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

CoCon: A Conference Management System with Formally Verified Document Confidentiality

Ping Hou · **Peter Lammich** · **Andrei Popescu**

the date of receipt and acceptance should be inserted later

Abstract We present a case study in formally verified security for realistic systems: the information flow security verification of the functional kernel of a web application, the CoCon conference management system. CoCon's kernel is implemented in the Isabelle theorem prover, where we specify and verify confidentiality properties, as well as complementary safety and accountability properties. The information flow expressiveness challenges posed by this development have led to *bounded-deducibility (BD) security*, a novel security model and verification method generally applicable to systems describable as input/output automata.

1 Introduction

Information flow security is concerned with preventing or facilitating (un)desired flow of information in computer systems, covering aspects such as confidentiality, integrity, and availability of information. Dieter Gollmann wrote in 2005 [25]: "Currently, information flow and noninterference models are areas of research rather than the bases of a practical methodology for the design of secure systems." The situation has undergone steady improvements in the past thirteen years. A number of practical systems, some of which are surveyed by Murray et al. [54], have been formally certified for information flow security—covering hardware, operating systems, programming languages, web browsers and web applications.

This paper gives a detailed presentation of the verification work that went into one such system. CoCon is a full-fledged conference management system, featuring multiple users and conferences and offering much of the functionality of widely used systems such as EasyChair [18] and HotCRP [19].

Ping Hou
Department of Computer Science, Middlesex University London, UK E-mail: phou@cs.cmu.edu

Peter Lammich
Fakultät für Informatik, Technische Universität München, Germany E-mail: lammich@in.tum.de

Andrei Popescu
Department of Computer Science, Middlesex University London, UK and Institute of Mathematics Simion Stoilow of the Romanian Academy, Romania E-mail: a.popescu@mdx.ac.uk

WARNING: HotCRP version 2.47 (commit range 94ca5a0e43bd7dd0565c2c8dc7d8f710a206ab49 through 9c1b45475411ecb85d46bad1f76064881792b038) was subject to an information exposure where some authors could see PC comments. Users of affected versions should upgrade or set the following option in `Code/options.inc: $Opt["disableCapabilities"] = true;`

Fig. 1: Confidentiality bug in HotCRP

Conference management systems are widely used in the scientific community. EasyChair alone claims more than two million users. Moreover, the information flow in these systems possesses enough complexity so that errors can sneak inside implementations, sometimes with bitter–comical consequences. In 2012, Popescu, as well as the authors of 267 papers submitted to a major security conference, initially received an acceptance notification, followed by a retraction [35]: "We are sorry to inform you that your paper was not accepted for this year's conference. We received 307 submissions and only accepted 40 of them … We apologize for an earlier acceptance notification, due to a system error."[1]

The above is an information integrity violation (a distorted decision was initially communicated to the authors) and could have been caused by a human error rather than a system error—but there is the question whether the system should not prevent even such human errors. The problem with a past version of HotCRP [19] shown in Fig. 1 is even more interesting: it describes a genuine confidentiality violation, probably stemming from the logic of the system, giving the authors capabilities to read confidential comments by the program committee (PC).

Although our methods would equally apply to integrity violations, guarding against confidentiality violations is the focus of this verification work. We verify properties such as the following (where DIS addresses the problem in Fig. 1):

- PAP_1: A group of users learn nothing about a paper unless one of them becomes an author of that paper or becomes a PC member at the paper's conference and the conference has reached the bidding phase
- PAP_2: A group of users learn nothing about a paper *beyond the last submitted version* unless one of them becomes an author of that paper
- REV: A group of users learn nothing about the content of a paper's review *beyond the last submitted version before the discussion phase and the later versions* unless one of them is that review's author
- DIS: The authors learn nothing about the discussion of their paper

In general, we will be concerned with properties restricting the information flow from the various sensitive documents maintained by the system (papers, reviews, comments, decisions) towards the users of the system. The restrictions refer to certain conditions (e.g., authorship, PC membership) as well as to upper bounds (e.g., at most the last submitted version) for information release.

Here is the structure of this paper. We start with a high-level presentation of the system architecture and its verified and trusted components (Section 2), after which we delve into the Isabelle [56, 57] specification of the system's kernel as an executable input/output (I/O) automaton (Section 3).

Then we move to describing the first main contribution of this paper: a novel security model called *bounded-deducibility (BD) security*, born from confronting notions from

[1] After reading the initial acceptance notification, Popescu went out to celebrate; it was only hours later when he read the retraction.

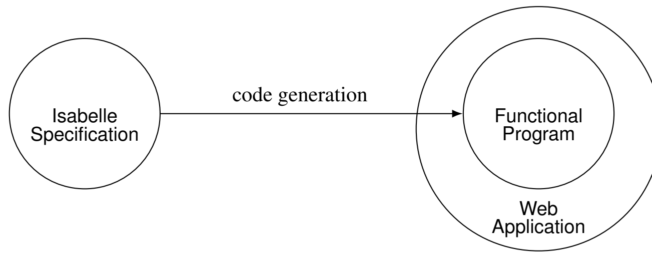

Fig. 2: System Architecture

the literature with the challenges posed by our system (Section 4). The result is a reusable framework, applicable to any I/O automaton. Its main novelty is wide flexibility: it allows the precise formulation of role-based and time-based declassification triggers and of declassification upper bounds. The framework is instantiated to provide a comprehensive coverage of CoCon's confidentiality properties of interest, including the ones discussed in this introduction. To address information flow security concerns more thoroughly, we discover the need for a form of accountability properties (Section 4.6), which naturally complement BD security by showing that the declassification triggers cannot be forged.

The second main contribution is a verification infrastructure for BD security, centered around an unwinding proof technique (Section 5), which we have deployed for CoCon's confidentiality properties. In the process of verifying confidentiality, we need to prove as lemmas several safety properties (system invariants). The formal Isabelle scripts, covering both the abstract framework and the CoCon instances, are made available from this paper's website [34].

The current paper is an extended version of a conference paper presented at CAV 2014 [37]. In addition to the material in the conference paper, it includes:

- the detailed description of some of the verified confidentiality properties (in Section 4.4) and of the unwinding relations used in their verification (in Section 5.3)
- a presentation of the accountability properties (as the newly added Section 4.6)
- the full definition of the abstract unwinding conditions (in Section 5.1) and their compositionality oriented theorems (in Section 5.2)
- an up-to-date discussion of related work (as the newly added Section 6)

2 Overall Architecture and Security Guarantees

The architecture of our system (Fig. 2) follows the paradigm of security by design:

- We formalize and verify the kernel of the system in the Isabelle proof assistant
- The formalization is automatically translated into a functional programming language
- The translated program is wrapped in a web application

Isabelle Specification We specify CoCon's kernel as an I/O automaton (Mealy machine), with the inputs called "actions." We first define, using Isabelle's records, the notions of state (holding information about users, conferences, and papers) and user action (representing user requests for manipulating documents and rights in the system: upload and download

papers, edit reviews, assign reviewers, etc.). Then we define the step function that takes a state and an action and returns a new state and an output.

Scala Functional Program The specification was designed to fall within the executable fragment of Isabelle. This allows us to automatically synthesize, using Isabelle's code generator [30], a program in the functional fragment of Scala [3], which is isomorphic to the specification. The types of data used in the specification (numbers, strings, tuples, records) are mapped to the corresponding Scala types. An exception is the Isabelle type of paper contents, which is mapped to the Scala/JVM file type.

Web Application Finally, the Scala program is wrapped in a web application, offering a menu-based user interface. Upon login, a user sees his conferences and his roles for each of them; the menus offer role-sensitive choices, e.g., assign reviewers (for chairs) or upload papers (for authors).

Verified and Trusted Components Our Isabelle verification targets information flow properties of CoCon's kernel. These properties express that, for any possible trace of the system, there is no way to infer from certain observations on that trace (e.g., actions performed by designated users), certain values extracted from that trace (e.g., the paper uploads by other users). In order for these guarantees to apply to the overall system (the entire web application), there are a few components that we need to trust (or, in the future, verify).

First, we need to trust Isabelle's code generator. Its general-purpose design is quite flexible, supporting program and data refinement [30]. In the presence of these rich features, the code generator is only known to preserve partial correctness, hence safety properties [29,30]. However, here we use the code generator in a very restrictive manner, to "refine" an already deterministic specification which is an implementation in its own right—the code generator simply translates it from the functional language of Isabelle to that of Scala. In addition, all the used Isabelle functions are proved to terminate, and nontrivial data refinement is disabled. These allow us to (informally) conclude that the synthesized implementation is trace-isomorphic to the specification, hence the former leaks as little information as the latter. (This meta-argument does not cover timing channels, but these seem to be of little importance for leaking document content.)

Second, we need to trust that no further leakage occurs via the web application wrapper, which acts mostly as a stateless interface to the step function: upon a user request, it invokes this step function (once or multiple times) with input from the user and then processes and displays the output of the step function. This constitutes a fundamentally safe interaction with the verified kernel. An exception to statelessness is the user identity management component, which performs the password hashing and stores session tokens as an outside-of-kernel state. This trusted component is of course critical to the overall confidentiality guarantees. The third-party libraries used by our web application also have to be trusted not to be vulnerable to exploits.

Finally, our verification targets only the server side implementation logic—lower-level attacks, as well as browser-level forging are out of its reach, but are orthogonal issues that could in principle be mitigated separately.

3 System Specification

CoCon is inspired by EasyChair [18], a popular conference system created by Andrei Voronkov. It hosts multiple users and conferences, allowing the creation of new users and conferences at any time. It has a superuser, which we call *voronkov* as a tribute to EasyChair. The

voronkov is the first user of the system, and his role is to approve new-conference requests. A conference goes through several phases.

No-Phase Any user can apply for a new conference, with the effect of registering it in the system as initially having "no phase." After approval from the voronkov, the conference moves to the setup phase, with the applicant becoming a conference chair.
Setup A conference chair can add new chairs and new regular PC members. From here on, moving the conference to successor phases can be done by the chairs.

Submission Any user can list the conferences awaiting submissions (i.e., being in the submission phase). He can submit a paper, upload new versions, or indicate other users as coauthors thereby granting them reading and editing rights.

Bidding Authors are no longer allowed to upload or register new papers, and PC members are allowed to view the submitted papers. PC members can place bids, indicating for each paper one of the following preferences: "want to review", "would review", "no preference", "would not review", and "conflict". If the preference is "conflict", the PC member cannot be assigned that paper, and will not see its discussion. "Conflict" is assigned automatically to papers authored by a PC member.

Reviewing Chairs can assign papers to PC members for reviewing either manually or by invoking an external program to establish fair assignment based on some parameters: preferences, number of papers per PC member, and number of reviewers per paper. The assigned reviewers can edit their reviews.

Discussion All PC members having no conflict with a paper can see its reviews and can add comments. The reviewers can still edit their reviews, but in a transparent manner—so that the overwritten versions are still visible to the non-conflict PC members. Also, chairs can edit the decision.

Notification The authors can read the reviews and the accept/reject decision, which no one can edit any longer.

Closing The conference becomes inactive. All users can still read the documents previously readable, but nothing is editable any longer.

3.1 State, Output, Actions, and Step Function

The **state** stores the lists of registered conference, user, and paper IDs and, for each ID, actual conference, user, or paper information. For user IDs, the state also stores (hashed) passwords. In the context of a conference, each user is assigned one or more of the roles described by the following Isabelle datatype:

```
datatype Role = Chair | PC | Aut PaperID | Rev PaperID Nat
```

with the following meanings, assuming *pid* is a paper ID and *n* is a number:

Chair: The user is a chair of the conference
PC: The user is a member of the program committee
Aut *pid*: The user is an author of the paper with ID *pid*
Rev *pid n*: The user is the *n*th reviewer of the paper with ID *pid*

In a state, each paper ID is assigned a paper having title, abstract, content, and, in due time, a list of reviews, a discussion text, and a decision. We keep different versions

of the decision and of each review, as they may transparently change during the discussion phase. This means that a decision is a list of strings representing its different versions, $\mathsf{Dec} = \mathsf{List}\,(\mathsf{String})$. Similarly, a review is a list of review contents representing its different versions, $\mathsf{Review} = \mathsf{List}\,(\mathsf{Review_Content})$, where Review_Content consists of triples (expertise, text, score).

In addition, the state stores: for each conference, the list of (IDs of) papers submitted to that conference, the list of news updated by the chairs, and the current phase; for each user and paper, the preferences resulted from biddings; for each user and conference, a list of roles. We will mainly manipulate the roles through discriminators. For example, isPC *s cid uid* returns True just in case in state *s* the user *uid* is a PC member for conference *cid*. Here is the formal structure of the state:

```
record State =
    confIDs : List (ConfID)                 conf : ConfID → Conf
    userIDs : List (UserID)                 pass : UserID → Pass
    user : UserID → User                    roles : ConfID → UserID → List (Role)
    paperIDs : ConfID → List (PaperID)      paper : PaperID → Paper
    pref : UserID → PaperID → Pref          voronkov : UserID
    news : ConfID → List (String)           phase : ConfID → Phase
```

The **initial state** of the system, $\mathsf{istate} \in \mathsf{State}$, is the one with a single user, the voronkov, and no conferences.

```
istate =
    confIDs = []                     conf = (λ cid. emptyConf)
    userIDs = [“voronkov”]           pass = (λ uid. emptyPass)
    user = (λ uid. emptyUser)        roles = (λ cid uid. [])
    paperIDs = (λ cid. [])           paper = (λ pid. emptyPaper)
    pref = (λ uid pid. NoPref)       voronkov = “voronkov”
    news = (λ cid. [])               phase = (λ cid. noPh)
```

Actions are parameterized by user IDs and passwords. There are 45 actions forming five categories: creation, update, undestructive update (u-update), reading and listing.

The **creation actions** register new objects (users, conferences, chairs, PC members, papers, authors), assign reviewers (by registering new review objects), and declare conflicts. For example, cPaper *cid uid pw pid title abs* is an action by user *uid* with password *pw* attempting to register to conference *cid* a new paper *pid* with indicated title and abstract. Moreover, cAuthor *cid uid pw pid* uid' expresses an attempt of user *uid* with password *pw* to create a new (co)author for the paper *pid* in the context of the conference *cid*—namely, to set the user uid' as this new author.

The **update actions** modify the various documents of the system: user information and password, paper content, reviewing preference, review content, etc. For example, uPaperC *cid uid pw pid pct* is an attempt by user *uid* with password *pw* to upload a new version of paper *pid* by modifying its content to *pct*.

The **u-update actions** are similar, but also record the history of a document's versions. For example, if a reviewer decides to change his review during the discussion phase, then the previous version is still stored in the system and visible to the other PC members (although never to the authors). Other documents subject to u-updates are the news, the discussion, and the accept-reject decision.

The **reading actions** access the content of the system's documents: papers, reviews, comments, decisions, news. The **listing actions** produce lists of IDs satisfying various

filters—e.g., all conferences awaiting paper submissions, all PC members of a conference, all the papers submitted by a given user, etc.

The different categories of actions are wrapped in a single datatype through specific constructors:

$$\texttt{datatype}\ \mathsf{Act}\ = \mathsf{Cact}\ \mathsf{cAct} \mid \mathsf{Uact}\ \mathsf{uAct} \mid \mathsf{UUact}\ \mathsf{uuAct} \mid \mathsf{Ract}\ \mathsf{rAct} \mid \mathsf{Lact}\ \mathsf{lAct}$$

Note that the first three categories of actions are aimed at *modifying* the state, and the last two are aimed at *observing* the state through outputs. However, the modification actions also produce a simple output, since they may succeed or fail. Moreover, the observation actions can also be seen as changing the state to itself. Therefore we can assume that both types produce a pair consisting of a new state and an output.

Outputs include some generic output types, like outOK for a successful update action and outErr for a failed action. Moreover, outputs for various datatypes are defined, e.g., for Booleans, lists of strings, lists of pairs of strings, etc. Similarly to the case of actions, all these types of outputs are wrapped together in a single type Out.

Finally, we define the **step function** $\mathsf{step} : \mathsf{State} \to \mathsf{Act} \to \mathsf{Out} \times \mathsf{State}$ that operates by determining the type of the action and dispatching specialized handler functions. We illustrate the definition of step by zooming into one of its subcases:

$$
\begin{array}{l}
\mathsf{step}\ s\ a\ \equiv \\
\texttt{case}\ a\ \texttt{of}\ \mathsf{Cact}\ ca\ \Rightarrow\ \texttt{case}\ ca\ \texttt{of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{cAuthor}\ cid\ uid\ pw\ pid\ uid'\ \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad \texttt{if}\ \mathsf{e_createAuthor}\ s\ cid\ uid\ pw\ pid\ uid' \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{then}\ (\mathsf{outOK}, \mathsf{createAuthor}\ s\ cid\ uid\ pw\ pid\ uid') \\
\qquad\qquad\qquad\qquad\qquad\quad \texttt{else}\ (\mathsf{outErr}, s) \\
\qquad\qquad\qquad\qquad\quad\ \mid \mathsf{cConf}\ cid\ uid\ pw\ name\ abs\ \Rightarrow \ldots \\
\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\quad \mid \mathsf{Uact}\ ua\ \Rightarrow\ \ldots \\
\qquad\quad \mid \mathsf{UUact}\ uua\ \Rightarrow\ \ldots \\
\qquad\quad \mid \mathsf{Ract}\ ra\ \Rightarrow\ \ldots \\
\qquad\quad \mid \mathsf{Lact}\ la\ \Rightarrow\ \ldots
\end{array}
$$

Above, we only showed one subcase of the creation-action case in full. The semantics of each type of action (e.g., cAuthor, which is itself a subtype of creation actions) has an associated test for enabledness (here, e_createAuthor) and an effect function (here, createAuthor).

The enabledness test checks if it is allowed to perform the requested action: if the IDs of the involved users and conferences exist (expressed by a generic predicate IDsOK), if the password matches the acting user's ID, if the conference phase is appropriate, if the acting user holds the appropriate role, etc.:

$$
\begin{array}{l}
\mathsf{e_createAuthor}\ s\ cid\ uid\ pw\ pid\ uid'\ \equiv \\
\quad \mathsf{IDsOK}\ s\ [cid]\ [uid, uid'][pid]\ \wedge\ \mathsf{pass}\ s\ uid = pw\ \wedge \\
\quad \mathsf{phase}\ s\ cid = \mathsf{Submission}\ \wedge\ \mathsf{isAut}\ s\ uid\ pid\ \wedge\ uid \neq uid'
\end{array}
$$

The effect is only applied if the action is enabled; otherwise an error output is issued. In this example, the effect is to add an author uid' to the existing paper pid, as well as a conflict in the system database between the author and the paper:

$$
\begin{array}{l}
\mathsf{createAuthor}\ scid\ uid\ pw\ pid\ uid'\ \equiv \\
\quad \texttt{let}\ \mathsf{rls} = \mathsf{roles}\ s\ cid\ uid'\ \texttt{in} \\
\quad s\ (\mathsf{roles} := \mathsf{fun_upd}_2\ (\mathsf{roles}\ s)\ cid\ uid'\ (\mathsf{insert}\ (\mathsf{Aut}\ pid)\ \mathsf{rls}), \\
\qquad \mathsf{pref} := \mathsf{fun_upd}_2(\mathsf{pref}\ s)\ uid'\ pid\ \mathsf{Conflict})
\end{array}
$$

To the outside world, i.e., to the web application wrapper, our specification only exports the initial state $\mathsf{istate} : \mathsf{State} \to \mathsf{bool}$ and the step function $\mathsf{step} : \mathsf{State} \to \mathsf{Act} \to \mathsf{Out} \times \mathsf{State}$, i.e., it exports an I/O automaton.

4 Security Model

In what follows, we first analyze the literature for possible inspiration concerning a suitable security model for our system. Then we introduce our own notion, which is an extension of Sutherland's nondeducibility [66] that factors in declassification triggers and bounds.

4.1 A Look at the Relevant Literature

There is a vast amount of literature on information flow security models, with many variants of formalisms and verification techniques. An important distinction is between models that completely forbid information flow (between designated sources and sinks) and models that only restrict the flow, allowing some declassification, i.e., controlled information release. Historically, the former were introduced first, and the latter were subsequently introduced as generalizations.

Absence of Information Flow The information flow security literature starts in the late 1970s and early 1980s [14, 23, 59], motivated by the desire to express the absence of information leaks of systems more abstractly and more precisely than by means of access control [7, 40].

Applied to our case of interest, the debate concerning information flow control versus access control can be summarized as follows: Wouldn't properties such as "only users with a certain role can *access* a certain *data*" suffice, where the data is identified as a particular state component, such as a stored document? In other words, isn't the simpler *access data* a good substitute for *learn information*? More than twenty years ago, the security community has decisively established that the answer is "no"—McLean [49] gives a good early summary of the debate and its conclusion, which was reinforced by the subsequent abundant literature (including the one cited below). Indeed, while access control properties are partially reassuring, no collection of such properties can offer the level of assurance achieved by factoring in genuine information flow in the statements.

For example, proving that an author learns nothing about his reviews before the notification phase represents *much more* than proving that an author cannot access his reviews before the notification phase. Unlike the latter, the former is a global property of the system that excludes in one swoop a whole variety of potential leaks. Here is one leaking scenario: The PC members are shown all the papers, but the scores of the papers with which they have conflict are omitted; moreover, the PC members clearly have conflict with their own (authored) papers. But what if the current average score of the papers' reviews is used to determine the order in which the papers are listed? Then a PC member may learn the current average score for their authored paper with high accuracy—without accessing the review documents. An ad hoc access control property can be designed to cope with this particular scenario, but there is an endless supply of such scenarios, which an information flow property would exclude without having to consider explicitly.

Influential early contributions to information flow security were Goguen and Meseguer's notion of noninterference [23] and its associated proof by unwinding [24]. Unwinding is essentially a form of simulation that allows one to construct incrementally, from a perturbed

trace of the system, an alternative "corrected" trace that "closes the leak." Many other notions were introduced subsequently, either in specialized programming-language-based [63] or process-algebra-based [21, 62] settings or in purely semantic, event-system-based settings [47, 48, 58, 66]. (Here we are mostly interested in the last category.) These notions are aimed at extending noninterference to nondeterministic systems, closing Trojan-horse channels, or achieving compositionality. The unwinding technique has been generalized for some of these variants—McLean [49] and Mantel [45] give overviews.

Even ignoring our aimed declassification aspect, most of these notions do not adequately model our properties of interest exemplified in the introduction. One problem is that they are not flexible enough w.r.t. the observations. They state nondetectability of the presence or absence of certain events anywhere in a system trace. By contrast, we are interested in a very controlled positioning of such undetectable events: in the property PAP_2 from the introduction, the unauthorized user should not learn of preliminary (non-final) uploads of a paper. Moreover, we are not interested in whole events, but rather in certain relevant values extracted from the events: e.g., the content of the paper, and not the ID of one of the particular authors who uploads it.

A fortunate exception to the above flexibility problems is Sutherland's early notion of *nondeducibility* [66]. One considers a set of worlds World and two functions $F : \mathsf{World} \to J$ and $H : \mathsf{World} \to K$. For example, the worlds could be the valid traces of the system, F could select the actions of certain users (potential attackers), and H could select the actions of other users (intended as being secret). *Nondeducibility of H from F* says that the following holds for all $w \in \mathsf{World}$: for all k in the image of H, there exists $w_1 \in \mathsf{World}$ such that $F\ w_1 = F\ w$ and $H\ w_1 = k$. Intuitively, from what the attacker (modeled as F) knows about the actual world w, the secret actions (the value of H) could be anything (in the image of H)—hence cannot be "deduced." The generality of this framework allows one to fine-tune both the location of the relevant events in the trace and their values of interest. But generality is no free lunch: it is no longer clear how to provide an unwinding-like incremental proof method.

Halpern and O'Neill [31] recast nondeducibility as a property called secrecy maintenance, in a multi-agent framework of "runs-and-systems" [60] based on epistemic logic. Their formulation enables general-purpose epistemic logic primitives for deducing absence of leaks, but no unwinding or any other inductive reasoning technique.

Restriction of Information Flow A large body of work on declassification was pursued in a language-based setting. Sabelfeld and Sands [64] give an overview of the state of the art up to 2009. Although they target language-based declassification, they identify some generic dimensions of declassification that apply to our case:

- What information is released? Here, document content, e.g., of papers, reviews, etc.
- Where in the system is information released? In our case, the relevant "where" is a "from where" (referring to the source, not to the exit point): from selected places in the system trace, e.g., the last submitted version before the deadline.
- When can information be released? After a certain trigger occurs, e.g., authorship.
- Who releases the information? The users who are entitled, e.g., the authors of a document.

Sabelfeld and Sands consider another interesting instance of the "where" dimension, namely intransitive noninterference [44, 61]. This is an extension of noninterference that allows downgrading of information, say, from High to Low, via a controlled Declassifier level. It could be possible to encode aspects of our properties of interest as intransitive noninterference, e.g., we could encode the act of a user becoming an author as a declassifying action for the target paper. However, such an encoding would be rather technical and somewhat

artificial for our system; additionally, it is not clear how to factor in our aforementioned specific "where" dimension.

The "when" aspect of declassification has been included as first-class citizen in customized temporal logics [13, 17], which can express aspects of our desired properties, e.g., "unless/until he becomes an author," but do not have the flexibility to capture some of the fine-grained aspects, such as the "beyond" component of PAP$_2$ and REV.

4.2 Bounded-Deducibility Security

We introduce a novel notion of information flow security that:

- retains the precision and versatility of nondeducibility
- factors in declassification as required by our motivating examples
- is amenable to a general unwinding technique

We shall formulate security in general, not only for our concrete system from Section 3.1, but for any I/O automaton indicated by the following data, which will be considered fixed throughout this subsection: the sets of states, State, actions, Act, and outputs, Out, an initial state $\mathsf{istate} \in \mathsf{State}$, and a step function $\mathsf{step} : \mathsf{State} \to \mathsf{Act} \to \mathsf{Out} \times \mathsf{State}$.

We let Trans, the set of *transitions*, be $\mathsf{State} \times \mathsf{Act} \times \mathsf{Out} \times \mathsf{State}$. Thus, a transition trn is a tuple, written (s, a, o, s'); s indicates the source, a the action, o the output, and s' the target. trn is called valid if it is induced by the step function, namely $\mathsf{step}\ s\ a = (o, s')$.

A *trace* $tr \in \mathsf{Trace}$ is any list of transitions: $\mathsf{Trace} = \mathsf{List}\ (\mathsf{Trans})$. For any $s \in \mathsf{State}$, the set of valid traces starting in s, $\mathsf{Valid}_s \subseteq \mathsf{Trace}$, consists of the traces of the form $[(s_1, a_1, o_1, s_2), (s_2, a_2, o_2, s_3), \ldots, (s_{n-1}, a_{n-1}, o_n, s_n)]$ for some n, where $s_1 = s$ and each transition (s_i, a_i, o_i, s_i) is valid. We will be interested in the valid traces starting in the initial state istate—we simply call these *valid traces* and write Valid for $\mathsf{Valid}_{\mathsf{istate}}$.

Besides the I/O automaton, we assume that we are given the following data:

- a *value domain* Val, together with a *value filter* $\varphi : \mathsf{Trans} \to \mathsf{Bool}$ and a *value producer* $f : \mathsf{Trans} \to \mathsf{Val}$
- an *observation domain* Obs, together with an *observation filter* $\gamma : \mathsf{Trans} \to \mathsf{Bool}$ and an *observation producer* $g : \mathsf{Trans} \to \mathsf{Obs}$

We define the *value function* $\mathsf{V} : \mathsf{Trace} \to \mathsf{List}\ (\mathsf{Val})$ componentwise, filtering out values not satisfying φ and applying f:

$$\mathsf{V}\ [] \equiv [] \qquad\qquad \mathsf{V}([trn] \cdot tr) \equiv \text{if } \varphi\ trn \text{ then } (f\ trn) \cdot (\mathsf{V}\ tr) \text{ else } \mathsf{V}\ tr$$

We also define the *observation function* $\mathsf{O} : \mathsf{Trace} \to \mathsf{List}\ (\mathsf{Obs})$ just like V, but using γ as a filter and g as a producer.

We think of the above as an instantiation of the abstract framework for nondeducibility recalled in Section 4.1, where $\mathsf{World} = \mathsf{Valid}$, $F = \mathsf{O}$, and $H = \mathsf{V}$. Thus, nondeducibility states that the observer O may learn nothing about V. Here we are concerned with a more fine-grained analysis, asking ourselves *what* may the observer O learn about V.

Using the idea underlying nondeducibility, we can answer this question precisely: Given a trace $tr \in \mathsf{Valid}$, the observer sees $\mathsf{O}\ tr$ and therefore can infer that $\mathsf{V}\ tr$ belongs to the set of all values $\mathsf{V}\ tr_1$ for some $tr_1 \in \mathsf{Valid}$ such that $\mathsf{O}\ tr_1 = \mathsf{O}\ tr$. In other words, he can infer that the value is in the set $\mathsf{V}\ (\mathsf{O}^{-1}(\mathsf{O}\ tr) \cap \mathsf{Valid})$, and nothing beyond this. We call this set the *declassification* associated to tr, written Dec_{tr}.

We want to establish, under certain conditions, *upper* bounds for declassification, or, in terms of set-theoretic inclusion, *lower* bounds for Dec_{tr}. To this end, we further fix:

– a relation $\mathsf{B} : \mathsf{List}\ (\mathsf{Val}) \to \mathsf{List}\ (\mathsf{Val}) \to \mathsf{Bool}$, which we call *declassification bound*
– a predicate $\mathsf{T} : \mathsf{Trans} \to \mathsf{Bool}$, which we call *declassification trigger*

The system is called *bounded-deducibility-secure* (*BD-secure*) if for all $tr \in \mathsf{Valid}$ such that $\mathsf{never}\ \mathsf{T}\ tr$, it holds that $\{vl_1 \mid \mathsf{B}\ (\mathsf{V}\ tr)\ vl_1\} \subseteq \mathsf{Dec}_{tr}$ (where "$\mathsf{never}\ \mathsf{T}\ tr$" means "$\mathsf{T}$ holds for no transition in tr"). Informally, BD security expresses the following:

If the trigger T *never holds (i.e., unless* T *eventually holds, i.e., until* T *holds), then the observer* O *can learn nothing about the values* V *beyond* B

We can think of B positively, as an upper bound for declassification, or negatively, as a lower bound for uncertainty. On the other hand, T is a trigger removing the bound B—as soon as T becomes true, the containment of declassification is no longer guaranteed. In the extreme case of B being everywhere true and T everywhere false, we have no declassification, i.e., total uncertainty—in other words, standard nondeducibility.

Expanding some of the above definitions, we can alternatively express BD security as the following being true for all $tr \in \mathsf{Valid}$ and vl, $vl_1 \in \mathsf{List}\ (\mathsf{Val})$:

$$\mathsf{never}\ \mathsf{T}\ tr \wedge \mathsf{V}\ tr = vl \wedge \mathsf{B}\ vl\ vl_1 \ \to\ (\exists tr_1 \in \mathsf{Valid}.\ \mathsf{O}\ tr_1 = \mathsf{O}\ tr \wedge \mathsf{V}\ tr_1 = vl_1) \qquad (*)$$

4.3 Discussion

BD security is a natural extension of nondeducibility. If one considers the latter as reasonably expressing the *absence* of information leak, then one is likely to accept the former as a reasonable means to indicate *bounds* on the leak. Unlike previous notions in the literature, BD security allows to express the bounds *as precisely as desired*.

As an extension of nondeducibility, BD security is subject to the same criticism. The problem with nondeducibility [47, 49, 62] is that in some cases it is too weak, since it takes as *plausible* each possible explanation for an observation: if the observation sequence is ol, then any trace tr such that $\mathsf{O}\ tr = vl$ is plausible. But what if the low-level observers can synchronize their actions and observations with the actions of other entities, such as a high-level user or a Trojan horse acting on his behalf, or even a third-party entity that is neither high nor low? Even without synchronization, the low-level observers may learn, from outside the system, of certain behavior patterns of the high-level users. Then the set of plausible explanations can be reduced, leading to information leak.

In our case, the low-level observers are a group of users assumed to never acquire a certain status (e.g., authorship of a paper). The other users of the system are either "high-level" (e.g., the authors of the paper) or "third-party" (e.g., the non-author users not in the group of observers). Concerning the high-level users, it does not make sense to assume that they would cooperate to leak information *through the system*, since they certainly have better means to do that outside the system, e.g., via email. Users also do not have to trust external software, since everything is filtered through the system kernel—e.g., a chair can run an external linear-programming tool for assigning reviewers, but each assignment is still done through the verified step function. As for the possible third-party cooperation towards leaks of information, this is bypassed by our consideration of arbitrary groups of observers: in the worst case, all the unauthorized users can be placed in this group. However, the possibility to learn and exploit behavior patterns from outside the system is not explicitly addressed by BD security—it would be best dealt with by a probabilistic analysis.

4.4 Instantiation to Our Running Examples

Recall that BD security involves the following parameters:

- an I/O automaton $(\mathsf{State}, \mathsf{Act}, \mathsf{Out}, \mathsf{istate}, \mathsf{step})$
- infrastructures for values $(\mathsf{Val}, \varphi, f)$ and observations $(\mathsf{Obs}, \gamma, g)$
- a declassification specification: trigger T and bound B

In particular, this applies to our conference system automaton. As we are about to show, BD security captures our examples by suitably instantiating the observation and declassification parameters.

Common Observation Infrastructure For all our examples, we consider the same observation infrastructure. We fix UIDs, the set of IDs of the observing users. We let $\mathsf{Obs} = \mathsf{Act} \times \mathsf{Out}$. We take γ to hold for a transition iff its acting user is in UIDs, and g to return its action and output:

$$\gamma\,(s, a, o, s') \;\equiv\; \mathsf{userOf}\ a \in \mathsf{UIDs}$$
$$g\,(s, a, o, s') \;\equiv\; (a, o)$$

$\mathsf{O}\ tr$ thus purges tr keeping only actions performed (or merely attempted) by users in UIDs.

The value infrastructure depends on the considered type of document.

Value Infrastructure for PAP_1 and PAP_2 We fix PID, the ID of the paper of interest. We let $\mathsf{Val} = \mathsf{List}\,(\mathsf{Paper_Content})$. We take φ to hold iff the transition is a successful upload of paper PID's content, and f to return the uploaded content. $\mathsf{V}\ tr$ thus returns the list of all uploaded paper contents for PID:

$$\varphi\,(s, a, o, s') \;\equiv\; o = \mathsf{outOK} \,\wedge\, (\exists\, cid\ uid\ pw\ pct.\ a = \mathsf{Uact}\,(\mathsf{uPaperC}\ cid\ uid\ pw\ \mathsf{PID}\ pct))$$
$$f\,(s, a, o, s') \;\equiv\; pct$$

Above, the value pct from the righthand side of the definition of $f\,(s, a, o, s')$ is the one uniquely determined by the condition defining $\varphi\,(s, a, o, s')$. (When $\varphi\,(s, a, o, s')$ does not hold, the result returned by $f\,(s, a, o, s')$ is irrelevant.)

The declassification triggers and bounds are specific to each example.

Declassification Trigger and Bound for PAP_1 We define $\mathsf{T}(s, a, o, s')$ as "in state s', the paper PID is registered at some conference cid and some user in UIDs is an author of PID or a PC member of cid and the conference has reached the bidding phase," formally:

$$\exists uid \in \mathsf{UIDs}.\ \exists\, cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s'\ cid\ \wedge$$
$$(\mathsf{isAut}\ s'\ uid\ \mathsf{PID} \,\vee\, \mathsf{isPC}\ s'\ uid\ cid \,\wedge\, \mathsf{phase}\ s'\ cid \geq \mathsf{Bidding})$$

Intuitively, the intent with PAP_1 is that, provided T never holds, users in UIDs learn nothing about the various consecutive versions of PID. But is it true that they can learn *absolutely nothing*? There is the possibility that a user could infer that no version was submitted: Say the paper's conference has not yet reached the submission phase; then the trace of paper uploads must be empty. But indeed, nothing beyond this quite harmless information should leak: any nonempty value sequence vl might as well have been any other (possibly empty) sequence vl_1. Hence we define $\mathsf{B}\ vl\ vl_1$ as $vl \neq []$. It is interesting to notice here that, while a user could determine emptiness, this is not true for nonemptiness. This shows that declassification bounds can be naturally asymmetric.

Declassification Trigger and Bound for PAP_2 The trigger only involves authorship, ignoring PC membership at the paper's conference—we take $\mathsf{T}(s, a, o, s')$ to be

$$\exists uid \in \mathsf{UIDs}.\ \exists\, cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s'\ cid\ \wedge\ \mathsf{isAut}\ s'\ uid\ \mathsf{PID}$$

In the case of $\textsc{Pap}_2$, we have a genuine example of nontrivial declassification bound: since a PC member can learn the paper's content but only as its *last submitted version*, we take $\mathsf{B}\ vl\ vl_1$ to be

$$vl \neq [] \wedge vl_1 \neq [] \wedge \mathsf{last}\ vl = \mathsf{last}\ vl_1$$

where the function last returns the last element of a list.

Instantiation for Rev To uniquely identify a review, we fix not only a paper ID PID, but also a number N—with the understanding that the pair (PID, N) denotes the Nth review of the paper PID. The value infrastructure refers not only to the review's content but also to the conference phase: $\mathsf{Val} = \mathsf{List}\ (\mathsf{Phase} \times \mathsf{Review_Content})$. The functions φ and f are defined similarly to those for $\textsc{Pap}_1$ and $\textsc{Pap}_2$, *mutatis mutandis*. Thus, φ checks whether the transition is a successful update or u-update[2] of the given review, namely (PID, N), and f returns a pair consisting of the conference's current phase and the updated review's content; hence V returns a list of such pairs.

$$\begin{aligned} \varphi\ (s, a, o, s') \ \equiv\ & o = \mathsf{outOK} \wedge \\ & (\exists\, cid\ uid\ pw\ rct.\ a = \mathsf{Uact}\ (\mathsf{uReview}\ cid\ uid\ pw\ \mathsf{PID}\ \mathsf{N}\ rct) \vee \\ & \qquad a = \mathsf{UUact}\ (\mathsf{uuReview}\ cid\ uid\ pw\ \mathsf{PID}\ \mathsf{N}\ rct)) \\ f\ (s, a, o, s') \ \equiv\ & (\mathsf{phase}\ s\ cid,\ rct) \end{aligned}$$

The trigger T is similar to that of $\textsc{Pap}_2$ but refers to authorship of the paper's Nth review rather than paper authorship:

$$\mathsf{T}\ (s, a, o, s') \ \equiv\ \exists uid \in \mathsf{UIDs}.\ \exists\, cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s'\ cid \wedge \mathsf{isRevNth}\ s'\ uid\ \mathsf{PID}\ \mathsf{N}$$

One may wonder why do we keep the conference phase as part of the value infrastructure for Rev, in other words, why do we have f return the conference phase in addition to the review content. The answer is that we need this information in order to formulate an appropriate bound B, which is able to distinguish between updates occurring *before* the discussion phase and those occurring *starting from* the discussion phase—because these updates have different confidentiality statuses. It is *a priori* knowledge (i.e., knowledge that can be attained solely by studying the system's specification) that review updates can only occur during the review and discussion phases, in this order—i.e., that any produced value list vl has the form $ul \cdot wl$ such that the pairs in ul have Reviewing as first component and the pairs in wl have Discussion as first component. Moreover, any PC member having no conflict with PID can additionally learn $\mathsf{last}\ ul$ (the last submitted version before discussion), and wl (the versions updated during discussion); but (unless/until T holds) nothing beyond these. So we take $\mathsf{B}\ vl\ vl_1$ to state that vl decomposes as $ul \cdot wl$ as indicated above, vl_1 decomposes similarly as $ul_1 \cdot wl$, and $\mathsf{last}\ ul = \mathsf{last}\ ul_1$.

Instantiation for Dis The property Dis needs rephrasing in order to be captured as BD security. It can be decomposed into:

$\textsc{Dis}_1$: An author always has conflict with his papers
$\textsc{Dis}_2$: A group of users learn nothing[3] about a paper's discussion unless one of them becomes a PC member at the paper's conference *having no conflict with the paper*

[2] Unlike papers, reviews can also be updated undestructively, i.e., with the previous version remaining available —namely, in the discussion phase.

[3] More precisely, almost nothing, i.e., nothing beyond the absence of any edit.

	Source	Declassification Trigger	Declassification Bound
1	Paper	Paper Authorship	Last Uploaded Version
2		Paper Authorship or PC Membership$^{\mathsf{B}}$	Absence of Any Upload
3	Review	Review Authorship	Last Edited Version Before Discussion and All the Later Versions
4		Review Authorship or Non-Conflict PC Membership$^{\mathsf{D}}$	Last Edited Version Before Notification
5		Review Authorship or Non-Conflict PC Membership$^{\mathsf{D}}$ or Paper Authorship$^{\mathsf{N}}$	Absence of Any Edit
6	Discussion	Non-Conflict PC Membership	Absence of Any Edit
7	Decision	Non-Conflict PC Membership	Last Edited Version
8		Non-Conflict PC Membership or PC Membership$^{\mathsf{N}}$ or Paper Authorship$^{\mathsf{N}}$	Absence of Any Edit
9	Reviewer Assignment	Non-Conflict PC Membership$^{\mathsf{R}}$	Non-Conflict PC Membership of Reviewers and No. of Reviews
10		Non-Conflict PC Membership$^{\mathsf{R}}$ or Paper Authorship$^{\mathsf{N}}$	Non-Conflict PC Membership of Reviewers

Phase Stamps: B = Bidding, D = Discussion, N = Notification, R = Review

Fig. 3: CoCon's confidentiality properties

$\textsc{Dis}_1$ is a safety property (holding for all reachable states of the system). $\textsc{Dis}_2$ is an instance of BD security defined as expected (in light of our previous analysis). In particular, the value infrastructure focuses on the actions that (undestructively) update the discussion section with comments.

$$
\begin{aligned}
\varphi\,(s,a,o,s') &\equiv o = \mathsf{outOK} \wedge (\exists\, cid\ uid\ pw\ com.\ a = \mathsf{UUact}\ (\mathsf{uuDis}\ cid\ uid\ pw\ \mathsf{PID}\ com))\\
f\,(s,a,o,s') &\equiv com\\
\mathsf{T}\,(s,a,o,s') &\equiv \exists uid \in \mathsf{UIDs}.\ \exists\, cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s'\ cid \wedge\\
&\qquad \mathsf{isPC}\ s'\ cid\ uid \wedge \mathsf{pref}\ s'\ uid\ \mathsf{PID} \neq \mathsf{Conflict}\\
\mathsf{B}\ vl\ vl_1 &\equiv vl \neq []
\end{aligned}
$$

4.5 More Instances

Fig. 3 shows, in informal notation, the entire array of confidentiality properties we have formulated as BD security. The observation infrastructure is always the same, given by the actions and outputs of a fixed group of observer users, as in Section 4.4.

There are several information sources, each yielding a different value infrastructure. In rows 1–8, the sources are actual documents: paper content, review, discussion, decision. The properties $\textsc{Pap}_1$, $\textsc{Pap}_2$, $\textsc{Rev}$ and $\textsc{Dis}_2$ form the rows 2, 1, 3, and 6, respectively. In rows 9 and 10, the source is the data about the reviewers assigned to a paper.

The declassification triggers express paper or review authorship (being or becoming an author of the indicated document) or PC membership at the paper's conference. Some triggers are also listed with "phase stamps" that strengthen the statements. For example, "PC membership$^{\mathsf{B}}$" should be read as "PC membership and paper's conference phase being at least bidding."

Some of the triggers require lack of conflict with the paper, which is often important for the security statement to be sufficiently strong. This is the case of $\textsc{Dis}_2$ (row 6), since without the non-conflict assumption $\textsc{Dis}_2$ and $\textsc{Dis}_1$ would no longer imply the desired property $\textsc{Dis}$. By contrast, lack of conflict cannot be added to PC membership in $\textsc{Pap}_1$ (row 2), since such a stronger version would not hold: even if a PC member decides to indicate conflict with a paper, this happens *after* he had the opportunity to see the paper's content.

Note that the listed properties capture exhaustively the information flow from the indicated sources, in the sense that they identify all the relevant roles that can influence these flows. This can be seen by traversing the rows for each source upwards—in the increasing order of the bound's permissiveness, which is also the decreasing order of the trigger's permissiveness—and recording the differences with respect to the triggers.

For example, for the review source, we have the following cases:

Row 5: If a user is not the review's author, not a non-conflict PC member in the discussion phase, and not the reviewed paper's author in the notification phase, then he could learn about the absence of any edit—but nothing more

Subtracting row 4 from row 5: In addition, the reviewed paper's authors will learn in the notification phase of the last edited version of the review before notification—but nothing more

Subtracting row 3 from row 4: In addition, non-conflict PC members will learn in the discussion phase of all the intermediate versions starting from the last one before the discussion phase and all the later versions (produced during the discussion phase)—but nothing more

The role that persists even in the least permissive trigger (in row 3) is that of the review's author, which obviously has no restriction.

As another example, consider the reviewer assignment source. We have the following cases:

Row 10: If a user is not a non-conflict PC member in the reviewing phase and not the paper's author in the notification phase, then he will have access to the *a priori* knowledge that reviewers are non-conflict PC members—but nothing more

Subtracting row 9 from row 10: In addition, the paper's authors will learn in the notification phase of the *number* of reviewers (of course, inferring it from the number of reviews they receive as authors)—but nothing more

Here, the role that persists in the least permissive trigger (in row 9) is that of PC member in the reviewing phase.

4.6 Accountability Properties

Our confidentiality properties show upper bounds on information release that are valid unless/until some trigger T occurs, e.g., chairness, PC membership, authorship, or the conference reaching a given phase. While T is allowed to depend on all four components of a transition (s, o, a, s'), our CoCon instances only depend on s', employing predicates such as $\mathsf{isAut}\ s'\ uid\ \mathsf{PID}$ and $\mathsf{isPC}\ s'\ uid\ cid$. Two questions arise.

First, why do we consider the target state s' and not the source state s? This is because our choice gives the more intuitive result: $\mathsf{never}\ \mathsf{T}$ holding for a valid trace $[(s_1, a_1, o_1, s_2), (s_2, a_2, o_2, s_3), \ldots, (s_{n-1}, a_{n-1}, o_n, s_n)]$ means that the corresponding state condition fails

for $s_2, \dots, s_n$ (importantly, also including the last state s_n); and all our trigger conditions fail trivially for the initial state $s_1 = \mathsf{istate}$, therefore not covering this state is not a problem.

Second, why do we formulate T as a state-based condition and not as an action-based condition? For example, instead of asking that a user $uid \in \mathsf{UIDs}$ *be* an author in the transition's target state ($\mathsf{isAut}\ s'\ uid\ \mathsf{PID}$), why not ask that the action of such a user *becoming* an author has occurred in the trace? A first answer to this is that the two choices are equivalent, while state-based conditions are easier to formulate.

However, the state-based versus action-based question leads us to a more fundamental concern about the security guarantees. We have proved that one does not acquire a certain information unless one acquires a certain role. But how can we know that only "lawfully" appointed users acquire that role? To fully answer this question, we track back, within valid traces, all possible chains of events that could have led to certain roles and other information flow enabling situations—leading to a form of *accountability* properties.

For instance, we prove that, if a user is currently a chair then he either must have been the original chair (who registered the conference), or, inductively, must have been appointed by another chair—and this of course in a well-founded fashion, in that the chain of chair appointments can always be traced back to the original chair and the registration of the conference.

Formally, we achieve this by introducing an alternative "is chair" predicate $\mathsf{isChair}'$: $\mathsf{Trace} \to \mathsf{ConfID} \to \mathsf{UserID} \to \mathsf{Bool}$, which is defined inductively to account for the lawful chair-appointment transitions on the trace:

Create Conference: $$\frac{trn = (_, \mathsf{Cact}\ (\mathsf{cConf}\ cid\ uid\ _\ _\ _), \mathsf{outOK}, _)}{\mathsf{isChair}'\ (tr \cdot [trn])\ cid\ uid}$$

Add Chair: $$\frac{\mathsf{isChair}'\ tr\ cid\ uid' \qquad trn = (_, (\mathsf{Cact}\ (\mathsf{cChair}\ cid\ uid'\ _\ uid)), \mathsf{outOK}, _)}{\mathsf{isChair}'\ (tr \cdot [trn])\ cid\ uid}$$

Irrelevant Transition: $$\frac{\mathsf{isChair}'\ tr\ cid\ uid}{\mathsf{isChair}'\ (tr \cdot [trn])\ cid\ uid}$$

The chair role accountability rests on the equivalence between the original (state-based) predicate and this alternative trace-based predicate:

Prop 1 For all valid traces tr ending in state s, we have that

$$\mathsf{isChair}\ s\ cid\ uid \quad \leftrightarrow \quad \mathsf{isChair}'\ tr\ cid\ uid$$

We formulate (and prove) such accountability properties for all the trigger components used in our security properties:

1. If a user is an author of a paper then either he has registered the paper in the first place or, inductively, has been appointed as coauthor by another author
2. If a user is a PC member then he either must have been the original chair or must have been appointed by a chair
3. If a user is a paper's reviewer, then he must have been appointed by a chair
4. If a user has conflict with a paper, than he is either an author of the paper or the conflict has been declared by the user himself or by a paper's author, in such a way that between the moment when the conflict has been last declared and the current moment there is no transition that successfully removes the conflict
5. If a conference is in a given phase different from "no phase," then this has happened as a consequence of either a conference approval action by the voronkov (if the phase is Setup) or a phase change action by a chair (otherwise)

As expected, some of the above accountability schemes rely on the others. For example, the accountability scheme of the PC member role relies on that of the chair role, and that of reviewer relies on those of chair and PC member.

In conclusion, the BD security instances for CoCon state that information disclosure is bounded, provided certain triggers are not fired. To complement these, we formulate accountability properties, stating that users cannot improperly fire the triggers.

5 Verification

To cope with general declassification bounds, BD security speaks about system traces in conjunction with value sequences that must be produced by these traces. We extend the unwinding proof technique to cope with this situation and employ the result to the verification of our confidentiality properties.

5.1 Unwinding Proof Method

We see from Section 4.2's definition $(*)$ that to prove BD security, one starts with a valid tr (starting in s and having value sequence vl) and an "alternative" value sequence vl_1 such that $\mathsf{B}\ vl\ vl_1$, and one needs to produce an "alternative" valid trace tr_1 starting in s whose value sequence is vl_1 and whose observation sequence is the same as that of tr.

Following the tradition of unwinding for noninterference [24, 61], we wish to construct tr_1 from tr *incrementally*: as tr grows, tr_1 should grow nearly synchronously. If we adopted the traditional setting, we would take an unwinding relation to be a relation $\theta : \mathsf{State} \to \mathsf{State} \to \mathsf{Bool}$ that connects the states reached by tr and tr_1, satisfying some conditions ensuring that for all possible moves of tr there is a suitable match by tr_1—as in a two player game where we have control over the tr_1 moves and the adversary has control over the tr moves. In order for tr_1 to have the same observation sequence (produced by O) as tr, we would need to require that the observable transitions of tr_1 (i.e., those for which γ holds) are perfectly synchronized with those of tr and produce the same observations.

So far, so good. However, when dealing with the value sequences (produced by V), we face the following problem. In contrast to the traditional setting, we must consider an additional parameter, namely the *a priori given* value sequence vl_1 that needs to be produced by tr_1. In fact, it appears that one would need to maintain, besides an unwinding relation on states $\theta : \mathsf{State} \to \mathsf{State} \to \mathsf{Bool}$, also an "evolving" generalization of the declassification trigger B; but then θ and B would certainly need to be synchronized. We resolve this by enlarging the domain of the unwindings to quaternary relations

$$\Delta : \mathsf{State} \to \mathsf{List}\ (\mathsf{Val}) \to \mathsf{State} \to \mathsf{List}\ (\mathsf{Val}) \to \mathsf{Bool}$$

that generalize both θ and B. Intuitively, $\Delta\ s\ vl\ s_1\ vl_1$ keeps track of the current state of tr, the remaining value sequence of tr, the current state of tr_1, and the remaining value sequence of tr_1.

Let the predicate $\mathsf{consume}\ trn\ vl\ vl'$ mean that the transition trn either produces a value that is consumed from vl yielding vl' or produces no value and $vl = vl'$. Formally:

$$\text{if } \varphi\ trn \text{ then } (vl \neq [] \wedge f\ trn = \mathsf{head}\ vl \wedge vl' = \mathsf{tail}\ vl) \text{ else } (vl' = vl)$$

In light of the above discussion, we are tempted to define an unwinding as a relation Δ such that $\Delta\ s\ vl\ s_1\ vl_1$ implies *either* of the following conditions:

- REACTION: For any valid transition (s, a, o, s') and lists of values vl, vl' such that $\mathsf{consume}\ (s, a, o, s')\ vl\ vl'$ holds, either of the following holds:
  - IGNORE: The transition yields no observation $(\neg\, \gamma\ a\ o)$ and $\Delta\ s'\ vl'\ s_1\ vl_1$ holds
  - MATCH: There exist a valid transition (s_1, a_1, o_1, s'_1) and a list of values vl'_1 such that $\mathsf{consume}\ (s_1, a_1, o_1, s'_1)\ vl_1\ vl'_1$ and $\Delta\ s'\ vl'\ s'_1\ vl'_1$ hold, and the transitions trn and trn_1 either are both unobservable or are both observable and yield the same observation (i.e., $\gamma\ trn$ holds iff $\gamma\ trn_1$ holds and, in case these hold, $g\ trn = g\ trn_1$)
- INDEPENDENT ACTION: There exist a valid transition (s_1, a_1, o_1, s'_1) that yields no observation $(\neg\, \gamma\ a_1\ o_1)$ and a list of values vl'_1 such that $\mathsf{consume}\ (s_1, a_1, o_1, s'_1)\ vl_1\ vl'_1$ and $\Delta\ s\ vl\ s'_1\ vl'_1$ hold

The intent is that BD security should hold if there exists an unwinding Δ that "initially includes" B. A trace tr_1 could then be constructed incrementally from tr, vl and vl_1, applying REACTION or INDEPENDENT ACTION until the three lists become empty.

Progress However, such an argument faces difficulties. First, INDEPENDENT ACTION is not guaranteed to decrease any of the three lists (tr, vl and vl_1). To address this, we strengthen INDEPENDENT ACTION by adding the requirement that $\varphi\ (s_1, a_1, o_1, s'_1)$ holds—this ensures that vl_1 decreases (i.e., vl'_1 is strictly shorter then vl_1). This way, we know that each REACTION and INDEPENDENT ACTION decreases at least one list: the former tr and the latter vl_1; and since vl is empty whenever tr is, the progress problem seems resolved.

Yet, there is a second, more subtle difficulty: after tr has become empty, how can we know that vl_1 will start decreasing? With the restrictions so far, one may still choose REACTION with parameters that leave vl_1 unaffected. So we need to make sure that the following implication holds: if $tr = []$ and $vl_1 \neq []$, then vl_1 will be consumed. Since from inside the unwinding relation we cannot (and do not want to!) see tr, but only vl, we weaken the assumption of this implication to "if $vl = []$ and $vl_1 \neq []$"; moreover, we strengthen its conclusion to requiring that only the INDEPENDENT ACTION choice (guaranteed to shorten vl_1) be available. Equivalently, we condition the alternative choice of REACTION by the negation of the above, namely $vl \neq [] \ \vee\ vl_1 = []$.

The above discussion shows that, unlike with traditional unwinding which is a purely *coinductive* notion, in the style of (bi)simulation [65] (meaning we win if we manage to stay in a certain game indefinitely), here we have a blend of coinduction and *induction*, the latter requiring that we also make some form of progress in the game.

Exit Condition The third observation is not concerned with a difficulty, but with an optimization. We note that BD security holds trivially if the original trace tr cannot saturate the value list vl, i.e., if $\mathsf{V}\ tr \neq vl$—this happens if and only if, at some point, an element v of vl can no longer be saturated, i.e., for some decompositions $tr = tr' \cdot tr''$ and $vl = vl' \cdot [v] \cdot vl''$ of tr and vl, it holds that $\mathsf{V}\ tr' = vl'$ and $\forall trn \in tr''.\ \varphi\ trn \to f\ trn \neq v$. Can we detect such a situation from within Δ? The answer is (an over-approximated) yes: after $\Delta\ s\ vl\ s_1\ vl_1$ evolves by REACTION and INDEPENDENT ACTION to $\Delta\ s'\ ([v] \cdot vl'')\ s'_1\ vl'_1$ for some s', s'_1 and vl'_1 (presumably consuming tr' and saturating the vl' prefix of vl), then one can safely exit the game if one proves that no valid trace tr'' starting from s' can ever saturate v, in that it satisfies $\forall trn \in tr''.\ \varphi\ trn \to f\ trn \neq v$.

The final definition of BD unwinding is given below, where $\mathsf{reach} : \mathsf{State} \to \mathsf{Bool}$ is the state reachability predicate and $\mathsf{reach}_{\neg\mathsf{T}} : \mathsf{State} \to \mathsf{Bool}$ is its strengthening to reachability

by transitions that do not satisfy T:

$$\begin{aligned}\mathsf{unwind}\,\Delta \;\equiv\; & \forall s\; vl\; s_1\; vl_1.\; \mathsf{reach}_{\neg\mathsf{T}}\; s \,\wedge\, \mathsf{reach}\; s_1 \,\wedge\, \Delta\; s\; vl\; s_1\; vl_1 \;\rightarrow \\ & ((vl \neq [] \,\vee\, vl_1 = []) \,\wedge\, \mathsf{reaction}\;\Delta\; s\; s\; vl\; s_1\; vl_1) \,\vee \\ & \mathsf{iaction}\,\Delta\; s\; s\; vl\; s_1\; vl_1 \,\vee \\ & (vl \neq [] \,\wedge\, \mathsf{exit}\; s\; (\mathsf{head}\; vl))\end{aligned}$$

The predicates reaction and iaction formalize REACTION and INDEPENDENT ACTION (with its aforementioned strengthening), the former involving a disjunction of predicates formalizing IGNORE and MATCH:

$$\begin{aligned}& \mathsf{reaction}\,\Delta\; s\; vl\; s_1\; vl_1 \;\equiv\; \forall a\; o\; s'.\; \text{let } trn = (s, a, o, s') \text{ in} \\ & \quad trn \in \mathsf{Valid} \,\wedge\, \neg\,\mathsf{T}\; trn \,\wedge\, \mathsf{consume}\; trn\; vl\; vl' \;\rightarrow \\ & \quad \mathsf{match}\,\Delta\; s\; s_1\; vl_1\; a\; o\; s'\; vl' \,\vee\, \mathsf{ignore}\,\Delta\; s\; s_1\; vl_1\; a\; o\; s'\; vl'\end{aligned}$$

where:

$$\mathsf{ignore}\,\Delta\; s\; s_1\; vl_1\; a\; o\; s' vl' \equiv \neg\,\gamma\,(s, a, o, s') \,\wedge\, \Delta\; s'\; vl'\; s_1\; vl_1$$

$$\begin{aligned}& \mathsf{match}\,\Delta\; s\; s_1\; vl_1\; a\; o\; s'\; vl' \equiv \\ & \quad \exists a_1\; o_1\; s_1'\; vl_1'.\; \text{let } trn = (s, a, o, s') \text{ and } trn_1 = (s_1, a_1, o_1, s_1') \text{ in} \\ & \quad trn_1 \in \mathsf{Valid} \,\wedge\, \mathsf{consume}\; trn_1\; vl_1\; vl_1' \,\wedge\, (\gamma\; trn \;\leftrightarrow\; \gamma\; trn_1) \,\wedge \\ & \quad (\gamma\; trn \rightarrow g\; trn = g\; trn_1) \,\wedge\, \Delta\; s'\; vl'\; s_1'\; vl_1'\end{aligned}$$

$$\begin{aligned}& \mathsf{iaction}\,\Delta\; s\; vl\; s_1\; vl_1 \equiv \\ & \quad \exists a_1\; o_1\; s_1'\; vl_1'.\; \text{let } trn_1 = (s_1, a_1, o_1, s_1') \text{ in} \\ & \quad trn_1 \in \mathsf{Valid} \,\wedge\, \mathsf{consume}\; trn_1\; vl_1\; vl_1' \,\wedge\, \varphi\; trn_1 \;\wedge\, \neg\,\gamma\; trn_1 \,\wedge\, \Delta\; s\; vl\; s_1'\; vl_1'\end{aligned}$$

The predicate exit is defined as

$$\mathsf{exit}\; s\; v \;\equiv\; \forall\; tr\; trn.\; tr \cdot [trn] \in \mathsf{Valid}_s \,\wedge\, \varphi\; trn \rightarrow f\; trn \neq v$$

It essentially expresses a safety property, and therefore can be verified in a trace-free manner by exhibiting an invariant $K : \mathsf{State} \rightarrow \mathsf{Bool}$ and proving that it holds for s. Intuitively, the potential invariant K ensures that the value v can never be produced.

Lemma 2 Assume that the following hold for all valid transitions $trn = (s, a, o, s')$ such that $K\; s$ holds:

- $\varphi\; trn \rightarrow f\; trn \neq v$
- $K\; s'$

Then $\forall s.\; K\; s \rightarrow \mathsf{exit}\; s\; v$

We can prove that indeed any unwinding relation constructs an "alternative" trace tr_1 from any trace tr starting in a $\neg\mathsf{T}$-reachable state:

Lemma 3 $\mathsf{unwind}\,\Delta \,\wedge\, \mathsf{reach}_{\neg\mathsf{T}}\; s \,\wedge\, \mathsf{reach}\; s_1 \,\wedge\, \Delta\; s\; vl\; s_1\; vl_1 \,\wedge\, tr \in \mathsf{Valid}_s \,\wedge\, \mathsf{never}\;\mathsf{T}\; tr \,\wedge\, \mathsf{V}\; tr = vl \;\rightarrow\; (\exists tr_1.\; tr_1 \in \mathsf{Valid}_{s_1} \,\wedge\, \mathsf{O}\; tr_1 = \mathsf{O}\; tr \,\wedge\, \mathsf{V}\; tr_1 = vl_1)$

Proof By induction on $\mathsf{length}\; tr + \mathsf{length}\; vl_1$, formalizing our previous analysis concerning progress.

Theorem 4 (Unwinding Theorem) Assume that the following hold:

- $\forall vl\; vl_1.\; \mathsf{B}\; vl\; vl_1 \rightarrow \Delta\; \mathsf{istate}\; vl\; \mathsf{istate}\; vl_1$
- $\mathsf{unwind}\,\Delta$

Then the system is BD-secure.

Proof From Lemma 3, taking $s_1 = s = \mathsf{istate}$.

According to the theorem, BD unwinding is a *sound proof method for BD security*: to check BD security it suffices to define a relation Δ and prove that it coincides with B on the initial state and that it is a BD unwinding.

5.2 Compositional Reasoning

To keep each reasoning step manageable, we replace the monolithic unwinding relation Δ with a network of relations, such that any relation may unwind to any number of relations in the network. To achieve this, we replace the single requirement unwind Δ with a set of requirements unwind_to Δ Δs with Δs being a set of relations. The predicate unwind_to is defined similarly to unwind, but employing disjunctions of the predicates in Δs, written disj Δs:

$$\begin{aligned}\mathsf{unwind_to}\,\Delta\,\Delta s \equiv\ & \forall s\ vl\ s_1\ vl_1.\ \mathsf{reach}_{\neg\top}\ s \wedge \mathsf{reach}\ s_1 \wedge \Delta\ s\ vl\ s_1\ vl_1 \rightarrow \\ & ((vl \neq [] \vee vl_1 = []) \wedge \mathsf{reaction}\ (\mathsf{disj}\,\Delta s)\ s\ s\ vl\ s_1\ vl_1) \vee \\ & \mathsf{iaction}\ (\mathsf{disj}\,\Delta s)\ s\ s\ vl\ s_1\ vl_1 \vee \\ & (vl \neq [] \wedge \mathsf{exit}\ s\ (\mathsf{head}\ vl))\end{aligned}$$

This enables a form of sound compositional reasoning: if we verify a condition as above for each component relation, we obtain an overall secure system.

Corollary 5 (Compositional Unwinding Theorem) Let Δs be a set of relations. For each $\Delta \in \Delta s$, let $\mathsf{next}_\Delta \subseteq \Delta s$ be a (possibly empty) "continuation" of Δ, and let $\Delta_{\mathsf{init}} \in \Delta s$ be a chosen "initial" relation. Assume the following hold:

- $\forall vl\ vl_1.\ \mathsf{B}\ vl\ vl_1 \rightarrow \Delta_{\mathsf{init}}\ \mathsf{istate}\ vl\ \mathsf{istate}\ vl_1$
- $\forall \Delta \in \Delta s.\ \mathsf{unwind_to}\ \Delta\ \mathsf{next}_\Delta$

Then the system is BD-secure.

Proof One can show that unwind (disj Δs) holds and use the original unwinding theorem.

The network of components can in principle form any directed graph, the only requirement being that each node has an outgoing edge —Fig. 5 shows an example. However, the unwinding proofs for our CoCon instances will essentially follow the temporal evolution of the conference as witnessed by the phase change and other events. Hence the following linear network will suffice (Fig. 6): each Δ_i unwinds either to itself, or to Δ_{i+1} (if $i \neq n$), or to an exit component Δ_{e} that invariably chooses the "exit" unwinding condition. To capture this type of situation, we employ the predicate unwind_cont that restricts the unwinding of Δ_i to proper continuations (i.e., no exits) and the predicate unwind_exit that restricts the unwinding of Δ_{e} to exits (as depicted in Fig. 6):

$$\begin{aligned}\mathsf{unwind_cont}\,\Delta\,\Delta s \equiv\ & \forall s\ vl\ s_1\ vl_1.\ \mathsf{reach}_{\neg\top}\ s \wedge \mathsf{reach}\ s_1 \wedge \Delta\ s\ vl\ s_1\ vl_1 \rightarrow \\ & ((vl \neq [] \vee vl_1 = []) \wedge \mathsf{reaction}\ (\mathsf{disj}\,\Delta s)\ s\ s\ vl\ s_1\ vl_1) \vee \\ & \mathsf{iaction}\ (\mathsf{disj}\,\Delta s)\ s\ s\ vl\ s_1\ vl_1 \\ \mathsf{unwind_exit}\,\Delta \equiv\ & \forall s\ vl\ s_1\ vl_1.\ \mathsf{reach}_{\neg\top}\ s \wedge \mathsf{reach}\ s_1 \wedge \Delta\ s\ vl\ s_1\ vl_1 \rightarrow \\ & vl \neq [] \wedge \mathsf{exit}\ s\ (\mathsf{head}\ vl)\end{aligned}$$

Corollary 6 (Sequential Unwinding Theorem) Consider the indexed set of relations $\{\Delta_1, \ldots, \Delta_n\}$ such that the following hold:

- $\forall vl\ vl_1.\ \mathsf{B}\ vl\ vl_1 \rightarrow \Delta_1\ \mathsf{istate}\ vl\ \mathsf{istate}\ vl_1$
- $\forall i \in \{1, \ldots, n-1\}.\ \mathsf{unwind_cont}\ \Delta_i\ \{\Delta_i, \Delta_{i+1}, \Delta_{\mathsf{e}}\}$
- $\mathsf{unwind_cont}\ \Delta_n\ \{\Delta_n, \Delta_{\mathsf{e}}\}$
- $\mathsf{unwind_exit}\ \Delta_{\mathsf{e}}$

Then the system is BD-secure.

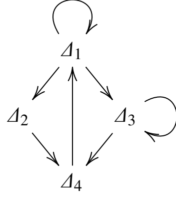

Fig. 5: A network of unwinding components

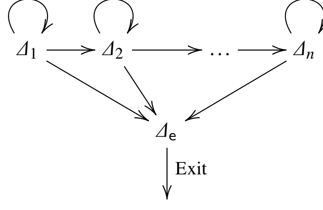

Fig. 6: A linear network with exit

Proof From the compositional unwinding theorem, given that unwind_cont and unwind_exit are both subsumed by unwind_to.

We found the sequential unwinding theorem to represent a sweet spot between generality and ease of instantiation for our concrete unwinding proofs, which we discuss next. In fact, we even went a little further and partially instantiated this theorem with various fixed small numbers of non-terminal relations, namely 3, 4 and 5.

5.3 Verification of Concrete Instances

We have employed the sequential unwinding theorem to verify all our CoCon instances of BD security. To explain our unwinding proofs, it is helpful to resort again to the "alternative trace" intuition of BD security: an unwinding proof essentially constructs an alternative trace tr_1 (which produces the value sequence vl_1) from the original trace tr (which produces the value sequence vl). The choice of the relations Δ_i required by the sequential unwinding theorem is guided by milestones in the journey of tr and tr_1: changing a conference's phase, registering a paper, registering a relevant agent such as a chair, a PC member or a reviewer, declaring or removing a conflict, etc. For example, here are the unwinding relations we used in the proof of PAP$_2$:

$\Delta_1\ s\ vl\ s_1\ vl_1$	$\neg\,(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid)\ \wedge\ s = s_1\ \wedge\ \mathsf{B}\ vl\ vl_1$
$\Delta_2\ s\ vl\ s_1\ vl_1$	$(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \wedge \mathsf{phase}\ s\ cid = \mathsf{Submission}) \wedge\ s =_{\mathsf{PID}} s_1 \wedge \mathsf{B}\ vl\ vl_1$
$\Delta_3\ s\ vl\ s_1\ vl_1$	$(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid) \wedge\ s = s_1\ \wedge\ vl = vl_1 = []$
$\Delta_{\mathsf{e}}\ s\ vl\ s_1\ vl_1$	$(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid\ \wedge\ \mathsf{phase}\ s\ cid > \mathsf{Submission})\ \wedge\ vl \neq []$

And here are the ones we used in the proof of REV:

$\Delta_1\ s\ vl\ s_1\ vl_1$	$(\forall cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \rightarrow \mathsf{phase}\ s\ cid < \mathsf{Reviewing}) \wedge s = s_1 \wedge \mathsf{B}\ vl\ vl_1$
$\Delta_2\ s\ vl\ s_1\ vl_1$	$(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid\ \wedge\ \mathsf{phase}\ s\ cid = \mathsf{Reviewing}\ \wedge$ $\neg\,(\exists uid.\ \mathsf{isRevNth}\ s\ uid\ \mathsf{PID}\ \mathsf{N})) \wedge$ $s = s_1 \wedge \mathsf{B}\ vl\ vl_1$
$\Delta_3\ s\ vl\ s_1\ vl_1$	$(\exists cid\ uid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \wedge \mathsf{phase}\ s\ cid = \mathsf{Reviewing}\ \wedge\ \mathsf{isRevNth}\ s\ uid\ \mathsf{PID}\ \mathsf{N}) \wedge$ $s =_{\mathsf{PID,N}} s_1\ \wedge \mathsf{B}\ vl\ vl_1$
$\Delta_4\ s\ vl\ s_1\ vl_1$	$(\exists cid\ uid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \wedge\ \mathsf{phase}\ s\ cid \geq \mathsf{Reviewing}\ \wedge \mathsf{isRevNth}\ s\ uid\ \mathsf{PID}\ \mathsf{N}) \wedge$ $s = s_1\ \wedge (\ \exists wl.\ vl = vl_1 = \mathsf{map}\ (\mathsf{Pair}\ \mathsf{Discussion})\ wl\ )$
$\Delta_{\mathsf{e}}\ s\ vl\ s_1\ vl_1$	$vl \neq [] \wedge$ $((\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \wedge \mathsf{phase}\ s\ cid > \mathsf{Reviewing} \wedge \neg\,(\exists uid.\ \mathsf{isRevNth}\ s\ uid\ \mathsf{PID}\ \mathsf{N}))$ $\vee$ $(\exists cid.\ \mathsf{PID} \in \mathsf{paperIDs}\ s\ cid \wedge \mathsf{phase}\ s\ cid > \mathsf{Reviewing}\ \wedge\ \mathsf{fst}\ (\mathsf{head}\ vl) = \mathsf{Reviewing}))$

Above, B vl vl_1 denotes the respective declassification bounds for these instances, and the changes from Δ_i to Δ_{i+1} have been emphasized.

Each BD security instance has one or more critical phases, the only phases when the value sequences vl and vl_1 can be produced. For PAP$_2$, value production means paper uploading, which is only possible in the submission phase—meaning that submission is the single critical phase. For REV, value production means review update; there is an update action available in the reviewing phase, and an u-update action available in the discussion phase—so both these phases are critical. Until the critical phases, (the construction of) tr_1 proceeds perfectly synchronously to tr, taking the same actions—consequently, the states s and s_1 stay equal in Δ_1 for PAP$_2$ and in Δ_1 and Δ_2 for REV.

In the critical phases, the traces tr and tr_1 will partly diverge, due to the need of producing possibly different (but B-related) value sequences. As a result, the equality between s and s_1 is replaced with the weaker relation of *equality everywhere except on certain components of the state*. This is the case with the relation Δ_2 for PAP$_2$, where $=_{\mathsf{PID}}$ denotes equality everywhere except on the content of PID. Similarly, in Δ_3 for REV, $=_{\mathsf{PID,N}}$ denotes equality everywhere except on the content of PID's Nth review.

At the end of the critical phases, tr_1 will usually need to resynchronize with tr and hereafter proceed with identical actions. Consequently, s and s_1 will become connected by a stronger "equality everywhere except" relation or even plain equality again—which is the case with Δ_3 for PAP$_2$ and with Δ_4 for REV.

Besides the phase changes, other relevant events in the unwinding proofs of PAP$_2$ and REV are the registration of the considered paper or review. For PAP$_2$, here is the informal reading of Δ_1–Δ_3 in light of such events:

- Δ_1: The paper PID is not registered yet, so the two states s and s_1 are equal
- Δ_2: The paper is registered and the phase is Submission; now the two states can diverge on the content of PID
- Δ_3: The paper is registered, and both the original trace and the alternative trace have exhausted their to-be-produced values

And here is the informal reading of the relations in the case of REV:

- Δ_1: Either the paper PID is not registered yet or the phase is not yet Reviewing, so the two states are equal
- Δ_2: The paper is registered and the phase is Reviewing but the paper's Nth review is not registered yet, so the two states are still equal
- Δ_3: Both the paper and its Nth review are registered and the phase is Reviewing; now the two states can diverge on the content of the review
- Δ_4: The phase is either Reviewing or higher (e.g., Discussion), both traces have exhausted their Reviewing-tagged values, meaning that the remaining to-be-produced values must be Discussion-tagged[4] and are required to be equal; now the states must be equal too

The smooth transition between consecutive components Δ_i and Δ_{i+1} that impose different state equalities is ensured by a suitable INDEPENDENT-ACTION/REACTION strategy—which does not show up in the relations themselves, but only in our proofs that the relations constitute a linear network of unwindings. For PAP$_2$, the crucial part in the proof is the strategy for transitioning from Δ_2 to Δ_3, with emptying the value lists vl and vl_1 *at the same*

[4] Remember that, for REV, the values are pairs, each consisting of a review content tagged with a conference phase that witnesses when the content has been added.

time: by INDEPENDENT ACTION, tr_1 will produce all values in vl_1 save for the last one, which will be produced by REACTION in sync with *tr* when *tr* reaches the last value in *vl*; this is possible since B guarantees $\mathsf{last}\ vl = \mathsf{last}\ vl_1$. And REV has a similar strategy for the crucial move from Δ_3 to Δ_4, this time with emptying not the entire value lists, but only their Reviewing-tagged components.

The exit component Δ_e collects unsound situations (s, vl) (that cannot be produced from any system trace *tr*), in order to exclude them via Exit. For PAP_2, such a situation is the paper's conference phase exceeding Submission while there are still values *vl* to be produced. The transition from Δ_2 to Δ_e occurs if a "premature" change-phase action is taken (from Submission to Bidding), while *vl* is still nonempty. For REV, Δ_e witnesses two unsound situations: when the phase exceeds Reviewing and either there is no Nth review or *vl* still contains Reviewing-tagged values.

In summary, employing the sequential unwinding theorem in our unwinding proofs had the benefit of allowing (and encouraging) a separation of concerns: the Δ_i's and the transitions between them constitute the main sequential flow of the phase-directed proof, while Δ_e collects all unsound situations, taking them out of our way.

About 20 safety properties are needed in the unwinding proofs, among which:

- A paper is never registered at two conferences
- An author always has conflict with his papers (DIS_1)
- A paper always has at least one author
- A user never reviews a paper with which he has conflict
- A user never gets to write more than one review for a given paper

For example, the first property in the above list was needed in the proof of PAP_2, to make sure that no value can be produced (i.e., $\varphi\ (\mathsf{head}\ vl)$ does not hold) from within Δ_1 or Δ_2, since no paper upload is possible without prior registration.

The verification took us three person months, which also counts the development of reusable proof infrastructure and automation. Eventually, we could prove the auxiliary safety properties quasi-automatically. By contrast, the unwinding proofs required interaction for indicating the INDEPENDENT-ACTION/REACTION strategy.

6 More Related Work

In Section 4.1 we have mostly analyzed literature covering theoretical security models, as a preparation for introducing our novel security model. In this section, we focus on related work with respect to the verification aspect.

6.1 Information Flow Security for Conference Management Systems

Arapinis et al. [4] introduce ConfiChair, a conference management system that improves on the state of the art by guaranteeing that the cloud, consisting of the system provider, cannot learn the content of the papers and reviews and cannot link users with their written reviews. This is achieved by a cryptographic protocol based on key translations and mixes. They encode the desired properties as strong secrecy (a property similar to Goguen-Meseguer noninterference) and verify them using the ProVerif [9] tool specialized in security protocols. Our work differs from theirs in three major aspects. First, they propose a cryptography-based enhancement, while we focus on a traditional conference system. Second, they manage to

encode and verify the desired properties automatically, while we use interactive theorem proving. While their automatic verification is an impressive achievement, we cannot hope for the same with our targeted properties which, while having a similar nature, are more nuanced and complex. For example, the properties PAP_2 and REV, with such precise indication of declassification bounds, go far beyond strong secrecy and require interactive verification. Finally, we synthesize functional code isomorphic to the specification, whereas they provide a separate implementation, not linked to the specification which abstracts away from many functionality aspects.

Qapla [50] is a middleware tool for enforcing access control policies for database systems. It has been deployed for the HotCRP conference management system. An interesting future work would be to use BD security to analyze the information flow content of the enforced policies. We expect that such an analysis would yield a certain overlap between the CoCon properties we have verified and the HotCRP properties that can be inferred from the Qapla policies.

6.2 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 [51], the CompCert C compiler [41] and the seL4 operating system kernel [38]. More recent developments include a range of microprocessors [32], Java and ML compilers [39, 43], and model checkers [20, 67].

Major holistic verification case studies in the area of information flow security are less well represented, perhaps due to the more complex nature of the involved properties compared to traditional safety and liveness [46]. They include a hardware architecture with information flow primitives [16] and a separation kernel [15], and noninterference for seL4 [52, 53]. A substantial contribution to web client security is the Quark verified browser [36]. Our own line of work is concerned with proof assistant verification of web-based system confidentiality grounded in BD security: it started in 2014 with CoCon and continued with the CoSMed social media platform [5] and its extension to a distributed model, CoSMeDis [6].

Outside the realm of proof-assistant based work, Ironclad [33] provides end-to-end security guarantees down to the binary code level and across the network. The information flow properties discussed in [33] focus on controlling *where* in the program information is declassified, e.g., in trusted declassification functions. 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.

6.3 Automatic Analysis of Information Flow

There are quite a few programming languages and tools aimed at supporting information flow secure programming—such as Jif [1] and its distributed extension Fabric [42], LIO [22] and its distribuetd extension Hails [22], Paragon [10], Spark [2], Jeeves [68] and Ur-Web [11]—as well as information flow tracking tools for the client side of web applications [8, 12,27]. The properties specifiable in these tools are significantly weaker (and more tractable) compared to those we considered in this paper.

We believe the future of information flow security verification will see an increased cooperation between fully automatic tools and proof assistants; the former being employed for wide-covering lightweight properties and the latter being employed more sparingly, for heavier properties of clearly isolated relatively small cores of systems. Compositionality results for information flow security [6, 26, 28, 45, 55] will play a key role in achieving such a cooperation on a well-understood semantic basis.

Acknowledgments

Much of the work reported here has been pursued while the second and third authors were employed at the Chair of Logic and Verification of Technische Universität München, Germany. We are indebted to the reviewers of the conference version of this paper for useful comments and suggestions, which led to the significant improvement of the presentation. We gratefully acknowledge support from DFG through grant "Security Type Systems and Deduction" (Ni 491/13-2), part of "RS^3 – Reliably Secure Software Systems" (SPP 1496) and from EPSRC through the grant "Verification of Web-based Systems (VOWS)" (EP/N019547/1). The authors are listed in alphabetical order.

References

1. Jif: Java + information flow, 2014. `http://www.cs.cornell.edu/jif`.
2. SPARK, 2014. `http://www.spark-2014.org`.
3. The Scala Programming Language, 2016. `http://www.scala-lang.org`.
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. CoSMed: A Confidentiality-Verified Confernece Management System. In *ITP*, 2016.
6. 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.
7. E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation, 1975. Technical Report MTR-2997, MITRE, Bedford, MA.
8. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, pp. 159–178, 2014.
9. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *LICS*, pp. 331–340, 2005.
10. 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.
11. A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL*, pp. 153–165, 2015.
12. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, pp. 50–62, 2009.
13. M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *POST*, pp. 265–284, 2014.
14. E. S. Cohen. Information transmission in computational systems. In *SOSP*, pp. 133–139, 1977.
15. 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.
16. 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. *Journal of Computer Security*, 24(6):689–734, 2016.
17. R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. Model checking information flow in reactive systems. In *VMCAI*, pp. 169–185, 2012.
18. The EasyChair conference system, 2014. `http://easychair.org`.
19. The HotCRP conference management system, 2014. `http://read.seas.harvard.edu/~kohler/hotcrp`.

20. 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.
21. R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In *FOSAD*, pp. 331–396, 2000.
22. D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. *Journal of Computer Security*, 25(4-5):427–461, 2017.
23. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.
24. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pp. 75–87, 1984.
25. D. Gollmann. *Computer Security*. Wiley, 2nd ed., 2005.
26. S. Greiner and D. Grahl. Non-interference with what-declassification in component-based systems. In *CSF*, pp. 253–267. IEEE Computer Society, 2016.
27. 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.
28. 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.
29. F. Haftmann. *Code Generation from Specifications in Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2009.
30. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117, 2010.
31. J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1), 2008.
32. 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.
33. 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.
34. P. Hou, P. Lammich, and A. Popescu. This paper's website. `http://andreipopescu.uk/papers/CoConExtended.html`.
35. IEEE Symposium on Security and Privacy. Email notification, 2012.
36. D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security*, pp. 113–128, 2012.
37. S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *CAV*, pp. 167–183, 2014.
38. 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.
39. R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: a verified implementation of ML. In *POPL*, pp. 179–192, 2014.
40. B. W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.
41. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
42. J. Liu, O. Arden, M. D. George, and A. C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.
43. A. Lochbihler. Java and the Java memory model—A unified, machine-checked formalisation. In *ESOP*, pp. 497–517, 2012.
44. H. Mantel. Information flow control and applications - bridging a gap. In *FME*, pp. 153–172, 2001.
45. H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, University of Saarbrücken, 2003.
46. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 605–607. 2011.
47. D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, 1987.
48. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *In Proc. IEEE Symposium on Security and Privacy*, pp. 79–93, 1994.
49. J. McLean. Security models. In *Encyclopedia of Software Engineering*, 1994.
50. A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy compliance for database-backed systems. In *USENIX Security*, pp. 1463–1479, 2017.

51. J. S. Moore, T. W. Lynch, and M. Kaufmann. A mechanically checked proof of the $\mathrm{amd5_k86^{tm}}$ floating point division program. *IEEE Trans. Computers*, 47(9):913–926, 1998.
52. 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.
53. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *CPP*, pp. 126–142, 2012.
54. T. C. Murray, A. Sabelfeld, and L. Bauer. Special issue on verified information flow security. *Journal of Computer Security*, 25(4-5):319–321, 2017.
55. T. C. Murray, R. Sison, and K. Engelhardt. COVERN: A logic for compositional verification of information flow control. In *EuroS&P*, pp. 16–30. IEEE, 2018.
56. T. Nipkow and G. Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.
57. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
58. C. O'Halloran. A calculus of information flow. In *ESORICS*, pp. 147–159, 1990.
59. G. J. Popek and D. A. Farber. A model for verification of data security in operating systems. *Commun. ACM*, 21(9):737–749, 1978.
60. Y. M. Ronald Fagin, Joseph Y. Halpern and M. Vardi. *Reasoning about knowledge*. MIT Press, 2003.
61. J. Rushby. Noninterference, transitivity, and channel-control security policies. Tech. report, dec 1992.
62. P. Y. A. Ryan. Mathematical models of computer security. In *FOSAD*, pp. 1–62, 2000.
63. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
64. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
65. D. Sangiorgi. On the bisimulation proof method. *Math. Struct. Comp. Sci.*, 8(5):447–479, 1998.
66. D. Sutherland. A model of information. In *9th National Security Conf.*, pp. 175–183, 1986.
67. S. Wimmer and P. Lammich. Verified model checking of timed automata. In *TACAS*, pp. 61–78, 2018.
68. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, pp. 85–96, 2012.