CoSMed: A Confidentiality-Verified Social Media Platform*

Thomas Bauereiß[1] Armando Pesenti Gritti[2,3] Andrei Popescu[2,4] Franco Raimondi[2]

[1] German Research Center for Artificial Intelligence (DFKI) Bremen, Germany
[2] School of Science and Technology, Middlesex University, UK
[3] Global NoticeBoard, UK
[4] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania

Abstract. This paper describes progress with our agenda of formal verification of information-flow security for realistic systems. We present CoSMed, a social media platform with verified document confidentiality. The system's kernel is implemented and verified in the proof assistant Isabelle/HOL. For verification, we employ the framework of *Bounded-Deducibility (BD) Security*, previously introduced for the conference system CoCon. CoSMed is a second major case study in this framework. For CoSMed, the static topology of declassification bounds and triggers that characterized previous instances of BD security has to give way to a dynamic integration of the triggers as part of the bounds.

1 Introduction

Web-based systems are pervasive in our daily activities. Examples include enterprise systems, social networks, e-commerce sites and cloud services. Such systems pose notable challenges regarding confidentiality [1].

Recently, we have started a line of work aimed at addressing information flow security problems of realistic web-based systems by interactive theorem proving—using our favorite proof assistant, Isabelle/HOL [26, 27]. We have introduced a security notion that allows a very fine-grained specification of what an attacker can observe about the system, and what information is to be kept confidential and in which situations. In our case studies, we assume the observers to be users of the system, and our goal is to verify that, by interacting with the system, the observers cannot learn more about confidential information than what we have specified. As a first case study, we have developed CoCon [18], a conference system (*à la* EasyChair) verified for confidentiality. We have verified a comprehensive list of confidentiality properties, systematically covering the relevant sources of information from CoCon's application logic [18, §4.5]. For example, besides authors, only PC members are allowed to learn about the content of submitted papers, and nothing beyond the last submitted version before the deadline.

This paper introduces a second major end product of this line of work: CoSMed, a confidentiality-verified social media platform. CoSMed allows users to register and post information, and to restrict access to this information based on friendship relationships established between users. Architecturally, CoSMed is an I/O automaton formalized in Isabelle, exported as Scala code, and wrapped in a web application (§2).

* This is a preprint of a paper presented at ITP 2016. The final publication is available at Springer via `http://dx.doi.org/10.1007/978-3-319-43144-4_6`

For CoCon, we had proved that information only flows from the stored documents to the users in a suitably *role-triggered* and *bounded* fashion. In CoSMed's case, the "documents" of interest are friendship requests, friendship statuses, and posts by the users. The latter consist of title, text, and an optional image. The roles in CoSMed include admin, owner and friend. Modeling the restrictions on CoSMed's information flow poses additional challenges (§3), since here the roles vary dynamically. For example, assume we prove a property analogous to those for CoCon: A user U1 learns nothing about the friend-only posts posted by a user U2 *unless* U1 becomes a friend of U2. Although this property makes sense, it is too weak—given that U1 may be "friended" and "unfriended" by U2 multiple times. A stronger confidentiality property would be: U1 learns nothing about U2's friend-only posts *beyond* the updates performed *while* U1 and U2 were friends. For the verification of both CoCon and CoSMed, we have employed Bounded-Deducibility (BD) Security (§3.2), a general framework for the verification of rich information flow properties of input/output automata. BD security is parameterized by declassification *bounds* and *triggers* . While for CoCon a fixed topology of bounds and triggers was sufficient, CoSMed requires a more dynamic approach, where the bounds incorporate trigger information on a dynamic basis (§3.3). The verification proceeds by providing suitable unwinding relations, closely matching the bounds (§4).

CoSMed has been developed to fulfill the functionality and security needs of a charity organization [4]. The current version is a prototype, not yet deployed for the charity usage. Both the formalization and the running website are publicly available [5].

Notation. Given $f : A \to B$, $a : A$ and $b : B$, we write $f(a := b)$ for the function that returns b for a and otherwise acts like f. $[]$ denotes the empty list and @ denotes list concatenation. Given a list xs, we write $\mathsf{last}\ xs$ for its last element. Given a predicate P, we write $\mathsf{filter}\ P\ xs$ for the sublist of xs consisting of those elements satisfying P. Given a function f, we write $\mathsf{map}\ f\ xs$ for the list resulting from applying the function f to each element of xs. Given a record σ, field labels $l_1, \ldots, l_n$ and values $v_1, \ldots, v_n$ respecting the types of the labels, we write $\sigma(l_1 := v_1, \ldots, l_n := v_n)$ for σ with the values of the fields l_i updated to v_i. We let $l_i\ \sigma$ be the value of field l_i stored in σ.

2 System Description

In this section we describe the system functionality as formalized in Isabelle (§2.1)—we provide enough detail so that the reader can have a good grasp of the formal confidentiality properties discussed later. Then we sketch CoSMed's overall architecture (§2.2).

2.1 Isabelle Specification

Abstractly, the system can be viewed as an I/O automaton, having a state and offering some actions through which the user can affect the state and retrieve outputs. The state stores information about users, posts and the relationships between them, namely:

- user information: pending new-user requests, the current user IDs and the associated user info, the system's administrator, the user passwords;
- post information: the current post IDs and the posts associated to them, including content and visibility information;

– post-user relationships: the post owners;
– user-user relationships: the pending friend requests and the friend relationships.

All in all, the **state** is represented as an Isabelle record:

```
RECORD state =
(* User info: *)
    pendingUReqs : userID list      userReq : userID → request      userIDs : userID list
    user : userID → user      pass : userID → password      admin : userID
(* Friend info: *)
    pendingFReqs : userID → userID list      friendReq : userID → userID → request
    friendIDs : userID → userID list
(* Post info: *)
    postIDs : postID list      post : postID → post      owner : postID → userID
```

Above, the types userID, postID, password, and request are essentially strings (more precisely, datatypes with one single constructor embedding strings). Each pending request (be it for user or for friend relationship) stores a request info (of type request), which contains a message of the requester for the recipient (the system admin or a given user). The type user contains user names and information. The type post of posts contains tuples $(\mathit{title}, \mathit{txt}, \mathit{img}, \mathit{vis})$, where the title and the text are essentially strings, *img* is an (optional) image file, and $\mathit{vis} \in \{\mathsf{Friend}, \mathsf{Public}\}$ is a visibility status that can be assigned to posts: Friend means visibility to friends only, whereas Public means visibility to all users.

The **initial state** of the system is completely empty: there are empty lists of registered users, posts, etc. Users can interact with the system via six categories of **actions**: start-up, creation, deletion, update, reading and listing.

The actions take varying numbers of parameters, indicating the user involved and optionally some data to be loaded into the system. Each action's behavior is specified by two functions:

– An effect function, actually performing the action, possibly changing the state and returning an output
– An enabledness predicate (marked by the prefix "e"), checking the conditions under which the action should be allowed

When a user issues an action, the system first checks if it is enabled, in which case its effect function is applied and the output is returned to the user. If it is not enabled, then an error message is returned and the state remains unchanged.

The **start-up action**, $\mathsf{startSys} : \mathsf{state} \to \mathsf{userID} \to \mathsf{password} \to \mathsf{state}$, initializes the system with a first user, who becomes the admin:

$$\begin{aligned}&\mathsf{startSys}\ \sigma\ \mathit{uid}\ p\ \equiv\\&\quad \sigma(\mathsf{admin} := \mathit{uid},\ \ \mathsf{userIDs} := [\mathit{uid}],\ \ \mathsf{user} := (\mathsf{user}\ \sigma)(\mathit{uid} := \mathsf{emptyUser}),\\&\qquad \mathsf{pass} := (\mathsf{pass}\ \sigma)(\mathit{uid} := p))\end{aligned}$$

The start-up action is enabled only if the system has no users:

$$\mathsf{e\_startSys}\ \sigma\ \mathit{uid}\ p\ \equiv\ \mathsf{userIDs}\ \sigma = []$$

Creation actions perform registration of new items in the system. They include: placing a new user registration request; the admin approving such a request, leading to registration of a new user; a user creating a post; a user placing a friendship request for another user; a user accepting a pending friendship request, thus creating a friendship connection.

The three main kinds of items that can be created/registered in the system are users, friends and posts. Post creation can be immediately performed by any user. By contrast, user and friend registration proceed in two stages: first a request is created by the interested party, which can later be approved by the authorized party. For example, a friendship request from *uid* to uid' is first placed in the pending friendship request queue for uid'. Then, upon approval by uid', the request turns into a friendship relationship. Since friendship is symmetric, both the list of uid''s friends and that of *uid*'s friends are updated, with *uid* and uid' respectively.

There is only one **deletion action** in the system, namely friendship deletion ("unfriending" an existing friend).

Update actions allow users with proper permissions to modify content in the system: user info, post content, visibility status, etc. For example, the following action is updating, on behalf of the user *uid*, the text of a post with ID *pid* to the value *txt*.

$$\begin{array}{l}\mathsf{updateTextPost}\ \sigma\ uid\ p\ pid\ txt\ \equiv \\ \quad \sigma\ (\mathsf{post} := (\mathsf{post}\ \sigma)(pid := \mathsf{setTextPost}\ (\mathsf{post}\ \sigma\ pid)\ txt))\end{array}$$

It is enabled if both the user ID and the post ID are registered, the given password matches the one stored in the state and the user is the post's owner. Besides the text, one can also update the title and the image of a post.

Reading actions allow users to retrieve content from the system. One can read user and post info, friendship requests and status, etc. Finally, the **listing actions** allow organizing and listing content by IDs. These include the listing of: all the pending user registration requests (for the admin); all users of the system; all posts; one's friendship requests, one's own friends, and the friends of them.

Action syntax and dispatch. So far we have discussed the action behavior, consisting of effect and enabledness. In order to keep the interface homogeneous, we distinguish between an action's behavior and its *syntax*. The latter is simply the input expected by the I/O automaton. The different kinds of actions (start-up, creation, deletion, update, reading and listing) are wrapped in a single datatype through specific constructors:

$$\text{DATATYPE}\ \mathsf{act} = \mathsf{Sact\ sAct} \mid \mathsf{Cact\ cAct} \mid \mathsf{Dact\ dAct} \mid \mathsf{Uact\ uAct} \mid \mathsf{Ract\ rAct} \mid \mathsf{Lact\ lAct}$$

In turn, each kind of action forms a datatype with constructors having varying numbers of parameters, mirroring those of the action behavior functions. For example, the following datatypes gather (the syntax of) all the update and reading actions:

```
DATATYPE uAct =
  uUser userID password password name info
| uTitlePost userID password postID title
| uTextPost userID password postID text
| uImgPost userID password postID img
| uVisPost userID password postID vis
```

```
DATATYPE rAct =
  rUser userID password userID
| rNUReq userID password userID
| rNAReq userID password appID
| rAmIAdmin userID password
| rTitlePost userID password postID
| rTextPost userID password postID
| rImgPost userID password postID
| rVisPost userID password postID
| rOwnerPost userID password postID
| rFriendReqToMe userID password userID
| rFriendReqFromMe userID password userID
```

We have more reading actions than update actions. Some items, such as new-user and new-friend request info, are readable but not updatable.

The naming convention we follow is that a constructor representing the syntax of an action is named by abbreviating the name of that action. For example, the constructor $\mathsf{uTextPost}$ corresponds to the effect function $\mathsf{updateTextPost}$.

The overall **step function**, $\mathsf{step} : \mathsf{state} \to \mathsf{act} \to \mathsf{out} \times \mathsf{state}$, proceeds as follows. When given a state σ and an action a, it first pattern-matches on a to discover what kind of action it is. For example, for the update action $\mathsf{Uact}\ (\mathsf{uTextPost}\ \mathit{uid}\ p\ \mathit{pid}\ \mathit{txt})$, the corresponding enabledness predicate is called on the current state (say, σ) with the given parameters, $\mathsf{e\_updateTextPost}\ \sigma\ \mathit{uid}\ p\ \mathit{pid}\ \mathit{txt}$. If this returns False, the result is $(\mathsf{outErr}, \sigma)$, meaning that the state has not changed and an error output is produced. If it returns True, the effect function is called, $\mathsf{updateTextPost}\ \sigma\ \mathit{uid}\ p\ \mathit{pid}\ \mathit{txt}$, yielding a new state σ'. The result is then $(\mathsf{outOK}, \sigma')$, containing the new state along with an output indicating that the update was successful.

Note that start, creation, deletion and update actions change the state but do not output non-trivial data (besides outErr or outOK). By contrast, reading actions do not change the state, but they output data such as user info, post content and friendship status. Likewise, listing actions output lists of IDs and other data. The datatype out, of the overall system outputs, wraps together all these possible outputs, including outErr and outOK.

In summary, all the heterogeneous parametrized actions and outputs are wrapped in the datatypes act and out, and the step function dispatches any request to the corresponding enabledness check and effect. The end product is a single I/O automaton.

2.2 Implementation

For CoSMed's implementation, we follow the same approach as for CoCon [18, §2]. The I/O automaton formalized by the initial state $\sigma_0 : \mathsf{state}$ and the step function $\mathsf{step} : \mathsf{state} \to \mathsf{act} \to \mathsf{out} \times \mathsf{state}$ represents CoSMed's kernel—it is this kernel that we formally verify. The kernel is automatically translated to isomorphic Scala code using Isabelle's code generator [15].

Around the exported code, there is a thin layer of trusted (unverified) code. It consists of an API written with the Scalatra framework and a web application that communicates with the API. Although this architecture involves trusted code, there are reasons to believe that the confidentiality guarantees of the kernel also apply to the overall system. Indeed, the Scalatra API is a thin layer: it essentially forwards requests back and forth between the kernel and the outside world. Moreover, the web application operates by calling combinations of primitive API operations, without storing any data itself. User authentication, however, is also part of this unverified code. Of course, complementing our secure kernel with a verification that "nothing goes wrong" in the outer layer (by some language-based tools) would give us stronger guarantees.

3 Stating Confidentiality

Web-based systems for managing online resources and workflows for multiple users, such as CoCon and CoSMed, are typically programmed by distinguishing between various roles (e.g., author, PC member, reviewer for CoCon, and admin, owner, friend for CoSMed). Under specified circumstances, members with specified roles are given access to (controlled parts of) the documents.

Access control is understood and enforced *locally*, as a property of the system's *reachable states*: that a given action is only allowed if the agent has a certain role and certain circumstances hold. However, the question whether access control achieves its purpose, i.e., really restricts undesired information flow, is a *global* question whose formalization simultaneously involves *all the system's execution traces*. We wish to restrict not only what an agent can access, but also what an agent can infer, or learn.

3.1 From CoCon to CoSMed

For CoCon, we verified properties with the pattern: A user can learn nothing about a document *beyond* a certain amount of information *unless* a certain event occurs. E.g.:

- A user can learn nothing about the uploads of a paper *beyond* the last uploaded version in the submission phase *unless* that user becomes an author.
- A user can learn nothing about the updates to a paper's review *beyond* the last updated version before notification *unless* that user is a non-conflicted PC member.

The "beyond" part expresses a *bound* on the amount of disclosed information. The "unless" part indicates a *trigger* in the presence of which the bound is not guaranteed to hold. This bound-trigger tandem has inspired our notion of BD security—applicable to I/O automata and instantiatable to CoCon. But let us now analyze the desired confidentiality properties for CoSMed. For a post, we may wish to prove:

> (P1) A user can learn nothing about the updates to a post content *unless that user is the post's owner, or he becomes friends with the owner, or the post is marked as public.*

And indeed, the system can be proved to satisfy this property. But is this strong enough? Note that the trigger, emphasized in (P1) above, expresses a condition in whose presence our property stops guaranteeing anything. Therefore, since both friendship and public visibility can be freely switched on and off by the owner at any time, relying on such a strong trigger simply means giving up too easily. We should aim to prove a stronger property, describing confidentiality along several iterations of issuing and disabling the trigger. A better candidate property is the following:[5]

> (P2) A user can learn nothing about the updates to a post content *beyond* those updates that are performed *while* one of the following holds: either that user is the post's owner, or he is a friend of the owner, or the post is marked as public.

In summary, the "beyond"-"unless" bound-trigger combination we employed for CoCon will need to give way to a "beyond"-"while" scheme, where "while" refers to the periods in a system run during which observers are allowed to learn about confidential information. We will call these periods "access windows." To formalize them, we will incorporate (and iterate) the trigger inside the bound. As we show below, this is possible with the price of enriching the notion of secret to record changes to the "openness" of the access window. In turn, this leads to more complex bounds having more subtle definitions. But first let us recall BD security formally.

[5] As it will turn out, this property needs to be refined in order to hold. We'll do this in §3.3.

3.2 BD Security Recalled

We focus on the security of systems specified as I/O automata. In such an automaton, we call the inputs "actions." We write state, act, and out for the types of states, actions, and outputs, respectively, $\sigma_0 : \mathsf{state}$ for the initial state, and $\mathsf{step} : \mathsf{state} \to \mathsf{act} \to \mathsf{out} \times \mathsf{state}$ for the one-step transition function. Transitions are tuples describing an application of step:

$$\textsc{datatype}\ \mathsf{trans} = \mathsf{Trans}\ \mathsf{state}\ \mathsf{act}\ \mathsf{out}\ \mathsf{state}$$

A transition $trn = \mathsf{Trans}\ \sigma\ a\ o\ \sigma'$ is called valid if it corresponds to an application of the step function, namely $\mathsf{step}\ \sigma\ a = (o, \sigma')$. Traces are lists of transitions:

$$\textsc{type\_synonym}\ \mathsf{trace} = \mathsf{trans}\ \mathsf{list}$$

A trace $tr = [trn_0, \ldots, trn_{n-1}]$ is called valid if it starts in the initial state σ_0 and all its transitions are valid and compose well, in that, for each $i < n - 1$, the target state of trn_i coincides with the source state of trn_{i+1}. Valid traces model the runs of the system: at each moment in the lifetime of the system, a certain trace has been executed. All our formalized security definitions and properties quantify over valid traces and transitions—to ease readability, we shall omit the validity assumption, and pretend that the types trans and trace contain only valid transitions and traces.

We want to verify that there are no unintended flows of information to attackers who can observe and influence certain aspects of the system execution. Hence, we specify

1. what the capabilities of the attacker are,
2. which information is (potentially) confidential, and
3. which flows are allowed.

The first point is captured by a function O taking a trace and returning the observable part of that trace. Similarly, the second point is captured by a function S taking a trace and returning the sequence of (potential) secrets occurring in that trace. For the third point, we add a parameter B, which is a binary relation on sequences of secrets. It specifies a lower *bound* on the uncertainty of the observer about the secrets, in other words, an upper bound on these secrets' *declassification*. In this context, BD security states that O *cannot learn anything about* S *beyond* B. Formally:

> For all valid system traces tr and sequence of secrets sl' such that $\mathsf{B}\ (\mathsf{S}\ tr)\ sl'$ holds, there exists a valid system trace tr' such that $\mathsf{S}\ tr' = sl'$ and $\mathsf{O}\ tr' = \mathsf{O}\ tr$.

Thus, BD security requires that, if $\mathsf{B}\ sl\ sl'$ holds, then observers cannot distinguish the sequence of secrets sl from sl'—if sl is consistent with a given observation, then so must be sl'. Classical nondeducibility [29] corresponds to B being the total relation—the observer can then deduce *nothing* about the secrets. Smaller relations B mean that an observer may deduce some information about the secrets, but nothing beyond B—for example, if B is an equivalence relation, then the observer may deduce the equivalence class, but not the concrete secret within the equivalence class.

The original formulation of BD security in [18] includes an additional parameter T, a *declassification trigger*: The above condition is only required to hold for traces tr where T does not occur. Hence, as soon as the trigger occurs, the security property no longer offers any guarantees. This was convenient for CoCon, but for CoSMed this is too coarse-grained, as discussed in §3.1. Since, in general, an instance of BD security *with* T can be transformed into one *without*,[6] in this paper we decide to drop T and use the above trigger-free formulation of BD security.

Regarding the parameters O and S, we assume that they are defined in terms of functions on individual transitions:

- $\mathsf{isSec} : \mathsf{trans} \to \mathsf{bool}$, filtering the transitions that produce secrets
- $\mathsf{getSec} : \mathsf{trans} \to \mathsf{secret}$, producing a secret out of a transition
- $\mathsf{isObs} : \mathsf{trans} \to \mathsf{bool}$, filtering the transitions that produce observations
- $\mathsf{getObs} : \mathsf{trans} \to \mathsf{obs}$, producing an observation out of a transition

We then define $\mathsf{O} = \mathsf{map}\ \mathsf{getObs} \circ \mathsf{filter}\ \mathsf{isObs}$ and $\mathsf{S} = \mathsf{map}\ \mathsf{getSec} \circ \mathsf{filter}\ \mathsf{isSec}$. Thus, O uses filter to select the transitions in a trace that are (partially) observable according to isObs, and then maps this sequence of transitions to the sequence of their induced observations, via getObs. Similarly, S produces sequences of secrets by filtering via isSec and mapping via getSec.

All in all, BD security is parameterized by the following data:

- an I/O automaton (state, act, out, σ_0, step)
- a security model, consisting of:
  - a secrecy infrastructure (secret, isSec, getSec)
  - an observation infrastructure (obs, isObs, getObs)
  - a declassification bound B

3.3 CoSMed Confidentiality as BD Security

Next we show how to capture CoSMed's properties as BD security. We first look in depth at one property, post confidentiality, expressed informally by (P2) from §3.1.

Let us attempt to choose appropriate parameters in order to formally capture a confidentiality property in the style of (P2). The I/O automaton will of course be the one described by the state, actions and outputs from §2.1.

For the security model, we first instantiate the observation infrastructure (obs, isObs, getObs). The observers are users. Moreover, instead of assuming a single user observer, we wish to allow coalitions of an arbitrary number of users—this will provide us with stronger security guarantees. Finally, from a transition $\mathsf{Trans}\ \sigma\ a\ o\ \sigma'$ issued by a user, it is natural to allow that user to observe both their own action a and the output o.

Formally, we take the type obs of observations to be $\mathsf{act} \times \mathsf{out}$ and the observation-producing function $\mathsf{getObs} : \mathsf{trans} \to \mathsf{obs}$ to be $\mathsf{getObs}\ (\mathsf{Trans}\ \_\ a\ o\ \_) \equiv (a, o)$. We fix a set UIDs of user IDs and define the observation filter $\mathsf{isObs} : \mathsf{trans} \to \mathsf{obs}$ by

$$\mathsf{isObs}\ (\mathsf{Trans}\ \sigma\ a\ o\ \sigma') \equiv \mathsf{userOf}\ a \in \mathsf{UIDs}$$

where $\mathsf{userOf}\ a$ returns the user who performs the action. In summary, the observations are all actions issued by members of a fixed set UIDs of users together with the outputs that these actions are producing.

Let us now instantiate the secrecy infrastructure (secret, isSec, getSec). Since the property (P2) talks about the text of a post, say, identified by $\mathsf{PID} : \mathsf{postID}$, a first natural choice for secrets would be the text updates stored in PID via $\mathsf{updateTextPost}$ actions.

[6] By modifying S to produce a dedicated value as soon as T occurs, and modifying B to only consider sequences without that value.

$$\frac{textl \neq [] \to textl' \neq []}{\mathsf{B}\ (\mathsf{map}\ \mathsf{TSec}\ textl)\ (\mathsf{map}\ \mathsf{TSec}\ textl')}\ (1) \qquad \mathsf{BO}\ (\mathsf{map}\ \mathsf{TSec}\ textl)\ (\mathsf{map}\ \mathsf{TSec}\ textl)\ (2)$$

$$\frac{\mathsf{BO}\ sl\ sl' \qquad textl \neq []\ \leftrightarrow\ textl' \neq [] \qquad textl \neq [] \to \mathsf{last}\ textl = \mathsf{last}\ textl'}{\mathsf{B}\ (\mathsf{map}\ \mathsf{TSec}\ textl\ @\ \mathsf{OSec}\ \mathsf{True}\ @\ sl)\ (\mathsf{map}\ \mathsf{TSec}\ textl'\ @\ \mathsf{OSec}\ \mathsf{True}\ @\ sl')}\ (3)$$

$$\frac{\mathsf{B}\ sl\ sl'}{\mathsf{BO}\ (\mathsf{map}\ \mathsf{TSec}\ textl\ @\ \mathsf{OSec}\ \mathsf{False}\ @\ sl)\ (\mathsf{map}\ \mathsf{TSec}\ textl\ @\ \mathsf{OSec}\ \mathsf{False}\ @\ sl')}\ (4)$$

Fig. 1: The bound for post text confidentiality

That is, we could have the filter isSec a hold just in case a is such a (successfully performed) action, say, updateTextPost σ uid p pid txt, and have the secret-producing function getSec a return the updated secret, here txt. But later, when we state the bound, how would we distinguish updates that should not be learned from updates that are OK to be learned because they happen while the access is legitimate for the observers—e.g., while a user in UIDs is the owner's friend? We shall refer to the portions of the trace when the observer access is legitimate as *open access windows*, and refer to the others as *closed access windows*. The bound clearly needs to distinguish these. Indeed, it states that nothing should be learned beyond the updates that occurred during open access windows.

To enable this distinction, we enrich the notion of secret to include not only the post text updates, but also marks for the shift between closed and open access windows. To this end, we define the state predicate open to express that PID is registered and one of the users in UIDs is entitled to access the text of PID—namely, is the owner or a friend of the owner, or the post is public.

$$\begin{aligned}\mathsf{open}\ \sigma \equiv\ & \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge \\ & \exists uid \in \mathsf{UIDs}.\ uid \in \mathsf{userIDs}\ \sigma\ \wedge \\ & \quad (uid = \mathsf{owner}\ \sigma\ pid\ \vee\ uid \in \mathsf{friendIDs}\ \sigma\ (\mathsf{owner}\ \sigma\ pid)\ \vee \\ & \quad\ \mathsf{visPost}\ (\mathsf{post}\ \sigma\ \mathsf{PID}) = \mathsf{Public})\end{aligned}$$

Now, the secret selector isSec : trans $\to$ bool will record both successful post-text updates and the changes in the truth value of open for the state of the transition:

$$\begin{aligned}\mathsf{isSec}\ (\mathsf{Trans}\ \_\ (\mathsf{Uact}\ (\mathsf{uTextPost}\ pid\ \_\ \_\ txt))\ o\ \_) &\equiv pid = \mathsf{PID}\ \wedge\ o = \mathsf{outOK} \\ \mathsf{isSec}\ (\mathsf{Trans}\ \sigma\ \_\ \_\ \sigma') &\equiv \mathsf{open}\ \sigma \neq \mathsf{open}\ \sigma'\end{aligned}$$

In consonance with the filter, the type of secrets will have two constructors

$$\textsc{datatype}\ \mathsf{secret} = \mathsf{TSec}\ \mathsf{text} \mid \mathsf{OSec}\ \mathsf{bool}$$

and the secret-producing function getSec : trans $\to$ secret will retrieve either the updated text or the updated openness status:

$$\begin{aligned}\mathsf{getSec}\ (\mathsf{Trans}\ \_\ (\mathsf{Uact}\ (\mathsf{uTitlePost}\ \_\ \_\ \_\ txt))\ \_\ \_) &\equiv \mathsf{TSec}\ txt \\ \mathsf{getSec}\ (\mathsf{Trans}\ \_\ \_\ \_\ \sigma') &\equiv \mathsf{OSec}\ (\mathsf{open}\ \sigma')\end{aligned}$$

In order to formalize the desired bound B, we first note that all sequences of secrets produced from system traces consist of:

– a (possibly empty) block of text updates TSec $txt_1^1, \ldots,$ TSec $txt_{n_1}^1$
– possibly followed by a shift to an open access window, OSec True

- possibly followed by another block of text updates $\mathsf{TSec}\ txt^2_1, \ldots, \mathsf{TSec}\ txt^2_{n_2}$
- possibly followed by a shift to a closed access window, $\mathsf{OSec\ False}$
- … and so on …

We wish to state that, given any such sequence of secrets *sl* (say, produced from a system trace *tr*), any other sequence sl' that coincides with *sl* on the open access windows (while being allowed to be *arbitrary* on the closed access windows) is equally possible as far as the observer is concerned—in that there exists a trace tr' yielding the same observations as *tr* and producing the secrets sl'.

The purpose of B is to capture this relationship between *sl* and sl', of coincidence on open access windows. But which part of a sequence of secrets *sl* represents such a window? It should of course include all the text updates that take place during the time when one of the observers has legitimate access to the post—namely, all blocks of *sl* that are immediately preceded by an OSec True secret.

But there are other secrets in the sequence that properly belong to this window: the last updated text before the access window is open, that is, the secret $\mathsf{TSec}\ txt^k_{n_k}$ occurring *immediately before* each occurrence of OSec True. For example, when the post becomes public, a user can see not only upcoming updates to its text, but also the current text, i.e., the last update before the visibility upgrade.

The definition of B reflects the above discussion, using an auxiliary predicate BO to cover the case when the window is open. The predicates are defined mutually inductively as in Figure 1.

Clause (1), the base case for B, describes the situation where the original system trace has made no shift from the original closed access window. Here, the produced sequence of secrets *sl* consists of text updates only, i.e., $sl = \mathsf{map\ TSec}\ textl$. It is indistinguishable from any alternative sequence of updates $sl' = \mathsf{map\ TSec}\ textl'$, save for the corner case where an observer can learn that *sl* is empty by inferring that the post does not exist, e.g. because the system has not been started yet, or because no users other than the observers exist who could have created the post. Such harmless knowledge is factored in by asking that sl' (i.e., $textl'$) be empty whenever *sl* (i.e., *textl*) is.

Clause (2), the base case for BO, handles sequences of secrets produced during open access windows. Since here information is entirely exposed, the corresponding sequence of secrets from the alternative trace has to be identical to the original.

Clause (3), the inductive case for B, handles sequences of secrets $\mathsf{map\ TSec}\ textl$ produced during closed access windows. The difference from clause (1) is that here we know that there will eventually be a shift to a closed access window—this is marked by the occurrences of OSec True in the conclusion, followed by a remaining sequence *sl*. As previously discussed, the only constraint on the sequence of secrets produced by the alternative trace, $\mathsf{map\ TSec}\ textl'$, is that it ends in the same secret—hence the condition that the sequences be empty at the same time and have the same last element. Finally, clause (4), the inductive case for BO, handles the secrets produced during open access window on a trace known to eventually move to an open access window

With all the parameters in place, we have a formalization of post text confidentiality: the BD-security instance for these parameters. However, we saw that the legitimate exposure of the posts is wider than initially suggested, hence (P2) is bogus as currently formulated. Namely, we need to factor in the last updates *before* open access windows in addition to the updates performed *during* open access windows. If we also factor in the generalization from a single user to a coalition of users, we obtain:

> (P3) A coalition of users can learn nothing about the updates to a post content beyond those updates that are performed while one of the following holds *or the last update before one of the following starts to hold*:
> – a user in the coalition is the post's owner or a friend of the post's owner, or
> – there is at least one user in the coalition and the post is marked as public.

3.4 More Confidentiality Properties

So far, we have discussed confidentiality for post content (i.e., text). However, a post also has a title and an image. For these, we want to verify the same confidentiality properties as in §3.3, only substituting text content by titles and images, respectively. In addition to posts, another type of information with confidentiality ramifications is that about friendship between users: who is friends with whom, and who has requested friendship with whom. We consider the confidentiality of the friendship information of two arbitrary but fixed users UID1 and UID2 who are *not* in the coalition of observers:

> (P4) A coalition of users UIDs can learn nothing about the updates to the friendship status between two users UID1 and UID2 beyond those updates that are performed while a member of the coalition is friends with UID1 or UID2, or the last update before there is a member of the coalition who becomes friends with UID1 or UID2.

> (P5) A coalition of users UIDs can learn nothing about the friendship requests between two users UID1 and UID2 beyond the existence of a request before each successful friendship establishment.

Formally, we declare open access window to friendship information when either an observer is friends with UID1 or UID2 (since the listing of friends of friends is allowed), or the two users have not been created yet (since observers know statically that there is no friendship if the users do not exist yet).

$$\begin{aligned}\mathsf{open}_F\ \sigma \equiv\ & (\exists \mathit{uid} \in \mathsf{UIDs}.\ \mathit{uid} \in \mathsf{friendIDs}\ \sigma\ \mathsf{UID1} \vee \mathit{uid} \in \mathsf{friendIDs}\ \sigma\ \mathsf{UID2}) \\ & \vee \mathsf{UID1} \notin \mathsf{userIDs}\ \sigma\ \vee\ \mathsf{UID2} \notin \mathsf{userIDs}\ \sigma\end{aligned}$$

The relevant transitions for the secrecy infrastructure are the creation of users and the creation and deletion of friends or friend requests. The creation and deletion of friendship between UID1 and UID2 produces an FSec True or FSec False secret, respectively. In the case of openness changes, OSec is produced just as for the post confidentiality. Moreover, for (P5), we let the creation of a friendship request between UID1 and UID2 produce FRSec *uid txt* secrets, where *uid* indicates the user that has placed the request, and *txt* is the request message.

The main inductive definition of the two phases of the declassification bounds for friendship (P4) is given in Figure 2, where *fs* ranges over friendship statuses, i.e., Booleans. Note that it follows the same "while"-"last update before" scheme as Figure 1 for the post confidentiality, but with FSec instead of TSec. The overall bound is

$$\mathsf{BO}_F\ (\mathsf{map\ FSec}\,\mathit{fs})\ (\mathsf{map\ FSec}\,\mathit{fs})\ (1) \qquad \mathsf{BC}_F\ (\mathsf{map\ FSec}\,\mathit{fs})\ (\mathsf{map\ FSec}\,\mathit{fs}')\ (2)$$

$$\frac{\mathsf{BO}_F\ \mathit{sl}\ \mathit{sl}' \qquad \mathit{fs} \neq [] \leftrightarrow \mathit{fs}' \neq [] \qquad \mathit{fs} \neq [] \rightarrow \mathsf{last}\,\mathit{fs} = \mathsf{last}\,\mathit{fs}'}{\mathsf{BC}_F\ (\mathsf{map\ FSec}\,\mathit{fs}\ @\ \mathsf{OSec\ True}\ @\ \mathit{sl})\ (\mathsf{map\ FSec}\,\mathit{fs}'\ @\ \mathsf{OSec\ True}\ @\ \mathit{sl}')}\ (3)$$

$$\frac{\mathsf{BC}_F\ \mathit{sl}\ \mathit{sl}'}{\mathsf{BO}_F\ (\mathsf{map\ FSec}\,\mathit{fs}\ @\ \mathsf{OSec\ False}\ @\ \mathit{sl})\ (\mathsf{map\ FSec}\,\mathit{fs}\ @\ \mathsf{OSec\ False}\ @\ \mathit{sl}')}\ (4)$$

Fig. 2: The bound on friendship status secrets

then defined as $\mathsf{BO}_F\ \mathit{sl}\ \mathit{sl}'$ (since we start in the open phase where UID1 and UID2 do not exist yet) plus a predicate on the values that captures the static knowledge of the observers: that the FSec's form an *alternating* sequence of "friending" and "unfriending."

For (P5), we additionally require that at least one FRSec and at most two FRSec secrets from different users have to occur before each FSec True secret. Beyond that, we require nothing about the request values. Hence, the bound for friendship requests states that observers learn nothing about the requests between UID1 and UID2 beyond the existence of a request before each successful friendship establishment. In particular, they learn nothing about the "orientation" of the requests (i.e., which of the two involved users has placed a given request) and the contents of the request messages.

4 Verifying Confidentiality

Next we recall the unwinding proof technique for BD security (§4.1) and show how we have employed it for CoSMed (§4.2).

4.1 BD Unwinding Recalled

In [18], we have presented a verification technique for BD security inspired by Goguen and Meseguer's *unwinding* technique for noninterference [13]. Classical noninterference requires that it must be possible to purge all secret transitions from a trace, without affecting the outputs of observable actions. The unwinding technique uses an equivalence relation on states, relating states with each other that are supposed to be indistinguishable for the observer. The proof obligations are that 1. equivalent states produce equal outputs for observable actions, 2. performing an observable action in two equivalent states again results in two equivalent states, and 3. the successor state of a secret transition is equivalent to the source state. This guarantees that purging secret transitions preserves observations. The proof proceeds via an induction on the original trace.

For BD security, the situation is different. Instead of purging all secret transitions, we have to *construct* a *different* trace tr' that produces the same observations as the original trace tr, but produces precisely a given sequence of secrets sl' for which $\mathsf{B}\ (\mathsf{S}\ \mathit{tr})\ \mathit{sl}'$ holds.

The idea is to construct tr' incrementally, in synchronization with tr, but "keeping an eye" on sl' as well. The unwinding relation [18, §5.1] is therefore not a relation on states, but a relation on $(\mathsf{state} \times \mathsf{secret\ list})$, or equivalently, a set of tuples $(\sigma, \mathit{sl}, \sigma', \mathit{sl}')$. Each of these tuples represents a possible configuration of the unwinding "synchronization

$$\begin{tikzcd}
 & & \Delta_0 \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
\Delta_1 & \Delta_4 \arrow[l] \arrow[rr, leftrightarrow] & & \Delta_3 \arrow[r] & \Delta_2
\end{tikzcd}$$

Fig. 3: Graph of unwinding relations

$$\begin{tikzcd}
 & & \Delta_0 \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
\Delta_1 & \Delta_4 \arrow[l] \arrow[rr] \arrow[drr, leftrightarrow] & & \Delta_3^1 \arrow[d] & \Delta_2 \\
 & & & \Delta_3^2 \arrow[ur] &
\end{tikzcd}$$

Fig. 4: Refined graph

game": σ and *sl* represent the current state reached by a potential original trace and the secrets that are still to be produced by it; and similarly for σ' and sl' w.r.t. the alternative trace.

To keep proof size manageable, the framework supports the decomposition of Δ into smaller unwinding relations $\Delta_0, \ldots, \Delta_n$ focusing on different phases of the synchronization game. The unwinding conditions require that, from any such configuration for which one of the relations hold, say, $\Delta_i\ \sigma\ sl\ \sigma'\ sl'$, the alternative trace can "stay in the game" by choosing to (1) either act independently or (2) wait for the original trace to act and then choose how to react to it: (1.a) either ignore that transition or (1.b) match it with an own transition. For the resulting configuration, one of the unwinding relations has to hold again. More precisely, the allowed steps in the synchronization game are the following:

INDEPENDENT ACTION: *There exists* a transition $trn' = \mathsf{Trans}\ \sigma'\ \_\ \_\ \sigma'_1$ that is unobservable (i.e., $\neg$ $\mathsf{isObs}\ trn'$), produces the first secret in sl', and leads to a configuration that is again in one of the relations, $\Delta_j\ \sigma\ sl\ \sigma'_1\ sl'_1$ for $j \in \{1, \ldots, n\}$

REACTION: *For all* possible transitions $trn = \mathsf{Trans}\ \sigma\ \_\ \_\ \sigma_1$ one of the following holds:

- **IGNORE:** *trn* is unobservable and again leads to a related configuration $\Delta_k\ \sigma_1\ sl_1\ \sigma'\ sl'$ for $k \in \{1, \ldots, n\}$
- **MATCH:** There exists an observationally equivalent transition $trn' = \mathsf{Trans}\ \sigma'\ \_\ \_\ \sigma'_1$ (i.e., $\mathsf{isObs}\ trn \leftrightarrow \mathsf{isObs}\ trn'$ and $\mathsf{isObs}\ trn \rightarrow \mathsf{getObs}\ trn = \mathsf{getObs}\ trn'$) that together with *trn* leads to a related configuration $\Delta_l\ \sigma_1\ sl_1\ \sigma'_1\ sl'_1$ for $l \in \{1, \ldots, n\}$

If one of these conditions is satisfied for any configuration, then the unwinding relations can be seen as forming a graph: For each i, Δ_i is connected to all the relations into which it "unwinds," i.e., the relations Δ_j, Δ_k or Δ_l appearing in the above conditions. We use these conditions in the inductive step of the proof of the soundness theorem below.

Finally, we require that the initial relation Δ_0 is a proper generalization of the bound for the initial state, $\forall sl\ sl'.\ \mathsf{B}\ sl\ sl' \rightarrow \Delta_0\ \sigma_0\ sl\ \sigma_0\ sl'$. This corresponds to initializing the game with a configuration that loads any two sequences of secrets satisfying the bound.

Theorem 1 [18] If $\Delta_0, \ldots, \Delta_n$ form a graph of unwinding relations, and $B\ sl\ sl'$ implies $\Delta_0\ \sigma_0\ sl\ \sigma_0\ sl'$ for all *sl* and sl', then (the given instance of) BD security holds.

4.2 Unwinding Relations for CoSMed

In a graph $\Delta_0, \ldots, \Delta_n$ of unwinding relations, Δ_0 generalizes the bound B. In turn, Δ_0 may unwind into other relations, and in general any relation in the graph may unwind into its successors. Hence, we can think of Δ_0 as "taking over the bound," and of all the relations as "maintaining the bound" together with state information. It is therefore natural to design the graph to reflect the definition of B.

We have applied this strategy to all our unwinding proofs. The graph in Figure 3 shows the unwindings of the post-text confidentiality property (P3). In addition to the initial relation Δ_0, there are 4 relations Δ_1–Δ_4 with Δ_i corresponding to clause (i) for the definition of B from Fig. 1. The edges correspond to the possible causalities between the clauses. For example, if $\mathsf{B}\ sl\ sl'$ has been obtained applying clause (3), then, due to the occurrence of BO in the assumptions, we know the previous clauses must have been either (2) or (4)—hence the edges from Δ_3 to Δ_2 and Δ_4. Each Δ_i also provides a relationship between the states σ and σ' that fits the situation. Since we deal with repeated opening and closing of the access window, we naturally require:

- that $\sigma = \sigma'$ when the window is open
- that $\sigma =_{\mathsf{PID}} \sigma'$, i.e., σ and σ' are equal everywhere save for the value of PID's text, when the window is closed

Indeed, only when the window is open the observer would have the power to distinguish different values for PID's text; hence, when the window is closed the secrets are allowed to diverge. Open windows are maintained by the clauses for BO, (2) and (4), and hence by Δ_2 and Δ_4. Closed windows are maintained by the clauses for B, (1) and (3), with the following exception for clause (3): When the open-window marker OSec True is reached, the PID text updates would have synchronized ($\mathsf{last}\ textl = \mathsf{last}\ textl'$), and therefore the relaxed equality $=_{\mathsf{PID}}$ between states would have shrunk to plain equality—this is crucial for the switch between open and closed windows.

To address this exception, we refine our graph as in Fig. 4, distinguishing between clause (3) applied to nonempty update prefixes where we only need $\sigma =_{\mathsf{PID}} \sigma'$, covered by Δ_3^1, and clause (3) with empty update prefixes where we need $\sigma = \sigma'$, covered by Δ_3^2. Fig. 5 gives the formal definitions of the relations. Δ_0 covers the prehistory of PID—from before it was created. In Δ_1–Δ_4, the conditions on sl and sl' essentially incorporate the inversion rules corresponding to clauses (1)-(4) in B's definition, while the conditions on σ and σ' reflect the access conditions, as discussed.

Proposition 2 The relations in Fig. 5 form a graph of unwinding relations, and therefore (by Thm. 1) the post-text confidentiality property (P3) holds.

For unwinding the friendship confidentiality properties, we proceed analogously. We define unwinding relations, corresponding to the different clauses in Figure 2, and prove that they unwind into each other and that $\mathsf{B}\ sl\ sl'$ implies $\Delta_0\ \sigma_0\ sl\ \sigma_0\ sl'$. In the open phase, we require that the two states are equal up to pending friendship requests between UID1 and UID2. In the closed phase, the two states may additionally differ on the friendship *status* of UID1 and UID2. Again, we need to converge back to the same friendship status when changing from the closed into the open phase. Hence, we maintain the invariant in the closed phase that if an OSec True secret follows later in the sequence of secrets, then the last updates before OSec True must be equal, analogous to Δ_3^1 for post texts, and the friendship status must be equal in the two states immediately before an OSec True secret, analogous to Δ_3^2 for post texts.

$$\begin{aligned}
\Delta_0\ \sigma\ sl\ \sigma'\ sl' &\equiv \neg\ \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma = \sigma' \\
\Delta_1\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma =_{\mathsf{PID}} \sigma'\ \wedge\ \neg\ \mathsf{open}\ \sigma\ \wedge \\
&\quad\ \exists textl\ textl'.\ sl = \mathsf{map\ TSec}\ textl\ \wedge\ sl' = \mathsf{map\ TSec}\ textl'\ \wedge \\
&\qquad textl = []\ \rightarrow\ textl' = [] \\
\Delta_2\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma = \sigma'\ \wedge\ \mathsf{open}\ \sigma\ \wedge \\
&\quad\ \exists textl.\ sl = \mathsf{map\ TSec}\ textl\ \wedge\ sl' = \mathsf{map\ TSec}\ textl \\
\Delta_3^1\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma =_{\mathsf{PID}} \sigma'\ \wedge\ \neg\ \mathsf{open}\ \sigma\ \wedge \\
&\quad\ \exists textl\ textl'\ sl_1\ sl_1'.\ sl = \mathsf{map\ TSec}\ textl\ @\ \mathsf{OSec\ True}\ \#\ sl_1\ \wedge \\
&\qquad sl' = \mathsf{map\ TSec}\ textl'\ @\ \mathsf{OSec\ True}\ \#\ sl_1'\ \wedge \\
&\qquad \mathsf{BO}\ sl_1\ sl_1'\ \wedge\ textl \neq []\ \wedge\ textl' \neq []\ \wedge\ \mathsf{last}\ textl = \mathsf{last}\ textl' \\
\Delta_3^2\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma = \sigma'\ \wedge\ \neg\ \mathsf{open}\ \sigma\ \wedge \\
&\quad\ \exists sl_1\ sl_1'.\ sl = \mathsf{OSec\ True}\ \#\ sl_1\ \wedge\ sl' = \mathsf{OSec\ True}\ \#\ sl_1'\ \wedge\ \mathsf{BO}\ sl_1\ sl_1' \\
\Delta_4\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma = \sigma'\ \wedge\ \mathsf{open}\ \sigma\ \wedge \\
&\quad\ \exists textl\ sl_1\ sl_1'.\ sl = \mathsf{map\ TSec}\ textl\ @\ \mathsf{OSec\ False}\ \#\ sl_1\ \wedge \\
&\qquad sl' = \mathsf{map\ TSec}\ textl'\ @\ \mathsf{OSec\ False}\ \#\ sl_1'\ \wedge\ \mathsf{B}\ sl_1\ sl_1'
\end{aligned}$$

Fig. 5: The unwinding relations for post-text confidentiality

5 Verification Summary

The whole formalization consists of around 9700 Isabelle lines of code (LOC). The (reusable) BD security framework takes 1800 LOC. CosMeD's kernel implementation represents 700 LOC. Specifying and verifying the confidentiality properties for CoSMeD represents the bulk, 6500 LOC. Some additional 200 LOC are dedicated to various safety properties to support the confidentiality proofs—e.g., that two users cannot be friends if there are pending friendship requests between them. Unlike the confidentiality proofs, which required explicit construction of unwindings, safety proofs were performed automatically (by reachable-state induction).

Yet another kind of properties were formulated in response to the following question: We have shown that a user can only learn about updates to posts that were performed during times of public visibility or friendship, and about the last updates before these time intervals. But how can we be sure that the public visibility status or the friendship status cannot be forged? We have proved that these statuses can indeed only occur by the standard protocols. These properties (taking 500 LOC), complement our proved confidentiality by a form of accountability: they show that certain statuses can only be forged by identity theft.

6 Related Work

Proof assistants are today's choice for *precise* and *holistic* formal verification of hardware and software systems. Already legendary verification works are the AMD microprocessor floating-point operations [24], the CompCert C compiler [21] and the seL4 operating system kernel [19]. More recent developments include a range of microprocessors [16], Java and ML compilers [20, 22], and a model checker [11].

Major "holistic" verification case studies in the area of information flow security are rather scarce, perhaps due to the more complex nature of the involved properties compared to traditional safety and liveness [23]. They include a hardware architecture with information-flow primitives [10] and a separation kernel [9], and noninterference for seL4 [25]. A substantial contribution to web client security is the Quark verified browser [17]. We hope that our line of work, putting CoCon and CoSMed in the spotlight but tuning a general verification framework backstage, will contribute a firm methodology for the holistic verification of server-side confidentiality.

Policy languages for social media platforms have been proposed in the context of Relationship-based Access Control [12], or using epistemic logic [28]. These approaches focus on specifying policies for granting or denying access to data based on the social graph, e.g. friendship relations. While our system implementation does make use of access control, our guarantees go beyond access control to information flow control. A formal connection between these policy languages and BD security would be interesting future work.

Finally, there are quite a few programming languages and tools aimed at supporting information-flow secure programming [2, 3, 7, 30], as well as information-flow tracking tools for the client side of web applications [6, 8, 14]. We foresee a future where such tools will cooperate with proof assistants to offer light-weight guarantees for free and stronger guarantees (like the ones we proved in this paper) on a need basis.

Conclusion CoSMed is the first social media platform with verified confidentiality guarantees. Its verification is based on BD security, a framework for information-flow security formalized in Isabelle. CoSMed's specific confidentiality needs require a dynamic topology of declassification bounds and triggers.

Acknowledgements We are indebted to the reviewers for useful comments and suggestions. We gratefully acknowledge support:

- from Innovate UK through the Knowledge Transfer Partnership 010041 between Caritas Anchor House and Middlesex University: "The Global Noticeboard (GNB): a verified social media platform with a charitable, humanitarian purpose",
- from EPSRC through grant "VOWS – Verification of Web-based Systems" (EP/N019547/1),
- from DFG through grants "Modelling and Refinement of Security Requirements on Data and Processes" (Hu 737/5-2) and "Security Type Systems and Deduction" (Ni 491/13-3) in the priority program "RS^3 – Reliably Secure Software Systems" (SPP 1496).

References

1. OWASP top ten project. `www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013`.
2. Jif: Java + information flow, 2014. `http://www.cs.cornell.edu/jif`.
3. SPARK, 2014. `http://www.spark-2014.org`.
4. Caritas Anchor House. `http://caritasanchorhouse.org.uk/`, 2016.
5. The CoSMed website. `https://cosmed.globalnoticeboard.com`, 2016. Anonymous access possible with username "demo" and password "demo".
6. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, pp. 159–178, 2014.
7. A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pp. 153–165, 2015.
8. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, pp. 50–62, 2009.

9. M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz. Formal verification of information flow security for a simple ARM-based separation kernel. In *CCS*, pp. 223–234, 2013.
10. A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL*, pp. 165–178, 2014.
11. J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus. A fully verified executable LTL model checker. In *CAV*, pp. 463–478, 2013.
12. P. W. L. Fong, M. M. Anwar, and Z. Zhao. A privacy preservation model for Facebook-style social network systems. In *ESORICS*, pp. 303–320, 2009.
13. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pp. 75–87, 1984.
14. W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS*, pp. 748–759, 2012.
15. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117, 2010.
16. D. S. Hardin, E. W. Smith, and W. D. Young. A robust machine code proof framework for highly secure applications. In P. Manolios and M. Wilding, eds., *ACL2*, pp. 11–20, 2006.
17. D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, pp. 113–128, 2012.
18. S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *CAV*, pp. 167–183, 2014.
19. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.
20. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL*, pp. 179–192, 2014.
21. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
22. A. Lochbihler. Java and the Java memory model—A unified, machine-checked formalisation. In *ESOP*, pp. 497–517, 2012.
23. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 605–607. 2011.
24. J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the amd5$_{\text{k}}$86$^{\text{tm}}$ floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998.
25. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. In *Security and Privacy*, pp. 415–429, 2013.
26. T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.
27. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
28. R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *SEFM*, pp. 378–392, 2014.
29. D. Sutherland. A model of information. In *9th National Security Conf.*, pp. 175–183, 1986.
30. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pp. 85–96, 2012.