CoSMed: A confidentiality-verified social media platform

Thomas Bauereiss   Andrei Popescu

March 19, 2025

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

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* ''''

[1] In 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* ≡ *Psw emptyStr*

datatype *req* = *ReqInfo String.literal*
definition *emptyReq* ≡ *ReqInfo emptyStr*

2.2 Identifiers

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

definition *emptyUserID* ≡ *Uid emptyStr*
definition *emptyPostID* ≡ *Nid emptyStr*

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

definition *getFreshUserID userIDs* ≡ *Uid* (*fresh* (*set* (*map userIDAsStr userIDs*)) (*STR ''2''*))

lemma *UserID-userIDAsStr*[*simp*]: *Uid* (*userIDAsStr userID*) = *userID*
⟨*proof*⟩

lemma *member-userIDAsStr-iff*[*simp*]: *str* ∈ *userIDAsStr* ‘ (*set userIDs*) ⟷ *Uid str* ∈∈ *userIDs*
⟨*proof*⟩

lemma *getFreshUserID*: ¬ *getFreshUserID userIDs* ∈∈ *userIDs*
⟨*proof*⟩

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

definition *getFreshPostID postIDs* ≡ *Nid* (*fresh* (*set* (*map postIDAsStr postIDs*)) (*STR ''3''*))

lemma *PostID-postIDAsStr*[*simp*]: *Nid* (*postIDAsStr postID*) = *postID*
⟨*proof*⟩

lemma *member-postIDAsStr-iff*[*simp*]: *str* ∈ *postIDAsStr* ‘ (*set postIDs*) ⟷ *Nid*

str ∈∈ *postIDs*
⟨*proof*⟩

lemma *getFreshPostID*: ¬ *getFreshPostID postIDs* ∈∈ *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* ⇒ *req*
userIDs :: *userID list*
user :: *userID* ⇒ *user*
pass :: *userID* ⇒ *password*

pendingFReqs :: *userID* ⇒ *userID list*
friendReq :: *userID* ⇒ *userID* ⇒ *req*
friendIDs :: *userID* ⇒ *userID list*

postIDs :: *postID list*
post :: *postID* ⇒ *post*
owner :: *postID* ⇒ *userID*
vis :: *postID* ⇒ *vis*

definition *IDsOK* :: *state* ⇒ *userID list* ⇒ *postID list* ⇒ *bool*
where
IDsOK s uIDs pIDs ≡
list-all (λ *uID*. *uID* ∈∈ *userIDs s*) *uIDs* ∧
list-all (λ *pID*. *pID* ∈∈ *postIDs s*) *pIDs*

3.2 The actions

3.2.1 Initialization of the system

definition *istate* :: *state*

where
istate $\equiv$
$(\!|$
$admin = emptyUserID,$

$pendingUReqs = [],$
$userReq = (\lambda\ uID.\ emptyReq),$
$userIDs = [],$
$user = (\lambda\ uID.\ emptyUser),$
$pass = (\lambda\ uID.\ emptyPass),$

$pendingFReqs = (\lambda\ uID.\ []),$
$friendReq = (\lambda\ uID\ uID'.\ emptyReq),$
$friendIDs = (\lambda\ uID.\ []),$

$postIDs = [],$
$post = (\lambda\ papID.\ emptyPost),$
$owner = (\lambda\ pID.\ emptyUserID),$
$vis = (\lambda\ pID.\ FriendV)$
$|\!)$

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
$e\text{-}startSys\ s\ uID\ p \equiv userIDs\ s = []$

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
$e\text{-}createNUReq\ s\ uID\ req \equiv$
$admin\ s \in\in userIDs\ s \wedge \neg\ uID \in\in userIDs\ s \wedge \neg\ uID \in\in pendingUReqs\ s$

definition *createUser* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *password* ⇒ *state*
where
createUser s uID p uID′ p′ ≡
 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* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-createUser s uID p uID′ p′ ≡
 IDsOK s [*uID*] [] ∧ *pass s uID* = *p* ∧ *uID* = *admin s* ∧ *uID′* ∈∈ *pendingUReqs s*

definition *createPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *title* ⇒ *state*
where
createPost s uID p pID title ≡
 s (|*postIDs* := *pID* # *postIDs s*,
  post := (*post s*) (*pID* := *Ntc title emptyText emptyImg*),
  owner := (*owner s*) (*pID* := *uID*)|)

definition *e-createPost* :: *state* ⇒ *userID* ⇒ *password* ⇒ *postID* ⇒ *title* ⇒ *bool*
where
e-createPost s uID p pID title ≡
 IDsOK s [*uID*] [] ∧ *pass s uID* = *p* ∧
 ¬ *pID* ∈∈ *postIDs s*

definition *createFriendReq* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *req* ⇒ *state*
where
createFriendReq s uID p uID′ req ≡
 let pfr = *pendingFReqs s in*
 s (|*pendingFReqs* := *pfr* (*uID′* := *pfr uID′* @ [*uID*]),
  friendReq := *fun-upd2* (*friendReq s*) *uID uID′ req*|)

definition *e-createFriendReq* :: *state* ⇒ *userID* ⇒ *password* ⇒ *userID* ⇒ *req* ⇒ *bool*
where
e-createFriendReq s uID p uID′ req ≡
 IDsOK s [*uID*,*uID′*] [] ∧ *pass s uID* = *p* ∧

$\neg$ *uID* $\in\in$ *pendingFReqs s uID'* $\wedge$ $\neg$ *uID* $\in\in$ *friendIDs s uID'*

definition *createFriend* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *userID* $\Rightarrow$ *state*
where
createFriend s uID p uID' $\equiv$
let fr = *friendIDs s*; *pfr* = *pendingFReqs s in*
s (|*friendIDs* := *fr* (*uID* := *fr uID* @ [*uID'*], *uID'* := *fr uID'* @ [*uID*]),
pendingFReqs := *pfr* (*uID* := *remove1 uID'* (*pfr uID*), *uID'* := *remove1 uID* (*pfr uID'*)),
friendReq := *fun-upd2* (*fun-upd2* (*friendReq s*) *uID' uID emptyReq*) *uID uID' emptyReq*|)

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

3.2.4 Updating actions

definition *updateUser* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *password* $\Rightarrow$ *name* $\Rightarrow$ *inform* $\Rightarrow$ *state*
where
updateUser s uID p p' name info $\equiv$
s (|*user* := (*user s*) (*uID* := *Usr name info*),
pass := (*pass s*) (*uID* := *p'*)|)

definition *e-updateUser* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *password* $\Rightarrow$ *name* $\Rightarrow$ *inform* $\Rightarrow$ *bool*
where
e-updateUser s uID p p' name info $\equiv$
IDsOK s [*uID*] [] $\wedge$ *pass s uID* = *p*

definition *updatePost* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *postID* $\Rightarrow$ *post* $\Rightarrow$ *state*
where
updatePost s uID p pID pst $\equiv$
s (|*post* := (*post s*) (*pID* := *pst*)|)

definition *e-updatePost* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *postID* $\Rightarrow$ *post* $\Rightarrow$ *bool*
where
e-updatePost s uID p pID pst $\equiv$
IDsOK s [*uID*] [*pID*] $\wedge$ *pass s uID* = *p* $\wedge$
owner s pID = *uID*

definition *updateVisPost* :: *state* $\Rightarrow$ *userID* $\Rightarrow$ *password* $\Rightarrow$ *postID* $\Rightarrow$ *vis* $\Rightarrow$ *state*
where

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

definition $e\text{-}updateVisPost :: state \Rightarrow userID \Rightarrow password \Rightarrow postID \Rightarrow vis \Rightarrow bool$
where
$e\text{-}updateVisPost\ s\ uID\ p\ pID\ vs \equiv$
$IDsOK\ s\ [uID]\ [pID] \wedge pass\ s\ uID = p \wedge$
$owner\ s\ pID = uID \wedge vs \in \{FriendV,\ PublicV\}$

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 $e\text{-}deleteFriend :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}deleteFriend\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID,uID']\ []\ \wedge pass\ s\ uID = p \wedge$
$uID' \in\in friendIDs\ s\ uID$

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 $e\text{-}readNUReq :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}readNUReq\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID]\ []\ \wedge pass\ s\ uID = p \wedge$
$uID = admin\ s \wedge uID' \in\in pendingUReqs\ s$

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 $e\text{-}readUser :: state \Rightarrow userID \Rightarrow password \Rightarrow userID \Rightarrow bool$
where
$e\text{-}readUser\ s\ uID\ p\ uID' \equiv$
$IDsOK\ s\ [uID,uID']\ []\ \wedge pass\ s\ uID = p$

definition $readAmIAdmin :: state \Rightarrow userID \Rightarrow password \Rightarrow bool$
where
$readAmIAdmin\ s\ uID\ p \equiv uID = admin\ s$

definition $\textit{e-readAmIAdmin} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{bool}$
where
$\textit{e-readAmIAdmin } s \ uID \ p \equiv$
$\ IDsOK \ s \ [uID] \ [] \wedge pass \ s \ uID = p$

definition $\textit{readPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{post}$
where
$\textit{readPost } s \ uID \ p \ pID \equiv post \ s \ pID$

definition $\textit{e-readPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{bool}$
where
$\textit{e-readPost } s \ uID \ p \ pID \equiv$
$\ let \ post = post \ s \ pID \ in$
$\ IDsOK \ s \ [uID] \ [pID] \wedge pass \ s \ uID = p \wedge$
$(owner \ s \ pID = uID \vee uID \in\in friendIDs \ s \ (owner \ s \ pID) \vee vis \ s \ pID = PublicV)$

definition $\textit{readVisPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{vis}$
where
$\textit{readVisPost } s \ uID \ p \ pID \equiv vis \ s \ pID$

definition $\textit{e-readVisPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{bool}$
where
$\textit{e-readVisPost } s \ uID \ p \ pID \equiv$
$\ let \ post = post \ s \ pID \ in$
$\ IDsOK \ s \ [uID] \ [pID] \wedge pass \ s \ uID = p \wedge$
$(owner \ s \ pID = uID \vee uID \in\in friendIDs \ s \ (owner \ s \ pID) \vee vis \ s \ pID = PublicV)$

definition $\textit{readOwnerPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{userID}$
where
$\textit{readOwnerPost } s \ uID \ p \ pID \equiv owner \ s \ pID$

definition $\textit{e-readOwnerPost} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{postID} \Rightarrow \textit{bool}$
where
$\textit{e-readOwnerPost } s \ uID \ p \ pID \equiv$
$\ let \ post = post \ s \ pID \ in$
$\ IDsOK \ s \ [uID] \ [pID] \wedge pass \ s \ uID = p \wedge$
$(owner \ s \ pID = uID \vee uID \in\in friendIDs \ s \ (owner \ s \ pID) \vee vis \ s \ pID = PublicV)$

definition $\textit{readFriendReqToMe} :: \textit{state} \Rightarrow \textit{userID} \Rightarrow \textit{password} \Rightarrow \textit{userID} \Rightarrow \textit{req}$
where
$\textit{readFriendReqToMe } s \ uID \ p \ uID' \equiv friendReq \ s \ uID' \ uID$

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

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

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

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

3.2.7 Listing actions

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

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

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

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

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

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

definition *listPosts* :: *state* ⇒ *userID* ⇒ *password* ⇒ (*userID* × *postID*) *list*
where
listPosts s uID p ≡
[(*owner s pID*, *pID*).
pID ← *postIDs s*,
vis s pID = *PublicV* ∨ *uID* ∈∈ *friendIDs s* (*owner s pID*) ∨ *uID* = *owner s pID*
]

definition *e-listPosts* :: *state* ⇒ *userID* ⇒ *password* ⇒ *bool*
where
e-listPosts s uID p ≡ *IDsOK s* [*uID*] [] ∧ *pass s uID* = *p*

3.3 The step function

datatype *out* =

outOK | *outErr* |

outBool bool| *outNI name* × *inform* | *outPost post* |
outImg img | *outVis vis* | *outReq req* |

outUID userID | *outUIDL userID list* |
outUIDNIDL (*userID* × *postID*)*list*

datatype *sActt* =
sSys userID password

lemmas *s-defs* =
e-startSys-def startSys-def

fun *sStep* :: *state* ⇒ *sActt* ⇒ *out* ∗ *state* **where**
sStep s (*sSys uID p*) =
(*if e-startSys s uID p*
then (*outOK*, *startSys s uID p*)
else (*outErr*, *s*))

fun *sUserOfA* :: *sActt* ⇒ *userID* **where**
sUserOfA (*sSys uID p*) = *uID*

datatype *cActt* =
cNUReq userID req
|*cUser userID password userID password*
|*cFriendReq userID password userID req*
|*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* =
uUser 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 $rObs :: state \Rightarrow rActt \Rightarrow out$ **where**
$rObs\ s\ (rNUReq\ uID\ p\ uID') =$
$(if\ e\text{-}readNUReq\ s\ uID\ p\ uID'\ then\ outReq\ (readNUReq\ s\ uID\ p\ uID')\ else\ outErr)$
$|$
$rObs\ s\ (rUser\ uID\ p\ uID') =$
$(if\ e\text{-}readUser\ s\ uID\ p\ uID'\ then\ outNI\ (readUser\ s\ uID\ p\ uID')\ else\ outErr)$
$|$
$rObs\ s\ (rAmIAdmin\ uID\ p) =$
$(if\ e\text{-}readAmIAdmin\ s\ uID\ p\ then\ outBool\ (readAmIAdmin\ s\ uID\ p)\ else\ outErr)$
$|$
$rObs\ s\ (rPost\ uID\ p\ pID) =$
$(if\ e\text{-}readPost\ s\ uID\ p\ pID\ then\ outPost\ (readPost\ s\ uID\ p\ pID)\ else\ outErr)$
$|$
$rObs\ s\ (rVisPost\ uID\ p\ pID) =$
$(if\ e\text{-}readVisPost\ s\ uID\ p\ pID\ then\ outVis\ (readVisPost\ s\ uID\ p\ pID)\ else\ outErr)$
$|$
$rObs\ s\ (rOwnerPost\ uID\ p\ pID) =$
$(if\ e\text{-}readOwnerPost\ s\ uID\ p\ pID\ then\ outUID\ (readOwnerPost\ s\ uID\ p\ pID)\ else\ outErr)$
$|$
$rObs\ s\ (rFriendReqToMe\ uID\ p\ uID') =$
$(if\ e\text{-}readFriendReqToMe\ s\ uID\ p\ uID'\ then\ outReq\ (readFriendReqToMe\ s\ uID\ p\ uID')\ else\ outErr)$
$|$
$rObs\ s\ (rFriendReqFromMe\ uID\ p\ uID') =$
$(if\ e\text{-}readFriendReqFromMe\ s\ uID\ p\ uID'\ then\ outReq\ (readFriendReqFromMe\ s\ uID\ p\ uID')\ else\ outErr)$

fun $rUserOfA :: rActt \Rightarrow userID\ option$ **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* =
lPendingUReqs 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 $lObs :: state \Rightarrow lActt \Rightarrow out$ **where**
$lObs\ s\ (lPendingUReqs\ uID\ p) =$
$(if\ e\text{-}listPendingUReqs\ s\ uID\ p\ then\ outUIDL\ (listPendingUReqs\ s\ uID\ p)\ else\ outErr)$
|
$lObs\ s\ (lAllUsers\ uID\ p) =$
$(if\ e\text{-}listAllUsers\ s\ uID\ p\ then\ outUIDL\ (listAllUsers\ s\ uID\ p)\ else\ outErr)$
|
$lObs\ s\ (lFriends\ uID\ p\ uID') =$
$(if\ e\text{-}listFriends\ s\ uID\ p\ uID'\ then\ outUIDL\ (listFriends\ s\ uID\ p\ uID')\ else\ outErr)$
|
$lObs\ s\ (lPosts\ uID\ p) =$
$(if\ e\text{-}listPosts\ s\ uID\ p\ then\ outUIDNIDL\ (listPosts\ s\ uID\ p)\ else\ outErr)$

fun $lUserOfA :: lActt \Rightarrow userID\ option$ **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* ∧ *userReq s* = *userReq s1* ∧ *userIDs s* = *userIDs s1* ∧
postIDs s = *postIDs s1* ∧ *admin s* = *admin s1* ∧
user s = *user s1* ∧ *pass s* = *pass s1* ∧ *pendingFReqs s* = *pendingFReqs s1* ∧
friendReq s = *friendReq s1* ∧ *friendIDs s* = *friendIDs s1* ∧
post s = *post s1* ∧
owner s = *owner s1* ∧
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-elims* = *step.elims sStep.elims cStep.elims dStep.elims uStep.elims*

declare *sstep-Cons*[*simp*]

lemma *Lact-Ract-noStateChange*[*simp*]:
assumes *a* ∈ *Lact* ‘ *UNIV* ∪ *Ract* ‘ *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* $\Longrightarrow$ *step s a* = (*ou*,*s'*) $\Longrightarrow$ *pID* $\in\in$ *postIDs s'*
⟨*proof*⟩

lemma *reach-visPost*: *reach s* $\Longrightarrow$ *vis s pID* $\in$ {*FriendV*, *PublicV*}
⟨*proof*⟩

lemma *reach-owner-userIDs*: *reach s* $\Longrightarrow$ *pID* $\in\in$ *postIDs s* $\Longrightarrow$ *owner s pID* $\in\in$ *userIDs s*
⟨*proof*⟩

lemma *reach-friendIDs-symmetric*:
reach s $\Longrightarrow$ *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* $\Longrightarrow$ *uid'* $\notin$ *set* (*friendIDs s uid*)
and *uid'* $\in\in$ *pendingFReqs s uid* $\Longrightarrow$ *uid* $\notin$ *set* (*friendIDs s uid'*)
⟨*proof*⟩

lemma *remove1-in-set*: *x* $\in\in$ *remove1 y xs* $\Longrightarrow$ *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'*] [] (**is** *?p*)
and *uid* $\in\in$ *friendIDs s uid'* $\longrightarrow$ *IDsOK s* [*uid*, *uid'*] [] (**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'* (**is** *?u*)
and *pid* $\in\in$ *postIDs s* $\longrightarrow$ *pid* $\in\in$ *postIDs s'* (**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 γ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
γ (*Trans* - *a* - -) =
 (*userOfA a* ∈ *Some* ‘ *UIDs*)

fun *g* :: (*state*,*act*,*out*)*trans* ⇒ *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 ntcs ntcs1 $\equiv$
$\forall$ *pid*. *pid* $\neq$ *PID* $\longrightarrow$ *ntcs pid* = *ntcs1 pid*

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

lemma *eeqButPID-eeq*[*simp*,*intro*!]: *eeqButPID ntcs ntcs*
⟨*proof*⟩

lemma *eeqButPID-sym*:
assumes *eeqButPID ntcs ntcs1* **shows** *eeqButPID ntcs1 ntcs*
⟨*proof*⟩

lemma *eeqButPID-trans*:
assumes *eeqButPID ntcs ntcs1* **and** *eeqButPID ntcs1 ntcs2* **shows** *eeqButPID ntcs ntcs2*
⟨*proof*⟩

lemma *eeqButPID-cong*:
assumes *eeqButPID ntcs ntcs1*
and *PID* = *PID* $\Longrightarrow$ *eqButT uu uu1*
and *pid* $\neq$ *PID* $\Longrightarrow$ *uu* = *uu1*
shows *eeqButPID* (*ntcs* (*pid* := *uu*)) (*ntcs1*(*pid* := *uu1*))
⟨*proof*⟩

lemma *eeqButPID-not-PID*:
⟦*eeqButPID ntcs ntcs1*; *pid* $\neq$ *PID*⟧ $\Longrightarrow$ *ntcs pid* = *ntcs1 pid*
⟨*proof*⟩

lemma *eeqButPID-toEq*:
assumes *eeqButPID ntcs ntcs1*
shows *ntcs* (*PID* := *pst*) = *ntcs1* (*PID* := *pst*)
⟨*proof*⟩

lemma *eeqButPID-update-post*:
assumes *eeqButPID ntcs ntcs1*
shows *eeqButPID* (*ntcs* (*pid* := *ntc*)) (*ntcs1* (*pid* := *ntc*))
⟨*proof*⟩

definition *eqButPID* :: *state* ⇒ *state* ⇒ *bool* **where**
eqButPID s s1 ≡
admin s = *admin s1* ∧

pendingUReqs s = *pendingUReqs s1* ∧ *userReq s* = *userReq s1* ∧
userIDs s = *userIDs s1* ∧ *user s* = *user s1* ∧ *pass s* = *pass s1* ∧

pendingFReqs s = *pendingFReqs s1* ∧ *friendReq s* = *friendReq s1* ∧ *friendIDs s* = *friendIDs s1* ∧

postIDs s = *postIDs s1* ∧ *admin s* = *admin s1* ∧
eeqButPID (*post s*) (*post s1*) ∧
owner s = *owner s1* ∧
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*
⟨*proof*⟩

lemma *eqButPID-sym*:
assumes *eqButPID s s1* **shows** *eqButPID s1 s*
⟨*proof*⟩

lemma *eqButPID-trans*:
assumes *eqButPID s s1* **and** *eqButPID s1 s2* **shows** *eqButPID s s2*
⟨*proof*⟩

lemma *eqButPID-stateSelectors*:
eqButPID s s1 ⟹
admin s = *admin s1* ∧

pendingUReqs s = *pendingUReqs s1* ∧ *userReq s* = *userReq s1* ∧
userIDs s = *userIDs s1* ∧ *user s* = *user s1* ∧ *pass s* = *pass s1* ∧

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

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

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

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

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 $eeqButPID\ ((post\ s)\ (pid := ntc))\ ((post\ s1)\ (pid := ntc))$
$\langle proof \rangle$

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

$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|pendingUReqs := uu1|\!))\ (s1\ (\!|pendingUReqs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|userReq := uu1|\!))\ (s1\ (\!|userReq := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|userIDs := uu1|\!))\ (s1\ (\!|userIDs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|user := uu1|\!))\ (s1\ (\!|user := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|pass := uu1|\!))\ (s1\ (\!|pass := uu2|\!))$

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

$(s1\ (\!|postIDs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|owner := uu1|\!))\ (s1\ (\!|owner := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow eeqButPID\ uu1\ uu2 \Longrightarrow eqButPID\ (s\ (\!|post := uu1|\!))\ (s1\ (\!|post := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|vis := uu1|\!))\ (s1\ (\!|vis := uu2|\!))$

$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|pendingFReqs := uu1|\!))\ (s1\ (\!|pendingFReqs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|friendReq := uu1|\!))\ (s1\ (\!|friendReq := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButPID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButPID\ (s\ (\!|friendIDs := uu1|\!))\ (s1\ (\!|friendIDs := uu2|\!))$

⟨*proof*⟩

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**
$openToUIDs\ s \equiv$
$\exists\ uid \in UIDs.$
$uid \in\in userIDs\ s\ \wedge$
$(uid = owner\ s\ PID \vee uid \in\in friendIDs\ s\ (owner\ s\ PID) \vee$
$vis\ s\ PID = PublicV)$

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 $openToUIDs\ s \longleftrightarrow openToUIDs\ s1$
⟨*proof*⟩

lemma *eqButPID-open*:
assumes *eqButPID s s1*
shows $open\ s \longleftrightarrow open\ s1$
⟨*proof*⟩

lemma *not-open-eqButPID*:
assumes *1*: $\neg$ *open s* **and** *2*: *eqButPID s s1*

shows $\neg$ *open s1*
$\langle proof \rangle$

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

lemma φ-*def2*:
  assumes *step s a* = (*ou*,s')
  shows
    φ (*Trans s a ou* s') $\longleftrightarrow$
    ($\exists$ *uid p pst*. *a* = *Uact* (*uPost uid p PID pst*) $\wedge$ *ou* = *outOK*) $\vee$
     open s $\neq$ *open* s'
$\langle proof \rangle$

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

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

lemma *eqButPID-step*:
assumes *ss1*: *eqButPID s s1*
and *step*: *step s a* = (*ou*,s')
and *step1*: *step s1 a* = (*ou1*,*s1*$'$)
shows *eqButPID* s' *s1*$'$
$\langle proof \rangle$

lemma *eqButPID-step-*φ*-imp*:
assumes *ss1*: *eqButPID s s1*
and *step*: *step s a* = (*ou*,s') **and** *step1*: *step s1 a* = (*ou1*,*s1*$'$)
and φ: φ (*Trans s a ou* s')
shows φ (*Trans s1 a ou1 s1*$'$)
$\langle proof \rangle$

lemma *eqButPID-step-*φ:
assumes $s's1'$: *eqButPID s s1*
and *step*: *step s a* = (*ou*,s') **and** *step1*: *step s1 a* = (*ou1*,*s1*$'$)
shows φ (*Trans s a ou* s') = φ (*Trans s1 a ou1 s1*$'$)

⟨*proof*⟩

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

6.3 Declassification bound

fun T :: (*state*,*act*,*out*) *trans* ⇒ *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* ⇒ *value list* ⇒ *bool*
and BO :: *value list* ⇒ *value list* ⇒ *bool*
where
 B-TVal[*simp*,*intro*!]:
  (*pstl* = [] ⟶ *pstl1* = []) ⟹ B (*map TVal pstl*) (*map TVal pstl1*)
|*B-BO*[*intro*]:
  BO vl vl1 ⟹ (*pstl* = [] ⟷ *pstl1* = []) ⟹ (*pstl* ≠ [] ⟹ *last pstl* = *last pstl1*)
⟹
   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 ⟹
   BO (*map TVal pstl* @ *OVal False* # *vl*) (*map TVal pstl* @ *OVal False* # *vl1*)

lemma *B-not-Nil*: *B vl vl1* ⟹ *vl* = [] ⟹ *vl1* = []
⟨*proof*⟩

lemma *B-OVal-True*:
assumes B (*OVal True* # *vl′*) *vl1*

shows $\exists$ *vl1'*. *BO vl' vl1'* $\wedge$ *vl1* = *OVal True # 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*⟩

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 γ: γ (*Trans s a ou s'*)
shows *ou* = *ou1*
⟨*proof*⟩

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'*
⟨*proof*⟩

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

definition Δ*1* :: *state* $\Rightarrow$ *value list* $\Rightarrow$ *state* $\Rightarrow$ *value list* $\Rightarrow$ *bool* **where**
Δ*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 Δ*2* :: *state* $\Rightarrow$ *value list* $\Rightarrow$ *state* $\Rightarrow$ *value list* $\Rightarrow$ *bool* **where**
Δ*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 Δ*31* :: *state* $\Rightarrow$ *value list* $\Rightarrow$ *state* $\Rightarrow$ *value list* $\Rightarrow$ *bool* **where**
Δ*31 s vl s1 vl1* $\equiv$

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

definition $\Delta 32 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
$\Delta 32\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ vll\ vll1.$
$BO\ vll\ vll1 \wedge$
$vl = OVal\ True\ \#\ vll \wedge vl1 = OVal\ True\ \#\ vll1) \wedge$
$s = s1 \wedge \neg\ open\ s$

definition $\Delta 4 :: state \Rightarrow value\ list \Rightarrow state \Rightarrow value\ list \Rightarrow bool$ **where**
$\Delta 4\ s\ vl\ s1\ vl1 \equiv$
$PID \in\in postIDs\ s \wedge$
$(\exists\ pstl\ vll\ vll1.$
$B\ vll\ vll1 \wedge$
$vl = map\ TVal\ pstl\ @\ OVal\ False\ \#\ vll \wedge vl1 = map\ TVal\ pstl\ @\ OVal\ False$
$\#\ vll1) \wedge$
$s = s1 \wedge open\ s$

lemma *istate-*$\Delta 0$:
assumes B: $B\ vl\ vl1$
shows $\Delta 0\ istate\ vl\ istate\ vl1$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 0$: $unwind\text{-}cont\ \Delta 0\ \{\Delta 0,\Delta 1,\Delta 2,\Delta 31,\Delta 32,\Delta 4\}$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 1$: $unwind\text{-}cont\ \Delta 1\ \{\Delta 1\}$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 2$: $unwind\text{-}cont\ \Delta 2\ \{\Delta 2\}$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 31$: $unwind\text{-}cont\ \Delta 31\ \{\Delta 31,\Delta 32\}$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 32$: $unwind\text{-}cont\ \Delta 32\ \{\Delta 2,\Delta 32,\Delta 4\}$
⟨*proof*⟩

lemma *unwind-cont-*$\Delta 4$: $unwind\text{-}cont\ \Delta 4\ \{\Delta 1,\Delta 31,\Delta 32,\Delta 4\}$
⟨*proof*⟩

definition Gr **where**
$Gr =$

{
(Δ*0*, {Δ*0*,Δ*1*,Δ*2*,Δ*31*,Δ*32*,Δ*4*}),
(Δ*1*, {Δ*1*}),
(Δ*2*, {Δ*2*}),
(Δ*31*, {Δ*31*,Δ*32*}),
(Δ*32*, {Δ*2*,Δ*32*,Δ*4*}),
(Δ*4*, {Δ*1*,Δ*31*,Δ*32*,Δ*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* ⇒ *userID list* ⇒ *userID list* ⇒ *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′ uidl*) (*remove1 uid′ uidl1*)
⟨*proof*⟩

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl uid uidl uidl1*
and *uid′* ∈∈ *uidl* ⟷ *uid′* ∈∈ *uidl1*
shows *eqButUIDl uid* (*uidl* ## *uid′*) (*uidl1* ## *uid′*)
⟨*proof*⟩

definition *eqButUIDf* **where**
eqButUIDf frds frds1 ≡
  eqButUIDl UID2 (*frds UID1*) (*frds1 UID1*)
∧ *eqButUIDl UID1* (*frds UID2*) (*frds1 UID2*)
∧ (∀ *uid*. *uid* ≠ *UID1* ∧ *uid* ≠ *UID2* ⟶ *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*
⟨*proof*⟩

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 $uid = UID1 \Longrightarrow eqButUIDl\ UID2\ uu\ uu1$
and $uid = UID2 \Longrightarrow eqButUIDl\ UID1\ uu\ uu1$
and $uid \neq UID1 \Longrightarrow uid \neq UID2 \Longrightarrow uu = uu1$
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*:
$[\![eqButUIDf\ frds\ frds1;\ uid \neq UID1;\ uid \neq UID2]\!] \Longrightarrow frds\ uid = frds1\ uid$
⟨*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*
⟨*proof*⟩

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

and $\neg$ (*uid*,*uid*′) $\in$ {(*UID1*,*UID2*), (*UID2*,*UID1*)} $\Longrightarrow$ *uu* = *uu1*
shows *eqButUID12* (*fun-upd2 freq uid uid′ uu*) (*fun-upd2 freq1 uid uid′ uu1*)
⟨*proof*⟩

lemma *eqButUID12-not-UID*:
⟦*eqButUID12 freq freq1*; $\neg$ (*uid*,*uid*′) $\in$ {(*UID1*,*UID2*), (*UID2*,*UID1*)}⟧ $\Longrightarrow$ *freq uid uid′* = *freq1 uid uid′*
⟨*proof*⟩

definition *eqButUID* :: *state* ⇒ *state* ⇒ *bool* **where**
eqButUID s s1 ≡
admin s = *admin s1* ∧

pendingUReqs s = *pendingUReqs s1* ∧ *userReq s* = *userReq s1* ∧
userIDs s = *userIDs s1* ∧ *user s* = *user s1* ∧ *pass s* = *pass s1* ∧

eqButUIDf (*pendingFReqs s*) (*pendingFReqs s1*) ∧
eqButUID12 (*friendReq s*) (*friendReq s1*) ∧
eqButUIDf (*friendIDs s*) (*friendIDs s1*) ∧

postIDs s = *postIDs s1* ∧ *admin s* = *admin s1* ∧
post s = *post s1* ∧
owner s = *owner s1* ∧
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*
⟨*proof*⟩

lemma *eqButUID-sym*[*sym*]:
assumes *eqButUID s s1* **shows** *eqButUID s1 s*
⟨*proof*⟩

lemma *eqButUID-trans*[*trans*]:
assumes *eqButUID s s1* **and** *eqButUID s1 s2* **shows** *eqButUID s s2*
⟨*proof*⟩

lemma *eqButUID-stateSelectors*:
$eqButUID\ s\ s1 \Longrightarrow$
$admin\ s = admin\ s1\ \wedge$

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

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

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

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

lemma *eqButUID-eqButUID2*:
$eqButUID\ s\ s1 \Longrightarrow eqButUIDl\ UID2\ (friendIDs\ s\ UID1)\ (friendIDs\ s1\ UID1)$
$\langle proof \rangle$

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

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

$\bigwedge uu1\ uu2.\ eqButUID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButUID\ (s\ (\!|pendingUReqs := uu1|\!))\ (s1\ (\!|pendingUReqs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButUID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButUID\ (s\ (\!|userReq := uu1|\!))\ (s1\ (\!|userReq := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButUID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButUID\ (s\ (\!|userIDs := uu1|\!))\ (s1\ (\!|userIDs := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButUID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButUID\ (s\ (\!|user := uu1|\!))\ (s1\ (\!|user := uu2|\!))$
$\bigwedge uu1\ uu2.\ eqButUID\ s\ s1 \Longrightarrow uu1 = uu2 \Longrightarrow eqButUID\ (s\ (\!|pass := uu1|\!))\ (s1\ (\!|pass := uu2|\!))$

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

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* $=$ *uu2* $\Longrightarrow$ *eqButUID* (*s* (|*vis* := *uu1*|)) (*s1* (|*vis* := *uu2*|))

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*pendingFReqs* := *uu1*|)) (*s1* (|*pendingFReqs* := *uu2*|))
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUID12 uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*friendReq* := *uu1*|)) (*s1* (|*friendReq* := *uu2*|))
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* (|*friendIDs* := *uu1*|)) (*s1* (|*friendIDs* := *uu2*|))

⟨*proof*⟩

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* ⇒ *bool* — Openness by absence
where *openByA s* ≡ ¬ *UID1* ∈∈ *userIDs s* ∨ ¬ *UID2* ∈∈ *userIDs s*

definition *openByF* :: *state* ⇒ *bool* — Openness by friendship
where *openByF s* ≡ ∃ *uid* ∈ *UIDs. uid* ∈∈ *friendIDs s UID1* ∨ *uid* ∈∈ *friendIDs s UID2*

definition *open* :: *state* ⇒ *bool*
where *open s* ≡ *openByA s* ∨ *openByF s*

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

definition *friends12* :: *state* ⇒ *bool*
where *friends12 s* ≡ *UID1* ∈∈ *friendIDs s UID2* ∧ *UID2* ∈∈ *friendIDs s UID1*

fun φ :: (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
φ (*Trans s* (*Cact* (*cFriend uid p uid'*)) *ou s'*) =
((*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
open s ≠ *open s'*)
|
φ (*Trans s* (*Dact* (*dFriend uid p uid'*)) *ou s'*) =
((*uid*,*uid'*) ∈ {(*UID1*,*UID2*), (*UID2*,*UID1*)} ∧ *ou* = *outOK* ∨
open s ≠ *open s'*)
|
φ (*Trans s* (*Cact* (*cUser uid p uid' p'*)) *ou s'*) =
(*open s* ≠ *open s'*)
|

φ - = *False*

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

lemma φE:
assumes φ: $\varphi\ (Trans\ s\ a\ ou\ s')$ (**is** $\varphi\ ?trn$)
and *step*: $step\ s\ a = (ou,\ s')$
and *rs*: $reach\ s$
obtains $(Friend)\ uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')\ ou = outOK\ f$
$?trn = FrVal\ True$
$uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' =$
$UID1$
$IDsOK\ s\ [UID1,\ UID2]\ []$
$\neg friends12\ s\ friends12\ s'$
| $(Unfriend)\ uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')\ ou = outOK\ f$
$?trn = FrVal\ False$
$uid = UID1 \wedge uid' = UID2 \vee uid = UID2 \wedge uid' =$
$UID1$
$IDsOK\ s\ [UID1,\ UID2]\ []$
$friends12\ s\ \neg friends12\ s'$
| $(OpenF)\ uid\ p\ uid'$ **where** $a = Cact\ (cFriend\ uid\ p\ uid')$
$(uid \in UIDs \wedge uid' \in \{UID1,UID2\}) \vee (uid' \in UIDs\ \wedge$
$uid \in \{UID1,UID2\})$
$ou = outOK\ f\ ?trn = OVal\ True\ \neg openByF\ s\ openByF\ s'$
$\neg openByA\ s\ \neg openByA\ s'$
| $(CloseF)\ uid\ p\ uid'$ **where** $a = Dact\ (dFriend\ uid\ p\ uid')$
$(uid \in UIDs \wedge uid' \in \{UID1,UID2\}) \vee (uid' \in UIDs$
$\wedge\ uid \in \{UID1,UID2\})$
$ou = outOK\ f\ ?trn = OVal\ False\ openByF\ s\ \neg openByF$
s'
$\neg openByA\ s\ \neg openByA\ s'$
| $(CloseA)\ uid\ p\ uid'\ p'$ **where** $a = Cact\ (cUser\ uid\ p\ uid'\ p')$
$uid' \in \{UID1,UID2\}\ openByA\ s\ \neg openByA\ s'$
$\neg openByF\ s\ \neg openByF\ s'$
$ou = outOK\ f\ ?trn = OVal\ False$
$\langle proof\rangle$

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

lemma *step-friends12-φ*:
assumes $step\ s\ a = (ou,\ s')$
and $friends12\ s \neq friends12\ s'$
shows $\varphi\ (Trans\ s\ a\ ou\ s')$
⟨*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 $set\ (friendIDs\ s\ uid) = set\ (friendIDs\ s1\ uid)$
⟨*proof*⟩

lemma *distinct-remove1-idem*: $distinct\ xs \Longrightarrow remove1\ y\ (remove1\ y\ xs) = remove1\ y\ xs$
⟨*proof*⟩

lemma *Cact-cFriend-step-eqButUID*:
assumes *step*: $step\ s\ (Cact\ (cFriend\ uid\ p\ uid')) = (ou,s')$
and *s*: *reach s*
and *uids*: $(uid = UID1 \wedge uid' = UID2) \vee (uid = UID2 \wedge uid' = UID1)$ (**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* $\Longrightarrow$ *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*: $a \neq$ *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 φ (*Trans s a ou s'*) = φ (*Trans s1 a ou1 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*
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**
$alternatingFriends\ []\ \text{-} = True$
$|\ alternatingFriends\ (FrVal\ st\ \#\ vl)\ st' \longleftrightarrow st' = (\neg st) \wedge alternatingFriends\ vl\ st$
$|\ alternatingFriends\ (OVal\ \text{-}\ \#\ vl)\ st = alternatingFriends\ vl\ st$

inductive $BO :: value\ list \Rightarrow value\ list \Rightarrow bool$
and $BC :: value\ list \Rightarrow value\ list \Rightarrow bool$
where
$BO\text{-}FrVal[simp,intro!]$:
$BO\ (map\ FrVal\ fs)\ (map\ FrVal\ fs)$
$|BO\text{-}BC[intro]$:
$BC\ vl\ vl1 \Longrightarrow$
$BO\ (map\ FrVal\ fs\ @\ OVal\ False\ \#\ vl)\ (map\ FrVal\ fs\ @\ OVal\ False\ \#\ vl1)$

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

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

lemma $BO\text{-}Nil\text{-}Nil$: $BO\ vl\ vl1 \Longrightarrow vl = [] \Longrightarrow vl1 = []$
$\langle proof \rangle$

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** $B = B$
$\langle proof \rangle$

7.4 Unwinding proof

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

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*]
$\langle$*proof*$\rangle$

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*]
$\langle$*proof*$\rangle$

definition $\Delta 0$ :: *state* $\Rightarrow$ *value list* $\Rightarrow$ *state* $\Rightarrow$ *value list* $\Rightarrow$ *bool* **where**
$\Delta 0$ *s vl s1 vl1* $\equiv$
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* = [] $\longleftrightarrow$ *fs1* = []) $\wedge$
(*fs* $\neq$ [] $\longrightarrow$ *last fs* = *last fs1*) $\wedge$
(*fs* = [] $\longrightarrow$ *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 = [] $\longleftrightarrow$ *fs1* = [] *fs* $\neq$ [] $\longrightarrow$ *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* ⇒ *userID list* ⇒ *userID list* ⇒ *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′ uidl*) (*remove1 uid′ uidl1*)
⟨*proof*⟩

lemma *eqButUIDl-snoc-cong*:
assumes *eqButUIDl uid uidl uidl1*
and *uid′* ∈∈ *uidl* ⟷ *uid′* ∈∈ *uidl1*
shows *eqButUIDl uid* (*uidl* ## *uid′*) (*uidl1* ## *uid′*)
⟨*proof*⟩

definition *eqButUIDf* **where**
eqButUIDf frds frds1 ≡
  eqButUIDl UID2 (*frds UID1*) (*frds1 UID1*)
∧ *eqButUIDl UID1* (*frds UID2*) (*frds1 UID2*)
∧ (∀ *uid*. *uid* ≠ *UID1* ∧ *uid* ≠ *UID2* ⟶ *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*
⟨*proof*⟩

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* ⟹ *eqButUIDl UID2 uu uu1*
and *uid* = *UID2* ⟹ *eqButUIDl UID1 uu uu1*
and *uid* ≠ *UID1* ⟹ *uid* ≠ *UID2* ⟹ *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*:
⟦*eqButUIDf frds frds1*; *uid* ≠ *UID1*; *uid* ≠ *UID2*⟧ ⟹ *frds uid* = *frds1 uid*
⟨*proof*⟩

lemma *eqButUIDf-not-UID′*:

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

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

lemma *eqButUID12-not-UID*:
$[\![$*eqButUID12 freq freq1*; $\neg\ (uid,uid') \in \{(UID1,UID2), (UID2,UID1)\}]\!] \Longrightarrow$ *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 ∧ admin s = admin s1 ∧
post s = post s1 ∧
owner s = owner s1 ∧
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*
⟨*proof*⟩

lemma *eqButUID-sym*[*sym*]:
assumes *eqButUID s s1* **shows** *eqButUID s1 s*
⟨*proof*⟩

lemma *eqButUID-trans*[*trans*]:
assumes *eqButUID s s1* **and** *eqButUID s1 s2* **shows** *eqButUID s s2*
⟨*proof*⟩

lemma *eqButUID-stateSelectors*:
eqButUID s s1 ⟹
admin s = admin s1 ∧

pendingUReqs s = pendingUReqs s1 ∧ userReq s = userReq s1 ∧
userIDs s = userIDs s1 ∧ user s = user s1 ∧ pass s = pass s1 ∧

eqButUIDf (pendingFReqs s) (pendingFReqs s1) ∧
eqButUID12 (friendReq s) (friendReq s1) ∧
eqButUIDf (friendIDs s) (friendIDs s1) ∧

postIDs s = postIDs s1 ∧ admin s = admin s1 ∧
post s = post s1 ∧
owner s = owner s1 ∧
vis s = vis s1 ∧

IDsOK s = IDsOK s1
⟨*proof*⟩

lemma *eqButUID-eqButUID2*:
eqButUID s s1 ⟹ eqButUIDl UID2 (friendIDs s UID1) (friendIDs s1 UID1)
⟨*proof*⟩

lemma *eqButUID-not-UID*:
eqButUID s s1 ⟹ uid ≠ UID ⟹ post s uid = post s1 uid
⟨*proof*⟩

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

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*admin* := *uu1*⦈) (*s1* ⦇*admin* := *uu2*⦈)

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*pendingUReqs* := *uu1*⦈) (*s1* ⦇*pendingUReqs* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*userReq* := *uu1*⦈) (*s1* ⦇*userReq* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*userIDs* := *uu1*⦈) (*s1* ⦇*userIDs* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*user* := *uu1*⦈) (*s1* ⦇*user* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*pass* := *uu1*⦈) (*s1* ⦇*pass* := *uu2*⦈)

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*postIDs* := *uu1*⦈) (*s1* ⦇*postIDs* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*owner* := *uu1*⦈) (*s1* ⦇*owner* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*post* := *uu1*⦈) (*s1* ⦇*post* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *uu1* = *uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*vis* := *uu1*⦈) (*s1* ⦇*vis* := *uu2*⦈)

$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*pendingFReqs* := *uu1*⦈) (*s1* ⦇*pendingFReqs* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUID12 uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*friendReq* := *uu1*⦈) (*s1* ⦇*friendReq* := *uu2*⦈)
$\bigwedge$ *uu1 uu2. eqButUID s s1* $\Longrightarrow$ *eqButUIDf uu1 uu2* $\Longrightarrow$ *eqButUID* (*s* ⦇*friendIDs* := *uu1*⦈) (*s1* ⦇*friendIDs* := *uu2*⦈)

⟨*proof*⟩

8.2 Value Setup

datatype *fUser* = *U1* | *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* ⇒ *bool* — Openness by absence
where *openByA s* ≡ ¬ *UID1* ∈∈ *userIDs s* ∨ ¬ *UID2* ∈∈ *userIDs s*

definition *openByF* :: *state* ⇒ *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**
φ (*Trans s* (*Cact* (*cFriendReq uid p uid' req*)) *ou s'*) =
  ((*uid*,*uid'*) $\in$ {(*UID1*,*UID2*), (*UID2*,*UID1*)} $\wedge$ *ou* = *outOK*)
|
φ (*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'*)
|
φ (*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'*)
|
φ (*Trans s* (*Cact* (*cUser uid p uid' p'*)) *ou s'*) =
  (*open s* $\neq$ *open s'*)
|
φ - = *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 φ: φ (*Trans s a ou s'*) (**is** φ *?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*
f ?trn = *FRVal u req u* = *U1 IDsOK s* [*UID1*, *UID2*] []
¬*friends12 s* ¬*friends12 s'* *open s'* = *open s*
UID1 ∈∈ *pendingFReqs s' UID2 UID1* ∉ *set* (*pendingFReqs s UID2*)
UID2 ∈∈ *pendingFReqs s' UID1* ⟷ *UID2* ∈∈ *pendingFReqs s UID1*
| (*FReq2*) *u p req* **where** *a* = *Cact* (*cFriendReq UID2 p UID1 req*) *ou* = *outOK*
f ?trn = *FRVal u req u* = *U2 IDsOK s* [*UID1*, *UID2*] []
¬*friends12 s* ¬*friends12 s'* *open s'* = *open s*
UID2 ∈∈ *pendingFReqs s' UID1 UID2* ∉ *set* (*pendingFReqs s UID1*)
UID1 ∈∈ *pendingFReqs s' UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*
| (*Friend*) *uid p uid'* **where** *a* = *Cact* (*cFriend uid p uid'*) *ou* = *outOK f ?trn* = *FVal True*
uid = *UID1* ∧ *uid'* = *UID2* ∨ *uid* = *UID2* ∧ *uid'* = *UID1*
IDsOK s [*UID1*, *UID2*] []
¬*friends12 s friends12 s' uid'* ∈∈ *pendingFReqs s uid*
UID1 ∉ *set* (*pendingFReqs s' UID2*)
UID2 ∉ *set* (*pendingFReqs s' UID1*)
| (*Unfriend*) *uid p uid'* **where** *a* = *Dact* (*dFriend uid p uid'*) *ou* = *outOK f ?trn* = *FVal False*
uid = *UID1* ∧ *uid'* = *UID2* ∨ *uid* = *UID2* ∧ *uid'* = *UID1*
IDsOK s [*UID1*, *UID2*] []
friends12 s ¬*friends12 s'*
UID1 ∉ *set* (*pendingFReqs s' UID2*)
UID1 ∉ *set* (*pendingFReqs s UID2*)
UID2 ∉ *set* (*pendingFReqs s' UID1*)
UID2 ∉ *set* (*pendingFReqs s UID1*)
| (*OpenF*) *uid p uid'* **where** *a* = *Cact* (*cFriend uid p uid'*)
(*uid* ∈ *UIDs* ∧ *uid'* ∈ {*UID1*,*UID2*}) ∨ (*uid'* ∈ *UIDs* ∧ *uid* ∈ {*UID1*,*UID2*})
ou = *outOK f ?trn* = *OVal True* ¬*openByF s openByF s'*
¬*openByA s* ¬*openByA s'*
friends12 s' = *friends12 s*
UID1 ∈∈ *pendingFReqs s' UID2* ⟷ *UID1* ∈∈ *pendingFReqs s UID2*
UID2 ∈∈ *pendingFReqs s' UID1* ⟷ *UID2* ∈∈ *pendingFReqs s UID1*
| (*CloseF*) *uid p uid'* **where** *a* = *Dact* (*dFriend uid p uid'*)
(*uid* ∈ *UIDs* ∧ *uid'* ∈ {*UID1*,*UID2*}) ∨ (*uid'* ∈ *UIDs* ∧ *uid* ∈ {*UID1*,*UID2*})

$ou = outOK\ f\ ?trn = OVal\ False\ openByF\ s\ \neg openByF$
s'
$\neg openByA\ s\ \neg openByA\ s'$
$friends12\ s' = friends12\ s$
$UID1 \in\in pendingFReqs\ s'\ UID2 \longleftrightarrow UID1 \in\in$
$pendingFReqs\ s\ UID2$
$UID2 \in\in pendingFReqs\ s'\ UID1 \longleftrightarrow UID2 \in\in$
$pendingFReqs\ s\ UID1$
| $(CloseA)\ uid\ p\ uid'\ p'$ **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*: $distinct\ xs \Longrightarrow remove1\ y\ (remove1\ y\ xs) = remove1\ y\ xs$
$\langle proof \rangle$

lemma *Cact-cFriend-step-eqButUID*:
assumes *step*: *step s* (*Cact* (*cFriend uid p uid′*)) = (*ou*,*s′*)
and *s*: *reach s*
and *uids*: (*uid* = *UID1* ∧ *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* ⟹ *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*: $\forall$ *req*. $a \neq$ *Cact* (*cFriend UID1* (*pass s UID1*) *UID2*) $\wedge$
$a \neq$ *Cact* (*cFriend UID2* (*pass s UID2*) *UID1*) $\wedge$
$a \neq$ *Cact* (*cFriendReq UID1* (*pass s UID1*) *UID2 req*) $\wedge$
$a \neq$ *Cact* (*cFriendReq UID2* (*pass s UID2*) *UID1 req*) $\wedge$
$a \neq$ *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*: $\forall$ *req*. $a \neq$ *Cact* (*cFriend UID1* (*pass s UID1*) *UID2*) $\wedge$
$a \neq$ *Cact* (*cFriend UID2* (*pass s UID2*) *UID1*) $\wedge$
$a \neq$ *Cact* (*cFriendReq UID1* (*pass s UID1*) *UID2 req*) $\wedge$
$a \neq$ *Cact* (*cFriendReq UID2* (*pass s UID2*) *UID1 req*) $\wedge$
$a \neq$ *Dact* (*dFriend UID1* (*pass s UID1*) *UID2*) $\wedge$
$a \neq$ *Dact* (*dFriend UID2* (*pass s UID2*) *UID1*)
shows φ (*Trans s a ou s'*) = φ (*Trans s1 a ou1 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* ⟷ (¬*st*) ∧ (¬*r1*) ∧ *validValSeq vl st True r2*
| *validValSeq* (*FRVal U2 req* # *vl*) *st r1 r2* ⟷ (¬*st*) ∧ (¬*r2*) ∧ *validValSeq vl st r1 True*
| *validValSeq* (*FVal True* # *vl*) *st r1 r2* ⟷ (¬*st*) ∧ (*r1* ∨ *r2*) ∧ *validValSeq vl True False False*
| *validValSeq* (*FVal False* # *vl*) *st r1 r2* ⟷ *st* ∧ (¬*r1*) ∧ (¬*r2*) ∧ *validValSeq vl False False False*
| *validValSeq* (*OVal True* # *vl*) *st r1 r2* ⟷ *validValSeq vl st r1 r2*
| *validValSeq* (*OVal False* # *vl*) *st r1 r2* ⟷ *validValSeq vl st r1 r2*

abbreviation *validValSeqFrom* :: *value list* ⇒ *state* ⇒ *bool*
where *validValSeqFrom vl s*
≡ *validValSeq vl* (*friends12 s*) (*UID1* ∈∈ *pendingFReqs s UID2*) (*UID2* ∈∈ *pendingFReqs s UID1*)

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

inductive BO :: *value list* $\Rightarrow$ *value list* $\Rightarrow$ *bool*
and BC :: *value list* $\Rightarrow$ *value list* $\Rightarrow$ *bool*
where
BO-FVal[*simp*,*intro*!]:
BO (*map FVal fs*) (*map FVal fs*)
|*BO-BC*[*intro*]:
$BC\ vl\ vl1 \Longrightarrow$
BO (*map FVal fs* @ *OVal False* # *vl*) (*map FVal fs* @ *OVal False* # *vl1*)

|*BC-FVal*[*simp*,*intro*!]:
BC (*map FVal fs*) (*map FVal fs1*)
|*BC-BO*[*intro*]:
$BO\ vl\ vl1 \Longrightarrow (fs = [] \longleftrightarrow fs1 = []) \Longrightarrow (fs \neq [] \Longrightarrow last\ fs = last\ fs1) \Longrightarrow$
BC (*map FVal fs* @ *OVal True* # *vl*)
(*map FVal fs1* @ *OVal True* # *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 False* if *vl* begins with *OVal 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 False*), in which case the observer can deduce that no friendship request has happened yet.

definition $B\ vl\ vl1 \equiv BO$ (*filter* (*Not o isFRVal*) *vl*) (*filter* (*Not o isFRVal*) *vl1*) $\wedge$
validValSeqFrom vl1 istate $\wedge$
$(vl = [] \longrightarrow vl1 = []) \wedge$
$(vl \neq [] \wedge hd\ vl = OVal\ False \longrightarrow vl1 \neq [] \wedge hd\ vl1 = OVal\ False)$

lemma *BO-Nil-iff*: $BO\ vl\ vl1 \Longrightarrow vl = [] \longleftrightarrow vl1 = []$
⟨*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** $B = B$

$\langle proof \rangle$

lemma *validFrom-validValSeq*:
assumes *validFrom s tr*
and *reach s*
shows *validValSeqFrom* (*V tr*) *s*
$\langle proof \rangle$

lemma *validFrom istate tr* $\Longrightarrow$ *validValSeqFrom* (*V tr*) *istate*
$\langle proof \rangle$

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 γ: γ (*Trans s a ou s'*)
and *os*: *open s* $\longrightarrow$ *friendIDs s* = *friendIDs s1*
shows *ou* = *ou1*
$\langle proof \rangle$

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 φ (*Trans s a outOK s'*)
  and *f* (*Trans s a outOK s'*) = *FRVal u req*
  and *validValSeqFrom vl s'*
$\langle proof \rangle$

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 φ (*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* = []

| (*FVal*) *fv fs* **where** *vl* = *FVal fv* # *map FVal fs* **and** *vl1* = []
| (*FVal1*) *fv fs fs1* **where** *vl* = *map FVal fs* **and** *vl1* = *FVal fv* # *map FVal fs1*
| (*BO-FVal*) *fv fv′ fs vl′ vl1′* **where** *vl* = *FVal fv* # *map FVal fs* @ *FVal fv′* # *OVal True* # *vl′*
and *vl1* = *FVal fv′* # *OVal True* # *vl1′* **and** *BO vl′ vl1′*
| (*BO-FVal1*) *fv fv′ fs fs1 vl′ vl1′* **where** *vl* = *map FVal fs* @ *FVal fv′* # *OVal True* # *vl′*
and *vl1* = *FVal fv* # *map FVal fs1* @ *FVal fv′* # *OVal True* # *vl1′*
and *BO vl′ vl1′*
| (*FVal-BO*) *fv vl′ vl1′* **where** *vl* = *FVal fv* # *OVal True* # *vl′*
and *vl1* = *FVal fv* # *OVal True* # *vl1′* **and** *BO vl′ vl1′*
| (*OVal*) *vl′ vl1′* **where** *vl* = *OVal True* # *vl′* **and** *vl1* = *OVal True* # *vl1′* **and** *BO vl′ vl1′*
⟨*proof*⟩

definition Δ*0* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*0 s vl s1 vl1* ≡
s = *s1* ∧ *B vl vl1* ∧ *open s* ∧ (¬*IDsOK s* [*UID1*, *UID2*] [])

definition Δ*1* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*1 s vl s1 vl1* ≡
eqButUID s s1 ∧ *friendIDs s* = *friendIDs s1* ∧ *open s* ∧
BO (*filter* (*Not o isFRVal*) *vl*) (*filter* (*Not o isFRVal*) *vl1*) ∧
validValSeqFrom vl1 s1 ∧
IDsOK s1 [*UID1*, *UID2*] []

definition Δ*2* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*2 s vl s1 vl1* ≡ (∃ *fs fs1*.
eqButUID s s1 ∧ ¬*open s* ∧
validValSeqFrom vl1 s1 ∧
filter (*Not o isFRVal*) *vl* = *map FVal fs* ∧
filter (*Not o isFRVal*) *vl1* = *map FVal fs1*)

definition Δ*3* :: *state* ⇒ *value list* ⇒ *state* ⇒ *value list* ⇒ *bool* **where**
Δ*3 s vl s1 vl1* ≡ (∃ *fs fs1 vlr vlr1*.
eqButUID s s1 ∧ ¬*open s* ∧ *BO vlr vlr1* ∧
validValSeqFrom vl1 s1 ∧
(*fs* = [] ⟷ *fs1* = []) ∧
(*fs* ≠ [] ⟶ *last fs* = *last fs1*) ∧
(*fs* = [] ⟶ *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*)

lemma Δ*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 Δ*2 s vl s1 vl1*
⟨*proof*⟩

lemma Δ*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 Δ*3 s vl s1 vl1*
⟨*proof*⟩

lemma *istate-*Δ*0*:
assumes *B*: *B vl vl1*
shows Δ*0 istate vl istate vl1*
⟨*proof*⟩

lemma *unwind-cont-*Δ*0*: *unwind-cont* Δ*0* {Δ*0*,Δ*1*,Δ*2*,Δ*3*}
⟨*proof*⟩

lemma *unwind-cont-*Δ*1*: *unwind-cont* Δ*1* {Δ*1*,Δ*2*,Δ*3*}
⟨*proof*⟩

lemma *unwind-cont-*Δ*2*: *unwind-cont* Δ*2* {Δ*2*, Δ*1*}
⟨*proof*⟩

lemma *unwind-cont-*Δ*3*: *unwind-cont* Δ*3* {Δ*3*,Δ*1*}
⟨*proof*⟩

definition *Gr* **where**
Gr =
{
(Δ*0*, {Δ*0*,Δ*1*,Δ*2*,Δ*3*}),
(Δ*1*, {Δ*1*,Δ*2*,Δ*3*}),
(Δ*2*, {Δ*2*,Δ*1*}),
(Δ*3*, {Δ*3*,Δ*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 uidd trn means "The transaction *trn* is a successful post creation by user *uidd*"

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

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

fun *SNVU* :: *userID* ⇒ *vis* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SNVU uidd vvs (*Trans s* (*Uact* (*uVisPost uid p pid vs*)) *ou s'*) =
(*ou* = *outOK* ∧ (*uid*,*pid*) = (*uidd*,*PID*) ∧ *vs* = *vvs*)
|
SNVU uidd vvis - = *False*

definition *proper* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper tr ≡
VIS = *FriendV*
∨
(∃ *uid tr1 trn tr2 trnn tr3*.
tr = *tr1* @ *trn* # *tr2* @ *trnn* # *tr3* ∧
SNC uid trn ∧ *SNVU uid VIS trnn* ∧ (∀ *vis*. *never* (*SNVU uid vis*) *tr3*))

definition *proper1* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper1 tr ≡
∃ *tr2 trnn tr3*.
tr = *tr2* @ *trnn* # *tr3* ∧
SNVU (*owner* (*srcOf trnn*) *PID*) *VIS trnn*

lemma *not-never-ex*:
assumes ¬ *never P xs*
shows ∃ *xs1 x xs2*. *xs* = *xs1* @ *x* # *xs2* ∧ *P x* ∧ *never P xs2*
⟨*proof*⟩

lemma *SNVU-postIDs*:
assumes *validTrans trn* **and** *SNVU uid vs trn*
shows *PID* ∈∈ *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* ∈∈ *postIDs* (*srcOf trn*)

shows *owner* (*srcOf trn*) *PID* = *owner* (*tgtOf trn*) *PID*
⟨*proof*⟩

lemma *owner-valid*:
assumes *valid tr* **and** *PID* ∈∈ *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* ∈∈ *postIDs* (*srcOf trn*)
and ∀ *vs*. ¬ *SNVU* (*owner* (*srcOf trn*) *PID*) *vs trn*
shows *vis* (*srcOf trn*) *PID* = *vis* (*tgtOf trn*) *PID*
⟨*proof*⟩

lemma *SNVU-vis-valid*:
assumes *valid tr* **and** *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
and ∀ *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 *vtr*: *valid tr* **and** *PID*: *PID* ∈∈ *postIDs* (*srcOf* (*hd tr*))
and *tr*: *proper1 tr* **and** *v*: *vis* (*tgtOf* (*last tr*)) *PID* = *VIS*
shows ∃ *tr2 trnn tr3*.
tr = *tr2* @ *trnn* # *tr3* ∧
SNVU (*owner* (*srcOf trnn*) *PID*) *VIS trnn* ∧ (∀ *vis*. *never* (*SNVU* (*owner* (*srcOf trnn*) *PID*) *vis*) *tr3*)
⟨*proof*⟩

lemma *SNVU-validTrans*:
assumes *validTrans trn*
and *PID* ∈∈ *postIDs* (*srcOf trn*)
and *vis* (*srcOf trn*) *PID* ≠ *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* ∈∈ *postIDs* (*srcOf* (*hd tr*))
shows *PID* ∈∈ *postIDs* (*tgtOf* (*last tr*))
⟨*proof*⟩

lemma *proper1-valid*:
assumes *V*: *VIS* ≠ *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*) *trans trace* ⇒ *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* ∈∈ *postIDs* (*tgtOf trn*)
⟨*proof*⟩

lemma *SNC-owner*:
assumes *validTrans trn* **and** *SNC uid trn*
shows *uid* = *owner* (*tgtOf trn*) *PID*
⟨*proof*⟩

theorem *post-accountability*:
assumes *v*: *valid tr* **and** *i*: *srcOf* (*hd tr*) = *istate*
and *PIDin*: *PID* ∈∈ *postIDs* (*tgtOf* (*last tr*))
and *PID*: *vis* (*tgtOf* (*last tr*)) *PID* = *VIS*
shows *proper tr*
⟨*proof*⟩

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* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFRC uidd uidd′ (*Trans s* (*Cact* (*cFriendReq uid p uid′* -)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFRC uidd uidd′ - = *False*

SFC means "is a successful friend creation"

fun *SFC* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFC uidd uidd′ (*Trans s* (*Cact* (*cFriend uid p uid′*)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFC uidd uidd′ - = *False*

SFD means "is a successful friend deletion"

fun *SFD* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans* ⇒ *bool* **where**
SFD uidd uidd′ (*Trans s* (*Dact* (*dFriend uid p uid′*)) *ou s′*) = (*ou* = *outOK* ∧ (*uid*,*uid′*) = (*uidd*,*uidd′*))
|
SFD uidd uidd′ - = *False*

definition *proper1* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper1 tr ≡
∃ *trr trnn tr3*. *tr* = *trr* @ *trnn* # *tr3* ∧
(*SFC UID UID′ trnn* ∨ *SFC UID′ UID trnn*) ∧
never (*SFD UID UID′*) *tr3* ∧ *never* (*SFD UID′ UID*) *tr3*

lemma *SFC-validTrans*:
assumes *validTrans trn*
and ¬ *UID′* ∈∈ *friendIDs* (*srcOf trn*) *UID*
and *UID′* ∈∈ *friendIDs* (*tgtOf trn*) *UID*
shows *SFC UID UID′ trn* ∨ *SFC UID′ UID trn*
⟨*proof*⟩

lemma *SFD-validTrans*:
assumes *validTrans trn*
and *UID′* ∈∈ *friendIDs* (*tgtOf trn*) *UID*
shows ¬ *SFD UID UID′ trn* ∧ ¬ *SFD UID′ UID trn*
⟨*proof*⟩

lemma *SFC-SFD*:
assumes *SFC uid1 uid2 trn* **shows** ¬ *SFD uid3 uid4 trn*
⟨*proof*⟩

lemma *proper1-valid*:
assumes *valid tr*

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

lemma *istate-friendIDs*:
$\neg$ *UID′* $\in\in$ *friendIDs* (*istate*) *UID*
⟨*proof*⟩

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

definition *proper2* :: *userID* ⇒ *userID* ⇒ (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper2 uid uid′ tr ≡
∃ *tr1 trnn tr2*. *tr* = *tr1* @ *trnn* # *tr2* ∧ *SFRC uid uid′ trnn*

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

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

lemma *istate-pendingFReqs*:
$\neg$ *uid* $\in\in$ *pendingFReqs* (*istate*) *uid′*
⟨*proof*⟩

lemma *proper2-valid-istate*:
assumes *valid tr* **and** *srcOf* (*hd tr*) = *istate*
and *uid* $\in\in$ *pendingFReqs* (*tgtOf* (*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′*
⟨*proof*⟩

definition *proper* :: (*state*,*act*,*out*) *trans trace* ⇒ *bool* **where**
proper tr ≡
∃ *tr1 trn tr2 trnn tr3*. *tr* = *tr1* @ *trn* # *tr2* @ *trnn* # *tr3* ∧
(*SFRC UID′ UID trn* ∧ *SFC UID UID′ trnn* ∨
SFRC UID UID′ trn ∧ *SFC UID′ UID trnn*) ∧
never (*SFD UID UID′*) *tr3* ∧ *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*
⟨*proof*⟩

end

References

[1] T. Bauereiß, 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. Bauereiß, 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 für 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.