Noname manuscript No.
(will be inserted by the editor)

CoSMed: A Confidentiality-Verified Social Media Platform

Thomas Bauereiß · Armando Pesenti Gritti · Andrei Popescu · Franco Raimondi

the date of receipt and acceptance should be inserted later

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. We also show that, from a theoretical viewpoint, the removal of triggers from the notion of BD Security does not restrict its expressiveness.

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. We believe that proof assistant technology is growing to be the proper environment to host such large verification tasks, for the following reasons:

This is a post-peer-review, pre-copyedit version of an article published in the Journal of Automated Reasoning. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10817-017-9443-3

Thomas Bauereiß
German Research Center for Artificial Intelligence (DFKI) Bremen, Germany E-mail: thomas@bauereiss.name

Andrei Popescu
Department of Computer Science, Middlesex University London, UK E-mail: a.popescu@mdx.ac.uk

Armando Pesenti Gritti
Global NoticeBoard, UK E-mail: arpesenti@gmail.com

Franco Raimondi
Department of Computer Science, Middlesex University London, UK E-mail: f.raimondi@mdx.ac.uk

1. Thanks to the expressiveness of their logic, proof assistants are versatile frameworks for system specification.
2. Thanks to their increasingly powerful automation, they can be used to discharge sufficiently simple "component" goals fully automatically.
3. Thanks to the code extraction facilities, they establish themselves as frameworks where "real programming," and not only abstract specification, can be pursued.

Our proof assistant of choice, Isabelle/HOL [49, 50], offers a good blend of these qualities:

1. It is based on a quite expressive logic, sufficient for most system specification purposes—polymorphic classic higher-order logic—and offers advanced specification and structuring mechanisms (parametric reasoning blocks [37], a structured proof language [55], inductive and coinductive datatypes [16], and Haskell-style type classes [33]).
2. It has good internal automation, as well as a possibility to capitalize state of the art fully automatic proving technology via the Sledgehammer highway [52].
3. Its automatic code generator [31,32] effectively organizes this prover into a front end for certified programming in efficient functional languages such as Haskell, ML or Scala.

Our particular focus is to take advantage of these capabilities for verifying web application security. We employ 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 developed CoCon [38], a conference system (*à la* EasyChair) verified for confidentiality. It has been built not as a toy system, but having usability in mind—and indeed, CoCon has been deployed for TABLEAUX 2015 [24] and ITP 2016 [17]. At the same time, CoCon has a fairly small kernel, manageable for verification. We verified a comprehensive list of confidentiality properties, systematically covering the relevant sources of information from CoCon's application logic [38, §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 presents a second major end product of this line of work: CoSMed, a running social media platform built around a kernel that is verified for confidentiality. 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 input/output (I/O) automaton formalized in Isabelle, exported as Scala code, and wrapped in a web application (Section 2).

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 (consisting 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 (Section 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 of 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 employed Bounded-Deducibility (BD) security (cf. [38] and Section 3.2), a general framework for the verification of rich information flow properties of I/O automata. BD Security is parameterized by *bounds* and *triggers*, specifying what and when confidential information may be released/declassified. It turns out that triggers, while convenient, are not essential for practical formulations of BD Security (Section 3.3). In addition, unlike CoCon, for which 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 (Section 3.4). The verification proceeded by providing suitable unwinding relations, closely matching the bounds (Section 4).

In addition to confidentiality, we proved safety and accountability properties (Section 5). The former were used in the proofs of confidentiality. The latter are natural complements that strengthen the security guarantees of confidentiality. Indeed, confidentiality states: Unless a document becomes public or a user acquires such role, he cannot learn such information. But can a user not forge the acquisition of that role or the publication of the document? The accountability properties we proved show that this is not possible (except by identity theft).

In the verification process, we took full advantage of Isabelle's aforementioned structuring and automation capabilities (Section 6). This allowed us to keep the whole verification effort manageable.

CoSMed was developed to fulfill the functionality and security needs of a charity organization [6]. The current version is a prototype, not yet deployed for the charity usage. The CoSMed homepage [2] has links to the formal proofs (also discussed in Section 6), the source code, documentation, and a deployed demo version of the system.

The current paper is an extended version of the conference paper [10] presented at ITP 2016. In addition to the material in the conference paper, it includes:

- the presentation of a technique for capturing the triggers inside the bounds, generally applicable to any BD Security property (Section 3.3)
- a significantly wider discussion of the employed verification technology (Section 6)
- more details and explanations on the technical contributions and on related work

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. We write $[x_1, \ldots, x_n]$ for the list consisting of the elements $x_1, \ldots, x_n$; in particular, we write $[x]$ for a singleton list and $[]$ for the empty list. We write @ for 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.

We will make use of Isabelle's (ML-style) records. These are products containing tuples where each component (field) has a label. 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. For record access, we use functional notations: $l_i\ \sigma$ is the value of field l_i stored in σ.

Finally, we will make heavy use of Isabelle's (ML-style) datatypes. In their non-recursive version used here, datatypes are essentially sums (or disjoint unions) of types, where the different variants of the sum are named by indicated labels, called constructors. The constructors are used for building particular elements and for pattern-matching on arbitrary elements of a datatype.

2 System Description

In this section, we describe the system functionality as formalized in Isabelle (Section 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 (Section 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.

Formally, 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 $(title, txt, img, vis)$, where the title and the text are essentially strings, *img* is an (optional) image file, and $vis \in \{\mathsf{Friend}, \mathsf{Public}\}$ is a visibility status that can be assigned to posts: Friend means visibility to friends only, whereas Public means visibility to all users.

Note that the state member functions are total, whereas one might expect partial functions. For example, the function userReq does not make sense for all inputs of type userID (i.e., for all strings), but only for those that correspond to actual existing requests—stored in the list pendingUReqs. We chose to use total functions because this leads to simpler definitions and proofs. For the non-meaningful inputs, our functions return empty items of the desired types, e.g., empty strings. The behavior on non-meaningful inputs is irrelevant though, because we only apply the functions to meaningful inputs—e.g., we apply userReq to user IDs only after testing that they belong to pendingUReqs.

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:

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

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

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

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:

$$\begin{array}{l}\mathsf{e\_updateTextPost}\ \sigma\ uid\ p\ pid\ txt\ \equiv \\ \quad \mathsf{IDsOK}\ \sigma\ [uid]\ [pid]\ []\ \wedge\ \mathsf{pass}\ \sigma\ uid = p\ \wedge\ \mathsf{owner}\ \sigma\ pid = uid\end{array}$$

Besides the text, one can also update the title and the image of a post. The general-purpose predicate IDsOK takes a lists of user IDs and post IDs and checks if they are registered in the system. This check is part of the enabledness predicates for most of the actions.

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:

```
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:

```
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 uTextPost corresponds to the effect function updateTextPost.

The overall **step function**, step : state $\to$ act $\to$ out $\times$ 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 Uact (uTextPost *uid p pid txt*), the corresponding enabledness predicate is called on the current state (say, σ) with the given parameters, e_updateTextPost σ *uid p pid txt*. If this returns False, the result is (outErr, σ), meaning that the state has not changed and an error output is produced. If it returns True, the effect function is called, updateTextPost σ *uid p pid txt*, yielding a new state σ'. The result is then (outOK, σ'), 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 [38, §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 [32].

Around the exported code, there is a layer of trusted (unverified) code. It consists of an API written with the Scalatra framework and a web application that communicates with the API. This trusted code is necessary as a mediator between the purely functional code of the kernel and the object-oriented API of the web framework. In terms of lines of code, the wrapper turns out to be bigger than the kernel (cf. Section 6). Nevertheless, there are reasons to believe that the confidentiality guarantees of the kernel also apply to the overall system, including the wrapper. From a functionality point of view, the API layer 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, mostly without storing data itself. The only exception is session management: The web application caches passwords of users while they are logged in, using them when making requests to the kernel.[1]

Thus, the wrapper is a *trusted* layer around the *verified* application logic (the latter residing in the kernel). An interesting direction for future work is extending the verification scope to cover the wrapper as well, using language-based tools, and composing the wrapper and kernel properties to obtain a more holistic security guarantee.

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. For example:

[1] In principle, password storage could be moved to the wrapper completely, but for our prototype we chose to store all persistent data in the kernel.

– 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:[2]

> (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.

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:

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

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:

$$\text{TYPE\_SYNONYM } \mathsf{trace} = \mathsf{trans\ list}$$

[2] As it will turn out, this property needs to be refined in order to hold. We will do this in Section 3.4.

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 two parameters, B and T:

- B : secret list → secret list → bool is a binary relation on sequences of secrets. It specifies which secret sequences have to be indistinguishable for an observer. Hence, it gives a lower *bound* on the uncertainty of the observer about the secrets, or in other words, an upper bound on these secrets' *declassification*.
- T : trans → bool is a unary predicate on transitions. It specifies a "way out" of the bound, that is, a condition under which the bound is not required to hold—such as an observer legitimately acquiring a role granting him access.

In this context, BD Security states that O *cannot learn anything about* S *beyond* B *unless* T *occurs*. Formally:

> For all valid system traces tr and sequence of secrets sl' such that B (S tr) sl' and never T tr hold, there exists a valid system trace tr' such that S $tr' = sl'$ and O $tr' =$ O tr.

Above, never T tr states that T does not hold for any transition of the trace tr. Thus, BD Security requires that, if 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 [54] corresponds to B being the total relation and T being the empty (vacuously false) predicate—BD Security then requires that *any* secret is possible together with *any* observation. Hence, the observer can deduce *nothing* about the secrets in this case. Smaller relations B mean that an observer may deduce some information about the secrets, but nothing beyond B. In particular, if B is an equivalence relation, then (unless the trigger becomes true) the observer may deduce the equivalence class, but not the concrete secret within the equivalence class. For example, the bound B sl $sl' \equiv$ (length sl = length sl') specifies that observers may learn about the number of secrets that have occurred, but nothing about their content.

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

- isSec : trans → bool, filtering the transitions that produce secrets
- getSec : trans → secret, producing a secret out of a transition
- isObs : trans → bool, filtering the transitions that produce observations
- getObs : trans → 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 $(\mathsf{state}, \mathsf{act}, \mathsf{out}, \sigma_0, \mathsf{step})$
- a security model, consisting of:
  - a secrecy infrastructure $(\mathsf{secret}, \mathsf{isSec}, \mathsf{getSec})$
  - an observation infrastructure $(\mathsf{obs}, \mathsf{isObs}, \mathsf{getObs})$
  - a declassification bound B
  - a declassification trigger T

3.3 Capturing the Trigger in the Bound

The triggers were very convenient for CoCon's verification, since they naturally expressed role-acquiring phenomena, which "lift" the bound restrictions. For example, if a user is added as an author to a paper, any possible bound about the content of that paper should no longer apply.

For CoSMed, the situation is more dynamic. Triggers may be repeatedly fired and canceled, e.g., an observer first becoming friends with the post owner, and the post owner later canceling that friendship again. As we show in the next subsection, we can capture these dynamic triggers by incorporating them *into the bound*, rendering static triggers T unnecessary.

An interesting theoretical question is whether in general we can formulate BD Security properties without the trigger, with no loss of generality. Here, we give a partially positive answer to this question, showing that the bound can be enriched to cater for any desired trigger. The transformed BD Security property is *almost* equivalent to the original one, but slightly stronger (as we discuss below).

Given a BD Security property as introduced above with a static trigger T, we transform it into another instance of BD Security with T' vacuously false. For this purpose, we first extend the notion of *secret* by adding a designated value $\bot$ that represents the occurrence of the trigger, i.e., $\mathsf{secret}' = \mathsf{secret} \uplus \{\bot\}$. Furthermore, isSec' and getSec' are extended so that this value is produced whenever the trigger fires.

$$\mathsf{isSec}'(trn) \equiv \mathsf{isSec}(trn) \vee \mathsf{T}(trn)$$

$$\mathsf{getSec}'(trn) \equiv \begin{cases} \mathsf{getSec}(trn) & \text{if } \neg\mathsf{T}(trn) \\ \bot & \text{if } \mathsf{T}(trn) \end{cases}$$

This extended secrecy infrastructure, $(\mathsf{secret}', \mathsf{isSec}', \mathsf{getSec}')$, allows us to talk about the (non)occurrence of the trigger in the bound B'.

$$\mathsf{B}'\ sl\ sl' \equiv \mathsf{B}\ sl\ sl' \wedge \mathsf{never}_\bot(sl) \wedge \mathsf{never}_\bot(sl')$$

where $\mathsf{never}_\bot \equiv \mathsf{never}\ (\lambda s.\ s = \bot)$. Recall that BD Security only considers traces tr where the trigger never occurs. This nonoccurrence is captured in the transformed bound B' using the predicate $\mathsf{never}_\bot$. The transformation does not weaken the original security property:

Proposition 1 If an I/O automaton Aut satisfies BD Security w.r.t. (secret', isSec', getSec'), (obs, isObs, getObs), B', and T', then it satisfies BD Security w.r.t. (secret, isSec, getSec), (obs, isObs, getObs), B, and T.

In fact, the transformed property is slightly stronger than the original BD Security property: The converse of the implication in the above lemma does not hold in general. This is due to the fact that BD Security does not specify whether the trigger is allowed to occur in the *alternative* trace tr', while the transformed bound B' specifies that it *must not* occur. The latter is necessary in the transformed setup, because otherwise, we would have to specify explicitly *when* the trigger *must* occur. This is not reasonably possible without further knowledge of the system.

These considerations suggest a strengthening of the original notion of BD Security that rules out the occurrence of the trigger in the alternative trace. We say that an I/O automaton Aut satisfies *trigger-preserving BD Security* w.r.t. (secret, isSec, getSec), (obs, isObs, getObs), B, and T if, for all valid system traces tr and sequence of secrets sl' such that $\mathsf{B}\ (\mathsf{S}\ tr)\ sl'$ and $\mathsf{never}\ \mathsf{T}\ tr$ hold, there exists a valid system trace tr' such that $\mathsf{S}\ tr' = sl'$, $\mathsf{O}\ tr' = \mathsf{O}\ tr$, and $\mathsf{never}\ \mathsf{T}\ tr'$ hold. The transformed property is equivalent to this strengthened notion of BD Security:

Proposition 2 Aut satisfies BD Security w.r.t. (secret', isSec', getSec'), (obs, isObs, getObs), B', and T' iff it satisfies trigger-preserving BD Security w.r.t. (secret, isSec, getSec), (obs, isObs, getObs), B, and T.

Let us discuss the difference between the original and trigger-preserving variants of BD Security. Both notions require that, for each modification of the secret that is required by the bound, the system can produce an alternative trace while preserving the observation. The difference is that the original notion is slightly more flexible, in that it allows the system to let the trigger fire in the alternative trace. However, there does not seem much to be gained from this flexibility. Indeed, we have not used this flexibility in our case studies so far (for CoSMed and CoCon), so we could also have used the trigger-preserving variant instead. Moreover, the latter has the pleasant mathematical property that it implies transitivity of the bound:

Proposition 3 Aut satisfies trigger-preserving BD Security w.r.t. (secret, isSec, getSec), (obs, isObs, getObs), B, and T iff it satisfies trigger-preserving BD Security w.r.t. the same parameters with B replaced by its reflexive-transitive closure, B^*.

This result can be used to reduce the verification effort by focusing on a subset of the bound, and then obtaining the reflexive-transitive closure for free. This approach is used, for example, in the MAKS framework [43]. Its Basic Security Predicates (BSP) focus on the insertion or deletion of *one* confidential event at a time. The possibility of modifying *sequences* of confidential events follows by transitivity.

Nevertheless, we will use the original notion of BD Security in the rest of this paper, since we do not need the transitive closure or static triggers for CoSMed—we always take T to be vacuously false in the first place. In the following sections, we show how to incorporate the application-specific *dynamic* triggers of CoSMed into the bound.

In the rest of the paper, when formulating BD Security properties, we shall ignore triggers (which is equivalent to taking them to be vacuously false).

3.4 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 Section 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 Section 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 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. Note that this includes failed actions, i.e., errors must not leak confidential information.

Let us now instantiate the secrecy infrastructure (secret, isSec, 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. 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.

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

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

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

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

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

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

Fig. 1: The bound for post text confidentiality

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

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

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

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

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 $\mathsf{TSec}\ txt_1^1, \ldots, \mathsf{TSec}\ txt_{n_1}^1$
- possibly followed by a shift to an open access window, OSec True
- possibly followed by another block of text updates $\mathsf{TSec}\ txt_1^2, \ldots, \mathsf{TSec}\ txt_{n_2}^2$
- 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 $\mathsf{TSec}\ txt_{n_k}^k$ 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., $sl = \mathsf{map}\ \mathsf{TSec}\ textl$. It is indistinguishable from any alternative sequence of updates $sl' = \mathsf{map}\ \mathsf{TSec}\ textl'$, 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 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 an open 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, 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 a closed 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. (In other words, (P2) fails because the last version of a post before an open access window, e.g., before the observer is marked as a friend, is obviously accessible to the observer, whereas (P2) would claim that this last version would not leak—only conceding information flows from the versions resulting from later updates.)

If we also factor in the generalization from a single user to a coalition of users, we obtain the following correct version:

> (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.

We have reached a very precise formulation of a flow policy. One may reasonably argue that this formulation is rather complex-looking. This complexity is not specific to social media platforms, but arises each time we have a system that repeatedly grants and removes access rights for its users. In the future, it would be worth designing a simpler language, based on BD security, where such properties are expressed more naturally—an extension of the Paragon language [18], which allows iterated locking and unlocking of flows, might provide an elegant solution here.

3.5 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 Section 3.4, 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.

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

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

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

Fig. 2: The bound for friendship status confidentiality

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 $\mathsf{BO}_F\ \mathit{sl}\ \mathit{sl}'$ (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 (Section 4.1) and show how we have employed it for CoSMed (Section 4.2).

4.1 BD Unwinding Recalled

In [38], we have presented a verification technique for BD Security inspired by Goguen and Meseguer's *unwinding* technique for noninterference [28]. 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 $\mathsf{B}\ (\mathsf{S}\ tr)\ sl'$ holds.

The idea is to construct tr' incrementally, in synchronization with tr, but "keeping an eye" on sl' as well. The unwinding relation [38, §5.1] is therefore be not a relation on states, but a relation on $\mathsf{state} \times \mathsf{secret\ list}$, or equivalently, a set of tuples $(\sigma, sl, \sigma', sl')$. Each of these tuples represents a possible configuration of the unwinding "synchronization 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 the unwinding relation into smaller 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 $k \in \{1, \ldots, n\}$

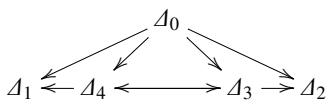

Fig. 3: Graph of unwinding relations

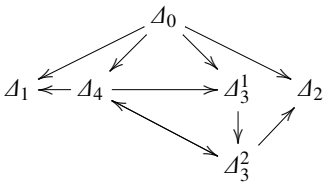

Fig. 4: Refined graph

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. This corresponds to initializing the game with a configuration that loads any two sequences of secrets satisfying the bound.

Theorem 4 [38] 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

$$
\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 \mathit{textl}\ \mathit{textl}'.\ sl = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}\ \wedge\ sl' = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}'\ \wedge \\
&\quad\ \ \mathit{textl} = [] \rightarrow \mathit{textl}' = [] \\
\Delta_2\ \sigma\ sl\ \sigma'\ sl' &\equiv \mathsf{PID} \in \mathsf{postIDs}\ \sigma\ \wedge\ \sigma = \sigma'\ \wedge\ \mathsf{open}\ \sigma\ \wedge \\
&\quad \exists \mathit{textl}.\ sl = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}\ \wedge\ sl' = \mathsf{map}\ \mathsf{TSec}\ \mathit{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 \mathit{textl}\ \mathit{textl}'\ sl_1\ sl_1'.\ sl = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}\ @\ [\mathsf{OSec\ True}]\ @\ sl_1\ \wedge \\
&\quad\ \ sl' = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}'\ @\ [\mathsf{OSec\ True}]\ @\ sl_1'\ \wedge \\
&\quad\ \ \mathsf{BO}\ sl_1\ sl_1'\ \wedge\ \mathit{textl} \neq []\ \wedge\ \mathit{textl}' \neq []\ \wedge\ \mathsf{last}\ \mathit{textl} = \mathsf{last}\ \mathit{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 \mathit{textl}\ sl_1\ sl_1'.\ sl = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}\ @\ [\mathsf{OSec\ False}]\ @\ sl_1\ \wedge \\
&\quad\ \ sl' = \mathsf{map}\ \mathsf{TSec}\ \mathit{textl}'\ @\ [\mathsf{OSec\ False}]\ @\ sl_1'\ \wedge\ \mathsf{B}\ sl_1\ sl_1'
\end{aligned}
$$

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

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 (last textl = 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 5 The relations in Fig. 5 form a graph of unwinding relations, and therefore (by Theorem 4) 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 $sl\ sl'$ 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.

5 Complementing Confidentiality with Safety and Accountability

While confidentiality was the main target of our verification, we also proved several safety properties as auxiliaries used for confidentiality (Section 5.1). In addition, we proved some accountability properties to strengthen the overall security guarantees (Section 5.2).

5.1 Safety Properties

It was helpful to establish some properties as global invariants of reachable states, which otherwise only appear locally or implicitly in the pre- and post-conditions of individual actions. For example, we proved that, in each reachable state:

1. The owner of an existing post in the system is an existing user.
2. Friendship is symmetric.
3. The lists of friends and friend requests contain no duplicates.
4. If a pending friend request exists from one user to another, then the two are not friends.

This allowed us to write the unwinding relations more succinctly and to simplify the proofs in several places. For example, the action by UID1 to add UID2 as a friend has the precondition that the two are not friends already, but with property 4 above, it is sufficient to know that a request from UID2 to UID1 exists in order to know that the action is enabled—which was very handy in the process of matching actions within the proof of friendship confidentiality.

5.2 Accountability Properties

We have shown that a user can only learn about updates to posts that were performed immediately before or during times of public visibility or friendship. But how can we be sure that the public visibility or the friendship status cannot be forged? To address this type of questions, we complement our proved confidentiality properties by a form of *accountability*, showing that forging confidentiality-relevant roles cannot be achieved without identity theft.

For friendship status, we proved the following: If, at some point t on a system trace, the users uid and uid' are friends, then one of the following holds:

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

In other words, an existing friendship can always be traced back to the standard protocol of a request from one user, followed by acceptance by the other, and the absence of subsequent canceling by either user. This ensures there is no back door to friendship.

We have formally stated this property by requiring that, given any valid system trace tr starting in the initial state for which the end state has uid and uid' as friends, we can decompose tr as $tr_1 @ [trn] @ tr_2 @ [trnn] @ tr_3$, where trn and $trnn$ are transitions and tr_1, tr_2, tr_3 are traces such that:

- trn represents the transition with the relevant friend request (from uid to uid' or vice versa)

– *trnn* represents the transition with the approval of this request
– tr_3 contains no successful unfriending action between the two users

For post visibility, we proved an accountability property similar to friend status accountability: If, at some point t on a system trace, the visibility of a post *pid* has a value *vis*, then one of the following holds:

– Either *vis* is Friend (as in the initial state)
– Or the post's owner had issued a successful "update visibility" action setting the visibility of *pid* to *vis*, and no other successful update actions to *pid*'s visibility occurs between that action and t

This was formalized by splitting any valid trace tr similarly to friend status accountability, as $tr_1 @ [trn] @ tr_2 @ [trnn] @ tr_3$, where:

– *trn* represents the transition where the post was created by some user *uid* (who becomes the owner)
– *trnn* represents the (last) update of *pid*'s visibility to *vis* (necessarily by *uid*)
– tr_3 contains no successful update to the post visibility

6 Verification Technology Aspects and Statistics

The whole formalization, which took us three person-months, consists of 9900 Isabelle lines of code (LOC). The system specification consists of 600 LOC, which gets automatically translated to 1400 lines of Scala code that forms the kernel of the running system. Around it, 2500 lines of manually written Scala code comprise the web application wrapper. The *statement* of the security properties takes 300 lines of Isabelle code—this is what needs to be manually inspected if one wishes to validate the adequacy of the properties we proved. The remaining code of the formalization comprises the machine-checked proofs of the properties and the (reusable) BD Security framework, which takes 1800 LOC. The framework has two parts: one dealing with generalities on I/O automata (350 LOC), and the other with BD Security specific aspects (the remaining 1450 LOC).

The I/O-automata part defines traces and state reachability. For both notions, it proves the equivalence between alternative inductive definitions (starting "from the left" or "from the right"), as well as the equivalence between the inductive trace-free notion of reachability and the existence of a trace from the initial state.

The BD Security specific part first defines BD Security and then proves sound the unwinding proof technique: initially for only one unwinding relation, then for a graph of relations—by reducing the latter to the former. It also includes the reduction of the triggers into the bounds discussed in Section 3.3. Having established the aforementioned equivalences for I/O automata was very useful in the transition between the heavy, trace-based formulation of BD Security and the lighter, state-based notion of unwinding.

The framework is contained in two locales,[3] reflecting its two parts: one for I/O automata (parameterized by arbitrary initial state σ_0 and step function step) and the other for BD Security (additionally parameterized by the secrecy (secret, isSec, getSec) and observation (obs, isObs, getObs) infrastructures, a bound B and a trigger T). Inside the latter locale,

[3] Locales [37] are Isabelle/HOL-specific structuring mechanisms. They allow for the development of theorems parameterized by abstract data and assumptions and automate the process of instantiating the theorems: The user provides concrete instances for the data and discharges the assumptions; in exchange, they obtain an unconditional version of the theorems for the given instance.

we express BD Security as a Boolean predicate secure. The end product of this locale is a slightly enhanced version of Theorem 4, which in Isabelle-like syntax looks as follows:

```
assumes 1: ∀ Δ ∈ Domain Gr. ∃ Δs ∈ Domain Gr. (Δ,Δs) ∈ Gr
and 2: Δ0 ∈ Domain Gr and 3: ∀ sl sl'. B sl sl' → Δ0 σ0 sl σ0 sl'
and 4: ∀ Δ. (∀ Δs. (Δ,Δs) ∈ Gr → unwind_cont Δ Δs) ∨ unwind_exit Δ
shows secure
```

The theorem's conclusion simply says "secure"—with no explicit parameters, but implicitly referring to the locale's parameters. Gr represents an unwinding graph as a set of pairs $(\Delta,\Delta s)$, where Δ is a relation on state $\times$ secret list and Δs is a set of such relations. It is required that each relation be connected to a set of continuations (assumption 1). A special initial relation is designated (assumption 2) and required to generalize the bound (assumption 3). Finally, it is required (assumption 4) that every relation Δ either unwinds into its continuations Δs (as explained in Section 4.2) or satisfies an exit condition. The exit condition is an enhancement, meant to simplify the proofs. It allows "winning" the unwinding game via a shortcut: proving that, in the reached configuration $(\sigma, sl, \sigma', sl')$, the remaining list of values sl cannot be produced by any partial trace starting in state σ—therefore making BD Security trivially true from this point onwards.

This theorem was instantiated to the verification of CoSMed's concrete properties: (P3) (post content confidentiality), the corresponding title and image versions of (P3), (P4) (friendship status) and (P5) (friendship request status). In each of these cases, we instantiated the parameters of the framework: σ_0 and step with the particular CoSMed I/O automaton, (obs, isObs, getObs) with the observations made by a fixed (but arbitrary) set of users UIDs, T as vacuously false (as explained in Section 3.4), and (secret, isSec, getSec), B and T in specific ways for each instance. Then we defined explicitly the unwinding relations and their graph Gr and verified the four assumptions for them. (In other words, we proved an analogue of Prop. 5 for each concrete instance.)

Expectedly, the formalization of these concrete instances constitutes the bulk of the formalization. It required a lot of elaboration and interaction—from the definitions of the unwinding relations to the establishment of some key lemmas to the verification of the unwinding conditions—totaling 6700 LOC.

As explained in Section 4.2, our unwinding relations include notions of "relaxed" state equality, e.g., equality everywhere save for the content of a given post, or equality everywhere save for the status of friendship between two given users. When verifying the unwinding conditions, we had to prove that these relaxed equalities are preserved by transitions and are indistinguishable under certain observations and secret productions. To support this reasoning, we employed four kinds of key lemmas, showing that, when hit with the same action a, two relaxed-equal states $\sigma_1 =_{\ldots} \sigma_2$ behave according to the following prescriptions, where we assume step $(\sigma_i, a) = (o_i, \sigma_i')$ and $trn_i = (\sigma_i, a, o_i, \sigma_i')$ for $i \in \{1, 2\}$:

1. They always yield relaxed-equal states, $\sigma_1' =_{\ldots} \sigma_2'$.
2. Under specific conditions, they yield the same output, $o_1 = o_2$.
3. Either both or none produce a secret, isSec $trn_1 \leftrightarrow$ isSec trn_2.
4. For specific actions and outputs, they are "closing in," yielding *strictly* equal states, $\sigma_1' = \sigma_2'$.

For example, for the "everywhere but on the content of the post PID" equality, $\sigma_1 =_{\mathsf{PID}} \sigma_2$, the conditions for property 2 were isObs $trn \wedge \neg$ open s (observable transition in a closed access window) and for property 4 we required that o_i is outOK and a has the form Uact (uTextPost $uid\ p$ PID txt) (successful update to the post PID). The proofs of types 1

and 2 lemmas required large case distinctions on the format of the actions, occasionally with further inner case distinctions involving equality of IDs—e.g., whether the post ID for an update action equals the fixed PID. By contrast, proving the lemmas of type 3 and 4 was straightforward, since the format of the action was predetermined (either directly or via the isSec selector)—so mere simplification usually did the job.

With the key lemmas in place, the unwinding proofs proceeded quite smoothly, but tediously. They required distinguishing various cases, concerning:

- the type of the current action involved
- the observability or secret-productivity of the associated transition
- or the status of the sequence-of-secrets part of the unwinding's current tuple $(\sigma, sl, \sigma', sl')$—e.g., whether sl or sl' have been emptied or not

Based on these cases, different decisions were made in the unwinding strategy: whether to proceed independently or react, and in the latter case whether to ignore or match. The key lemmas were all employed in the process of matching, with type 1 being used for most of the actions and type 4 for switching from open to closed access windows (reflected in the switch from relaxed to strict equality).

Employing the Isar structured proof language[4] was instrumental in managing the complexity of such nested case distinctions—especially since we often needed to temporarily leave an unwinding proof with some unfinished leaves in various nested contexts in order to prove the necessary lemmas. Without Isar, that is, in a so called "apply style" environment (which is the only option in most mainstream provers), switching back and forth between these leaves and outer lemmas would have been quite difficult. For example, to prove that the relation Δ_3^1 in Section 4.2 satisfies the unwinding conditions, we started a structured Isar proof where we first distinguished on whether the length of $textl_1$ (consisting of the post text content of sl_1) is ≥ 2 or not. For the condition being true, "independent action" was appropriate to finish the proof. Then we assumed the condition was false, and went for a "matching" strategy. In order to decide if "matching" should further be refined to "react" or to "ignore," we needed to further test whether the considered action was a successful text update of the given post ID PID (i.e., whether $\exists\, uid\ p\ txt.\ a = \mathsf{Uact}\ (\mathsf{uTextPost}\ uid\ p\ \mathsf{PID}\ txt) \wedge o = \mathsf{outOK}$ holds). If this was true, then we could infer that $textl$ had txt as its head. But to decide between "react" and "ignore," we still needed to know if the tail of $textl$ was empty or not. Only if it was empty we could go for "match"—and when trying to prove the matching condition, we realized we needed a safety-like property: that, on reachable states, no action can decrease the list of stored post IDs. We went outside the current proof to establish this auxiliary property; when we returned to the proof, we could easily regain mental context, since the hole corresponding to the nested case of interest was located at the end of a clearly delimited path in a pen-and-paper-like proof text. Later on, we discovered that this auxiliary property can be used to establish some intermediary facts which were useful in other cases as well—thanks to the structured proof text, it was easy to locate the most general (i.e., topmost) position in the proof where these intermediate facts could be stated and proved, for maximum reusability.

As mentioned, safety properties were used to support the unwinding proofs. The safety properties themselves were proved mostly automatically, by reachable-state induction. They number 15 properties and together took only 200 LOC. Finally, the two accountability properties were completely independent from the BD Security properties. Their proofs, totaling

[4] Isar [55] is a scripting language for Isabelle that allows to express structured proofs in forward, pen-and-paper style, with stating intermediate facts for later use and the possibility to resort to fully automated proofs for simple enough facts. It was inspired by the language used in the Mizar proof assistant [47].

700 LOC, proceeded by a conceptually straightforward, but quite tedious induction on the definition of traces.

In the proofs, we employed both the Isabelle internal automation ("auto" and friends) [48] and the Sledgehammer tool [52].[5] The internal automation worked very well for the properties involving reachable states and actions on them—including the sufficiently simple subgoals of the concrete unwinding proofs and their key lemmas. Here, the desired properties had a fairly regular pattern, determining them to usually surrender after case distinctions followed by auto.

On the other hand, Sledgehammer came into its own for properties involving traces, where it was able to find fairly complex chains of reasoning inside trace inductions on which internal automation would fail. This applied to both the abstract framework and the accountability properties. Indeed, we had 50 successful invocations of Sledgehammer in the 1800 LOC of the former, and 15 in the 700 LOC of the latter. Roughly one fifth of these invocations were able to return proofs by auto or fastforce, taking advantage of a recent improvement in Sledgehammer's feedback with the internal automation [14]. By contrast, we had only 28 invocations in the entire 6700 LOC of confidentiality proofs.

7 Related Work

We next describe the work from the literature that is related to ours according to different aspects—focusing on social media platforms, on holistic verification of systems, and on information-flow security design and verification/analysis.

7.1 Access Control for Social Media Platforms

Policy languages for social media platforms have been proposed in the context of relationship based access control [27], also including techniques based on epistemic logic [51]. These approaches focus on specifying policies for granting or denying access to data based on the social graph, e.g. friendship relations.

The main difference to our work is that we follow the paradigm of *information flow control*: We define a global notion of confidential information (complete histories of content updates, in the case of CoCon and CoSMed) and specify a bound on what aspects of this information an observer may learn. We then verify that there is no combination of system actions allowing the observer to learn more than the bound specifies. While our system implementation does make use of access control, our guarantees go beyond access control, to information flow control.

It would be an interesting line of future work to establish a formal connection between the different paradigms, i.e., whether enforcing an access control policy specified in a language such as the ones from [27,51] generally implies that a certain information flow policy is satisfied.

[5] Sledgehammer differs from the internal automation in that it requires no instrumentation (of what facts to invoke in the proof, to add to the simplifier, etc.). Instead, Sledgehammer applies a relevance filter to identify facts that are likely to be useful for the stated goal; these facts are translated to first-order logic and handed over to the automatic provers; a possible positive answer from any of the provers (which also contains the much smaller set of actually used facts) is translated back into Isabelle/HOL's logic, where the original goal is discharged [15, §7].

7.2 Holistic Verification of Systems

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 [45], the CompCert C compiler [41] and the seL4 operating system kernel [39]. More recent developments include a range of microprocessors [34], Java and ML compilers [40, 42], and a model checker [25].

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 [44]. They include a hardware architecture with information-flow primitives [23] and a separation kernel [22], and noninterference for seL4 [46]. A substantial contribution to web client security is the Quark verified browser [36].

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. A very recent addition to this line of work is CoSMeDis [3], an extension of CoSMed into a distributed, diaspora*-style [7] social media platform—which we have verified using a theory of compositionality for BD security [11].

Outside the realm of proof-assistant based work, ConfiChair [8] is a competitor for CoCon verified using the ProVerif [13] process algebra tool. It proposes a cryptography-based cloud model where authors and reviews cannot be linked to their documents—not even by the system's administrator.

Ironclad [35] provides end-to-end security guarantees down to the binary code level and across the network, in a two-stage approach. In the first stage, properties of interest are verified using a combination of automatic tools. The information flow properties discussed in [35] focus on controlling *where* in the program information is declassified, e.g., in trusted declassification functions—by contrast, in this paper we focus on controlling *what* information is released and *when*. In the second stage, a verified Ironclad app is deployed on a server, and a Trusted Platform Module certifies to remote users of the app that the code running on the server indeed corresponds to the verified code. In principle, these two stages are orthogonal, and the second could be applied to CoSMed to provide guarantees to end-users that the code running on a CoSMed server is authentic.

7.3 Automatic Analysis of Information Flow

There are quite a few programming languages and tools aimed at supporting information-flow secure programming (such as Jif [4], Spark [5], Jeeves [56] and Ur-Web [19]) as well as information-flow tracking tools for the client side of web applications [12, 21, 29].

We foresee a future where such tools, including emerging information-flow model checkers [26], 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. CertiCrypt [9] is an excellent example of a cooperation between a proof assistant (Coq) and an automatic tool (Z3) for a specific purpose—automating cryptography proofs.

7.4 Frameworks for Information-Flow Security

There has been a lot of work on notions of confidentiality with controlled declassification, often focused on specific programming languages. Sabelfeld and Sands [53] give an

overview of the literature and discuss different potentially desirable dimensions of declassification control. Our work is focused on the aspects of *what* information may be declassified *when*, e.g. only the current version of a private post when an observer becomes friends with the post owner.

The Paragon language [18] controls when information may be released using flow locks that may be opened or closed by the program. Information release is guarded by conditions on the lock state. This is very similar in spirit to our approach of defining declassification bounds with open and closed phases for information release. The main difference to our declassification bounds is that flow locks do not attempt to provide fine-grained control over *what* information is declassified.

Recently, Guttman and Rowe introduced a notion of *blur operators* [30] which is very similar to the notion of declassification bounds in BD security, in that they specify upper bounds on the declassification or, equivalently, lower bounds on which confidential traces have to be possible together with a given observation. Similarly, Chong and van der Meyden [20] introduce a notion of information flow policies with *filter functions* specifying what information may flow between security domains. Guttman and Rowe leave open the question how the local assumptions on the system components are verified, while Chong and van der Meyden provide a set of conditions that can be used to implement an information flow policy using access control. However, due to differences in the definition of the security notion, these conditions differ from ours. While Chong and van der Meyden prove that *every* pair of traces with indistinguishable secrets produces the same observation, we construct *one* alternative system trace for each given observation and alternative secret in our unwinding proofs. Since we only want to prove the *possibility* of different secrets for a given observation, this is sufficient.

8 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.

Acknowledgments

We are indebted to the reviewers of both the conference and the journal versions of this paper for useful comments and suggestions, which led to the significant improvement of the presentation. 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",
- EPSRC through grants "VOWS" (EP/N019547/1) and "VRBMAS" (EP/K033921/1),
- DFG through grants "MORES" (Hu 737/5-2) and "SecDed" (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. The CoSMed Homepage. http://andreipopescu.uk/CoSMed.html.
3. The CoSMeDis Homepage. http://andreipopescu.uk/CoSMeDis.html.
4. Jif: Java + information flow, 2014. http://www.cs.cornell.edu/jif.
5. SPARK, 2014. http://www.spark-2014.org.
6. Caritas Anchor House. http://caritasanchorhouse.org.uk/, 2016.
7. The diaspora* project. https://diasporafoundation.org/, 2016.
8. M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *POST*, pp. 89–108, 2012.
9. G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pp. 90–101, 2009.
10. T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi. CoSMed: A confidentiality-verified conference management system. In *ITP*, 2016.
11. T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. CoSMeDis: A distributed social media platform with formally verified confidentiality guarantees. In *IEEE Symposium on Security and Privacy*, pp. 729–748, 2017.
12. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, pp. 159–178, 2014.
13. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *LICS*, pp. 331–340, 2005.
14. J. C. Blanchette, S. Böhme, M. Fleury, S. J. Smolka, and A. Steckermeier. Semi-intelligible isar proofs from machine-generated proofs. *J. Autom. Reasoning*, 56(2):155–200, 2016.
15. J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.
16. J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, eds., *ITP 2014*, vol. 8558 of *LNCS*, pp. 93–110. Springer, 2014.
17. J. C. Blanchette and S. Merz, eds. *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, vol. 9807, 2016.
18. N. Broberg, B. van Delft, and D. Sands. Paragon - practical programming with information flow control. *Journal of Computer Security*, 25(4-5):323–365, 2017.
19. A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pp. 153–165, 2015.
20. S. Chong and R. V. D. Meyden. Using Architecture to Reason About Information Security. *ACM Trans. Inf. Syst. Secur.*, 18(2):8:1–8:30, Dec. 2015.
21. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, pp. 50–62, 2009.
22. 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.
23. 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.

24. H. de Nivelle, ed. *Automated Reasoning with Analytic Tableaux and Related Methods - 24th International Conference, TABLEAUX 2015, Wrocław, Poland, September 21-24, 2015. Proceedings*, vol. 9323, 2015.
25. 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.
26. B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking hyperltl and hyperctl ^*. In *CAV*, pp. 30–48.
27. 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.
28. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pp. 75–87, 1984.
29. 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.
30. J. D. Guttman and P. D. Rowe. A cut principle for information flow. In C. Fournet, M. W. Hicks, and L. Viganò, eds., *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, pp. 107–121. IEEE, 2015.
31. F. Haftmann. *Code Generation from Specifications in Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2009.
32. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117, 2010.
33. F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In *TYPES*, pp. 160–174, 2006.
34. 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.
35. C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pp. 165–181, 2014.
36. D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, pp. 113–128, 2012.
37. F. Kammüller, M. Wenzel, and L. C. Paulson. Locales—a sectioning concept for Isabelle. In *TPHOLs'99*, pp. 149–166, 1999.
38. S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *CAV*, pp. 167–183, 2014.
39. 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.
40. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL*, pp. 179–192, 2014.
41. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
42. A. Lochbihler. Java and the Java memory model—A unified, machine-checked formalisation. In *ESOP*, pp. 497–517, 2012.
43. H. Mantel. Possibilistic definitions of security - an assembly kit. In *CSFW*, pp. 185–199, 2000.
44. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 605–607. 2011.

45. J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the amd5$_k$86^{tm} floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998.
46. 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.
47. A. Naumowicz, Adamand Korniłowicz. A brief overview of Mizar. In *TPHOLs*, pp. 67–72, 2009.
48. T. Nipkow. *Programming and Proving in Isabelle/HOL*, 2017. `https://isabelle.in.tum.de/dist/Isabelle2016-1/doc/prog-prove.pdf`.
49. T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL.* Springer, 2014.
50. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
51. R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *SEFM*, pp. 378–392, 2014.
52. L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *IWIL*, 2010.
53. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
54. D. Sutherland. A model of information. In *9th National Security Conf.*, pp. 175–183, 1986.
55. M. Wenzel. Isar—a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pp. 167–184, 1999.
56. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pp. 85–96, 2012.