiv$
 $\text{let } \text{post} = \text{post } s \text{ pID} \text{ in}$
 $IDsOK\ s\ [uID]\ [pID] \wedge \text{pass } s\ \text{uID} = p \wedge$
 $(\text{owner } s \text{ pID} = \text{uID} \vee \text{uID} \in \in \text{friendIDs } s (\text{owner } s \text{ pID}) \vee \text{vis } s \text{ pID} = \text{PublicV})$

definition $\text{readVisPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{vis}$

where

$\text{readVisPost } s \text{ uID } p \text{ pID} \equiv \text{vis } s \text{ pID}$

definition $e\text{-readVisPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readVisPost } s \text{ uID } p \text{ pID} \equiv$
 $\text{let } \text{post} = \text{post } s \text{ pID} \text{ in}$
 $IDsOK\ s\ [uID]\ [pID] \wedge \text{pass } s\ \text{uID} = p \wedge$
 $(\text{owner } s \text{ pID} = \text{uID} \vee \text{uID} \in \in \text{friendIDs } s (\text{owner } s \text{ pID}) \vee \text{vis } s \text{ pID} = \text{PublicV})$

definition $\text{readOwnerPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{userID}$

where

$\text{readOwnerPost } s \text{ uID } p \text{ pID} \equiv \text{owner } s \text{ pID}$

definition $e\text{-readOwnerPost} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{postID} \Rightarrow \text{bool}$

where

$e\text{-readOwnerPost } s \text{ uID } p \text{ pID} \equiv$
 $\text{let } \text{post} = \text{post } s \text{ pID} \text{ in}$
 $IDsOK\ s\ [uID]\ [pID] \wedge \text{pass } s\ \text{uID} = p \wedge$
 $(\text{owner } s \text{ pID} = \text{uID} \vee \text{uID} \in \in \text{friendIDs } s (\text{owner } s \text{ pID}) \vee \text{vis } s \text{ pID} = \text{PublicV})$

definition $\text{readFriendReqToMe} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{req}$

where

$\text{readFriendReqToMe } s \text{ uID } p \text{ uID}' \equiv \text{friendReq } s \text{ uID}' \text{ uID}$

definition $e\text{-readFriendReqToMe} :: \text{state} \Rightarrow \text{userID} \Rightarrow \text{password} \Rightarrow \text{userID} \Rightarrow \text{bool}$

where

e-readFriendReqToMe s uID p uID' \equiv
 $IDsOK\ s\ [uID, uID']\ [] \wedge pass\ s\ uID = p \wedge$
 $uID' \in \in pendingFReqs\ s\ uID$

definition *readFriendReqFromMe* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow req$
where
readFriendReqFromMe $s\ uID\ p\ uID' \equiv friendReq\ s\ uID\ uID'$

definition *e-readFriendReqFromMe* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow$
 $bool$
where
e-readFriendReqFromMe $s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ [] \wedge pass\ s\ uID = p \wedge$
 $uID \in \in pendingFReqs\ s\ uID'$

3.2.7 Listing actions

definition *listPendingUReqs* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID\ list$
where
listPendingUReqs $s\ uID\ p \equiv pendingUReqs\ s$

definition *e-listPendingUReqs* $:: state \Rightarrow userID \Rightarrow password \Rightarrow bool$
where
e-listPendingUReqs $s\ uID\ p \equiv$
 $IDsOK\ s\ [uID]\ [] \wedge pass\ s\ uID = p \wedge uID = admin\ s$

definition *listAllUsers* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID\ list$
where
listAllUsers $s\ uID\ p \equiv userIDs\ s$

definition *e-listAllUsers* $:: state \Rightarrow userID \Rightarrow password \Rightarrow bool$
where
e-listAllUsers $s\ uID\ p \equiv IDsOK\ s\ [uID]\ [] \wedge pass\ s\ uID = p$

definition *listFriends* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow userID\ list$
where
listFriends $s\ uID\ p\ uID' \equiv friendIDs\ s\ uID'$

definition *e-listFriends* $:: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
e-listFriends $s\ uID\ p\ uID' \equiv$
 $IDsOK\ s\ [uID, uID']\ [] \wedge pass\ s\ uID = p \wedge$
 $(uID = uID' \vee uID \in \in friendIDs\ s\ uID')$

definition $listPosts :: state \Rightarrow userID \Rightarrow password \Rightarrow (userID \times postID) list$
where
 $listPosts\ s\ userID\ p \equiv$
 $[(owner\ s\ pID, pID).$
 $\quad pID \leftarrow postIDs\ s,$
 $\quad vis\ s\ pID = PublicV \vee userID \in \in friendIDs\ s\ (owner\ s\ pID) \vee userID = owner\ s$
 $\quad pID$
 $\quad]$

definition $e-listPosts :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$
where
 $e-listPosts\ s\ userID\ p \equiv IDsOK\ s\ [userID] [] \wedge pass\ s\ userID = p$

3.3 The step function

datatype $out =$

$outOK \mid outErr \mid$
 $outBool\ bool \mid outNI\ name \times inform \mid outPost\ post \mid$
 $outImg\ img \mid outVis\ vis \mid outReq\ req \mid$
 $outUID\ userID \mid outUIDL\ userID\ list \mid$
 $outUIDNIDL\ (userID \times postID)list$

datatype $sActt =$
 $sSys\ userID\ password$

lemmas $s-defs =$
 $e-startSys-def\ startSys-def$

fun $sStep :: state \Rightarrow sActt \Rightarrow out * state$ **where**
 $sStep\ s\ (sSys\ userID\ p) =$
 $(if\ e-startSys\ s\ userID\ p$
 $\quad then\ (outOK, startSys\ s\ userID\ p)$
 $\quad else\ (outErr, s))$

fun $sUserOfA :: sActt \Rightarrow userID$ **where**
 $sUserOfA\ (sSys\ userID\ p) = userID$

datatype $cActt =$
 $cNUReq\ userID\ req$
 $\mid cUser\ userID\ password\ userID\ password$
 $\mid cFriendReq\ userID\ password\ userID\ req$
 $\mid cFriend\ userID\ password\ userID$

| *cPost userID password postID title*

lemmas *c-defs* =
e-createNUReq-def createNUReq-def
e-createUser-def createUser-def
e-createFriendReq-def createFriendReq-def
e-createFriend-def createFriend-def
e-createPost-def createPost-def

fun *cStep* :: *state* ⇒ *cActt* ⇒ *out* * *state* **where**
cStep *s* (*cNUReq* *uID* *req*) =
 (*if e-createNUReq* *s* *uID* *req*
 then (*outOK*, *createNUReq* *s* *uID* *req*)
 else (*outErr*, *s*))
|
cStep *s* (*cUser* *uID* *p* *uID'* *p'*) =
 (*if e-createUser* *s* *uID* *p* *uID'* *p'*
 then (*outOK*, *createUser* *s* *uID* *p* *uID'* *p'*)
 else (*outErr*, *s*))
|
cStep *s* (*cFriendReq* *uID* *p* *uID'* *req*) =
 (*if e-createFriendReq* *s* *uID* *p* *uID'* *req*
 then (*outOK*, *createFriendReq* *s* *uID* *p* *uID'* *req*)
 else (*outErr*, *s*))
|
cStep *s* (*cFriend* *uID* *p* *uID'*) =
 (*if e-createFriend* *s* *uID* *p* *uID'*
 then (*outOK*, *createFriend* *s* *uID* *p* *uID'*)
 else (*outErr*, *s*))
|
cStep *s* (*cPost* *uID* *p* *pID* *title*) =
 (*if e-createPost* *s* *uID* *p* *pID* *title*
 then (*outOK*, *createPost* *s* *uID* *p* *pID* *title*)
 else (*outErr*, *s*))

fun *cUserOfA* :: *cActt* ⇒ *userID* *option* **where**
cUserOfA (*cNUReq* *uID* *req*) = *Some uID*
| *cUserOfA* (*cUser* *uID* *p* *uID'* *p'*) = *Some uID*
| *cUserOfA* (*cFriendReq* *uID* *p* *uID'* *req*) = *Some uID*
| *cUserOfA* (*cFriend* *uID* *p* *uID'*) = *Some uID*
| *cUserOfA* (*cPost* *uID* *p* *pID* *title*) = *Some uID*

datatype *dActt* =
 dFriend userID password userID

```

lemmas d-defs =
  e-deleteFriend-def deleteFriend-def

fun dStep :: state ⇒ dActt ⇒ out * state where
  dStep s (dFriend uID p uID') =
    (if e-deleteFriend s uID p uID'
     then (outOK, deleteFriend s uID p uID')
     else (outErr, s))

fun dUserOfA :: dActt ⇒ userID where
  dUserOfA (dFriend uID p uID') = uID

datatype uActt =
  | User userID password password name inform
  | uPost userID password postID post
  | uVisPost userID password postID vis

lemmas u-defs =
  e-updateUser-def updateUser-def
  e-updatePost-def updatePost-def
  e-updateVisPost-def updateVisPost-def

fun uStep :: state ⇒ uActt ⇒ out * state where
  uStep s (uUser uID p p' name info) =
    (if e-updateUser s uID p p' name info
     then (outOK, updateUser s uID p p' name info)
     else (outErr, s))
  |
  uStep s (uPost uID p pID pst) =
    (if e-updatePost s uID p pID pst
     then (outOK, updatePost s uID p pID pst)
     else (outErr, s))
  |
  uStep s (uVisPost uID p pID visStr) =
    (if e-updateVisPost s uID p pID visStr
     then (outOK, updateVisPost s uID p pID visStr)
     else (outErr, s))

fun uUserOfA :: uActt ⇒ userID where
  uUserOfA (uUser uID p p' name info) = uID
  | uUserOfA (uPost uID p pID pst) = uID
  | uUserOfA (uVisPost uID p pID visStr) = uID

datatype rActt =
  | rNUReq userID password userID
  | rUser userID password userID

```

```

|rAmIAdmin userID password
|rPost userID password postID
|rVisPost userID password postID
|rOwnerPost userID password postID
|rFriendReqToMe userID password userID
|rFriendReqFromMe userID password userID

```

```

lemmas r-defs =
readNUReq-def e-readNUReq-def
readUser-def e-readUser-def
readAmIAdmin-def e-readAmIAdmin-def
readPost-def e-readPost-def
readVisPost-def e-readVisPost-def
readOwnerPost-def e-readOwnerPost-def
readFriendReqToMe-def e-readFriendReqToMe-def
readFriendReqFromMe-def e-readFriendReqFromMe-def

```

```

fun rObs :: state ⇒ rActt ⇒ out where
rObs s (rNUReq uID p uID') =
  (if e-readNUReq s uID p uID' then outReq (readNUReq s uID p uID') else outErr)
|
rObs s (rUser uID p uID') =
  (if e-readUser s uID p uID' then outNI (readUser s uID p uID') else outErr)
|
rObs s (rAmIAdmin uID p) =
  (if e-readAmIAdmin s uID p then outBool (readAmIAdmin s uID p) else outErr)
|
rObs s (rPost uID p pID) =
  (if e-readPost s uID p pID then outPost (readPost s uID p pID) else outErr)
|
rObs s (rVisPost uID p pID) =
  (if e-readVisPost s uID p pID then outVis (readVisPost s uID p pID) else outErr)
|
rObs s (rOwnerPost uID p pID) =
  (if e-readOwnerPost s uID p pID then outUID (readOwnerPost s uID p pID) else
  outErr)
|
rObs s (rFriendReqToMe uID p uID') =
  (if e-readFriendReqToMe s uID p uID' then outReq (readFriendReqToMe s uID p
  uID') else outErr)
|
rObs s (rFriendReqFromMe uID p uID') =
  (if e-readFriendReqFromMe s uID p uID' then outReq (readFriendReqFromMe s
  uID p uID') else outErr)

```

```

fun rUserOfA :: rActt ⇒ userID option where
  rUserOfA (rNUReq uID p uID') = Some uID
|rUserOfA (rUser uID p uID') = Some uID

```

```

|rUserOfA (rAmIAdmin uID p) = Some uID
|rUserOfA (rPost uID p pID) = Some uID
|rUserOfA (rVisPost uID p pID) = Some uID
|rUserOfA (rOwnerPost uID p pID) = Some uID
|rUserOfA (rFriendReqToMe uID p uID') = Some uID
|rUserOfA (rFriendReqFromMe uID p uID') = Some uID

```

```

datatype lActt =
  |PendingUReqs userID password
  |lAllUsers userID password
  |lFriends userID password userID
  |lPosts userID password

```

```

lemmas l-defs =
  listPendingUReqs-def e-listPendingUReqs-def
  listAllUsers-def e-listAllUsers-def
  listFriends-def e-listFriends-def
  listPosts-def e-listPosts-def

```

```

fun lObs :: state ⇒ lActt ⇒ out where
lObs s (lPendingUReqs uID p) =
  (if e-listPendingUReqs s uID p then outUIDL (listPendingUReqs s uID p) else
  outErr)
|
lObs s (lAllUsers uID p) =
  (if e-listAllUsers s uID p then outUIDL (listAllUsers s uID p) else outErr)
|
lObs s (lFriends uID p uID') =
  (if e-listFriends s uID p uID' then outUIDL (listFriends s uID p uID') else outErr)
|
lObs s (lPosts uID p) =
  (if e-listPosts s uID p then outUIDNIDL (listPosts s uID p) else outErr)

```

```

fun lUserOfA :: lActt ⇒ userID option where
  lUserOfA (lPendingUReqs uID p) = Some uID
  lUserOfA (lAllUsers uID p) = Some uID
  lUserOfA (lFriends uID p uID') = Some uID
  lUserOfA (lPosts uID p) = Some uID

```

```

datatype act =
  Sact sActt |

```

Cact cActt | *Dact dActt* | *Uact uActt* |

Ract rActt | *Lact lActt*

```
fun step :: state ⇒ act ⇒ out * state where
step s (Sact sa) = sStep s sa
|
step s (Cact ca) = cStep s ca
|
step s (Dact da) = dStep s da
|
step s (Uact ua) = uStep s ua
|
step s (Ract ra) = (rObs s ra, s)
|
step s (Lact la) = (lObs s la, s)
```

```
fun userOfA :: act ⇒ userID option where
userOfA (Sact sa) = Some (sUserOfA sa)
|
userOfA (Cact ca) = cUserOfA ca
|
userOfA (Dact da) = Some (dUserOfA da)
|
userOfA (Uact ua) = Some (uUserOfA ua)
|
userOfA (Ract ra) = rUserOfA ra
|
userOfA (Lact la) = lUserOfA la
```

3.4 Code generation

```
export-code step istate getFreshPostID in Scala
```

```
end
theory Automation-Setup
imports System-Specification
begin
```

```
lemma add-prop:
  assumes PROP (T)
  shows A ==> PROP (T)
  ⟨proof⟩
```

```
lemmas exhaust-elim =
```

```

sActt.exhaust[of x, THEN add-prop[where A=a=Sact x], rotated -1]
cActt.exhaust[of x, THEN add-prop[where A=a=Cact x], rotated -1]
uActt.exhaust[of x, THEN add-prop[where A=a=Uact x], rotated -1]
rActt.exhaust[of x, THEN add-prop[where A=a=Ract x], rotated -1]
lActt.exhaust[of x, THEN add-prop[where A=a=Lact x], rotated -1]
for x a

```

lemma *state-cong*:

fixes *s::state*

assumes

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge *userIDs s = userIDs s1* \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge

user s = user s1 \wedge *pass s = pass s1* \wedge *pendingFReqs s = pendingFReqs s1* \wedge

friendReq s = friendReq s1 \wedge *friendIDs s = friendIDs s1* \wedge

post s = post s1 \wedge

owner s = owner s1 \wedge

vis s = vis s1

shows *s = s1*

<proof>

end

4 Safety properties

theory *Safety-Properties*

imports *Automation-Setup Bounded-Deducibility-Security.Compositional-Reasoning*

begin

interpretation *IO-Automaton* **where**

istate = istate **and** *step = step*

<proof>

declare *if-splits*[*split*]

declare *IDsOK-def*[*simp*]

lemmas *eff-defs = s-defs c-defs d-defs u-defs*

lemmas *obs-defs = r-defs l-defs*

lemmas *all-defs = eff-defs obs-defs*

lemmas *step-elim = step.elims sStep.elims cStep.elims dStep.elims uStep.elims*

declare *sstep-Cons*[*simp*]

lemma *Lact-Ract-noStateChange*[*simp*]:

assumes *a* \in *Lact* \cup *UNIV* \cup *Ract* \cup *UNIV*

shows *snd (step s a) = s*

<proof>

lemma *Lact-Ract-noStateChange-set:*

assumes $set\ al \subseteq Lact\ 'UNIV \cup Ract\ 'UNIV$

shows $snd\ (sstep\ s\ al) = s$

<proof>

lemma *reach-postIDs-persist:*

$pid \in \in postIDs\ s \implies step\ s\ a = (ou, s') \implies pid \in \in postIDs\ s'$

<proof>

lemma *reach-visPost:* $reach\ s \implies vis\ s\ pid \in \{FriendV, PublicV\}$

<proof>

lemma *reach-owner-userIDs:* $reach\ s \implies pid \in \in postIDs\ s \implies owner\ s\ pid \in \in userIDs\ s$

<proof>

lemma *reach-friendIDs-symmetric:*

$reach\ s \implies uid1 \in \in friendIDs\ s\ uid2 \longleftrightarrow uid2 \in \in friendIDs\ s\ uid1$

<proof>

lemma *reach-not-postIDs-vis-FriendV:*

assumes $reach\ s \neg pid \in \in postIDs\ s$

shows $vis\ s\ pid = FriendV$

<proof>

lemma *reach-distinct-friends-reqs:*

assumes $reach\ s$

shows $distinct\ (friendIDs\ s\ uid)$ **and** $distinct\ (pendingFReqs\ s\ uid)$

and $uid' \in \in pendingFReqs\ s\ uid \implies uid' \notin set\ (friendIDs\ s\ uid)$

and $uid' \in \in pendingFReqs\ s\ uid \implies uid \notin set\ (friendIDs\ s\ uid')$

<proof>

lemma *remove1-in-set:* $x \in \in remove1\ y\ xs \implies x \in \in xs$

<proof>

lemma *reach-IDs-used-IDsOK[rule-format]:*

assumes $reach\ s$

shows $uid \in \in pendingFReqs\ s\ uid' \longrightarrow IDsOK\ s\ [uid, uid']\ []\ (\mathbf{is}\ ?p)$

and $uid \in \in friendIDs\ s\ uid' \longrightarrow IDsOK\ s\ [uid, uid']\ []\ (\mathbf{is}\ ?f)$

<proof>

lemma *IDs-mono[rule-format]:*

assumes $step\ s\ a = (ou, s')$

shows $uid \in \in userIDs\ s \longrightarrow uid \in \in userIDs\ s'\ (\mathbf{is}\ ?u)$

and $pid \in \in postIDs\ s \longrightarrow pid \in \in postIDs\ s'\ (\mathbf{is}\ ?n)$

<proof>

lemma *IDsOK-mono:*

```

assumes step s a = (ou, s')
and IDsOK s uIDs pIDs
shows IDsOK s' uIDs pIDs
<proof>

```

```

end

```

```

theory Observation-Setup
imports Safety-Properties
begin

```

5 The observation setup

The observers are a arbitrary but fixed set of users:

```

consts UIDs :: userID set

```

```

type-synonym obs = act * out

```

The observations are all their actions:

```

fun  $\gamma :: (state, act, out) trans \Rightarrow bool$  where
 $\gamma (Trans - a - -) =$ 
  (userOfA a \in Some 'UIDs)

```

```

fun  $g :: (state, act, out)trans \Rightarrow obs$  where
 $g (Trans - a ou -) = (a, ou)$ 

```

```

end

```

```

theory Post-Intro
  imports ../Safety-Properties ../Observation-Setup
begin

```

6 Post confidentiality

We prove the following property:

Given a group of users *UIDs* and a post *PID*,

that group cannot learn anything about the different versions of the post *PID* (the initial created version and the later ones obtained by updating the post)

beyond the updates performed while or last before one of the following holds:

- either a user in $UIDs$ is the post’s owner, a friend of the owner, or the admin
- or $UIDs$ has at least one registered user and the post is marked as “public”.

end

theory *Post-Value-Setup*
imports *Post-Intro*
begin

The ID of the confidential post:

consts $PID :: postID$

6.1 Preliminaries

definition *eeqButPID* **where**
 $eeqButPID\ ntc\ ncs1 \equiv$
 $\forall\ pid.\ pid \neq PID \longrightarrow ntc\ pid = ncs1\ pid$

lemmas *eeqButPID-intro* = *eeqButPID-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eeqButPID-eeq*[*simp,intro!*]: *eeqButPID ntc ncs*
 $\langle proof \rangle$

lemma *eeqButPID-sym*:
assumes *eeqButPID ntc ncs1* **shows** *eeqButPID ncs1 ntc*
 $\langle proof \rangle$

lemma *eeqButPID-trans*:
assumes *eeqButPID ntc ncs1* **and** *eeqButPID ncs1 ncs2* **shows** *eeqButPID ntc ncs2*
 $\langle proof \rangle$

lemma *eeqButPID-cong*:
assumes *eeqButPID ntc ncs1*
and $PID = PID \implies eqButT\ uu\ uu1$
and $pid \neq PID \implies uu = uu1$
shows *eeqButPID (ntcs (pid := uu)) (ntcs1 (pid := uu1))*
 $\langle proof \rangle$

lemma *eeqButPID-not-PID*:
 $\llbracket eeqButPID\ ntc\ ncs1;\ pid \neq PID \rrbracket \implies ntc\ pid = ncs1\ pid$
 $\langle proof \rangle$

lemma *eqButPID-toEq*:
assumes *eqButPID ntc ntc1*
shows *ntcs (PID := pst) = ntc1 (PID := pst)*
 \langle *proof* \rangle

lemma *eqButPID-update-post*:
assumes *eqButPID ntc ntc1*
shows *eqButPID (ntcs (pid := ntc)) (ntcs1 (pid := ntc))*
 \langle *proof* \rangle

definition *eqButPID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**
eqButPID s s1 \equiv
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

pendingFReqs s = pendingFReqs s1 \wedge *friendReq s = friendReq s1* \wedge *friendIDs s*
 $=$ *friendIDs s1* \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
eqButPID (post s) (post s1) \wedge
owner s = owner s1 \wedge
vis s = vis s1

lemmas *eqButPID-intro = eqButPID-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButPID-refl*[*simp,intro!*]: *eqButPID s s*
 \langle *proof* \rangle

lemma *eqButPID-sym*:
assumes *eqButPID s s1* **shows** *eqButPID s1 s*
 \langle *proof* \rangle

lemma *eqButPID-trans*:
assumes *eqButPID s s1* **and** *eqButPID s1 s2* **shows** *eqButPID s s2*
 \langle *proof* \rangle

lemma *eqButPID-stateSelectors*:
eqButPID s s1 \implies
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

$pendingFReqs\ s = pendingFReqs\ s1 \wedge friendReq\ s = friendReq\ s1 \wedge friendIDs\ s = friendIDs\ s1 \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$
 $eqButPID\ (post\ s)\ (post\ s1) \wedge$
 $owner\ s = owner\ s1 \wedge$
 $vis\ s = vis\ s1 \wedge$

$IDsOK\ s = IDsOK\ s1$
 $\langle proof \rangle$

lemma *eqButPID-not-PID*:
 $eqButPID\ s\ s1 \implies pid \neq PID \implies post\ s\ pid = post\ s1\ pid$
 $\langle proof \rangle$

lemma *eqButPID-actions*:
assumes $eqButPID\ s\ s1$
shows $listPosts\ s\ uid\ p = listPosts\ s1\ uid\ p$
 $\langle proof \rangle$

lemma *eqButPID-setPost*:
assumes $eqButPID\ s\ s1$
shows $(post\ s)(PID := pst) = (post\ s1)(PID := pst)$
 $\langle proof \rangle$

lemma *eqButPID-update-post*:
assumes $eqButPID\ s\ s1$
shows $eqButPID\ ((post\ s)\ (pid := ntc))\ ((post\ s1)\ (pid := ntc))$
 $\langle proof \rangle$

lemma *eqButPID-cong[simp, intro]*:
 $\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!admin := uu1))\ (s1\ (\!admin := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!pendingUReqs := uu1))\ (s1\ (\!pendingUReqs := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!userReq := uu1))\ (s1\ (\!userReq := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!userIDs := uu1))\ (s1\ (\!userIDs := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!user := uu1))\ (s1\ (\!user := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!pass := uu1))\ (s1\ (\!pass := uu2))$

$\bigwedge uu1\ uu2. eqButPID\ s\ s1 \implies uu1 = uu2 \implies eqButPID\ (s\ (\!postIDs := uu1))\ (s1\ (\!postIDs := uu2))$

$$\begin{aligned}
& (s1 \langle postIDs := uu2 \rangle) \\
& \wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s \langle owner := uu1 \rangle) (s1 \langle owner := uu2 \rangle) \\
& \wedge uu1 uu2. eqButPID s s1 \implies eqButPID uu1 uu2 \implies eqButPID (s \langle post := uu1 \rangle) (s1 \langle post := uu2 \rangle) \\
& \wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s \langle vis := uu1 \rangle) (s1 \langle vis := uu2 \rangle)
\end{aligned}$$

$$\begin{aligned}
& \wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s \langle pendingFReqs := uu1 \rangle) (s1 \langle pendingFReqs := uu2 \rangle) \\
& \wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s \langle friendReq := uu1 \rangle) (s1 \langle friendReq := uu2 \rangle) \\
& \wedge uu1 uu2. eqButPID s s1 \implies uu1 = uu2 \implies eqButPID (s \langle friendIDs := uu1 \rangle) (s1 \langle friendIDs := uu2 \rangle)
\end{aligned}$$

$\langle proof \rangle$

6.2 Value Setup

datatype *value* =

TVal post — updated content of the confidential post
| *OVal bool* — updated dynamic declassification trigger condition

Openness of the access window to the confidential information in a given state, i.e. the dynamic declassification trigger condition:

definition *openToUIDs* **where**

$$\begin{aligned}
openToUIDs s \equiv & \\
& \exists uid \in UIDs. \\
& uid \in \in userIDs s \wedge \\
& (uid = owner s PID \vee uid \in \in friendIDs s (owner s PID) \vee \\
& vis s PID = PublicV)
\end{aligned}$$

definition *open* **where** $open s \equiv PID \in \in postIDs s \wedge openToUIDs s$

lemmas *open-defs* = *openToUIDs-def open-def*

lemma *eqButPID-openToUIDs*:

assumes *eqButPID s s1*

shows $openToUIDs s \longleftrightarrow openToUIDs s1$

$\langle proof \rangle$

lemma *eqButPID-open*:

assumes *eqButPID s s1*

shows $open s \longleftrightarrow open s1$

$\langle proof \rangle$

lemma *not-open-eqButPID*:

assumes *1: $\neg open s$ and 2: eqButPID s s1*

shows $\neg \text{open } s1$
 $\langle \text{proof} \rangle$

fun $\varphi :: (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{bool}$ **where**
 $\varphi (\text{Trans } - (\text{Uact } (\text{uPost } \text{uid } p \text{ pid } \text{pst})) \text{ ou } -) = (\text{pid} = \text{PID} \wedge \text{ou} = \text{outOK})$
 $|$
 $\varphi (\text{Trans } s \text{ - } s') = (\text{open } s \neq \text{open } s')$

lemma $\varphi\text{-def2}$:

assumes $\text{step } s \ a = (\text{ou}, s')$
shows
 $\varphi (\text{Trans } s \ a \ \text{ou } s') \longleftrightarrow$
 $(\exists \text{uid } p \ \text{pst}. a = \text{Uact } (\text{uPost } \text{uid } p \ \text{PID } \text{pst}) \wedge \text{ou} = \text{outOK}) \vee$
 $\text{open } s \neq \text{open } s'$
 $\langle \text{proof} \rangle$

fun $f :: (\text{state}, \text{act}, \text{out}) \text{ trans} \Rightarrow \text{value}$ **where**
 $f (\text{Trans } s \ (\text{Uact } (\text{uPost } \text{uid } p \ \text{pid } \ \text{pst})) \text{ - } s') =$
 $(\text{if } \text{pid} = \text{PID} \text{ then } \text{TVal } \text{pst} \text{ else } \text{OVal } (\text{open } s'))$
 $|$
 $f (\text{Trans } s \text{ - } s') = \text{OVal } (\text{open } s')$

lemma $\text{Uact-uPost-step-eqButPID}$:

assumes $a = \text{Uact } (\text{uPost } \text{uid } p \ \text{PID } \ \text{pst})$
and $\text{step } s \ a = (\text{ou}, s')$
shows $\text{eqButPID } s \ s'$
 $\langle \text{proof} \rangle$

lemma eqButPID-step :

assumes $ss1: \text{eqButPID } s \ s1$
and $\text{step}: \text{step } s \ a = (\text{ou}, s')$
and $\text{step1}: \text{step } s1 \ a = (\text{ou1}, s1')$
shows $\text{eqButPID } s' \ s1'$
 $\langle \text{proof} \rangle$

lemma $\text{eqButPID-step-}\varphi\text{-imp}$:

assumes $ss1: \text{eqButPID } s \ s1$
and $\text{step}: \text{step } s \ a = (\text{ou}, s')$ **and** $\text{step1}: \text{step } s1 \ a = (\text{ou1}, s1')$
and $\varphi: \varphi (\text{Trans } s \ a \ \text{ou } s')$
shows $\varphi (\text{Trans } s1 \ a \ \text{ou1 } s1')$
 $\langle \text{proof} \rangle$

lemma $\text{eqButPID-step-}\varphi$:

assumes $s's1': \text{eqButPID } s \ s1$
and $\text{step}: \text{step } s \ a = (\text{ou}, s')$ **and** $\text{step1}: \text{step } s1 \ a = (\text{ou1}, s1')$
shows $\varphi (\text{Trans } s \ a \ \text{ou } s') = \varphi (\text{Trans } s1 \ a \ \text{ou1 } s1')$

<proof>

```
end  
theory Post  
imports ../Observation-Setup Post-Value-Setup  
begin
```

6.3 Declassification bound

```
fun T :: (state,act,out) trans  $\Rightarrow$  bool where T - = False
```

The bound may dynamically change from closed (B) to open (BO) access to the confidential information (or vice versa) when the openness predicate changes value. The bound essentially relates arbitrary value sequences in the closed phase (i.e. observers learn nothing about the updates during that phase) and identical value sequences in the open phase (i.e. observers may learn everything about the updates during that phase); when transitioning from a closed to an open access window (B - BO below), the last update in the closed phase, i.e. the current version of the post, is also declassified in addition to subsequent updates. This formalizes the “while-or-last-before” scheme in the informal description of the confidentiality property. Moreover, the empty value sequence is treated specially in order to capture harmless cases where the observers may deduce that no secret updates have occurred, e.g. if the system has not been initialized yet. See [2, Section 3.4] for a detailed discussion of the bound.

```
inductive B :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
```

```
and BO :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
```

```
where
```

```
B-TVal[simp,intro!]:
```

```
(pstl = []  $\longrightarrow$  pstl1 = [])  $\Longrightarrow$  B (map TVal pstl) (map TVal pstl1)
```

```
B-BO[intro]:
```

```
BO vl vl1  $\Longrightarrow$  (pstl = []  $\longleftrightarrow$  pstl1 = [])  $\Longrightarrow$  (pstl  $\neq$  []  $\Longrightarrow$  last pstl = last pstl1)  
 $\Longrightarrow$ 
```

```
B (map TVal pstl @ OVal True # vl)  
(map TVal pstl1 @ OVal True # vl1)
```

```
BO-TVal[simp,intro!]:
```

```
BO (map TVal pstl) (map TVal pstl)
```

```
BO-B[intro]:
```

```
B vl vl1  $\Longrightarrow$ 
```

```
BO (map TVal pstl @ OVal False # vl) (map TVal pstl @ OVal False # vl1)
```

```
lemma B-not-Nil: B vl vl1  $\Longrightarrow$  vl = []  $\Longrightarrow$  vl1 = []
```

```
<proof>
```

```
lemma B-OVal-True:
```

```
assumes B (OVal True # vl') vl1
```

shows $\exists vl1'. BO vl' vl1' \wedge vl1 = OVal True \# vl1'$
 $\langle proof \rangle$

unbundle *no relcomp-syntax*

interpretation *BD-Security-IO* **where**
istate = istate and step = step and
 $\varphi = \varphi$ and $f = f$ and $\gamma = \gamma$ and $g = g$ and $T = T$ and $B = B$
 $\langle proof \rangle$

6.4 Unwinding proof

lemma *eqButPID-step- γ -out:*
assumes *ss1: eqButPID s s1*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou1, s1')*
and *op: \neg open s*
and *sT: reachNT s and s1: reach s1*
and *$\gamma: \gamma (Trans s a ou s')$*
shows *ou = ou1*
 $\langle proof \rangle$

lemma *eqButPID-step-eq:*
assumes *ss1: eqButPID s s1*
and *a: a = Uact (uPost uid p PID pst) ou = outOK*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou', s1')*
shows *s' = s1'*
 $\langle proof \rangle$

definition $\Delta 0 :: state \Rightarrow value list \Rightarrow state \Rightarrow value list \Rightarrow bool$ **where**
 $\Delta 0 s vl s1 vl1 \equiv$
 $\neg PID \in \in postIDs s \wedge$
 $s = s1 \wedge B vl vl1$

definition $\Delta 1 :: state \Rightarrow value list \Rightarrow state \Rightarrow value list \Rightarrow bool$ **where**
 $\Delta 1 s vl s1 vl1 \equiv$
 $PID \in \in postIDs s \wedge$
 $(\exists pstl pstl1. (pstl = [] \longrightarrow pstl1 = [])) \wedge vl = map TVal pstl \wedge vl1 = map TVal$
 $pstl1) \wedge$
 $eqButPID s s1 \wedge \neg open s$

definition $\Delta 2 :: state \Rightarrow value list \Rightarrow state \Rightarrow value list \Rightarrow bool$ **where**
 $\Delta 2 s vl s1 vl1 \equiv$
 $PID \in \in postIDs s \wedge$
 $(\exists pstl. vl = map TVal pstl \wedge vl1 = map TVal pstl) \wedge$
 $s = s1 \wedge open s$

definition $\Delta 31 :: state \Rightarrow value list \Rightarrow state \Rightarrow value list \Rightarrow bool$ **where**
 $\Delta 31 s vl s1 vl1 \equiv$

$PID \in \in postIDs\ s \wedge$
 $(\exists\ pstl\ pstl1\ vll\ vll1.$
 $\quad BO\ vll\ vll1 \wedge pstl \neq [] \wedge pstl1 \neq [] \wedge last\ pstl = last\ pstl1 \wedge$
 $\quad vl = map\ TVal\ pstl\ @\ OVal\ True\ \# \ vll \wedge vl1 = map\ TVal\ pstl1\ @\ OVal\ True$
 $\# \ vll1) \wedge$
 $eqButPID\ s\ s1 \wedge \neg\ open\ s$

definition $\Delta32 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta32\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $(\exists\ vll\ vll1.$
 $\quad BO\ vll\ vll1 \wedge$
 $\quad vl = OVal\ True\ \# \ vll \wedge vl1 = OVal\ True\ \# \ vll1) \wedge$
 $s = s1 \wedge \neg\ open\ s$

definition $\Delta4 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**

$\Delta4\ s\ vl\ s1\ vl1 \equiv$
 $PID \in \in postIDs\ s \wedge$
 $(\exists\ pstl\ vll\ vll1.$
 $\quad B\ vll\ vll1 \wedge$
 $\quad vl = map\ TVal\ pstl\ @\ OVal\ False\ \# \ vll \wedge vl1 = map\ TVal\ pstl\ @\ OVal\ False$
 $\# \ vll1) \wedge$
 $s = s1 \wedge open\ s$

lemma *istate- $\Delta0$* :

assumes $B: B\ vl\ vl1$

shows $\Delta0\ istate\ vl\ istate\ vl1$

<proof>

lemma *unwind-cont- $\Delta0$* : $unwind-cont\ \Delta0\ \{\Delta0, \Delta1, \Delta2, \Delta31, \Delta32, \Delta4\}$

<proof>

lemma *unwind-cont- $\Delta1$* : $unwind-cont\ \Delta1\ \{\Delta1\}$

<proof>

lemma *unwind-cont- $\Delta2$* : $unwind-cont\ \Delta2\ \{\Delta2\}$

<proof>

lemma *unwind-cont- $\Delta31$* : $unwind-cont\ \Delta31\ \{\Delta31, \Delta32\}$

<proof>

lemma *unwind-cont- $\Delta32$* : $unwind-cont\ \Delta32\ \{\Delta2, \Delta32, \Delta4\}$

<proof>

lemma *unwind-cont- $\Delta4$* : $unwind-cont\ \Delta4\ \{\Delta1, \Delta31, \Delta32, \Delta4\}$

<proof>

definition Gr **where**

$Gr =$

```

{
( $\Delta_0$ , { $\Delta_0, \Delta_1, \Delta_2, \Delta_{31}, \Delta_{32}, \Delta_4$ }),
( $\Delta_1$ , { $\Delta_1$ }),
( $\Delta_2$ , { $\Delta_2$ }),
( $\Delta_{31}$ , { $\Delta_{31}, \Delta_{32}$ }),
( $\Delta_{32}$ , { $\Delta_2, \Delta_{32}, \Delta_4$ }),
( $\Delta_4$ , { $\Delta_1, \Delta_{31}, \Delta_{32}, \Delta_4$ })
}

```

```

theorem secure: secure
<proof>

```

```

end
theory Friend-Intro
imports ../Safety-Properties ../Observation-Setup
begin

```

7 Friendship status confidentiality

We prove the following property:

Given a group of users $UIDs$ and given two users $UID1$ and $UID2$ not in that group,

that group cannot learn anything about the changes in the status of friendship between $UID1$ and $UID2$

beyond what everybody knows, namely that

- there is no friendship between $UID1$ and $UID2$ before those users have been created, and
- the updates form an alternating sequence of friending and unfriending,

and beyond those updates performed while or last before a user in $UIDs$ is friends with $UID1$ or $UID2$.

```

end

```

```

theory Friend-Value-Setup
imports Friend-Intro
begin

```

The confidential information is the friendship status between two arbitrary but fixed users:

consts $UID1 :: userID$
consts $UID2 :: userID$

axiomatization where
 $UID1$ - $UID2$ - $UIDs$: $\{UID1, UID2\} \cap UIDs = \{\}$
and
 $UID1$ - $UID2$: $UID1 \neq UID2$

7.1 Preliminaries

fun $eqButUIDl :: userID \Rightarrow userID\ list \Rightarrow userID\ list \Rightarrow bool$ **where**
 $eqButUIDl\ uid\ uidl\ uidl1 = (remove1\ uid\ uidl = remove1\ uid\ uidl1)$

lemma $eqButUIDl$ - eq [*simp,intro!*]: $eqButUIDl\ uid\ uidl\ uidl$
 $\langle proof \rangle$

lemma $eqButUIDl$ -*sym*:
assumes $eqButUIDl\ uid\ uidl\ uidl1$
shows $eqButUIDl\ uid\ uidl1\ uidl$
 $\langle proof \rangle$

lemma $eqButUIDl$ -*trans*:
assumes $eqButUIDl\ uid\ uidl\ uidl1$ **and** $eqButUIDl\ uid\ uidl1\ uidl2$
shows $eqButUIDl\ uid\ uidl\ uidl2$
 $\langle proof \rangle$

lemma $eqButUIDl$ -*remove1-cong*:
assumes $eqButUIDl\ uid\ uidl\ uidl1$
shows $eqButUIDl\ uid\ (remove1\ uid'\ uidl)\ (remove1\ uid'\ uidl1)$
 $\langle proof \rangle$

lemma $eqButUIDl$ -*snoc-cong*:
assumes $eqButUIDl\ uid\ uidl\ uidl1$
and $uid' \in\in\ uidl \longleftrightarrow uid' \in\in\ uidl1$
shows $eqButUIDl\ uid\ (uidl\ \#\#\ uid')\ (uidl1\ \#\#\ uid')$
 $\langle proof \rangle$

definition $eqButUIDf$ **where**
 $eqButUIDf\ frds\ frds1 \equiv$
 $eqButUIDl\ UID2\ (frds\ UID1)\ (frds1\ UID1)$
 $\wedge eqButUIDl\ UID1\ (frds\ UID2)\ (frds1\ UID2)$
 $\wedge (\forall uid. uid \neq UID1 \wedge uid \neq UID2 \longrightarrow frds\ uid = frds1\ uid)$

lemmas $eqButUIDf$ -*intro* = $eqButUIDf$ -*def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma $eqButUIDf$ - eeq [*simp,intro!*]: $eqButUIDf\ frds\ frds$
 $\langle proof \rangle$

lemma *eqButUIDf-sym*:
assumes *eqButUIDf frds frds1* **shows** *eqButUIDf frds1 frds*
 ⟨*proof*⟩

lemma *eqButUIDf-trans*:
assumes *eqButUIDf frds frds1* **and** *eqButUIDf frds1 frds2*
shows *eqButUIDf frds frds2*
 ⟨*proof*⟩

lemma *eqButUIDf-cong*:
assumes *eqButUIDf frds frds1*
and $uid = UID1 \implies eqButUIDl\ UID2\ uu\ uu1$
and $uid = UID2 \implies eqButUIDl\ UID1\ uu\ uu1$
and $uid \neq UID1 \implies uid \neq UID2 \implies uu = uu1$
shows *eqButUIDf (frds (uid := uu)) (frds1 (uid := uu1))*
 ⟨*proof*⟩

lemma *eqButUIDf-eqButUIDl*:
assumes *eqButUIDf frds frds1*
shows *eqButUIDl UID2 (frds UID1) (frds1 UID1)*
and *eqButUIDl UID1 (frds UID2) (frds1 UID2)*
 ⟨*proof*⟩

lemma *eqButUIDf-not-UID*:
 $\llbracket eqButUIDf\ frds\ frds1; uid \neq UID1; uid \neq UID2 \rrbracket \implies frds\ uid = frds1\ uid$
 ⟨*proof*⟩

lemma *eqButUIDf-not-UID'*:
assumes *eq1: eqButUIDf frds frds1*
and $uid: (uid, uid') \notin \{(UID1, UID2), (UID2, UID1)\}$
shows $uid \in \in frds\ uid' \longleftrightarrow uid \in \in frds1\ uid'$
 ⟨*proof*⟩

definition *eqButUID12* **where**
eqButUID12 *freq* *freq1* \equiv
 $\forall\ uid\ uid'.\ if\ (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}\ then\ True\ else\ freq\ uid$
 $uid' = freq1\ uid\ uid'$

lemmas *eqButUID12-intro* = *eqButUID12-def*[*THEN meta-eq-to-obj-eq, THEN iffD2*]

lemma *eqButUID12-eeq[simp,intro!]*: *eqButUID12* *freq* *freq*
 ⟨*proof*⟩

lemma *eqButUID12-sym*:
assumes *eqButUID12* *freq* *freq1* **shows** *eqButUID12* *freq1* *freq*
 ⟨*proof*⟩

lemma *eqButUID12-trans*:
assumes *eqButUID12 freq freq1 and eqButUID12 freq1 freq2*
shows *eqButUID12 freq freq2*
 $\langle proof \rangle$

lemma *eqButUID12-cong*:
assumes *eqButUID12 freq freq1*

and $\neg (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \implies uu = uu1$
shows *eqButUID12 (fun-upd2 freq uid uid' uu) (fun-upd2 freq1 uid uid' uu1)*
 $\langle proof \rangle$

lemma *eqButUID12-not-UID*:
 $\llbracket eqButUID12 freq freq1; \neg (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \rrbracket \implies freq$
 $uid uid' = freq1 uid uid'$
 $\langle proof \rangle$

definition *eqButUID* :: *state* \Rightarrow *state* \Rightarrow *bool* **where**
eqButUID s s1 \equiv
admin s = admin s1 \wedge

pendingUReqs s = pendingUReqs s1 \wedge *userReq s = userReq s1* \wedge
userIDs s = userIDs s1 \wedge *user s = user s1* \wedge *pass s = pass s1* \wedge

eqButUIDf (pendingFReqs s) (pendingFReqs s1) \wedge
eqButUID12 (friendReq s) (friendReq s1) \wedge
eqButUIDf (friendIDs s) (friendIDs s1) \wedge

postIDs s = postIDs s1 \wedge *admin s = admin s1* \wedge
post s = post s1 \wedge
owner s = owner s1 \wedge
vis s = vis s1

lemmas *eqButUID-intro = eqButUID-def[THEN meta-eq-to-obj-eq, THEN iffD2]*

lemma *eqButUID-refl[simp,intro!]*: *eqButUID s s*
 $\langle proof \rangle$

lemma *eqButUID-sym[sym]*:
assumes *eqButUID s s1* **shows** *eqButUID s1 s*
 $\langle proof \rangle$

lemma *eqButUID-trans[trans]*:
assumes *eqButUID s s1 and eqButUID s1 s2* **shows** *eqButUID s s2*
 $\langle proof \rangle$

lemma *eqButUID-stateSelectors*:

$eqButUID\ s\ s1 \implies$
 $admin\ s = admin\ s1 \wedge$

$pendingUReqs\ s = pendingUReqs\ s1 \wedge userReq\ s = userReq\ s1 \wedge$
 $userIDs\ s = userIDs\ s1 \wedge user\ s = user\ s1 \wedge pass\ s = pass\ s1 \wedge$

$eqButUIDf\ (pendingFReqs\ s)\ (pendingFReqs\ s1) \wedge$
 $eqButUID12\ (friendReq\ s)\ (friendReq\ s1) \wedge$
 $eqButUIDf\ (friendIDs\ s)\ (friendIDs\ s1) \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$
 $post\ s = post\ s1 \wedge$
 $owner\ s = owner\ s1 \wedge$
 $vis\ s = vis\ s1 \wedge$

$IDsOK\ s = IDsOK\ s1$
 $\langle proof \rangle$

lemma *eqButUID-eqButUID2*:

$eqButUID\ s\ s1 \implies eqButUID1\ UID2\ (friendIDs\ s\ UID1)\ (friendIDs\ s1\ UID1)$
 $\langle proof \rangle$

lemma *eqButUID-not-UID*:

$eqButUID\ s\ s1 \implies uid \neq UID \implies post\ s\ uid = post\ s1\ uid$
 $\langle proof \rangle$

lemma *eqButUID-cong[simp, intro]*:

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|admin := uu1|))$
 $(s1\ (\!|admin := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|pendingUReqs :=$
 $uu1|))\ (s1\ (\!|pendingUReqs := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|userReq := uu1|))$
 $(s1\ (\!|userReq := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|userIDs := uu1|))$
 $(s1\ (\!|userIDs := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|user := uu1|))\ (s1$
 $(\!|user := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|pass := uu1|))\ (s1$
 $(\!|pass := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|postIDs := uu1|))$
 $(s1\ (\!|postIDs := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|owner := uu1|))$
 $(s1\ (\!|owner := uu2|))$

$\bigwedge uu1\ uu2. eqButUID\ s\ s1 \implies uu1 = uu2 \implies eqButUID\ (s\ (\!|post := uu1|))\ (s1$
 $(\!|post := uu2|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s \langle vis := uu1 \rangle) (s1 \langle vis := uu2 \rangle)$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s \langle pendingFReqs := uu1 \rangle) (s1 \langle pendingFReqs := uu2 \rangle)$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUID12 uu1 uu2 \implies eqButUID (s \langle friendReq := uu1 \rangle) (s1 \langle friendReq := uu2 \rangle)$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s \langle friendIDs := uu1 \rangle) (s1 \langle friendIDs := uu2 \rangle)$

$\langle proof \rangle$

7.2 Value Setup

datatype *value* =

FrVal *bool* — updated friendship status between *UID1* and *UID2*
OVal *bool* — updated dynamic declassification trigger condition

The dynamic declassification trigger condition holds, i.e. the access window to the confidential information is open, as long as the two users have not been created yet (so there cannot be friendship between them) or one of them is friends with an observer.

definition *openByA* :: *state* \Rightarrow *bool* — Openness by absence

where *openByA* *s* $\equiv \neg UID1 \in \in userIDs s \vee \neg UID2 \in \in userIDs s$

definition *openByF* :: *state* \Rightarrow *bool* — Openness by friendship

where *openByF* *s* $\equiv \exists uid \in UIDs. uid \in \in friendIDs s UID1 \vee uid \in \in friendIDs s UID2$

definition *open* :: *state* \Rightarrow *bool*

where *open* *s* $\equiv openByA s \vee openByF s$

lemmas *open-defs* = *open-def openByA-def openByF-def*

definition *friends12* :: *state* \Rightarrow *bool*

where *friends12* *s* $\equiv UID1 \in \in friendIDs s UID2 \wedge UID2 \in \in friendIDs s UID1$

fun φ :: (*state,act,out*) *trans* \Rightarrow *bool* **where**

$\varphi (Trans s (Cact (cFriend uid p uid')) ou s') =$
 $((uid,uid') \in \{(UID1,UID2), (UID2,UID1)\} \wedge ou = outOK \vee$
 $open s \neq open s')$

$\varphi (Trans s (Dact (dFriend uid p uid')) ou s') =$
 $((uid,uid') \in \{(UID1,UID2), (UID2,UID1)\} \wedge ou = outOK \vee$
 $open s \neq open s')$

$\varphi (Trans s (Cact (cUser uid p uid' p')) ou s') =$
 $(open s \neq open s')$

lemma *step-open-φ*:
assumes *step s a = (ou, s')*
and *open s ≠ open s'*
shows φ (*Trans s a ou s'*)
<proof>

lemma *step-friends12-φ*:
assumes *step s a = (ou, s')*
and *friends12 s ≠ friends12 s'*
shows φ (*Trans s a ou s'*)
<proof>

lemma *eqButUID-friends12-set-friendIDs-eq*:
assumes *ss1: eqButUID s s1*
and *f12: friends12 s = friends12 s1*
and *rs: reach s and rs1: reach s1*
shows $\text{set } (\text{friendIDs } s \text{ uid}) = \text{set } (\text{friendIDs } s1 \text{ uid})$
<proof>

lemma *distinct-remove1-idem*: $\text{distinct } xs \implies \text{remove1 } y (\text{remove1 } y \text{ xs}) = \text{remove1 } y \text{ xs}$
<proof>

lemma *Cact-cFriend-step-eqButUID*:
assumes *step: step s (Cact (cFriend uid p uid')) = (ou, s')*
and *s: reach s*
and *uids: (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ uid' = UID1) (is ?u12 ∨ ?u21)*
shows *eqButUID s s'*
<proof>

lemma *Cact-cFriendReq-step-eqButUID*:
assumes *step: step s (Cact (cFriendReq uid p uid' req)) = (ou, s')*
and *uids: (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ uid' = UID1) (is ?u12 ∨ ?u21)*
shows *eqButUID s s'*
<proof>

lemma *Dact-dFriend-step-eqButUID*:
assumes *step: step s (Dact (dFriend uid p uid')) = (ou, s')*
and *s: reach s*
and *uids: (uid = UID1 ∧ uid' = UID2) ∨ (uid = UID2 ∧ uid' = UID1) (is ?u12 ∨ ?u21)*
shows *eqButUID s s'*
<proof>

lemma *eqButUID-step*:
assumes *ss1*: *eqButUID s s1*
and *step*: *step s a = (ou,s')*
and *step1*: *step s1 a = (ou1,s1')*
and *rs*: *reach s*
and *rs1*: *reach s1*
shows *eqButUID s' s1'*
<proof>

lemma *eqButUID-openByA-eq*:
assumes *eqButUID s s1*
shows *openByA s = openByA s1*
<proof>

lemma *eqButUID-openByF-eq*:
assumes *ss1*: *eqButUID s s1*
shows *openByF s = openByF s1*
<proof>

lemma *eqButUID-open-eq*: *eqButUID s s1 \implies open s = open s1*
<proof>

lemma *eqButUID-step-friendIDs-eq*:
assumes *ss1*: *eqButUID s s1*
and *rs*: *reach s* **and** *rs1*: *reach s1*
and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*
and *a*: *a \neq Cact (cFriend UID1 (pass s UID1) UID2) \wedge a \neq Cact (cFriend UID2 (pass s UID2) UID1) \wedge*
a \neq Dact (dFriend UID1 (pass s UID1) UID2) \wedge a \neq Dact (dFriend UID2 (pass s UID2) UID1)
and *friendIDs s = friendIDs s1*
shows *friendIDs s' = friendIDs s1'*
<proof>

lemma *eqButUID-step- φ -imp*:
assumes *ss1*: *eqButUID s s1*
and *rs*: *reach s* **and** *rs1*: *reach s1*
and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*
and *a*: *a \neq Cact (cFriend UID1 (pass s UID1) UID2) \wedge a \neq Cact (cFriend UID2 (pass s UID2) UID1) \wedge*
a \neq Dact (dFriend UID1 (pass s UID1) UID2) \wedge a \neq Dact (dFriend UID2 (pass s UID2) UID1)
and *φ : φ (Trans s a ou s')*
shows *φ (Trans s1 a ou1 s1')*
<proof>

lemma *eqButUID-step- φ* :

assumes $ss1: eqButUID\ s\ s1$
and $rs: reach\ s$ **and** $rs1: reach\ s1$
and $step: step\ s\ a = (ou, s')$ **and** $step1: step\ s1\ a = (ou1, s1')$
and $a \neq Cact\ (cFriend\ UID1\ (pass\ s\ UID1)\ UID2) \wedge a \neq Cact\ (cFriend\ UID2\ (pass\ s\ UID2)\ UID1) \wedge$
 $a \neq Dact\ (dFriend\ UID1\ (pass\ s\ UID1)\ UID2) \wedge a \neq Dact\ (dFriend\ UID2\ (pass\ s\ UID2)\ UID1)$
shows $\varphi\ (Trans\ s\ a\ ou\ s') = \varphi\ (Trans\ s1\ a\ ou1\ s1')$
 $\langle proof \rangle$

lemma *createFriend-sym*: $createFriend\ s\ uid\ p\ uid' = createFriend\ s\ uid'\ p'\ uid$
 $\langle proof \rangle$

lemma *deleteFriend-sym*: $deleteFriend\ s\ uid\ p\ uid' = deleteFriend\ s\ uid'\ p'\ uid$
 $\langle proof \rangle$

lemma *createFriendReq-createFriend-absorb*:
assumes $e\text{-}createFriendReq\ s\ uid'\ p\ uid\ req$
shows $createFriend\ (createFriendReq\ s\ uid'\ p1\ uid\ req)\ uid\ p2\ uid' = createFriend\ s\ uid\ p3\ uid'$
 $\langle proof \rangle$

lemma *eqButUID-deleteFriend12-friendIDs-eq*:
assumes $ss1: eqButUID\ s\ s1$
and $rs: reach\ s$ **and** $rs1: reach\ s1$
shows $friendIDs\ (deleteFriend\ s\ UID1\ p\ UID2) = friendIDs\ (deleteFriend\ s1\ UID1\ p'\ UID2)$
 $\langle proof \rangle$

lemma *eqButUID-createFriend12-friendIDs-eq*:
assumes $ss1: eqButUID\ s\ s1$
and $rs: reach\ s$ **and** $rs1: reach\ s1$
and $f12: \neg friends12\ s\ \neg friends12\ s1$
shows $friendIDs\ (createFriend\ s\ UID1\ p\ UID2) = friendIDs\ (createFriend\ s1\ UID1\ p'\ UID2)$
 $\langle proof \rangle$

end
theory *Friend*
imports $../Observation-Setup\ Friend-Value-Setup$
begin

7.3 Declassification bound

fun $T :: (state, act, out)\ trans \Rightarrow bool$
where $T\ (Trans\ -\ -\ -) = False$

The bound follows the same “while-or-last-before” scheme as the bound for post confidentiality (Section 6.3), alternating between open (*BO*) and closed (*BC*) phases.

The access window is initially open, because the two users are known not to exist when the system is initialized, so there cannot be friendship between them.

The bound also incorporates the static knowledge that the friendship status alternates between *False* and *True*.

```
fun alternatingFriends :: value list  $\Rightarrow$  bool  $\Rightarrow$  bool where
  alternatingFriends [] - = True
| alternatingFriends (FrVal st # vl) st'  $\longleftrightarrow$  st' = ( $\neg$ st)  $\wedge$  alternatingFriends vl st
| alternatingFriends (OVal - # vl) st = alternatingFriends vl st
```

```
inductive BO :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
and BC :: value list  $\Rightarrow$  value list  $\Rightarrow$  bool
where
```

```
  BO-FrVal[simp,intro!]:
    BO (map FrVal fs) (map FrVal fs)
|BO-BC[intro]:
  BC vl vl1  $\Longrightarrow$ 
    BO (map FrVal fs @ OVal False # vl) (map FrVal fs @ OVal False # vl1)
```

```
|BC-FrVal[simp,intro!]:
  BC (map FrVal fs) (map FrVal fs1)
|BC-BO[intro]:
  BO vl vl1  $\Longrightarrow$  (fs = []  $\longleftrightarrow$  fs1 = [])  $\Longrightarrow$  (fs  $\neq$  []  $\Longrightarrow$  last fs = last fs1)  $\Longrightarrow$ 
    BC (map FrVal fs @ OVal True # vl)
      (map FrVal fs1 @ OVal True # vl1)
```

definition $B\ vl\ vl1 \equiv BO\ vl\ vl1 \wedge alternatingFriends\ vl1\ False$

lemma *BO-Nil-Nil*: $BO\ vl\ vl1 \Longrightarrow vl = [] \Longrightarrow vl1 = []$
<proof>

unbundle *no relcomp-syntax*

```
interpretation BD-Security-IO where
  istate = istate and step = step and
   $\varphi = \varphi$  and  $f = f$  and  $\gamma = \gamma$  and  $g = g$  and  $T = T$  and  $B = B$ 
<proof>
```

7.4 Unwinding proof

```
lemma eqButUID-step- $\gamma$ -out:
assumes ss1: eqButUID s s1
and step: step s a = (ou,s') and step1: step s1 a = (ou1,s1')
and  $\gamma$ :  $\gamma$  (Trans s a ou s')
and os: open s  $\longrightarrow$  friendIDs s = friendIDs s1
shows ou = ou1
<proof>
```

lemma *toggle-friends12-True*:
assumes *rs*: *reach s*
 and *IDs*: *IDsOK s [UID1, UID2]* []
 and *nf12*: \neg *friends12 s*
obtains *al oul*
where *sstep s al* = (*oul*, *createFriend s UID1 (pass s UID1) UID2*)
 and *al* \neq [] **and** *eqButUID s (createFriend s UID1 (pass s UID1) UID2)*
 and *friends12 (createFriend s UID1 (pass s UID1) UID2)*
 and *O (traceOf s al)* = [] **and** *V (traceOf s al)* = [*FrVal True*]
<proof>

lemma *toggle-friends12-False*:
assumes *rs*: *reach s*
 and *IDs*: *IDsOK s [UID1, UID2]* []
 and *f12*: *friends12 s*
obtains *al oul*
where *sstep s al* = (*oul*, *deleteFriend s UID1 (pass s UID1) UID2*)
 and *al* \neq [] **and** *eqButUID s (deleteFriend s UID1 (pass s UID1) UID2)*
 and \neg *friends12 (deleteFriend s UID1 (pass s UID1) UID2)*
 and *O (traceOf s al)* = [] **and** *V (traceOf s al)* = [*FrVal False*]
<proof>

definition $\Delta 0$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 0$ *s vl s1 vl1* \equiv
 eqButUID s s1 \wedge *friendIDs s = friendIDs s1* \wedge *open s* \wedge
 BO vl vl1 \wedge *alternatingFriends vl1 (friends12 s1)*

definition $\Delta 1$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 1$ *s vl s1 vl1* \equiv (\exists *fs fs1*.
 eqButUID s s1 \wedge \neg *open s* \wedge
 alternatingFriends vl1 (friends12 s1) \wedge
 vl = map FrVal fs \wedge *vl1 = map FrVal fs1*)

definition $\Delta 2$:: *state* \Rightarrow *value list* \Rightarrow *state* \Rightarrow *value list* \Rightarrow *bool* **where**
 $\Delta 2$ *s vl s1 vl1* \equiv (\exists *fs fs1 vlr vlr1*.
 eqButUID s s1 \wedge \neg *open s* \wedge *BO vlr vlr1* \wedge
 alternatingFriends vl1 (friends12 s1) \wedge
 (*fs* = [] \iff *fs1* = []) \wedge
 (*fs* \neq [] \implies *last fs* = *last fs1*) \wedge
 (*fs* = [] \implies *friendIDs s* = *friendIDs s1*) \wedge
 vl = map FrVal fs @ OVal True # vlr \wedge
 vl1 = map FrVal fs1 @ OVal True # vlr1)

lemma $\Delta 2$ -I:
assumes *eqButUID s s1* \neg *open s* *BO vlr vlr1* *alternatingFriends vl1 (friends12 s1)*
 fs = [] \iff *fs1* = [] *fs* \neq [] \implies *last fs* = *last fs1*

```

    fs = []  $\longrightarrow$  friendIDs s = friendIDs s1
    vl = map FrVal fs @ OVal True # vlr
    vl1 = map FrVal fs1 @ OVal True # vlr1
shows  $\Delta 2$  s vl s1 vl1
<proof>

lemma istate- $\Delta 0$ :
assumes B: B vl vl1
shows  $\Delta 0$  istate vl istate vl1
<proof>

lemma unwind-cont- $\Delta 0$ : unwind-cont  $\Delta 0$  { $\Delta 0, \Delta 1, \Delta 2$ }
<proof>

lemma unwind-cont- $\Delta 1$ : unwind-cont  $\Delta 1$  { $\Delta 1, \Delta 0$ }
<proof>

lemma unwind-cont- $\Delta 2$ : unwind-cont  $\Delta 2$  { $\Delta 2, \Delta 0$ }
<proof>

definition Gr where
Gr =
{
( $\Delta 0$ , { $\Delta 0, \Delta 1, \Delta 2$ }),
( $\Delta 1$ , { $\Delta 1, \Delta 0$ }),
( $\Delta 2$ , { $\Delta 2, \Delta 0$ })
}

theorem secure: secure
<proof>

end
theory Friend-Request-Intro
imports ../Safety-Properties ../Observation-Setup
begin

```

8 Friendship request confidentiality

We prove the following property:

Given a group of users *UIDs* and given two users *UID1* and *UID2* not in that group,
that group cannot learn anything about the friendship requests issued be-

tween $UID1$ and $UID2$
beyond what everybody knows, namely that

- there is no friendship between $UID1$ and $UID2$ before those users have been created, and
- friendship status updates form an alternating sequence of friending and unfriending, every successful friend creation is preceded by at least one and at most two requests,

and beyond those requests performed while or last before a user in $UIDs$ is friends with $UID1$ or $UID2$.

end

theory *Friend-Request-Value-Setup*
imports *Friend-Request-Intro*
begin

The confidential information is the friendship requests between two arbitrary but fixed users:

consts $UID1 :: userID$
consts $UID2 :: userID$

axiomatization **where**
 $UID1-UID2-UIDs: \{UID1, UID2\} \cap UIDs = \{\}$
and
 $UID1-UID2: UID1 \neq UID2$

8.1 Preliminaries

fun $eqButUIDl :: userID \Rightarrow userID list \Rightarrow userID list \Rightarrow bool$ **where**
 $eqButUIDl uid uidl uidl1 = (remove1 uid uidl = remove1 uid uidl1)$

lemma $eqButUIDl-eq[simp,intro!]: eqButUIDl uid uidl uidl$
<proof>

lemma $eqButUIDl-sym:$
assumes $eqButUIDl uid uidl uidl1$
shows $eqButUIDl uid uidl1 uidl$
<proof>

lemma $eqButUIDl-trans:$
assumes $eqButUIDl uid uidl uidl1$ **and** $eqButUIDl uid uidl1 uidl2$
shows $eqButUIDl uid uidl uidl2$
<proof>

lemma $eqButUIDl-remove1-cong:$
assumes $eqButUIDl uid uidl uidl1$

shows $eqButUIDl\ uid\ (remove1\ uid'\ uid)\ (remove1\ uid'\ uid1)$
 $\langle proof \rangle$

lemma $eqButUIDl\ snoc\ cong$:
assumes $eqButUIDl\ uid\ uid1\ uid11$
and $uid' \in\in\ uid1 \longleftrightarrow uid' \in\in\ uid11$
shows $eqButUIDl\ uid\ (uid1\ \#\#\ uid')\ (uid11\ \#\#\ uid')$
 $\langle proof \rangle$

definition $eqButUIDf\ where$
 $eqButUIDf\ frds\ frds1 \equiv$
 $eqButUIDl\ UID2\ (frds\ UID1)\ (frds1\ UID1)$
 $\wedge eqButUIDl\ UID1\ (frds\ UID2)\ (frds1\ UID2)$
 $\wedge (\forall uid.\ uid \neq UID1 \wedge uid \neq UID2 \longrightarrow frds\ uid = frds1\ uid)$

lemmas $eqButUIDf\ intro = eqButUIDf\ def[THEN\ meta\ eq\ to\ obj\ eq,\ THEN\ iffD2]$

lemma $eqButUIDf\ eeq[simp,intro!]$: $eqButUIDf\ frds\ frds$
 $\langle proof \rangle$

lemma $eqButUIDf\ sym$:
assumes $eqButUIDf\ frds\ frds1$ **shows** $eqButUIDf\ frds1\ frds$
 $\langle proof \rangle$

lemma $eqButUIDf\ trans$:
assumes $eqButUIDf\ frds\ frds1$ **and** $eqButUIDf\ frds1\ frds2$
shows $eqButUIDf\ frds\ frds2$
 $\langle proof \rangle$

lemma $eqButUIDf\ cong$:
assumes $eqButUIDf\ frds\ frds1$
and $uid = UID1 \implies eqButUIDl\ UID2\ uu\ uu1$
and $uid = UID2 \implies eqButUIDl\ UID1\ uu\ uu1$
and $uid \neq UID1 \implies uid \neq UID2 \implies uu = uu1$
shows $eqButUIDf\ (frds\ (uid := uu))\ (frds1\ (uid := uu1))$
 $\langle proof \rangle$

lemma $eqButUIDf\ eqButUIDl$:
assumes $eqButUIDf\ frds\ frds1$
shows $eqButUIDl\ UID2\ (frds\ UID1)\ (frds1\ UID1)$
and $eqButUIDl\ UID1\ (frds\ UID2)\ (frds1\ UID2)$
 $\langle proof \rangle$

lemma $eqButUIDf\ not\ UID$:
 $\llbracket eqButUIDf\ frds\ frds1;\ uid \neq UID1;\ uid \neq UID2 \rrbracket \implies frds\ uid = frds1\ uid$
 $\langle proof \rangle$

lemma $eqButUIDf\ not\ UID'$:

assumes $eq1: eqButUIDf\ frds\ frds1$
and $uid: (uid,uid') \notin \{(UID1,UID2), (UID2,UID1)\}$
shows $uid \in \in frds\ uid' \iff uid \in \in frds1\ uid'$
 $\langle proof \rangle$

definition $eqButUID12$ **where**

$eqButUID12\ freq\ freq1 \equiv$
 $\forall\ uid\ uid'.\ if\ (uid,uid') \in \{(UID1,UID2), (UID2,UID1)\}\ then\ True\ else\ freq\ uid$
 $uid' = freq1\ uid\ uid'$

lemmas $eqButUID12-intro = eqButUID12-def[THEN\ meta-eq-to-obj-eq,\ THEN$
 $iffD2]$

lemma $eqButUID12-eeq[simp,intro!]: eqButUID12\ freq\ freq$
 $\langle proof \rangle$

lemma $eqButUID12-sym:$

assumes $eqButUID12\ freq\ freq1$ **shows** $eqButUID12\ freq1\ freq$
 $\langle proof \rangle$

lemma $eqButUID12-trans:$

assumes $eqButUID12\ freq\ freq1$ **and** $eqButUID12\ freq1\ freq2$
shows $eqButUID12\ freq\ freq2$
 $\langle proof \rangle$

lemma $eqButUID12-cong:$

assumes $eqButUID12\ freq\ freq1$
and $\neg (uid,uid') \in \{(UID1,UID2), (UID2,UID1)\} \implies uu = uu1$
shows $eqButUID12\ (fun-upd2\ freq\ uid\ uid'\ uu)\ (fun-upd2\ freq1\ uid\ uid'\ uu1)$
 $\langle proof \rangle$

lemma $eqButUID12-not-UID:$

$\llbracket eqButUID12\ freq\ freq1; \neg (uid,uid') \in \{(UID1,UID2), (UID2,UID1)\} \rrbracket \implies freq$
 $uid\ uid' = freq1\ uid\ uid'$
 $\langle proof \rangle$

definition $eqButUID :: state \Rightarrow state \Rightarrow bool$ **where**

$eqButUID\ s\ s1 \equiv$
 $admin\ s = admin\ s1 \wedge$

$pendingUReqs\ s = pendingUReqs\ s1 \wedge userReq\ s = userReq\ s1 \wedge$
 $userIDs\ s = userIDs\ s1 \wedge user\ s = user\ s1 \wedge pass\ s = pass\ s1 \wedge$

$eqButUIDf\ (pendingFReqs\ s)\ (pendingFReqs\ s1) \wedge$
 $eqButUID12\ (friendReq\ s)\ (friendReq\ s1) \wedge$
 $eqButUIDf\ (friendIDs\ s)\ (friendIDs\ s1) \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$
 $post\ s = post\ s1 \wedge$
 $owner\ s = owner\ s1 \wedge$
 $vis\ s = vis\ s1$

lemmas $eqButUID-intro = eqButUID-def[THEN\ meta-eq-to-obj-eq,\ THEN\ iffD2]$

lemma $eqButUID-refl[simp,intro!]: eqButUID\ s\ s$
 $\langle proof \rangle$

lemma $eqButUID-sym[sym]:$
assumes $eqButUID\ s\ s1$ **shows** $eqButUID\ s1\ s$
 $\langle proof \rangle$

lemma $eqButUID-trans[trans]:$
assumes $eqButUID\ s\ s1$ **and** $eqButUID\ s1\ s2$ **shows** $eqButUID\ s\ s2$
 $\langle proof \rangle$

lemma $eqButUID-stateSelectors:$

$eqButUID\ s\ s1 \implies$
 $admin\ s = admin\ s1 \wedge$

$pendingUReqs\ s = pendingUReqs\ s1 \wedge userReq\ s = userReq\ s1 \wedge$
 $userIDs\ s = userIDs\ s1 \wedge user\ s = user\ s1 \wedge pass\ s = pass\ s1 \wedge$

$eqButUIDf\ (pendingFReqs\ s)\ (pendingFReqs\ s1) \wedge$
 $eqButUID12\ (friendReq\ s)\ (friendReq\ s1) \wedge$
 $eqButUIDf\ (friendIDs\ s)\ (friendIDs\ s1) \wedge$

$postIDs\ s = postIDs\ s1 \wedge admin\ s = admin\ s1 \wedge$
 $post\ s = post\ s1 \wedge$
 $owner\ s = owner\ s1 \wedge$
 $vis\ s = vis\ s1 \wedge$

$IDsOK\ s = IDsOK\ s1$
 $\langle proof \rangle$

lemma $eqButUID-eqButUID2:$
 $eqButUID\ s\ s1 \implies eqButUID1\ UID2\ (friendIDs\ s\ UID1)\ (friendIDs\ s1\ UID1)$
 $\langle proof \rangle$

lemma $eqButUID-not-UID:$
 $eqButUID\ s\ s1 \implies uid \neq UID \implies post\ s\ uid = post\ s1\ uid$
 $\langle proof \rangle$

lemma $eqButUID-cong[simp,\ intro]:$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|admin := uu1\!|)) (s1 (\!|admin := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|pendingUReqs := uu1\!|)) (s1 (\!|pendingUReqs := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|userReq := uu1\!|)) (s1 (\!|userReq := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|userIDs := uu1\!|)) (s1 (\!|userIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|user := uu1\!|)) (s1 (\!|user := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|pass := uu1\!|)) (s1 (\!|pass := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|postIDs := uu1\!|)) (s1 (\!|postIDs := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|owner := uu1\!|)) (s1 (\!|owner := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|post := uu1\!|)) (s1 (\!|post := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies uu1 = uu2 \implies eqButUID (s (\!|vis := uu1\!|)) (s1 (\!|vis := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\!|pendingFReqs := uu1\!|)) (s1 (\!|pendingFReqs := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUID12 uu1 uu2 \implies eqButUID (s (\!|friendReq := uu1\!|)) (s1 (\!|friendReq := uu2\!|))$

$\bigwedge uu1 uu2. eqButUID s s1 \implies eqButUIDf uu1 uu2 \implies eqButUID (s (\!|friendIDs := uu1\!|)) (s1 (\!|friendIDs := uu2\!|))$

$\langle proof \rangle$

8.2 Value Setup

datatype $fUser = U1 \mid U2$

datatype $value =$

$isFRVal: FRVal fUser req$ — friendship requests from $UID1$ to $UID2$ (or vice versa)

$isFVal: FVal bool$ — updates to the status of friendship between them

$isOVal: OVal bool$ — updated dynamic declassification trigger condition

The dynamic declassification trigger condition holds, i.e. the access window to the confidential information is open, as long as the two users have not been created yet (so there cannot be friendship between them) or one of them is friends with an observer.

definition $openByA :: state \Rightarrow bool$ — Openness by absence

where $openByA s \equiv \neg UID1 \in \in userIDs s \vee \neg UID2 \in \in userIDs s$

definition $openByF :: state \Rightarrow bool$ — Openness by friendship

where $openByF\ s \equiv \exists uid \in UIDs. uid \in\in friendIDs\ s\ UID1 \vee uid \in\in friendIDs\ s\ UID2$

definition $open :: state \Rightarrow bool$
where $open\ s \equiv openByA\ s \vee openByF\ s$

lemmas $open-defs = open-def\ openByA-def\ openByF-def$

definition $friends12 :: state \Rightarrow bool$
where $friends12\ s \equiv UID1 \in\in friendIDs\ s\ UID2 \wedge UID2 \in\in friendIDs\ s\ UID1$

fun $\varphi :: (state, act, out)\ trans \Rightarrow bool$ **where**
 $\varphi\ (Trans\ s\ (Cact\ (cFriendReq\ uid\ p\ uid'\ req))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK)$
 $|$
 $\varphi\ (Trans\ s\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK \vee$
 $open\ s \neq open\ s')$
 $|$
 $\varphi\ (Trans\ s\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou\ s') =$
 $((uid, uid') \in \{(UID1, UID2), (UID2, UID1)\} \wedge ou = outOK \vee$
 $open\ s \neq open\ s')$
 $|$
 $\varphi\ (Trans\ s\ (Cact\ (cUser\ uid\ p\ uid'\ p'))\ ou\ s') =$
 $(open\ s \neq open\ s')$
 $|$
 $\varphi\ - = False$

fun $f :: (state, act, out)\ trans \Rightarrow value$ **where**
 $f\ (Trans\ s\ (Cact\ (cFriendReq\ uid\ p\ uid'\ req))\ ou\ s') =$
 $(if\ uid = UID1 \wedge uid' = UID2\ then\ FRVal\ U1\ req$
 $else\ if\ uid = UID2 \wedge uid' = UID1\ then\ FRVal\ U2\ req$
 $else\ OVal\ True)$
 $|$
 $f\ (Trans\ s\ (Cact\ (cFriend\ uid\ p\ uid'))\ ou\ s') =$
 $(if\ (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}\ then\ FVal\ True$
 $else\ OVal\ True)$
 $|$
 $f\ (Trans\ s\ (Dact\ (dFriend\ uid\ p\ uid'))\ ou\ s') =$
 $(if\ (uid, uid') \in \{(UID1, UID2), (UID2, UID1)\}\ then\ FVal\ False$
 $else\ OVal\ False)$
 $|$
 $f\ (Trans\ s\ (Cact\ (cUser\ uid\ p\ uid'\ p'))\ ou\ s') = OVal\ False$
 $|$
 $f\ - = undefined$

lemma φE :
assumes $\varphi: \varphi\ (Trans\ s\ a\ ou\ s')\ (is\ \varphi\ ?trn)$

and *step*: $step\ s\ a = (ou, s')$
and *rs*: $reach\ s$
obtains $(FReq1)\ u\ p\ req$ **where** $a = Cact\ (cFriendReq\ UID1\ p\ UID2\ req)\ ou = outOK$

$$\begin{aligned}
& f\ ?trn = FRVal\ u\ req\ u = U1\ IDsOK\ s\ [UID1, UID2]\ [] \\
& \neg friends12\ s\ \neg friends12\ s'\ open\ s' = open\ s \\
& UID1 \in \in pendingFReqs\ s'\ UID2\ UID1 \notin set\ (pendingFReqs\ s\ UID2) \\
& UID2 \in \in pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in \in pendingFReqs\ s\ UID1
\end{aligned}$$

or $(FReq2)\ u\ p\ req$ **where** $a = Cact\ (cFriendReq\ UID2\ p\ UID1\ req)\ ou = outOK$

$$\begin{aligned}
& f\ ?trn = FRVal\ u\ req\ u = U2\ IDsOK\ s\ [UID1, UID2]\ [] \\
& \neg friends12\ s\ \neg friends12\ s'\ open\ s' = open\ s \\
& UID2 \in \in pendingFReqs\ s'\ UID1\ UID2 \notin set\ (pendingFReqs\ s\ UID1) \\
& UID1 \in \in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in \in pendingFReqs\ s\ UID2
\end{aligned}$$

or $(Friend)\ uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')\ ou = outOK\ f\ ?trn = FVal\ True$

$$\begin{aligned}
& uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1 \\
& IDsOK\ s\ [UID1, UID2]\ [] \\
& \neg friends12\ s\ friends12\ s'\ uid' \in \in pendingFReqs\ s\ uid \\
& UID1 \notin set\ (pendingFReqs\ s'\ UID2) \\
& UID2 \notin set\ (pendingFReqs\ s'\ UID1)
\end{aligned}$$

or $(Unfriend)\ uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')\ ou = outOK\ f\ ?trn = FVal\ False$

$$\begin{aligned}
& uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1 \\
& IDsOK\ s\ [UID1, UID2]\ [] \\
& friends12\ s\ \neg friends12\ s' \\
& UID1 \notin set\ (pendingFReqs\ s'\ UID2) \\
& UID1 \notin set\ (pendingFReqs\ s\ UID2) \\
& UID2 \notin set\ (pendingFReqs\ s'\ UID1) \\
& UID2 \notin set\ (pendingFReqs\ s\ UID1)
\end{aligned}$$

or $(OpenF)\ uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')$

$$\begin{aligned}
& (uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee (uid' \in UIDs \wedge uid \in \{UID1, UID2\}) \\
& ou = outOK\ f\ ?trn = OVal\ True\ \neg openByF\ s\ openByF\ s' \\
& \neg openByA\ s\ \neg openByA\ s' \\
& friends12\ s' = friends12\ s \\
& UID1 \in \in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in \in pendingFReqs\ s\ UID2 \\
& UID2 \in \in pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in \in pendingFReqs\ s\ UID1
\end{aligned}$$

or $(CloseF)\ uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')$

$$\begin{aligned}
& (uid \in UIDs \wedge uid' \in \{UID1, UID2\}) \vee (uid' \in UIDs \wedge uid \in \{UID1, UID2\})
\end{aligned}$$

s'

$ou = outOK f ?trn = OVal False openByF s \neg openByF$

$\neg openByA s \neg openByA s'$
 $friends12 s' = friends12 s$
 $UID1 \in \in pendingFReqs s' UID2 \longleftrightarrow UID1 \in \in$

$pendingFReqs s UID2$

$UID2 \in \in pendingFReqs s' UID1 \longleftrightarrow UID2 \in \in$

$pendingFReqs s UID1$

$| (CloseA) uid p uid' p' \mathbf{where} a = Cact (cUser uid p uid' p')$
 $uid' \in \{UID1, UID2\} openByA s \neg openByA s'$
 $\neg openByF s \neg openByF s'$
 $ou = outOK f ?trn = OVal False$
 $friends12 s' = friends12 s$
 $UID1 \in \in pendingFReqs s' UID2 \longleftrightarrow UID1 \in \in$

$pendingFReqs s UID2$

$UID2 \in \in pendingFReqs s' UID1 \longleftrightarrow UID2 \in \in$

$pendingFReqs s UID1$

$\langle proof \rangle$

lemma *step-open-φ*:
assumes $step s a = (ou, s')$
and $open s \neq open s'$
shows $\varphi (Trans s a ou s')$
 $\langle proof \rangle$

lemma *step-friends12-φ*:
assumes $step s a = (ou, s')$
and $friends12 s \neq friends12 s'$
shows $\varphi (Trans s a ou s')$
 $\langle proof \rangle$

lemma *step-pendingFReqs-φ*:
assumes $step s a = (ou, s')$
and $(UID1 \in \in pendingFReqs s UID2) \neq (UID1 \in \in pendingFReqs s' UID2)$
 $\vee (UID2 \in \in pendingFReqs s UID1) \neq (UID2 \in \in pendingFReqs s' UID1)$
shows $\varphi (Trans s a ou s')$
 $\langle proof \rangle$

lemma *eqButUID-friends12-set-friendIDs-eq*:
assumes $ss1: eqButUID s s1$
and $f12: friends12 s = friends12 s1$
and $rs: reach s$ **and** $rs1: reach s1$
shows $set (friendIDs s uid) = set (friendIDs s1 uid)$
 $\langle proof \rangle$

lemma *distinct-remove1-idem*: $distinct xs \implies remove1 y (remove1 y xs) = remove1 y xs$
 $\langle proof \rangle$

lemma *Cact-cFriend-step-eqButUID:*

assumes *step: step s (Cact (cFriend uid p uid')) = (ou,s')*

and *s: reach s*

and *uids: (uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1) (is ?u12 \vee ?u21)*

shows *eqButUID s s'*

<proof>

lemma *Cact-cFriendReq-step-eqButUID:*

assumes *step: step s (Cact (cFriendReq uid p uid' req)) = (ou,s')*

and *uids: (uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1) (is ?u12 \vee ?u21)*

shows *eqButUID s s'*

<proof>

lemma *Dact-dFriend-step-eqButUID:*

assumes *step: step s (Dact (dFriend uid p uid')) = (ou,s')*

and *s: reach s*

and *uids: (uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1) (is ?u12 \vee ?u21)*

shows *eqButUID s s'*

<proof>

lemma *eqButUID-step:*

assumes *ss1: eqButUID s s1*

and *step: step s a = (ou,s')*

and *step1: step s1 a = (ou1,s1')*

and *rs: reach s*

and *rs1: reach s1*

shows *eqButUID s' s1'*

<proof>

lemma *eqButUID-openByA-eq:*

assumes *eqButUID s s1*

shows *openByA s = openByA s1*

<proof>

lemma *eqButUID-openByF-eq:*

assumes *ss1: eqButUID s s1*

shows *openByF s = openByF s1*

<proof>

lemma *eqButUID-open-eq: eqButUID s s1 \implies open s = open s1*

<proof>

lemma *eqButUID-step-friendIDs-eq*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*

and *a*: $a \neq \text{Cact}(\text{cFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Cact}(\text{cFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Dact}(\text{dFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge a \neq \text{Dact}(\text{dFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1})$

and *friendIDs s = friendIDs s1*

shows *friendIDs s' = friendIDs s1'*

<proof>

lemma *eqButUID-step-φ-imp*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*

and *a*: $\forall \text{req. } a \neq \text{Cact}(\text{cFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Cact}(\text{cFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Cact}(\text{cFriendReq UID1}(\text{pass } s \text{ UID1}) \text{ UID2 req}) \wedge$

$a \neq \text{Cact}(\text{cFriendReq UID2}(\text{pass } s \text{ UID2}) \text{ UID1 req}) \wedge$

$a \neq \text{Dact}(\text{dFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Dact}(\text{dFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1})$

and φ : $\varphi(\text{Trans } s \text{ a } ou \text{ s}')$

shows $\varphi(\text{Trans } s1 \text{ a } ou1 \text{ s1}')$

<proof>

lemma *eqButUID-step-φ*:

assumes *ss1*: *eqButUID s s1*

and *rs*: *reach s* **and** *rs1*: *reach s1*

and *step*: *step s a = (ou,s')* **and** *step1*: *step s1 a = (ou1,s1')*

and *a*: $\forall \text{req. } a \neq \text{Cact}(\text{cFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Cact}(\text{cFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1}) \wedge$

$a \neq \text{Cact}(\text{cFriendReq UID1}(\text{pass } s \text{ UID1}) \text{ UID2 req}) \wedge$

$a \neq \text{Cact}(\text{cFriendReq UID2}(\text{pass } s \text{ UID2}) \text{ UID1 req}) \wedge$

$a \neq \text{Dact}(\text{dFriend UID1}(\text{pass } s \text{ UID1}) \text{ UID2}) \wedge$

$a \neq \text{Dact}(\text{dFriend UID2}(\text{pass } s \text{ UID2}) \text{ UID1})$

shows $\varphi(\text{Trans } s \text{ a } ou \text{ s}') = \varphi(\text{Trans } s1 \text{ a } ou1 \text{ s1}')$

<proof>

lemma *createFriend-sym*: *createFriend s uid p uid' = createFriend s uid' p' uid*

<proof>

lemma *deleteFriend-sym*: *deleteFriend s uid p uid' = deleteFriend s uid' p' uid*

<proof>

lemma *createFriendReq-createFriend-absorb*:

assumes *e-createFriendReq s uid' p uid req*

shows *createFriend (createFriendReq s uid' p1 uid req) uid p2 uid' = createFriend*

s uid p3 uid'
 ⟨proof⟩

lemma *eqButUID-deleteFriend12-friendIDs-eq*:
assumes *ss1: eqButUID s s1*
and *rs: reach s and rs1: reach s1*
shows *friendIDs (deleteFriend s UID1 p UID2) = friendIDs (deleteFriend s1 UID1 p' UID2)*
 ⟨proof⟩

lemma *eqButUID-createFriend12-friendIDs-eq*:
assumes *ss1: eqButUID s s1*
and *rs: reach s and rs1: reach s1*
and *f12: ¬friends12 s ¬friends12 s1*
shows *friendIDs (createFriend s UID1 p UID2) = friendIDs (createFriend s1 UID1 p' UID2)*
 ⟨proof⟩

end
theory *Friend-Request*
imports *../Observation-Setup Friend-Request-Value-Setup*
begin

8.3 Declassification bound

fun *T :: (state,act,out) trans ⇒ bool*
where *T (Trans - - -) = False*

Friendship updates form an alternating sequence of friending and unfriending, and every successful friend creation is preceded by one or two friendship requests.

fun *validValSeq :: value list ⇒ bool ⇒ bool ⇒ bool ⇒ bool where*
validValSeq [] - - - = True
 | *validValSeq (FRVal U1 req # vl) st r1 r2* $\longleftrightarrow (\neg st) \wedge (\neg r1) \wedge \text{validValSeq } vl \text{ st } True \text{ } r2$
 | *validValSeq (FRVal U2 req # vl) st r1 r2* $\longleftrightarrow (\neg st) \wedge (\neg r2) \wedge \text{validValSeq } vl \text{ st } r1 \text{ } True$
 | *validValSeq (FVal True # vl) st r1 r2* $\longleftrightarrow (\neg st) \wedge (r1 \vee r2) \wedge \text{validValSeq } vl \text{ } True \text{ } False \text{ } False$
 | *validValSeq (FVal False # vl) st r1 r2* $\longleftrightarrow st \wedge (\neg r1) \wedge (\neg r2) \wedge \text{validValSeq } vl \text{ } False \text{ } False \text{ } False$
 | *validValSeq (OVal True # vl) st r1 r2* $\longleftrightarrow \text{validValSeq } vl \text{ st } r1 \text{ } r2$
 | *validValSeq (OVal False # vl) st r1 r2* $\longleftrightarrow \text{validValSeq } vl \text{ st } r1 \text{ } r2$

abbreviation *validValSeqFrom :: value list ⇒ state ⇒ bool*
where *validValSeqFrom vl s*
 $\equiv \text{validValSeq } vl \text{ (friends12 } s) \text{ (UID1} \in \in \text{ pendingFReqs } s \text{ UID2)} \text{ (UID2} \in \in \text{ pendingFReqs } s \text{ UID1)}$

With respect to the friendship status updates, we use the same “while-or-last-before” bound as for friendship status confidentiality.

inductive $BO :: \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$

and $BC :: \text{value list} \Rightarrow \text{value list} \Rightarrow \text{bool}$

where

$BO\text{-}FVal[simp,intro!]:$

$BO (\text{map } FVal \text{ fs}) (\text{map } FVal \text{ fs})$

$|BO\text{-}BC[intro]:$

$BC \text{ vl } vl1 \Longrightarrow$

$BO (\text{map } FVal \text{ fs} @ OVal \text{ False} \# \text{ vl}) (\text{map } FVal \text{ fs} @ OVal \text{ False} \# \text{ vl1})$

$|BC\text{-}FVal[simp,intro!]:$

$BC (\text{map } FVal \text{ fs}) (\text{map } FVal \text{ fs1})$

$|BC\text{-}BO[intro]:$

$BO \text{ vl } vl1 \Longrightarrow (\text{fs} = [] \longleftrightarrow \text{fs1} = []) \Longrightarrow (\text{fs} \neq [] \Longrightarrow \text{last fs} = \text{last fs1}) \Longrightarrow$

$BC (\text{map } FVal \text{ fs} @ OVal \text{ True} \# \text{ vl})$

$(\text{map } FVal \text{ fs1} @ OVal \text{ True} \# \text{ vl1})$

Taking into account friendship requests, two value sequences vl and $vl1$ are in the bound if

- $vl1$ (with friendship requests) forms a valid value sequence,
- vl and $vl1$ are in BO (without friendship requests),
- $vl1$ is empty if vl is empty, and
- $vl1$ begins with $OVal \text{ False}$ if vl begins with $OVal \text{ False}$.

The last two points are due to the fact that $UID1$ and $UID1$ might not exist yet if vl is empty (or before $OVal \text{ False}$), in which case the observer can deduce that no friendship request has happened yet.

definition $B \text{ vl } vl1 \equiv BO (\text{filter } (Not \ o \ isFRVal) \ \text{vl}) (\text{filter } (Not \ o \ isFRVal) \ \text{vl1})$
 \wedge

$\text{validValSeqFrom } vl1 \ \text{istate} \wedge$

$(\text{vl} = [] \longrightarrow \text{vl1} = []) \wedge$

$(\text{vl} \neq [] \wedge \text{hd } \text{vl} = OVal \text{ False} \longrightarrow \text{vl1} \neq [] \wedge \text{hd } \text{vl1} = OVal$

$\text{False})$

lemma $BO\text{-}Nil\text{-iff}: BO \ \text{vl} \ \text{vl1} \Longrightarrow \text{vl} = [] \longleftrightarrow \text{vl1} = []$

$\langle \text{proof} \rangle$

unbundle $no \ \text{relcomp}\text{-syntax}$

interpretation $BD\text{-Security}\text{-IO}$ **where**

$\text{istate} = \text{istate}$ **and** $\text{step} = \text{step}$ **and**

$\varphi = \varphi$ **and** $f = f$ **and** $\gamma = \gamma$ **and** $g = g$ **and** $T = T$ **and** $B = B$

<proof>

lemma *validFrom-validValSeq*:
assumes *validFrom s tr*
and *reach s*
shows *validValSeqFrom (V tr) s*
<proof>

lemma *validFrom istate tr \implies validValSeqFrom (V tr) istate*
<proof>

8.4 Unwinding proof

lemma *eqButUID-step- γ -out*:
assumes *ss1: eqButUID s s1*
and *step: step s a = (ou, s')* **and** *step1: step s1 a = (ou1, s1')*
and *$\gamma: \gamma (\text{Trans } s \ a \ ou \ s')$*
and *os: open s \longrightarrow friendIDs s = friendIDs s1*
shows *ou = ou1*
<proof>

lemma *produce-FRVal*:
assumes *rs: reach s*
and *IDs: IDsOK s [UID1, UID2] []*
and *vVS: validValSeqFrom (FRVal u req # vl) s*
obtains *a uid uid' s'*
where *step s a = (outOK, s')*
and *a = Cact (cFriendReq uid (pass s uid) uid' req)*
and *uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1*
and *$\varphi (\text{Trans } s \ a \ outOK \ s')$*
and *f (Trans s a outOK s') = FRVal u req*
and *validValSeqFrom vl s'*
<proof>

lemma *toggle-friends12-True*:
assumes *rs: reach s*
and *IDs: IDsOK s [UID1, UID2] []*
and *nf12: \neg friends12 s*
and *vVS: validValSeqFrom (FVal True # vl) s*
obtains *a uid uid' s'*
where *step s a = (outOK, s')*
and *a = Cact (cFriend uid (pass s uid) uid')*
and *s' = createFriend s UID1 (pass s UID1) UID2*
and *uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' = UID1*
and *friends12 s'*
and *eqButUID s s'*
and *$\varphi (\text{Trans } s \ a \ outOK \ s')$*
and *f (Trans s a outOK s') = FVal True*

and $\neg\gamma$ (*Trans* *s a outOK s'*)
and *validValSeqFrom vl s'*
 ⟨*proof*⟩

lemma *toggle-friends12-False*:
assumes *rs: reach s*
and *IDs: IDsOK s [UID1, UID2] []*
and *f12: friends12 s*
and *vVS: validValSeqFrom (FVal False # vl) s*
obtains *a s'*
where *step s a = (outOK, s')*
and *a = Dact (dFriend UID1 (pass s UID1) UID2)*
and *s' = deleteFriend s UID1 (pass s UID1) UID2*
and \neg *friends12 s'*
and *eqButUID s s'*
and φ (*Trans s a outOK s'*)
and *f (Trans s a outOK s') = FVal False*
and $\neg\gamma$ (*Trans s a outOK s'*)
and *validValSeqFrom vl s'*
 ⟨*proof*⟩

lemma *toggle-friends12*:
assumes *rs: reach s*
and *IDs: IDsOK s [UID1, UID2] []*
and *f12: friends12 s \neq fv*
and *vVS: validValSeqFrom (FVal fv # vl) s*
obtains *a s'*
where *step s a = (outOK, s')*
and *friends12 s' = fv*
and *eqButUID s s'*
and φ (*Trans s a outOK s'*)
and *f (Trans s a outOK s') = FVal fv*
and $\neg\gamma$ (*Trans s a outOK s'*)
and *validValSeqFrom vl s'*
 ⟨*proof*⟩

lemma *BO-cases*:
assumes *BO vl vl1*
obtains (*Nil*) *vl = [] and vl1 = []*
 | (*FVal*) *fv vl' vl1' where vl = FVal fv # vl' and vl1 = FVal fv # vl1' and*
BO vl' vl1'
 | (*OVal*) *vl' vl1' where vl = OVal False # vl' and vl1 = OVal False # vl1'*
and *BC vl' vl1'*
 ⟨*proof*⟩

lemma *BC-cases*:
assumes *BC vl vl1*
obtains (*Nil*) *vl = [] and vl1 = []*

$$\begin{aligned}
& | (FVal) fv fs \textbf{ where } vl = FVal fv \# \text{ map } FVal fs \textbf{ and } vl1 = [] \\
& | (FVal1) fv fs fs1 \textbf{ where } vl = \text{ map } FVal fs \textbf{ and } vl1 = FVal fv \# \text{ map } FVal \\
& fs1 \\
& | (BO-FVal) fv fv' fs vl' vl1' \textbf{ where } vl = FVal fv \# \text{ map } FVal fs @ FVal fv' \\
& \# OVal True \# vl' \\
& \qquad \qquad \qquad \textbf{and } vl1 = FVal fv' \# OVal True \# vl1' \textbf{ and } BO \\
& vl' vl1' \\
& | (BO-FVal1) fv fv' fs fs1 vl' vl1' \textbf{ where } vl = \text{ map } FVal fs @ FVal fv' \# OVal \\
& True \# vl' \\
& \qquad \qquad \qquad \textbf{and } vl1 = FVal fv \# \text{ map } FVal fs1 @ FVal fv' \# \\
& OVal True \# vl1' \\
& \qquad \qquad \qquad \textbf{and } BO vl' vl1' \\
& | (FVal-BO) fv vl' vl1' \textbf{ where } vl = FVal fv \# OVal True \# vl' \\
& \qquad \qquad \qquad \textbf{and } vl1 = FVal fv \# OVal True \# vl1' \textbf{ and } BO vl' vl1' \\
& | (OVal) vl' vl1' \textbf{ where } vl = OVal True \# vl' \textbf{ and } vl1 = OVal True \# vl1' \\
& \textbf{and } BO vl' vl1' \\
& \langle \textit{proof} \rangle
\end{aligned}$$

definition $\Delta 0 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 0\ s\ vl\ s1\ vl1 \equiv$
 $s = s1 \wedge B\ vl\ vl1 \wedge open\ s \wedge (\neg IDsOK\ s\ [UID1,\ UID2])\ []$

definition $\Delta 1 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 1\ s\ vl\ s1\ vl1 \equiv$
 $eqButUID\ s\ s1 \wedge friendIDs\ s = friendIDs\ s1 \wedge open\ s \wedge$
 $BO\ (filter\ (Not\ o\ isFRVal)\ vl)\ (filter\ (Not\ o\ isFRVal)\ vl1) \wedge$
 $validValSeqFrom\ vl1\ s1 \wedge$
 $IDsOK\ s1\ [UID1,\ UID2]\ []$

definition $\Delta 2 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 2\ s\ vl\ s1\ vl1 \equiv (\exists fs\ fs1.$
 $eqButUID\ s\ s1 \wedge \neg open\ s \wedge$
 $validValSeqFrom\ vl1\ s1 \wedge$
 $filter\ (Not\ o\ isFRVal)\ vl = \text{ map } FVal\ fs \wedge$
 $filter\ (Not\ o\ isFRVal)\ vl1 = \text{ map } FVal\ fs1)$

definition $\Delta 3 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
 $\Delta 3\ s\ vl\ s1\ vl1 \equiv (\exists fs\ fs1\ vlr\ vlr1.$
 $eqButUID\ s\ s1 \wedge \neg open\ s \wedge BO\ vlr\ vlr1 \wedge$
 $validValSeqFrom\ vl1\ s1 \wedge$
 $(fs = [] \iff fs1 = []) \wedge$
 $(fs \neq [] \implies last\ fs = last\ fs1) \wedge$
 $(fs = [] \implies friendIDs\ s = friendIDs\ s1) \wedge$
 $filter\ (Not\ o\ isFRVal)\ vl = \text{ map } FVal\ fs @ OVal\ True\ \# vlr \wedge$
 $filter\ (Not\ o\ isFRVal)\ vl1 = \text{ map } FVal\ fs1 @ OVal\ True\ \# vlr1)$

lemma $\Delta 2-I$:

assumes $eqButUID\ s\ s1\ \neg open\ s$
 $validValSeqFrom\ vl1\ s1$
 $filter\ (Not\ o\ isFRVal)\ vl = map\ FVal\ fs$
 $filter\ (Not\ o\ isFRVal)\ vl1 = map\ FVal\ fs1$
shows $\Delta 2\ s\ vl\ s1\ vl1$
 $\langle proof \rangle$

lemma $\Delta 3-I$:
assumes $eqButUID\ s\ s1\ \neg open\ s\ BO\ vlr\ vlr1$
 $validValSeqFrom\ vl1\ s1$
 $fs = [] \longleftrightarrow fs1 = []\ fs \neq [] \longrightarrow last\ fs = last\ fs1$
 $fs = [] \longrightarrow friendIDs\ s = friendIDs\ s1$
 $filter\ (Not\ o\ isFRVal)\ vl = map\ FVal\ fs\ @\ Oval\ True\ \# \ vlr$
 $filter\ (Not\ o\ isFRVal)\ vl1 = map\ FVal\ fs1\ @\ Oval\ True\ \# \ vlr1$
shows $\Delta 3\ s\ vl\ s1\ vl1$
 $\langle proof \rangle$

lemma $istate-\Delta 0$:
assumes $B: B\ vl\ vl1$
shows $\Delta 0\ istate\ vl\ istate\ vl1$
 $\langle proof \rangle$

lemma $unwind-cont-\Delta 0$: $unwind-cont\ \Delta 0\ \{\Delta 0, \Delta 1, \Delta 2, \Delta 3\}$
 $\langle proof \rangle$

lemma $unwind-cont-\Delta 1$: $unwind-cont\ \Delta 1\ \{\Delta 1, \Delta 2, \Delta 3\}$
 $\langle proof \rangle$

lemma $unwind-cont-\Delta 2$: $unwind-cont\ \Delta 2\ \{\Delta 2, \Delta 1\}$
 $\langle proof \rangle$

lemma $unwind-cont-\Delta 3$: $unwind-cont\ \Delta 3\ \{\Delta 3, \Delta 1\}$
 $\langle proof \rangle$

definition Gr where
 $Gr =$
 $\{$
 $(\Delta 0, \{\Delta 0, \Delta 1, \Delta 2, \Delta 3\}),$
 $(\Delta 1, \{\Delta 1, \Delta 2, \Delta 3\}),$
 $(\Delta 2, \{\Delta 2, \Delta 1\}),$
 $(\Delta 3, \{\Delta 3, \Delta 1\})$
 $\}$

theorem $secure: secure$

<proof>

```
end  
theory Traceback-Intro  
  imports ../Safety-Properties  
begin
```

9 Traceback Properties

In this section, we prove traceback properties. These properties trace back the actions leading to:

- the current visibility status of a post
- the current friendship status of two users

They state that the current status can only occur via a “legal” sequence of actions. Because the BD properties have (dynamic triggers within) declassification bounds that refer to such statuses, the traceback properties complement BD Security in adding confidentiality assurance. [1, Section 5.2] gives more details and explanations.

```
end  
theory Post-Visibility-Traceback  
  imports Traceback-Intro  
begin
```

```
consts PID :: postID  
consts VIS :: vis
```

9.1 Tracing Back Post Visibility Status

We prove the following traceback property: If, at some point t on a system trace, the visibility of a post PID has a value VIS , then one of the following holds:

- Either VIS is *FriendV* (i.e., friends-only) which is the default at post creation
- Or the post’s owner had issued a successful “update visibility” action setting the visibility to VIS , and no other successful update actions to PID ’s visibility occur between the time of that action and t .

This will be captured in the predicate *proper*, and the main theorem states that *proper tr* holds for any trace tr that leads to post PID acquiring visibility VIS .

SNC uid *trn* means “The transaction *trn* is a successful post creation by user *uid*”

fun *SNC* :: *userID* \Rightarrow (*state,act,out*) *trans* \Rightarrow *bool* **where**
SNC uid (*Trans s* (*Cact* (*cPost uid p pid tit*) *ou s'*) = (*ou* = *outOK* \wedge (*uid,pid*) = (*uid,PID*))
|
SNC uid - = *False*

SNVU uid vis trn means "The transaction *trn* is a successful post visibility update for *PID*, by user *uid*, to value *vis*”

fun *SNVU* :: *userID* \Rightarrow *vis* \Rightarrow (*state,act,out*) *trans* \Rightarrow *bool* **where**
SNVU uid vis (*Trans s* (*Uact* (*uVisPost uid p pid vs*) *ou s'*) = (*ou* = *outOK* \wedge (*uid,pid*) = (*uid,PID*) \wedge *vs* = *vis*)
|
SNVU uid vis - = *False*

definition *proper* :: (*state,act,out*) *trans trace* \Rightarrow *bool* **where**
proper tr \equiv
VIS = *FriendV*
 \vee
 $(\exists$ *uid tr1 trn tr2 trnn tr3*.
tr = *tr1* @ *trn* # *tr2* @ *trnn* # *tr3* \wedge
SNC uid trn \wedge *SNVU uid VIS trnn* \wedge (\forall *vis*. *never* (*SNVU uid vis*) *tr3*))

definition *proper1* :: (*state,act,out*) *trans trace* \Rightarrow *bool* **where**
proper1 tr \equiv
 \exists *tr2 trnn tr3*.
tr = *tr2* @ *trnn* # *tr3* \wedge
SNVU (*owner* (*srcOf trnn*) *PID*) *VIS trnn*

lemma *not-never-ex*:

assumes \neg *never P xs*

shows \exists *xs1 x xs2*. *xs* = *xs1* @ *x* # *xs2* \wedge *P x* \wedge *never P xs2*
<*proof*>

lemma *SNVU-postIDs*:

assumes *validTrans trn* **and** *SNVU uid vs trn*

shows *PID* $\in\in$ *postIDs* (*srcOf trn*)

<*proof*>

lemma *SNVU-visib*:

assumes *validTrans trn* **and** *SNVU uid vs trn*

shows *vis* (*tgtOf trn*) *PID* = *vs*

<*proof*>

lemma *owner-validTrans*:

assumes *validTrans trn* **and** *PID* $\in\in$ *postIDs* (*srcOf trn*)

shows $owner (srcOf trn) PID = owner (tgtOf trn) PID$
<proof>

lemma *owner-valid*:

assumes $valid\ tr$ **and** $PID \in\in postIDs (srcOf (hd\ tr))$
shows $owner (srcOf (hd\ tr)) PID = owner (tgtOf (last\ tr)) PID$
<proof>

lemma *SNVU-vis-validTrans*:

assumes $validTrans\ trn$ **and** $PID \in\in postIDs (srcOf\ trn)$
and $\forall\ vis. \neg SNVU (owner (srcOf\ trn)\ PID)\ vis\ trn$
shows $vis (srcOf\ trn)\ PID = vis (tgtOf\ trn)\ PID$
<proof>

lemma *SNVU-vis-valid*:

assumes $valid\ tr$ **and** $PID \in\in postIDs (srcOf (hd\ tr))$
and $\forall\ vis. never (SNVU (owner (srcOf (hd\ tr))\ PID)\ vis)\ tr$
shows $vis (srcOf (hd\ tr))\ PID = vis (tgtOf (last\ tr))\ PID$
<proof>

lemma *proper1-never*:

assumes $utr: valid\ tr$ **and** $PID: PID \in\in postIDs (srcOf (hd\ tr))$
and $tr: proper1\ tr$ **and** $v: vis (tgtOf (last\ tr))\ PID = VIS$
shows $\exists\ tr2\ trnn\ tr3.$
 $tr = tr2 @ trnn \# tr3 \wedge$
 $SNVU (owner (srcOf\ trnn)\ PID)\ VIS\ trnn \wedge (\forall\ vis. never (SNVU (owner$
 $(srcOf\ trnn)\ PID)\ vis)\ tr3)$
<proof>

lemma *SNVU-validTrans*:

assumes $validTrans\ trn$
and $PID \in\in postIDs (srcOf\ trn)$
and $vis (srcOf\ trn)\ PID \neq VIS$
and $vis (tgtOf\ trn)\ PID = VIS$
shows $SNVU (owner (srcOf\ trn)\ PID)\ VIS\ trn$
<proof>

lemma *valid-mono-postID*:

assumes $valid\ tr$
and $PID \in\in postIDs (srcOf (hd\ tr))$
shows $PID \in\in postIDs (tgtOf (last\ tr))$
<proof>

lemma *proper1-valid*:

assumes $V: VIS \neq FriendV$
and $a: valid\ tr$

$PID \in \in postIDs (srcOf (hd tr))$
 $vis (srcOf (hd tr)) PID \neq VIS$
 $vis (tgtOf (last tr)) PID = VIS$
shows *proper1 tr*
 ⟨*proof*⟩

lemma *istate-postIDs*:
 $\neg PID \in \in postIDs istate$
 ⟨*proof*⟩

definition *proper2* :: $(state, act, out) \text{ trans trace} \Rightarrow bool$ **where**
proper2 tr \equiv
 $\exists uid \ tr1 \ trn \ tr2.$
 $tr = tr1 @ trn \# tr2 \wedge SNC \ uid \ trn$

lemma *SNC-validTrans*:
assumes $VIS \neq FriendV$ **and** *validTrans trn*
and $\neg PID \in \in postIDs (srcOf trn)$
and $PID \in \in postIDs (tgtOf trn)$
shows $\exists uid. SNC \ uid \ trn$
 ⟨*proof*⟩

lemma *proper2-valid*:
assumes $V: VIS \neq FriendV$
and *a: valid tr*
 $\neg PID \in \in postIDs (srcOf (hd tr))$
 $PID \in \in postIDs (tgtOf (last tr))$
shows *proper2 tr*
 ⟨*proof*⟩

lemma *proper2-valid-istate*:
assumes $V: VIS \neq FriendV$
and *a: valid tr*
 $srcOf (hd tr) = istate$
 $PID \in \in postIDs (tgtOf (last tr))$
shows *proper2 tr*
 ⟨*proof*⟩

lemma *SNC-visPost*:
assumes $VIS \neq FriendV$
and *validTrans trn SNC uid trn* **and** *reach (srcOf trn)*
shows $vis (tgtOf trn) PID \neq VIS$
 ⟨*proof*⟩

lemma *SNC-postIDs*:
assumes *validTrans trn and SNC uid trn*
shows $PID \in \in postIDs (tgtOf\ trn)$
 $\langle proof \rangle$

lemma *SNC-owner*:
assumes *validTrans trn and SNC uid trn*
shows $uid = owner (tgtOf\ trn)\ PID$
 $\langle proof \rangle$

theorem *post-accountability*:
assumes $v: valid\ tr$ **and** $i: srcOf\ (hd\ tr) = istate$
and $PIDin: PID \in \in postIDs (tgtOf\ (last\ tr))$
and $PID: vis (tgtOf\ (last\ tr))\ PID = VIS$
shows *proper tr*
 $\langle proof \rangle$

end
theory *Friend-Traceback*
imports *Traceback-Intro*
begin

9.2 Tracing Back Friendship Status

We prove the following traceback property: If, at some point t on a system trace, the users UID and UID' are friends, then one of the following holds:

- Either UID had issued a friend request to UID' , eventually followed by an approval (i.e., a successful UID -friend creation action) by UID' such that between that approval and t there was no successful UID' -unfriending (i.e., friend deletion) by UID or UID -unfriending by UID'
- Or vice versa (with UID and UID' swapped)

This property is captured by the predicate *proper*, which decomposes any valid system trace tr starting in the initial state for which the target state $tgtOf (last\ tr)$ has UID and UID' as friends, as follows: tr is the concatenation of $tr1$, trn , $tr2$, $trnn$ and $tr3$ where

- trn represents the time of the relevant friend request
- $trnn$ represents the time of the approval of this request
- $tr3$ contains no unfriending between the two users

The main theorem states that *proper tr* holds for any trace tr that leads to UID and UID' being friends.

consts $UID :: userID$
consts $UID' :: userID$

SFRC means “is a successful friend request creation”

fun *SFRC* :: $userID \Rightarrow userID \Rightarrow (state,act,out) \text{ trans} \Rightarrow bool$ **where**
SFRC $uidd\ uidd'$ ($\text{Trans } s \text{ (Cact (cFriendReq } uid\ p\ uid' \text{ -)) } ou\ s'$) = ($ou = outOK$
 $\wedge (uid,uid') = (uidd,uidd')$)
|
SFRC $uidd\ uidd'$ - = *False*

SFC means “is a successful friend creation”

fun *SFC* :: $userID \Rightarrow userID \Rightarrow (state,act,out) \text{ trans} \Rightarrow bool$ **where**
SFC $uidd\ uidd'$ ($\text{Trans } s \text{ (Cact (cFriend } uid\ p\ uid')) } ou\ s'$) = ($ou = outOK \wedge$
 $(uid,uid') = (uidd,uidd')$)
|
SFC $uidd\ uidd'$ - = *False*

SFD means “is a successful friend deletion”

fun *SFD* :: $userID \Rightarrow userID \Rightarrow (state,act,out) \text{ trans} \Rightarrow bool$ **where**
SFD $uidd\ uidd'$ ($\text{Trans } s \text{ (Dact (dFriend } uid\ p\ uid')) } ou\ s'$) = ($ou = outOK \wedge$
 $(uid,uid') = (uidd,uidd')$)
|
SFD $uidd\ uidd'$ - = *False*

definition *proper1* :: $(state,act,out) \text{ trans trace} \Rightarrow bool$ **where**
proper1 $tr \equiv$

$\exists trr\ trnn\ tr3. tr = trr @ trnn \# tr3 \wedge$
 $(SFC\ UID\ UID'\ trnn \vee SFC\ UID'\ UID\ trnn) \wedge$
 $never\ (SFD\ UID\ UID')\ tr3 \wedge never\ (SFD\ UID'\ UID)\ tr3$

lemma *SFC-validTrans*:

assumes *validTrans* trn
and $\neg UID' \in \in friendIDs\ (srcOf\ trn)\ UID$
and $UID' \in \in friendIDs\ (tgtOf\ trn)\ UID$
shows $SFC\ UID\ UID'\ trn \vee SFC\ UID'\ UID\ trn$
 $\langle proof \rangle$

lemma *SFD-validTrans*:

assumes *validTrans* trn
and $UID' \in \in friendIDs\ (tgtOf\ trn)\ UID$
shows $\neg SFD\ UID\ UID'\ trn \wedge \neg SFD\ UID'\ UID\ trn$
 $\langle proof \rangle$

lemma *SFC-SFD*:

assumes $SFC\ uid1\ uid2\ trn$ **shows** $\neg SFD\ uid3\ uid4\ trn$
 $\langle proof \rangle$

lemma *proper1-valid*:

assumes *valid* tr

and $\neg UID' \in \text{friendIDs} (\text{srcOf} (\text{hd } tr)) UID$
and $UID' \in \text{friendIDs} (\text{tgtOf} (\text{last } tr)) UID$
shows *proper1 tr*
 ⟨*proof*⟩

lemma *istate-friendIDs*:
 $\neg UID' \in \text{friendIDs} (\text{istate}) UID$
 ⟨*proof*⟩

lemma *proper1-valid-istate*:
assumes *valid tr* **and** $\text{srcOf} (\text{hd } tr) = \text{istate}$
and $UID' \in \text{friendIDs} (\text{tgtOf} (\text{last } tr)) UID$
shows *proper1 tr*
 ⟨*proof*⟩

definition *proper2* :: $\text{userID} \Rightarrow \text{userID} \Rightarrow (\text{state}, \text{act}, \text{out}) \text{ trans trace} \Rightarrow \text{bool}$ **where**
proper2 uid uid' tr \equiv
 $\exists tr1 \text{ trnn } tr2. tr = tr1 @ \text{trnn} \# tr2 \wedge \text{SFRC } uid \text{ uid' } \text{trnn}$

lemma *SFRC-validTrans*:
assumes *validTrans trn*
and $\neg uid \in \text{pendingFReqs} (\text{srcOf } trn) uid'$
and $uid \in \text{pendingFReqs} (\text{tgtOf } trn) uid'$
shows *SFRC uid uid' trn*
 ⟨*proof*⟩

lemma *proper2-valid*:
assumes *valid tr*
and $\neg uid \in \text{pendingFReqs} (\text{srcOf} (\text{hd } tr)) uid'$
and $uid \in \text{pendingFReqs} (\text{tgtOf} (\text{last } tr)) uid'$
shows *proper2 uid uid' tr*
 ⟨*proof*⟩

lemma *istate-pendingFReqs*:
 $\neg uid \in \text{pendingFReqs} (\text{istate}) uid'$
 ⟨*proof*⟩

lemma *proper2-valid-istate*:
assumes *valid tr* **and** $\text{srcOf} (\text{hd } tr) = \text{istate}$
and $uid \in \text{pendingFReqs} (\text{tgtOf} (\text{last } tr)) uid'$
shows *proper2 uid uid' tr*
 ⟨*proof*⟩

lemma *SFC-pendingFReqs*:
assumes *validTrans trn*

and $SFC\ uid' uid\ trn$
shows $uid \in \in pendingFReqs (srcOf\ trn)\ uid'$
 $\langle proof \rangle$

definition $proper :: (state, act, out)\ trans\ trace \Rightarrow bool$ **where**
 $proper\ tr \equiv$
 $\exists tr1\ trn\ tr2\ trnn\ tr3. tr = tr1 @ trn \# tr2 @ trnn \# tr3 \wedge$
 $(SFC\ UID'\ UID\ trn \wedge SFC\ UID\ UID'\ trnn \vee$
 $SFC\ UID\ UID'\ trn \wedge SFC\ UID'\ UID\ trnn) \wedge$
 $never\ (SFD\ UID\ UID')\ tr3 \wedge never\ (SFD\ UID'\ UID)\ tr3$

theorem $friend-accountability$:
assumes v : $valid\ tr$ **and** i : $srcOf\ (hd\ tr) = istate$
and UID : $UID' \in \in friendIDs\ (tgtOf\ (last\ tr))\ UID$
shows $proper\ tr$
 $\langle proof \rangle$

end

References

- [1] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.
- [2] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. CoSMed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.
- [3] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.
- [4] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum fr Informatik, 2021.

- [5] A. Popescu, P. Lammich, and T. Bauereiss. Bounded-deducibility security. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*, 2014.
- [6] A. Popescu, P. Lammich, and P. Hou. Cocon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.