CoSMed: A Confidentiality-Verified Social Media Platform

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

[1] German Research Center for Artificial Intelligence (DFKI) Bremen, Germany
[2] School of Science and Technology, Middlesex University, UK
[3] Global NoticeBoard, UK

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. Modeling CoSMed's confidentiality properties poses interesting challenges to BD Security, due to the more dynamic nature of information declassification. Namely, the static topology of declassification bounds and triggers that characterized previous instances of BD security has to give way to a dynamic integration of the triggers as part of the bounds.

1 Introduction

Recent years have seen an explosion of web-based systems aimed at sharing information between multiple users in a convenient, but controlled fashion. Examples include enterprise systems, social networks, e-commerce sites and cloud services. These systems often deal with confidential information such as credit card details, medical data, location information and sensitive documents. Unfortunately, most of these systems offer no guarantees concerning the prevention of *unintended flow* of information. Programming errors leading to information leakage can have different degrees of severity and can affect millions of users or registered individuals (as with, e.g., the Heartbleed bug [1]). This type of errors are likely to occur in practice, since web applications engage in intensive exchange of information with the environment. The Open Web Application Security Project (OWASP) includes "sensitive data exposure" (with typical "severe" impact) and the related "missing function level access control" (with "moderate" impact) in their influential list of ten most critical web application security risks [2].

The difficulty of preventing such errors often rests in the fact that information flow security is a complex, global property of a program, not entirely reducible to access control. For example, consider a system that stores information on medical patients and offers selective web access to individuals and companies. A life insurance company agent should not be given direct access to sensitive information such as a patient suffering from cancer. Yet, this protection may not prevent *propagation* of information: if it is known that certain discounts are only offered to cancer suffering patients, then the agent can infer that sensitive information if they can access a patient's available discounts.

Two years ago, we have started a line of work aimed at addressing information flow security problems of realistic web-based systems by interactive theorem proving (using our favorite proof assistant, Isabelle/HOL [23, 24]). We have introduced a security notion that allows a very fine-grained specification of what an attacker is assumed to be able to observe about the system, and what information is to be kept confidential in which situations. In our case studies, we assume the observers to be users of the system, and our goal is to verify that, by interacting with the system, the observers cannot learn more about confidential information than what we have specified.

As a first case study, we have developed CoCon [15], 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 already been deployed for TABLEAUX 2015 and ITP 2016. At the same time, CoCon has a fairly small kernel, manageable for verification. With little resources (a few person months) we have verified a comprehensive list of confidentiality properties, systematically covering the relevant sources of information from the application logic [15, §4.5]. For example, besides authors, only PC members are allowed to learn about the content of submitted papers, and nothing beyond the last submitted version before the deadline.

This paper introduces a second major end product of this line of work: CoSMed, a confidentiality-verified social media platform. CoSMed allows human users to register and post information in the form of notices (containing text and/or images), and to restrict access to this information based on friendship relationships established between users. In addition, it allows web application users ("apps") to register and selectively access information. Architecturally, CoSMed is an I/O automaton formalized in Isabelle, exported as Scala code, and wrapped in a web application (§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 notices that can be posted by the users. The latter consist of title, text, and an optional image. The roles in CoSMed include admin, owner, friend and registered app. Modeling the restrictions on CoSMed's information flow poses additional challenges (§3), since here the roles vary dynamically. For example, assume we prove a property analogous to those for CoCon: A user U1 learns nothing about the friend-only notices posted by a user U2 *unless* U1 becomes a friend of U2. Although this property makes sense, it is too weak—given that U1 may be "friended" and "unfriended" by U2 multiple times. A stronger confidentiality property would be: U1 learns nothing about U2's friend-only notices *beyond* the updates performed *while* U1 and U2 were friends. This stretches the limits of Bounded-Deducibility (BD) Security (§3.2), a framework designed for CoCon's verification. The previously encountered fixed structure of bounds and triggers gives way to more dynamically evolving bounds that incorporate trigger information (§3.3). The verification proceeds by providing suitable unwinding relations, closely matching the bounds (§4).

CoSMed has been developed to fulfill the functionality and security needs of a charity organization [3]. The current version is a prototype, not yet deployed for the charity usage. Both the formalization and the running website are publicly available [5, 6].

2 System Description

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

2.1 Isabelle Specification

Abstractly, the system can be viewed as an I/O automaton, having a state and offering some actions through which the user can affect the state and retrieve outputs. The state stores information about users, notices 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;
– notice information: the current notice IDs and the notices (notice content) associated to them, including visibility information;
– notice-user relationships: the notice owners;
– user-user relationships: the pending friend requests and the friend relationship.

In addition to (human) users, the system also allows "apps,“ that is, web applications authorized to retrieve notice content from the system—this is to enable web programmers to make CoSMed notice texts available on external websites, including other notice boards. Similarly to users who authenticate via passwords, apps authenticate via keys. All in all, the **state** is represented as an Isabelle record:

```
RECORD state =
(* User info: *)
    pendingUReqs : userID list    userReq : userID → req    userIDs : userID list
    user : userID → user    pass : userID → password    admin : userID
(* Friend info: *)
    pendingFReqs : userID → userID list    friendReq : userID → userID → req
    friendIDs : userID → userID list
(* App info: *)
    pendingAReqs : appID list    appReq : appID → req    appIDs : appID list
    key : appID  → key
(* Notice info: *)
    noticeIDs : noticeID list    notice : noticeID → notice    owner : noticeID → userID
```

Above, the types userID, appID, noticeID, password, key and req are essentially strings (more precisely, datatypes with one single constructor embedding strings). Each pending request (be it for user or app registration or for friend relationship) stores a request info (of type req), 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 notice of notices contains tuples $(title, text, img, vis)$, where the title and the text are essentially strings, img is an (optional) image file, and $vis \in \{\mathsf{FriendV}, \mathsf{UserV}, \mathsf{PublicV}\}$ is a visibility status that can be assigned to notices: FriendV means visibility to friends only, UserV means visibility to all the (human) users, and PublicV means visibility to all, including the registered apps.

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

The actions take varying numbers of parameters, indicating the agent (user or app) involved and optionally some data to be "posted" 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 an agent 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 agent. If it is not enabled, then an error message is returned to the agent and the state remains unchanged.

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

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

Here and elsewhere, the following Isabelle notations are used: Given a record s, field labels $l_1, \ldots, l_n$ and values $v_1, \ldots, v_n$ respecting the types of the labels, we write $s(l_1 := v_1, \ldots, l_n := v_n)$ for s with the values of the fields l_i updated to v_i. We let $l_i\ s$ be the value of field l_i stored in s. 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.

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

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

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

The four main kinds of items that can be created/registered in the system are users, apps, friends and notices. Notice creation can be immediately performed by any user. By contrast, user, app 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, notice content, visibility status, etc. For example, the following action is updating, on behalf of the user *uid*, the text of a notice with ID *nid* to the value *text*.

$$\begin{aligned}&\mathsf{updateTextNotice}\ s\ uid\ p\ nid\ text\ \equiv\\&\quad s\ (\mathsf{notice} := (\mathsf{notice}\ s)(nid := \mathsf{setTextNotice}\ (\mathsf{notice}\ s\ nid)\ text))\end{aligned}$$

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

Reading actions allow users and apps to retrieve content from the system. One can read user and notice 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 and app registration requests (for the admin); all users and apps of the system; all notices; 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
| uTitleNotice userID password noticeID title
| uTextNotice userID password noticeID text
| uImgNotice userID password noticeID img
| uVisNotice userID password noticeID vis

DATATYPE rAct =
  rUser userID password userID
| rNUReq userID password userID
| rNAReq userID password appID
| rAmIAdmin userID password
| rTitleNotice userID password noticeID
| rTextNotice userID password noticeID
| rImgNotice userID password noticeID
| rVisNotice userID password noticeID
| rOwnerNotice userID password noticeID
| rTitleNoticeByApp appID password noticeID
| rTextNoticeByApp appID password noticeID
| rFriendReqToMe userID password userID
| rFriendReqFromMe userID password userID
```

We have more reading actions than update actions. Some items, such as new-user, new-app 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 uTextNotice corresponds to the effect function updateTextNotice.

The overall **step function**, step : state $\rightarrow$ act $\rightarrow$ out $\times$ state, proceeds as follows. When given a state s and an action a, it first pattern-matches on a to discover what kind of action it is. For example, assume a is an update action, i.e., has the form Uact ua for ua : uAct. Then one pattern matches on the result, here ua, to discover the particular form of action. Assume ua has the form uTextNotice $uid\ p\ nid\ text$. In this case, one calls (on the current state s) the corresponding enabledness predicate with the given parameters, e_updateTextNotice $s\ uid\ p\ nid\ text$. If this returns False, the result is (outErr, s), meaning that the state has not changed and an error output is produced. If it returns True, the effect predicate is called, updateTextNotice $s\ uid\ p\ nid\ text$, yielding a new state s'. The result is then (outOK, s'), containing the new state along with an output indicating that the update was successful.

Note that start, creation and update actions change the state but do not output nontrivial data (besides outErr or outOK). By contrast, reading actions do not change the state, but output data such as user info, notice content or 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 [15, §2]. The I/O automaton formalized by the initial state $\mathsf{istate} : \mathsf{state}$ and the step function $\mathsf{step} : \mathsf{state} \to \mathsf{act} \to \mathsf{out} \times \mathsf{state}$ represents CoSMed's kernel, which we formally verify. This kernel is exported as Scala code using Isabelle's code generator [12].

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

3 Stating Confidentiality

Web-based systems for managing online resources and workflows for multiple users, such as CoCon and CoSMed, are typically programmed by distinguishing between various roles (e.g., author, PC member, reviewer for CoCon and admin, owner, friend, registered app 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 such action is only allowed if the agent has such role and such circumstance holds. However, the question whether access control achieves its purpose, i.e., really restricts undesired information flow, is a *global* question whose formalization involves the set of *all the system's execution traces*. In the end, we are interested in a bound not on what an agent can access, but on what an agent can infer, or learn.

3.1 From CoCon to CoSMed

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

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

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

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

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 user/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:[4]

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

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 when the access window is supposed to be open. In other words, we would need to incorporate (and iterate) the trigger inside the bound. As we show below, this is possible with the price of enriching the notion of observed value to include access-window data. 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

BD security is parameterized by the following data:

- an I/O automaton $(\mathsf{state}, \mathsf{act}, \mathsf{out}, \mathsf{istate}, \mathsf{step})$
- a security model, consisting of:
  - a value infrastructure $(\mathsf{val}, \varphi, f)$
  - an observation infrastructure $(\mathsf{obs}, \gamma, g)$
  - a declassification trigger T
  - a declassification bound B

In the automaton, we call the inputs "actions." Then state, act, and out are the types of states, actions, and outputs, respectively, $\mathsf{istate} : \mathsf{state}$ is the initial state, and $\mathsf{step} : \mathsf{state} \to \mathsf{act} \to \mathsf{out} \times \mathsf{state}$ is the one-step transition function. Transitions are tuples describing an application of step:

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

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

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

A trace $tr = [trn_0, \ldots, trn_{n-1}]$ is called valid if it starts in the initial state istate 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 contains only valid transitions and traces.

In the security model, we have the types:

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

- val, of values of interest (which could have also been called “secrets”)
- obs, of possible results of observations

as well as the functions

- $\varphi : \mathsf{trans} \to \mathsf{bool}$, filtering the transitions that produce values of interest
- $f : \mathsf{trans} \to \mathsf{val}$, producing a value out of a transition
- $\gamma : \mathsf{trans} \to \mathsf{bool}$, filtering the transitions that produce observations
- $g : \mathsf{trans} \to \mathsf{obs}$, producing an observation out of a transition

Given a system trace *tr*, we let:

- $\mathsf{V}\ tr$ be the list of values produced by filtering the transitions of *tr* with φ and applying f to them
- $\mathsf{O}\ tr$ be the list of observations produced by filtering the transitions of *tr* with γ and applying g to them

Formally, we have $\mathsf{V} = \mathsf{filtermap}\ \varphi\ f$ and $\mathsf{O} = \mathsf{filtermap}\ \gamma\ g$, where $\mathsf{filtermap} : (\alpha \to \mathsf{bool}) \to (\alpha \to \beta) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list}$ is a polymorphic combinator defined as follows:

$$\begin{aligned} \mathsf{filtermap}\ P\ h\ [] &\equiv [] \\ \mathsf{filtermap}\ P\ h\ (x \# xl) &\equiv (\text{if } P\ x \text{ then } [h\ x] \text{ else } []) \mathbin{@} \mathsf{filtermap}\ P\ h\ xl \end{aligned}$$

with # and @ being cons and append for lists.

The binary relation $\mathsf{B} : \mathsf{val\ list} \to \mathsf{val\ list} \to \mathsf{bool}$, called the *bound*, expresses a form of indistinguishability in the space of all possible sequences of values. Finally, $\mathsf{T} : \mathsf{trans} \to \mathsf{bool}$ is predicate on transitions called the *trigger*. We write $\mathsf{never}\ \mathsf{T}$ for the predicate on traces stating that T holds for none of the trace’s transition.

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

For all traces *tr* and value lists vl' such that $\mathsf{B}\ (\mathsf{V}\ tr)\ vl'$ and $\mathsf{never}\ \mathsf{T}\ tr$ hold, there exists a trace tr' such that $\mathsf{V}\ tr' = vl'$ and $\mathsf{O}\ tr' = \mathsf{O}\ tr$.

Intuitively, BD security requires that, if $\mathsf{B}\ vl\ vl'$ holds and the trigger has not occurred, then observers cannot distinguish vl from vl' via their observations—if vl is consistent with a given observation, then so must be vl'. Classical nondeducibility [25] corresponds to B being the total relation—the observer can then deduce *nothing* about the secret values. Smaller relations B mean that an observer may deduce some information about the value, but nothing beyond B—for example, if B is an equivalence relation, then the observer may deduce the equivalence class, but not the concrete value within the equivalence class.

3.3 CoSMed Confidentiality as BD Security

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

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

For the security model, we first instantiate the observation infrastructure $(\mathsf{obs}, \gamma, g)$. The agents that can observe the system are users and apps, so we factor in both categories. Moreover, instead of assuming a single observer, we wish to allow coalitions of an arbitrary number of agents—this will provide us with slightly stronger security guarantees. Finally, from a transition $\mathsf{Trans}\ s\ a\ o\ s'$ issued by a user or app agent, it is natural to allow the agent 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 $g : \mathsf{trans} \to \mathsf{obs}$ to be $g\ (\mathsf{Trans}\ \_\ a\ o\ \_) \equiv (a, o)$. We fix sets UIDs and AIDs of user IDs and app IDs and define the observation filter $\gamma : \mathsf{trans} \to \mathsf{obs}$ by

$$\begin{aligned}\gamma\ (\mathsf{Trans}\ s\ a\ o\ s') \equiv\ & (\exists uid \in \mathsf{UIDs}.\ \mathsf{userOf}\ a = \mathsf{Some}\ uid) \vee \\ & (\exists aid \in \mathsf{AIDs}.\ \mathsf{appOf}\ a = \mathsf{Some}\ aid)\end{aligned}$$

Above, for any action a, $\mathsf{userOf}\ a$ returns None if there is no user agent performing the action, and $\mathsf{Some}\ uid$ if the user agent with user ID uid performs the action. The function $\mathsf{appOf}\ a$ is defined in a similar way for apps. For example:

$$\begin{aligned}\mathsf{userOf}\ (\mathsf{updateTextNotice}\ s\ uid\ p\ nid\ text) &\equiv \mathsf{Some}\ uid \\ \mathsf{appOf}\ (\mathsf{readNoticeByApp}\ aid\ p\ nid) &\equiv \mathsf{Some}\ aid\end{aligned}$$

In summary, the observations are all actions issued by members of two fixed sets UIDs (of users) and AIDs (of apps), together with the outputs that these actions are producing.

Let us now instantiate the value infrastructure $(\mathsf{val}, \varphi, f)$. Since the property (P2) talks about the text of a notice, say, identified by $\mathsf{NID} : \mathsf{noticeID}$, a first natural choice for values would be the text updates stored in NID via $\mathsf{updateTextNotice}$ actions. That is, we could have the filter $\varphi\ a$ hold just in case a is such a (successfully performed) action, say, $\mathsf{updateTextNotice}\ s\ uid\ p\ nid\ text$, and have the value-producing function $f\ a$ return the updated value, here $text$. 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 window is open—e.g., while a user in UIDs is the owner's friend? The bound clearly needs this distinction—indeed, it states that nothing should be learned beyond the updates that occurred during an open access window.

To enable this distinction, we enrich the notion of value to include not only these notice-text updates, but also the changes in the open-closed status of the access window for the observers UIDs or AIDs w.r..t. NID. To this end, we define the following state predicates (where we write $\in$ for membership to both sets and lists):

$$\begin{aligned}\mathsf{openToUIDs}\ s \equiv\ & \exists uid \in \mathsf{UIDs}.\ uid \in \mathsf{userIDs}\ s\ \wedge \\ & \quad (uid = \mathsf{owner}\ s\ nid\ \vee\ uid \in \mathsf{friendIDs}\ s\ (\mathsf{owner}\ s\ nid) \vee \\ & \quad \mathsf{visNotice}\ (\mathsf{notice}\ s\ \mathsf{NID}) \in \{\mathsf{UserV}, \mathsf{PublicV}\}) \\ \mathsf{openToAIDs}\ s \equiv\ & \exists aid \in \mathsf{AIDs}.\ aid \in \mathsf{appIDs}\ s\ \wedge\ \mathsf{visNotice}\ (\mathsf{notice}\ s\ \mathsf{NID}) = \mathsf{PublicV} \\ \mathsf{open}\ s \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ (\mathsf{openToUIDs}\ s\ \vee\ \mathsf{openToAIDs}\ s)\end{aligned}$$

Thus, $\mathsf{openToUIDs}$ and $\mathsf{openToAIDs}$ mark the states when one of the users in UIDs and one of the apps in AIDs is entitled to access the text of NID. Openness, expressed by open, holds when NID is registered and one of these two access windows are open.

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

$$\begin{aligned}\varphi\ (\mathsf{Trans}\ \_\ (\mathsf{Uact}\ (\mathsf{uTextNotice}\ \mathsf{NID}\ \_\ \_\ text))\ o\ \_) &\equiv nid = \mathsf{NID}\ \wedge\ o = \mathsf{outOK} \\ \varphi\ (\mathsf{Trans}\ s\ \_\ \_\ s') &\equiv \mathsf{open}\ s \neq \mathsf{open}\ s'\end{aligned}$$

In consonance with the filter, the value-producing function $f : \mathsf{trans} \to \mathsf{val}$, where

$$\textsc{datatype}\ \mathsf{val} = \mathsf{TVal}\ text\ |\ \mathsf{OVal}\ \mathsf{bool}$$

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

$$\frac{\mathsf{BO}\ \mathit{vl}\ \mathit{vl}' \qquad \mathit{textl} \neq [] \leftrightarrow \mathit{textl}' \neq [] \qquad \mathit{textl} \neq [] \rightarrow \mathsf{last}\ \mathit{textl} = \mathsf{last}\ \mathit{textl}'}{\mathsf{B}\ (\mathsf{map\ TVal}\ \mathit{textl}\ @\ \mathsf{OVal\ True}\ @\ \mathit{vl})\ (\mathsf{map\ TVal}\ \mathit{textl}'\ @\ \mathsf{OVal\ True}\ @\ \mathit{vl}')}\ (3)$$

$$\frac{\mathsf{B}\ \mathit{vl}\ \mathit{vl}'}{\mathsf{BO}\ (\mathsf{map\ TVal}\ \mathit{textl}\ @\ \mathsf{OVal\ False}\ @\ \mathit{vl})\ (\mathsf{map\ TVal}\ \mathit{textl}\ @\ \mathsf{OVal\ False}\ @\ \mathit{vl}')}\ (4)$$

Fig. 1: The bound for notice text confidentiality

will retrieve either the updated text or the updated openness status:

$$f\ (\mathsf{Trans}\ \_\ (\mathsf{Uact}\ (\mathsf{uTitleNotice}\ \_\ \_\ \_\ \mathit{text}))\ \_\ \_) \equiv \mathsf{TVal}\ \mathit{text}$$
$$f\ (\mathsf{Trans}\ \_\ \_\ \_\ s') \equiv \mathsf{OVal}\ (\mathsf{open}\ s')$$

It remains to define the trigger $\mathsf{T} : \mathsf{trans} \rightarrow \mathsf{bool}$ and the bound $\mathsf{B} : \mathsf{val} \rightarrow \mathsf{val} \rightarrow \mathsf{bool}$. As discussed in §3.1, our bound will also take responsibility for tracking the repeated enabledness and disabledness of a trigger-like condition—so here we take the "static" trigger T to be vacuously false. Now, in order to formalize the desired bound B, we first note that the values produced from system traces consist of:

- a (possibly empty) block of text updates $\mathsf{TVal}\ \mathit{text}^1_1, \ldots, \mathsf{TVal}\ \mathit{text}^1_{n_1}$
- possibly followed by a change of status in the access window, OVal True
- possibly followed by another block of text updates $\mathsf{TVal}\ \mathit{text}^2_1, \ldots, \mathsf{TVal}\ \mathit{text}^2_{n_2}$
- possibly followed by a change of status in the access window, OVal False
- … and so on …

We wish to state that, given any such value sequence vl (say, produced from a system trace tr), any other value sequence vl' that coincides with vl on the open access windows (while being allowed be 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 values vl'.

The purpose of B is to capture this relationship between vl and vl', of coincidence on open access windows. But which part of a value sequence vl 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 notice text—namely, all blocks of vl that are immediately preceded by an OVal True value.[5]

But there are other values in the sequence that properly belong to this window: the last updated text before the access is open, that is, the value $\mathsf{TVal}\ \mathit{text}^k_{n_k}$ occurring *immediately before* each occurrence of OVal True. Indeed, for example, when the notice becomes public or user-visible, 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. Both predicates are defined mutually inductively as in Fig. 1. (We write @ for list concatenation and last for the function returning the last element in a list. Moreover, map TVal applied to a list $\mathit{textl} = [\mathit{text}_1, \ldots, \mathit{text}_n]$ of notice texts produces the list of corresponding values $[\mathsf{TVal}\ \mathit{text}_1, \ldots, \mathsf{TVal}\ \mathit{text}_n]$.)

[5] Recall from the definition of f that a value $\mathsf{OVal}\ b$ with b a Boolean marks the fact that the legitimate-access predicate open has just been made b.

	Source	**Declassification Bound**
1	Notice Text	Updates performed while or last before one of the following holds: User in coalition is notice owner or friend with notice owner Coalition has at least one user and notice is public or user-visible Coalition has at least one app and notice is public
2	Notice Title	Same as for Notice Text
3	Notice Image	Uploads performed while or last before one of the following holds: User in coalition is notice owner or friend with notice owner Coalition has at least one user and notice is public or user-visible
4	Friendship Status between two users	Status changes performed while or last before the following holds: User in coalition is friend with one of the two
5	Friendship Requests between two users	Existence of accepted requests while or last before the following holds: User in coalition is friend with one of the two

Fig. 2: CoSMed's confidentiality properties

Clause (1), the base case for B, describes the situation where the original system trace has made no change to the access status (w.r.t. the observers), which is initially entirely restricted. Here, the produced value sequence *vl* consists of text updates only, i.e., $vl = \mathsf{map\ TVal}\ text$. It is indistinguishable from any alternative sequence of updates $vl' = \mathsf{map\ TVal}\ text'$, save for the following corner case: An observer can learn that *vl* is empty, e.g., by inferring that a notice ID does not exist. Such harmless knowledge is factored in by asking that vl' (i.e., $textl'$) be empty whenever *vl* (i.e., *textl*) is.

Clause (2), the base case for BO, handles sequences of values produced during open access to the observers. Since here information is entirely exposed, the corresponding sequences of values from the alternative traces have to be identical to the original.

Clause (3), the inductive case for B, handles sequences of values $\mathsf{map\ TVal}\ textl$ produced while the access window is closed. The difference from clause (1) is that here we know that the window will eventually open—this is marked by the occurrences of $\mathsf{OVal\ True}$ in the conclusion, followed by a remaining value sequence *vl*. As previously discussed, the only constraint on the sequence of values produced by the alternative trace, $\mathsf{map\ TVal}\ textl'$, is that it ends in the same value—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 values produced during open access window on a trace known to eventually close the window.

With all the parameters in place, we have a formalization of notice text confidentiality, namely, the BD-security instance for these parameters. However, we saw that the open access windows need to be larger than initially suggested, hence (P2) is bogus as currently formulated. It needs to be refined by factoring in the last updates *before* access windows in addition to the updates performed *during* access windows. If we also factor in the generalization from a single user to a coalition of users and apps, we obtain:

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

3.4 More Confidentiality Properties

So far, we have discussed confidentiality for notice content (i.e., text). However, a notice also has a title and an image. In addition to notices, another type of information with confidentiality ramifications is that about friendship between users. The table in Fig. 2 summarizes all the confidentiality properties we have verified for CoSMed—where property 1 corresponds to notice content confidentiality (P3).

Notice titles (2) can be shown to have the same confidentiality status as contents. By contrast, notice images (3) are slightly more confidential, in that they are not accessible by apps—the bound is changed accordingly, by removing the condition involving apps.

The confidentiality of friendship status and friendship requests (4 and 5) have a structure similar to notice confidentiality. The observation setup consists of a coalition of users and apps, $(\mathsf{UIDs}, \mathsf{AIDs})$, and two arbitrary but fixed users $\mathsf{UID1}$ and $\mathsf{UID2}$ not belonging to the coalition. Since the system allows the listing of the friends of one's friends, observers can learn about the current friendship status between $\mathsf{UID1}$ and $\mathsf{UID2}$ (and updates to this status) once they become friends with one of them. Hence, the bound follows the same "while"-"last before" scheme as notice confidentiality. However, here the "produced" values are not text or image content, but rather:

- Boolean values for friendship status, indicating whether the status was changed to "friend" or "not friend", and
- pairs of user IDs and strings for friendship requests, indicating who has placed the request and what the message is.

Property (4) states that the friendship status of $\mathsf{UID1}$ and $\mathsf{UID2}$ remains unknown to non-friends. More precisely, the only information that flows to the coalition about this status consists of the "friending" and "unfriending" actions occurring while or last before one of the users in UIDs is friends with either $\mathsf{UID1}$ or $\mathsf{UID2}$.

Finally, property (5) states that friendship requests remain unknown to non-friends, and moreover, even to friends: their existence remains unknown unless they are accepted (hence turning into friendship); and their "orientation" (which of $\mathsf{UID1}$ and $\mathsf{UID2}$ has placed the request) and the messages remain unknown in any case.

4 Verifying Confidentiality

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

4.1 BD Unwinding Recalled

When proving a BD security property, we start with an "original" trace tr and an "alternative" value sequence vl' such that $\mathsf{B}\ (\mathsf{V}\ tr)\ vl'$ holds. We then need to provide an alternative trace tr' that never satisfies the trigger, produces the same observations as tr, and produces precisely the values vl.

To streamline this process, our BD security framework provides a notion of *unwinding à la* Goguen and Meseguer [11]. The idea is to construct tr' incrementally, in synchronization with tr, but "keeping an eye" on vl' as well. An unwinding relation [15, §5.1] is a tuple (s, vl, s', vl') representing a possible configuration of the unwinding synchronization game: s and vl represent the current state reached by a potential original

$$\begin{tikzcd}
 & & \varDelta_0 \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
\varDelta_1 & \varDelta_4 \arrow[l] \arrow[rr, leftrightarrow] & & \varDelta_3 \arrow[r] & \varDelta_2
\end{tikzcd}$$

Fig. 4: Graph of unwinding relations

$$\begin{tikzcd}
 & & \varDelta_0 \arrow[dll] \arrow[dl] \arrow[dr] \arrow[drr] & & \\
\varDelta_1 & \varDelta_4 \arrow[l] \arrow[rr] \arrow[drr, leftrightarrow] & & \varDelta_3^1 \arrow[d] & \varDelta_2 \\
 & & & \varDelta_3^2 \arrow[ur] &
\end{tikzcd}$$

Fig. 5: Refined graph

trace and the values that are still to be produced by it; and similarly for s' and vl' w.r.t. the alternative trace.

To keep proof size manageable, the framework supports interconnected unwinding relations, $\varDelta_0, \dots, \varDelta_n$. The unwinding conditions require that, from any such configuration for which one of the relations hold, say, $\varDelta_i\ s\ vl\ s'\ vl'$, 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. Namely, we require that one of the following holds:

INDEPENDENT ACTION: *There exists* an unobservable transition $trn' = \mathsf{Trans}\ s'\ \_\ \_\ t'$ leading to a tuple that is in one of the relations, $\varDelta_j\ s\ vl\ t'\ wl'$ for $j \in \{1, \dots, n\}$

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

- **IGNORE:** trn is unobservable (i.e., $\neg\ \gamma\ trn$) and again leads to a related tuple, $\varDelta_k\ t\ wl\ s'\ vl'$ for $k \in \{1, \dots, n\}$
- **MATCH:** There exists an equally observable transition $trn' = \mathsf{Trans}\ s'\ \_\ \_\ t'$ (i.e., such that $\gamma\ trn \leftrightarrow \gamma\ trn'$ and $\gamma\ trn \to g\ trn = g\ trn'$) that together with trn leads to a related tuple, $\varDelta_l\ t\ wl\ t'\ wl'$ for $l \in \{1, \dots, n\}$

Finally, we require that the initial relation $\varDelta_0$ is a proper generalization of the bound for the initial state, $\forall vl\ vl'.\ \mathsf{B}\ vl\ vl' \to \varDelta_0\ \mathsf{istate}\ vl\ \mathsf{istate}\ vl'$. This corresponds to initializing the game with a configuration that loads any two values satisfying the bound.

It is useful to think of the unwinding relations as forming a graph: For each i, $\varDelta_i$ is connected to all the relations into which it "unwinds," i.e., the relations $\varDelta_j$, $\varDelta_k$ or $\varDelta_l$ appearing in the above conditions.

Theorem 1 [15] If $\varDelta_0, \dots, \varDelta_n$ form a graph of unwinding relations then (the given instance of) BD security holds.

4.2 Unwinding Relations for CoSMed

In a graph $\varDelta_0, \dots, \varDelta_n$ of unwinding relations, $\varDelta_0$ generalizes the bound B. In turn, $\varDelta_0$ may unwind into other relations, and in general any relation in the graph may unwind into its successors. Hence, we can think of $\varDelta_0$ 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 Fig. 4 shows the unwindings of the notice-text confidentiality property (P3). In addition to the initial relation $\varDelta_0$, there are 4 relations $\varDelta_1$–$\varDelta_4$ with $\varDelta_i$ 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 $\mathsf{B}\ vl\ vl'$ 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 s and s' that fits the situation. Since we deal with repeated opening and closing of the access window, we naturally require:

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

Indeed, only when the window is open the observer would have the power to distinguish different values for NID's text; hence, when the window is closed the values 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 OVal True is reached, the NID text updates would have synchronized ($\mathsf{last}\ textl = \mathsf{last}\ textl'$), and therefore the relaxed equality $=_{\mathsf{NID}}$ 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. 5, distinguishing between clause (3) applied to nonempty update prefixes where we only need $s =_{\mathsf{NID}} s'$, covered by Δ_3^1, and clause (3) with empty update prefixes where we need $s = s'$, covered by Δ_3^2. Fig. 7 gives the formal definitions of the relations. Δ_0 covers the prehistory of NID—from before it was created. In Δ_1–Δ_4, the conditions on vl and vl' essentially incorporate the inversion rules corresponding to clauses (1)-(4) in B's definition, while the conditions on s and s' reflect the access conditions, as discussed.

Proposition 2 The relations in Fig. 7 form a graph of unwinding relations, and therefore (by Thm. 1) the notice-text confidentiality property (P3) holds.

5 Verification Summary

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

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

$$
\begin{aligned}
\Delta_0\ s\ vl\ s'\ vl' \equiv\ & \neg\ \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s = s' \\
\Delta_1\ s\ vl\ s'\ vl' \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s =_{\mathsf{NID}} s'\ \wedge\ \neg\ \mathsf{open}\ s\ \wedge \\
& \exists textl\ textl'.\ vl = \mathsf{map\ TVal}\ textl\ \wedge\ vl' = \mathsf{map\ TVal}\ textl'\ \wedge \\
& \quad textl = [] \rightarrow textl' = [] \\
\Delta_2\ s\ vl\ s'\ vl' \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s = s'\ \wedge\ \mathsf{open}\ s\ \wedge \\
& \exists textl.\ vl = \mathsf{map\ TVal}\ textl\ \wedge\ vl' = \mathsf{map\ TVal}\ textl \\
\Delta_3^1\ s\ vl\ s'\ vl' \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s =_{\mathsf{NID}} s'\ \wedge\ \neg\ \mathsf{open}\ s\ \wedge \\
& \exists textl\ textl'\ wl\ wl'.\ vl = \mathsf{map\ TVal}\ textl\ @\ \mathsf{OVal\ True}\ \#\ wl\ \wedge \\
& \quad vl' = \mathsf{map\ TVal}\ textl'\ @\ \mathsf{OVal\ True}\ \#\ wl'\ \wedge \\
& \quad \mathsf{BO}\ wl\ wl'\ \wedge\ textl \neq []\ \wedge\ textl' \neq []\ \wedge\ \mathsf{last}\ textl = \mathsf{last}\ textl' \\
\Delta_3^2\ s\ vl\ s'\ vl' \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s = s'\ \wedge\ \neg\ \mathsf{open}\ s\ \wedge \\
& \exists wl\ wl'.\ vl = \mathsf{OVal\ True}\ \#\ wl\ \wedge\ vl' = \mathsf{OVal\ True}\ \#\ wl'\ \wedge\ \mathsf{BO}\ wl\ wl' \\
\Delta_4\ s\ vl\ s'\ vl' \equiv\ & \mathsf{NID} \in \mathsf{noticeIDs}\ s\ \wedge\ s = s'\ \wedge\ \mathsf{open}\ s\ \wedge \\
& \exists textl\ wl\ wl'.\ vl = \mathsf{map\ TVal}\ textl\ @\ \mathsf{OVal\ False}\ \#\ wl\ \wedge \\
& \quad vl' = \mathsf{map\ TVal}\ textl'\ @\ \mathsf{OVal\ False}\ \#\ wl'\ \wedge\ \mathsf{B}\ wl\ wl'
\end{aligned}
$$

Fig. 7: The unwinding relations for notice-text confidentiality

6 Related Work

Proof assistants are today's choice for *precise* and *holistic* formal verification of hardware and software systems. Already legendary verification works are the AMD microprocessor floating-point operations [21], the CompCert C compiler [18] and the seL4 operating system kernel [16]. More recent developments include a range of microprocessors [13], Java and ML compilers [17, 19], and a model checker [10].

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 [20]. They include a hardware architecture with information-flow primitives [9] and a separation kernel [8], and noninterference for seL4 [22]. A substantial contribution to web client security is the Quark verified browser [14]. 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.

Outside the realm of proof-assistant based work, ConfiChair [4] is a competitor for CoCon verified using the ProVerif process algebra tool. It proposes a cloud model where authors and reviews cannot be linked to their documents—not even by the system's administrator. Finally, there are quite a few programming languages and tools aimed at supporting information-flow secure programming such as Jif or Spark, including web programming specifically [7], as well as information-flow tracking tools for the client side. We foresee a future where such tools will cooperate with proof assistants to offer light-weight guarantees for free and stronger guarantees (like the ones we proved for CoCon and CoSMed) on a by-need bases.

Conclusion CoSMed is the first social media platform with confidentiality guarantees. Its verification is based on BD security, a framework for information-flow security formalized in Isabelle. CoSMed's specific confidentiality needs have challenged this framework and brought us more insight into how to instantiate it.

Acknowledgement. We gratefully acknowledge support from UK Innovate through KTP 010041, from DFG through grants Ni 491/13-3 and Hu 737/5-3 and from EPSRC through grant EP/N019547/1.

References

1. The Heartbleed bug. `http://heartbleed.com/`.
2. OWASP top ten project. `www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013`.
3. Caritas Anchor House. `http://caritasanchorhouse.org.uk/`, 2016.
4. M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *POST*, pp. 89–108, 2012.
5. T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi. The CoSMed website. `https://cosmed.globalnoticeboard.com`, 2016. Anonymous access possible with username "demo" and password "demo".
6. T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi. The Isabelle formalization of CoSMed. `http://andreipopescu.uk/cosmed.zip`, 2016.
7. A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pp. 153–165, 2015.
8. 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.
9. A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL*, pp. 165–178, 2014.
10. 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.
11. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pp. 75–87, 1984.
12. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117, 2010.
13. 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.
14. D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In T. Kohno, ed., *USENIX Security '12*, pp. 113–128. USENIX, 2012.
15. S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *CAV*, pp. 167–183, 2014.
16. 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.
17. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL*, pp. 179–192, 2014.
18. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
19. A. Lochbihler. Java and the Java memory model—A unified, machine-checked formalisation. In *ESOP*, pp. 497–517, 2012.
20. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 605–607. 2011.
21. 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.
22. 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 *SP 2013*, pp. 415–429, 2013.
23. T. Nipkow and G. Klein. *Concrete Semantics. A Proof Assistant Approach*. December 2014. 310 pp. `http://www.in.tum.de/~nipkow/Concrete-Semantics`.
24. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
25. D. Sutherland. A model of information. In *9th National Security Conf.*, pp. 175–183, 1986.

APPENDIX: More Details on the Verified Properties

A Confidentiality

We already explained the notice confidentiality properties in detail in §3.3. We now provide more details on the friendship status and request confidentiality (properties 4 and 5 of Figure 2). As explained in §3.4, we consider these properties wrt. the friendship of two arbitrary but fixed users UID1 and UID2, while UIDs and AIDs are coalitions of observing users and apps, respectively.

We define the access window to the friendship information to be *open* if either an observer is friends with UID1 or UID2, or if the two users have not been created yet (because observers know statically that there is no friendship if the users do not exist yet).

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

The relevant transitions for the value setup are the creation of users (for the openness) and the creation and deletion of friends (and friend requests in the case of P5). The creation and deletion of friendship between UID1 and UID2 produces an FVal True or FVal False value, respectively. In the case of openness changes, and OVal is produced just as for the notice confidentiality. Moreover, for the friend request confidentiality, we let cFriendReq transitions involving UID1 and UID2 produce FRVal *uid text* values, where *uid* indicates the user that has placed the request, and *text* is the request message.

$$\mathsf{BO}\ (\mathsf{map}\ \mathsf{FVal}\ fs)\ (\mathsf{map}\ \mathsf{FVal}\ fs)\ (1) \qquad \mathsf{BC}\ (\mathsf{map}\ \mathsf{FVal}\ fs)\ (\mathsf{map}\ \mathsf{FVal}\ fs')\ (2)$$

$$\frac{\mathsf{BO}\ vl\ vl' \qquad fs \neq [] \leftrightarrow fs' \neq [] \qquad fs \neq [] \rightarrow \mathsf{last}\ fs = \mathsf{last}\ fs'}{\mathsf{BC}\ (\mathsf{map}\ \mathsf{FVal}\ fs\ @\ \mathsf{OVal}\ \mathsf{True}\ @\ vl)\ (\mathsf{map}\ \mathsf{FVal}\ fs'\ @\ \mathsf{OVal}\ \mathsf{True}\ @\ vl')}\ (3)$$

$$\frac{\mathsf{BC}\ vl\ vl'}{\mathsf{BO}\ (\mathsf{map}\ \mathsf{FVal}\ fs\ @\ \mathsf{OVal}\ \mathsf{False}\ @\ vl)\ (\mathsf{map}\ \mathsf{FVal}\ fs\ @\ \mathsf{OVal}\ \mathsf{False}\ @\ vl')}\ (4)$$

Fig. 8: The bound on friendship status values

The main inductive definition of the two phases of the declassification bounds for (P4) and (P5) is given in Figure 8. Note that it follows the same "while"-"last before" scheme as Figure 1 for the notice confidentiality, but with FVal instead of TVal. The overall bounds are then defined as BO *vl vl′* (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. For (P4), the predicate says that the FVal's form an *alternating* sequence of "friending" and "unfriending". For (P5), it additionally requires that at least one FRVal and at most two FRVal values from different users have to occur before each FVal True value. Beyond that, we require nothing about the request values. Hence, the bound for (P5) states that observers learn nothing about the friendship 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 (who has placed them) and the contents of the request messages, as summarized already in §3.4.

For unwinding the friendship confidentiality properties, we proceed analogously to the notice confidentiality. We define unwinding relations, corresponding to the different clauses in Figure 8, and prove that they unwind into each other and that B $vl\ vl'$ implies Δ_0 istate vl istate vl'. 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, but with the invariant that if an OVal True value follows later in the value sequence, then either the status has to be same in the two states, or the last updates before OVal True must be equal, as defined in clause (3) of Figure 1. This allows us to converge back to the same friendship status before the transition into the open phase.

B Safety

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

1. The owner of an existing notice 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.

C Accountability

For friendship status, we have 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)

We have formally stated this 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 time of the relevant friend request (from *uid* to uid' or vice versa)
– *trnn* represents the time of the approval of this request
– tr_3 contains no successful unfriending action between the two users

For notice visibility, we have proved an accountability property similar to friend status accountability: If, at some point *t* on a system trace, the visibility of a notice *nid* has a value *vis*, then one of the following holds:
– Either *vis* is FriendV (as in the initial state)
– Or the notice's owner had issued a successful "update visibility" action setting the visibility of *nid* to *vis*, and no other successful update actions to *nid*'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 time the notice was created by some user *uid* (who becomes the owner)
– *trnn* represents the (last) update of *nid*'s visibility to *vis* (necessarily by *uid*)
– tr_3 contains no successful update to the notice visibility