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 last $xs$ for its last element. Given a predicate $P$, we write filter $P$ $xs$ for the sublist of $xs$ consisting of those elements satisfying $P$. Given a function $f$, we write map $f$ $xs$ for the list resulting from applying the function $f$ to each element of $xs$. Given a record $\sigma$, 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 $\sigma$ 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 $\sigma$.

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:

```isabelle
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 (title, txt, img, vis), where the title and the text are essentially strings, img is an (optional) image file, and vis ∈ {Friend, 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, startSys : state → userID → password → state, initializes the system with a first user, who becomes the admin:

```isabelle
startSys σ uid p ≡
  σ(admin := uid, userIDs := [uid], user := (user σ)(uid := emptyUser),
  pass := (pass σ)(uid := p))
```

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

```isabelle
e_startSys σ uid p ≡ userIDs σ = []
```

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 (“un-friending” 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 $txt$.

\[
\text{updateTextPost} \sigma uid p pid txt \equiv \\
\sigma (\text{post} := (\text{post} \sigma))(\text{setTextPost} (\text{post} \sigma pid) txt))
\]

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:

\[
\text{DATATYPE} \ act = Sact sAct | Cact cAct | Dact dAct | Uact uAct | Ract rAct | Lact lAct
\]

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:

\[
\text{DATATYPE} \ uAct = \\
\text{uUser userID password password name info} | \text{uTitlePost userID password postID title} | \text{uTextPost userID password postID text} | \text{uImgPost userID password postID img} | \text{uVisPost userID password postID vis} \\
\text{DATATYPE} \ rAct = \\
\text{rUser userID password userID} | \text{rNUReq userID password userID} | \text{rNAReq userID password appID} | \text{rAdmin userID password} | \text{rTitlePost userID password postID} | \text{rTextPost userID password postID} | \text{rImgPost userID password postID} | \text{rVisPost userID password postID} | \text{rOwnerPost userID password postID} | \text{rFriendReqToMe userID password userID} | \text{rFriendReqFromMe userID password userID}
\]
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 uTextPost corresponds to the effect function updateTextPost.

The overall step function, \( \text{step} : \text{state} \rightarrow \text{act} \rightarrow \text{out} \times \text{state} \), proceeds as follows. When given a state \( \sigma \) and an action \( a \), it first pattern-matches on \( a \) to discover what kind of action it is. For example, for the update action Uact (uTextPost uid pid txt), the corresponding enabledness predicate is called on the current state (say, \( \sigma \)) with the given parameters, \( \text{e_updateTextPost} \); if this returns \( \text{False} \), the result is \( (\text{outErr}, \sigma) \), meaning that the state has not changed and an error output is produced. If it returns \( \text{True} \), the effect function is called, updateTextPost \( \sigma \) uid pid txt, yielding a new state \( \sigma' \). The result is then \( (\text{outOK}, \sigma') \), 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 \( \text{outErr} \) or \( \text{outOK} \)). 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 \( \text{out} \), of the overall system outputs, wraps together all these possible outputs, including \( \text{outErr} \) and \( \text{outOK} \).

In summary, all the heterogeneous parametrized actions and outputs are wrapped in the datatypes \( \text{act} \) and \( \text{out} \), 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 \( \sigma_0 : \text{state} \) and the step function \( \text{step} : \text{state} \rightarrow \text{act} \rightarrow \text{out} \times \text{state} \) 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 var-
ious 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 $\text{state}$, $\text{act}$, and $\text{out}$ for the types of states, actions, and outputs, respectively, $\sigma_0 : \text{state}$ for the initial state, and $\text{step} : \text{state} \rightarrow \text{act} \rightarrow \text{out} \times \text{state}$ for the one-step transition function. Transitions are tuples describing an application of $\text{step}$:

$$\text{DATATYPE trans} = \text{Trans state act out state}$$

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

$$\text{TYPE SYNONYM trace} = \text{trans list}$$

A trace $\text{tr} = [\text{trn}_0, \ldots, \text{trn}_{n-1}]$ is called valid if it starts in the initial state $\sigma_0$ and all its transitions are valid and compose well, in that, for each $i < n - 1$, the target state of $\text{trn}_i$ coincides with the source state of $\text{trn}_{i+1}$. 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 $\text{trans}$ and $\text{trace}$ 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 $\text{tr}$ and sequence of secrets $s \ s'$ such that $B (S \text{tr}) s'$ holds, there exists a valid system trace $\text{tr}'$ such that $S \text{tr}' = s'$ and $O \text{tr}' = O \text{tr}$.

Thus, BD security requires that, if $B s \ s'$ holds, then observers cannot distinguish the sequence of secrets $s \ s'$ from $s'$—if $s$ is consistent with a given observation, then so must be $s'$. 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 $\text{tr}$ 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:

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

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

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

- an I/O automaton (state, act, out, $\sigma_0$, step)
- a security model, consisting of:
  - a secrecy infrastructure (secret, $\text{isSec}$, $\text{getSec}$)
  - an observation infrastructure (obs, $\text{isObs}$, $\text{getObs}$)
  - 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, $\text{isObs}$, $\text{getObs}$). 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 $\text{Trans} \sigma a o \sigma'$ 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 $\text{act} \times \text{out}$ and the observation-producing function $\text{getObs} : \text{trans} \to \text{obs}$ to be $\text{getObs} (\text{Trans} _a o _) \equiv (a, o)$. We fix a set UIDs of user IDs and define the observation filter $\text{isObs} : \text{trans} \to \text{obs}$ by

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

where 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, $\text{isSec}$, $\text{getSec}$). Since the property (P2) talks about the text of a post, say, identified by PID : postID, a first natural choice for secrets would be the text updates stored in PID via 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.
\[\text{textl} \neq [] \rightarrow \text{textl}' \neq []\]
\[\text{B (map TSec textl) (map TSec textl')}\]
\[\text{BO (map TSec textl) (map TSec textl')}\]
\[\text{BO sl s l'} \text{ textl} \neq [] \leftrightarrow \text{textl}' \neq [] \rightarrow \text{textl} = \text{last textl}'\]
\[\text{B (map TSec textl @ OSec True @ sl) (map TSec textl' @ OSec True @ s l')}\]
\[\text{B sl s l'} \text{ BO (map TSec textl @ OSec False @ sl) (map TSec textl' @ OSec False @ s l')}\]

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 \(\sigma\; uid\; p\; pid\; txt\), 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.

\[
\text{open } \sigma \equiv \text{PID} \in \text{postIDs} \sigma \land \exists \text{uid} \in \text{UIDs}. \text{uid} \in \text{userIDs} \sigma \land (\text{uid} = \text{owner} \sigma \text{pid} \lor \text{uid} \in \text{friendIDs} \sigma \land (\text{owner} \sigma \text{pid}) \lor \text{visPost}(\text{post} \sigma \text{PID}) = \text{Public})
\]

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

\[
\text{isSec} (\text{Trans } _\_ (\text{Uact} (\text{uTextPost pid } _ _ _ \text{txt}) ) _ _ ) \equiv \text{pid} = \text{PID} \land o = \text{outOK}
\]

\[
\text{isSec} (\text{Trans } \sigma _ _ _ \sigma') \equiv \text{open } \sigma \neq \text{open } \sigma'
\]

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

\[
\text{DATATYPE secret = TSec text | OSec bool}
\]

and the secret-producing function getSec : \text{trans} \rightarrow \text{secret} will retrieve either the updated text or the updated openness status:

\[
\text{getSec} (\text{Trans } _\_ (\text{Uact} (\text{uTitlePost } _ _ _ \text{txt}) ) _ _ ) \equiv \text{TSec txt}
\]

\[
\text{getSec} (\text{Trans } _ \_ _ \sigma') \equiv \text{OSec (open } \sigma')
\]

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 \(\text{TSec txt}_1, \ldots, \text{TSec txt}_n\)
- possibly followed by a shift to an open access window, \text{OSec True}
– possibly followed by another block of text updates $TSec \, ttxt_1^2, \ldots, TSec \, ttxt_n^2$
– possibly followed by a shift to a closed access window, $OSec \, False$
– … 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 $TSec \, txt_{nk}^k$ 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., $sl = \text{map } TSec \, txtl$. It is indistinguishable from any alternative sequence of updates $sl' = \text{map } TSec \, txtl'$, 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., $txtl'$) be empty whenever $sl$ (i.e., $txtl$) 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 $\text{map } TSec \, txtl$ 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, $\text{map } TSec \, txtl'$, 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).

\[
\text{open}_F \equiv (\exists u\text{id} \in \text{UIDs}. u\text{id} \in \text{friendIDs} \sigma \land \text{UID1} \lor u\text{id} \in \text{friendIDs} \sigma \land \text{UID2}) \\
\lor \text{UID1} \notin \text{userIDs} \sigma \land \text{UID2} \notin \text{userIDs} \sigma
\]

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
then defined as $\text{BO}_F (\text{map FSec } fs)$ (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 “friend” and “unfriend.”

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 $B (S tr) sl'$ 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 (state × secret list), or equivalently, a set of tuples $(s, sl, s')$. Each of these tuples represents a possible configuration of the unwinding “synchronization
game”: $\sigma$ 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 $\sigma'$ and $sl'$ w.r.t. the alternative trace.

To keep proof size manageable, the framework supports the decomposition of $\Delta$ 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 $trn' = Trans \sigma' \_ \_ \sigma'_1$ that is unobservable (i.e., $\neg isObs trn'$), 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 $trn = Trans \sigma _ _ \sigma_1$ 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 $trn' = Trans \sigma' _ _ \sigma'_1$ (i.e., $isObs trn \leftrightarrow isObs trn'$ and $isObs trn \rightarrow getObs trn = getObs trn'$) 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 $i$, $\Delta_i$ is connected to all the relations into which it “unwinds,” i.e., the relations $\Delta_j, \Delta_k$ or $\Delta_l$ 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 $\Delta_0$ is a proper generalization of the bound for the initial state, $\forall sl sl', B sl sl' \rightarrow \Delta_0 \sigma_0 sl \sigma_0 sl'$. 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, $\Delta_0$ generalizes the bound $B$. In turn, $\Delta_0$ may unwind into other relations, and in general any relation in the graph may unwind...
into its successors. Hence, we can think of \( A_0 \) 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 \( A_0 \), there are 4 relations \( A_1 \)–\( A_4 \) with \( A_i \) corresponding to clause \((i)\) for the definition of \( B \) from Fig. 1. The edges correspond to the possible causalties between the clauses. For example, if \( B \) \( s/s' \) 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 \( A_3 \) to \( A_2 \) and \( A_4 \). Each \( A_i \) also provides a relationship between the states \( \sigma \) and \( \sigma' \) 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 \( \sigma =_{PID} \sigma' \), i.e., \( \sigma \) and \( \sigma' \) 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 \( A_2 \) and \( A_4 \). 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 (last textl = last textl'), and therefore the relaxed equality \( =_{PID} \) 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 \( \sigma =_{PID} \sigma' \), covered by \( A_1 \), and clause \((3)\) with empty update prefixes where we need \( \sigma = \sigma' \), covered by \( A_3 \). Fig. 5 gives the formal definitions of the relations. \( A_0 \) covers the prehistory of PID—from before it was created. In \( A_1 \)–\( A_3 \), the conditions on \( s/s' \) and \( s/s' \) essentially incorporate the inversion rules corresponding to clauses \((1)-(4)\) in \( B \)'s definition, while the conditions on \( \sigma \) and \( \sigma' \) 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 \( B \) \( s/s' \) implies \( A_0 \sigma_0 s/s_0 \sigma_0 s'/s' \). 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 \( A_1 \) for post texts, and the friendship status must be equal in the two states immediately before an OSec True secret, analogous to \( A_3 \) for post texts.
\[ A_0 \sigma \sigma' \sigma'' \equiv \neg \text{PID} \in \text{postIDs} \land \sigma = \sigma' \land \sigma'' \]
\[ A_1 \sigma \sigma' \sigma'' \equiv \text{PID} \in \text{postIDs} \land \sigma = \text{PID} \land \neg \text{open} \land \sigma'' \]
\[ \exists \text{textl, textl'}. \text{sl} = \text{map TSec textl} \land \text{sl'} = \text{map TSec textl'} \land \text{textl} = \text{[]} \rightarrow \text{textl'} = \text{[]} \]
\[ A_2 \sigma \sigma' \sigma'' \equiv \text{PID} \in \text{postIDs} \land \sigma = \sigma' \land \text{open} \land \sigma'' \]
\[ \exists \text{textl, textl'}. \text{sl} = \text{map TSec textl} \land \text{sl'} = \text{map TSec textl'} \land \text{BO sl sl' textl = textl' textl' last textl = last textl} \]
\[ A_3 \sigma \sigma' \sigma'' \equiv \text{PID} \in \text{postIDs} \land \sigma = \text{PID} \land \neg \text{open} \land \sigma'' \]
\[ \exists \text{textl, textl'}. \text{sl} = \text{OSec True sl sl' textl} \land \text{sl'} = \text{OSec True sl' textl} \land \text{BO sl sl' textl = textl'} \]
\[ A_4 \sigma \sigma' \sigma'' \equiv \text{PID} \in \text{postIDs} \land \sigma = \sigma' \land \text{open} \land \sigma'' \]
\[ \exists \text{textl sl sl'}. \text{sl} = \text{map TSec textl @ OSec True sl sl' textl} \land \text{sl'} = \text{map TSec textl @ OSec False sl sl' textl} \land \text{BO sl sl' textl'} \]

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 noninter-
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 “RS³ – Reliably Secure Software Systems” (SPP 1496).

**References**

5. The CoSMed website. [https://cosmed.globalnoticeboard.com](https://cosmed.globalnoticeboard.com), 2016. Anonymous access possible with username “demo” and password “demo”.