A Conference Management System with Verified Document Confidentiality

Sudeep Kanav, Peter Lammich, and Andrei Popescu

Fakultät für Informatik, Technische Universität München, Germany

Abstract. We present a case study in verified security for realistic systems: the implementation of a conference management system, whose functional kernel is faithfully represented in the Isabelle theorem prover, where we specify and verify confidentiality properties. The various theoretical and practical challenges posed by this development led to 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 [15]: "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 improved somewhat in the past ten years, with mature software systems such as Jif [1] offering powerful and scalable information flow technology integrated with programming.

However, the state of the art in information-flow security models [24] is still far from finding its way towards applications to real-world systems. If we further restrict attention to *mechanically verified* work, the situation is even more dramatic, with examples of realistic system verification [3,8,28] being brave exceptions. This is partly explained by the complexity of information-flow properties, which is much greater than that of traditional functional properties [23]. However, this situation is certainly undesirable, in a world where confidentiality and secrecy raise higher and higher challenges.

In this paper, we take on the task of implementing, and verifying the confidentiality of, a realistic system: CoCon,[1] a full-fledged conference system, featuring multiple users and conferences and offering much of the functionality of widely used systems such as EasyChair [10] and HotCRP [11].

Conference systems are widely used in the scientific community—EasyChair alone claims one million users. Moreover, the information flow in such systems possesses enough complexity so that errors can sneak inside implementations, sometimes with bitter–comical consequences. Recently, Popescu, as well as the authors of 267 papers submitted to a major security conference, initially received an acceptance notification, followed by a retraction [19]: "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."[2]

[1] A running version of CoCon, as well as the formal proof sources, are available at [20].

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

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

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 [11] 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 a PC member at the paper's conference

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 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

We will be concerned with properties restricting the information flow from the various 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.

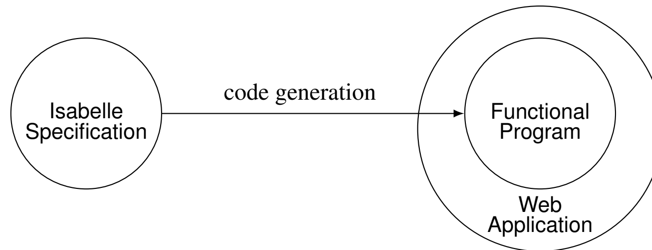
We specify CoCon's kernel using the proof assistant Isabelle [29, 30], with which we formulate and verify confidentiality. The functional implementation of this kernel is automatically synthesized from the specification and wrapped into a web application offering the expected behavior of a conference system as a menu-based interface.

A first contribution of this paper is the engineering approach behind the system specification and implementation (§2). To keep the Isabelle specification (§3) manageable, yet faithful to the implementation and therefore reach a decent balance between trust and usability, we employ state-of-the-art theorem proving and code synthesis technology towards a security-preserving layered architecture.

A second contribution is a novel security model called *bounded-deducibility* (BD) security, born from confronting notions from the literature with the challenges posed by our system (§4). The result is a reusable framework, applicable to any IO 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. We endow this framework with a declassification-oriented unwinding proof technique (§5).

Our third and last contribution is the verification itself: the BD security framework, its general unwinding theorem, and the unwinding proofs for CoCon's confidentiality properties expressed as instances of BD security are all mechanized in Isabelle.

2 Overall Architecture and Security Guarantees



The architecture of our system 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 in a functional programming language
- The translated program is wrapped in a web application

Isabelle Specification We specify the system as an input–output 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/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 [17], a program in the functional fragment of Scala [2] 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).

Overall Security Guarantees Our Isabelle verification targets information-flow properties. 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). The question arises as to what guarantees we have that the properties we verified formally for the specification also hold for the overall system. E.g., if we prove in Isabelle that users never learn the content of other users' papers, how can we be sure that this is actually the case when using the web interface? We do not have a formal answer to this, but only an informal argument in terms of the trustworthiness of two trusted steps.

First, we need to trust Isabelle's code generator. Its general-purpose design is very flexible, supporting program and data refinement [17]. In the presence of these rich features, the code generator is only known to preserve partial correctness, hence safety properties [16, 17]. 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. To acquire this trust, we make sure that the web application acts as a stateless interface to the step function: upon a user request, all it does is invoke "step" (one or multiple times) with input from the user and then process and display the output of the step function. The third-party libraries used by our web application also have to be trusted to not be vulnerable to exploits.

In summary, the formal guarantees we provide in Isabelle have to be combined with a few trusted steps to apply to the whole system. Our verification targets only the system's implementation logic—lower-level attacks such as browser-level forging are out of its reach, but are orthogonal issues that could in principle be mitigated separately.

3 System Specification

The system behaves similarly to EasyChair [10], 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. The system 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 with "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 A user can list the conferences awaiting submissions (i.e., being in 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.

Discussion All PC members having no conflict with a paper can see its reviews and can add comments. 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.

3.1 State, 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. 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: $\mathsf{Paper} = \mathsf{String} \times \mathsf{String} \times \mathsf{Paper_Content} \times \mathsf{List}(\mathsf{Review}) \times \mathsf{Dis} \times \mathsf{Dec}$

We keep different versions of the decision and of each review, as they may transparently change during discussion: $\mathsf{Dec} = \mathsf{List}(\mathsf{String})$ and $\mathsf{Review} = \mathsf{List}(\mathsf{Review_Content})$ 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: chair, PC member, paper author, or paper reviewer (the last two roles also containing paper IDs).

```
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   news : ConfID → List(String)   phase : ConfID → Phase
```

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. E.g., 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.

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 ct* is an attempt to upload a new version of paper *pid* by modifying its content to *ct*. The **u-update actions** are similar, but also record the history of a document's versions. E.g., 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.

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 action.

We define the 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. The initial state of the system, $\mathsf{istate} \in \mathsf{State}$, is the one with a single user, the voronkov, and a dummy password (which can be changed immediately). The step function and the initial state are the only items exported by our specification to the outside world.

4 Security Model

Here 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 [38] that factors in declassification triggers and bounds.

4.1 Relevant Literature

There is a vast amount of literature on information-flow security, with many variants of formalisms and verification techniques. An important distinction is between notions that completely forbid information flow (between designated sources and sinks) and notions that only restrict the flow, allowing some declassification. 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 [7, 13, 32], motivated by the desire to express the absence of information leaks of systems more abstractly and more precisely than by means of access control [4, 21]. Very influential were Goguen and Meseguer's notion of noninterference [13] and its associated proof by unwinding [14]. 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 [36] or process-algebra-based [12,35] settings or in purely semantic, event-system-based settings [25,26,31,38]. (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 [27] and Mantel [23] 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 absence or occurrence 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 $\textsc{Pap}_2$ 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* [38]. 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 [18] recast nondeducibility as a property called secrecy maintenance, in a multi-agent framework of "runs-and-systems" [33] 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.

On the practical verification side, Arapinis et al. [3] introduce ConfiChair, a conference system that improves on standard systems such as EasyChair by guaranteeing that "the cloud", consisting of the system provider/administrator, 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 [5] 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 systems not involving cryptography. 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. E.g., properties like $\textsc{Pap}_2$ and $\textsc{Rev}$, with such flexible indications 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.

Restriction of Information Flow A large body of work on declassification was pursued in a language-based setting. Sabelfeld and Sands [37] give an overview of the state of the art up to 2009. Although they target language-based declassification, they phrase some generic dimensions of declassification most of which 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.

Sabelfeld and Sands consider another interesting instance of the "where" dimension, namely intransitive noninterference [22, 34]. 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.

Recently, the "when" aspect of declassification has been included as first-class citizen in customized temporal logics [6, 9], which can express aspects of our desired properties, e.g., "unless/until he becomes an author". Their work is focused on efficiently model-checking finite systems, whereas we are interested in verifying an infinite system. Combining model checking with infinite-to-finite abstraction is an interesting prospect, but reflecting information-flow security properties under abstraction is difficult problem.

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 §3.1, but for any IO automaton indicated by the following data. We fix 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 IO 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 §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 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 set-theoretic terms, *lower* bounds for Dec_{tr}. To this end, we further fix:

- a declassification bound $\mathsf{B} : \mathsf{List}(\mathsf{Val}) \to \mathsf{List}(\mathsf{Val}) \to \mathsf{Bool}$
- a declassification trigger $\mathsf{T} : \mathsf{Trans} \to \mathsf{Bool}$

The system is called *bounded-deducibility-secure* (*BD-secure*) if for all $tr \in \mathsf{Trace}$ such that never $\mathsf{T}\ tr$, it holds that $\{vl_1 \mid \mathsf{B}\ (\mathsf{V}\ tr)\ vl_1\} \subseteq \mathsf{Dec}_{tr}$ (where "never $\mathsf{T}\ tr$" means "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), 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.

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

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

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 [25, 27, 35] 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 observer 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 IO 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. BD security then captures our examples by suitably instantiating the observation and declassification parameters. For all our examples, we have the same observation infrastructure. We fix UIDs, the set of IDs of the observing users. We let $\mathsf{Obs} = \mathsf{Act} \times \mathsf{Out}$. Given a transition, γ holds iff the action's subject is a user in UIDs, and g returns the pair (action,output). $\mathsf{O}\ tr$ thus purges tr keeping only actions of users in UIDs.

The value infrastructure depends on the considered type of document. For PAP_1 and PAP_2 we fix PID, the ID of the paper of interest. We let $\mathsf{Val} = \mathsf{List}(\mathsf{Paper_Content})$. Given a transition, φ holds iff the action is an upload of paper PID, and f returns the uploaded content. $\mathsf{V}\ tr$ thus returns the list of all uploaded paper contents for PID.

The declassification triggers and bounds are specific to each example. For PAP_1, we define $\mathsf{T}(s, a, o, s')$ as "in state s', some user in UIDs is an author of PID or a PC member of some conference cid where PID is registered," formally:

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

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 a remote possibility that a user could infer that no version was submitted (by probing the current phases of the conferences in the system and noticing that none has reached the submission phase). 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 $vl \neq []$.

For PAP_2, the trigger involves only authorship, ignoring PC membership at the paper's conference—we take $\mathsf{T}(s, a, o, s')$ to be $\exists uid \in \mathsf{UIDs}.\ \mathsf{isAut}\ s'\ uid\ \mathsf{PID}$. Here 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 define $\mathsf{B}\ vl\ vl_1$ as $vl \neq [] \neq vl_1 \wedge \mathsf{last}\ vl = \mathsf{last}\ vl_1$, where the function last returns the last element of a list.

For REV, 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 paper contents, mutatis mutandis; in particular, f returns a pair (ph, rct) consisting of the conference's current phase and the updated review's content; hence V returns a list of such pairs. The trigger T is similar to that of PAP_2 but refers to review authorship rather than paper authorship. The bound B is more complex. Any user can infer that the only possiblities for the phase are Reviewing and Discussion, in this order—i.e., that 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. Moreover, any PC member having no conflict with PID can learn $\mathsf{last}\ ul$ (the last submitted version before Discussion), and wl (the versions updated during Discussion, public to non-conflict PC members); but (until T holds) nothing beyond these. So $\mathsf{B}\ vl\ vl_1$ states that vl decomposes as $ul \cdot wl$ as indicated above, vl_1 decomposes similarly as $ul_1 \cdot wl$, and $\mathsf{last}\ ul = \mathsf{last}\ ul_1$.

DIS needs rephrasing to be captured as BD security. It can be decomposed into:

- DIS_1: An author always has conflict with his papers
- DIS_2: A group of users learn nothing about a paper's discussion unless one of them becomes a PC member at the paper's conference *having no conflict with the paper*

DIS_1 is a safety property. DIS_2 is an instance of BD security defined as expected.

	Source	Declassification Trigger	Declassification Bound
1	Paper Content	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 PC Membership$^{\mathsf{N}}$ 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

4.5 More Instances

The above table shows an array of confidentiality properties formulated as BD security. They provide a classification of relevant roles, statuses and conference phases that are necessary conditions for degrees of information release. The observation infrastructure is always the same, given by the actions and outputs of a fixed group of users as in §4.4.

The table lists 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 PAP_1, PAP_2, REV and DIS_2 form the rows 2, 1, 3, and 6. In rows 9 and 10, the sources are the identities of the reviewers assigned to the 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, with or without the requirement of lack of conflict with the paper. Some triggers are also listed with "phase stamps" that strengthens the statements. E.g., row 2 contains a strengthening of the trigger discussed so far for PAP_1: "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 strong enough. This is the case of DIS_2 (row 6), since without the non-conflict assumption DIS_2 and DIS_1 would no longer imply DIS. By contrast, lack of conflict cannot be added to PC membership in 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.

Most of the declassification bounds are similar to those from §4.4. The row 10 property states that, unless one becomes a PC member having no conflict with a paper in the reviewing phase or a paper's author in the notification phase, one can't learn anything about the paper's assigned reviewers beyond what everyone knows: that reviewers are non-conflict PC members. If we remove the non-authorship restriction, then the user may also infer the number of reviewers—but, as row 9 states, nothing beyond this.

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 this situation and employ the result to the verification of our confidentiality properties.

5.1 Unwinding Proof Method

We see from 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" trace tr_1 starting in s whose value sequence is vl_1 and whose observation sequence is the same as that of *tr*.

In the tradition of unwinding for noninterference [14, 34], we wish to construct tr_1 from *tr incrementally*: as *tr* grows, tr_1 should grow nearly synchronously. In order for tr_1 to have the same observation sequence (produced by O) as *tr*, we need to require that the observable transitions of tr_1 (i.e., for which γ holds) be *identical* to those of *tr*.

As for the value sequences (produced by V), we face the following problem. In contrast to the unwinding relations studied so far in the literature, 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; then θ and B would certainly need to be synchronized. We resolve this by enlarging the domain of the unwindings to quaternary relations $\varDelta : \mathsf{State} \to \mathsf{List}(\mathsf{Val}) \to \mathsf{State} \to \mathsf{List}(\mathsf{Val}) \to \mathsf{Bool}$ that generalize both θ and B. Intuitively, $\varDelta\ s\ vl\ s_1\ vl_1$ 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 $\varDelta$ such that $\varDelta\ s\ vl\ s_1\ vl_1$ 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 $\varDelta\ s'\ vl'\ s_1\ vl_1$ holds
  - MATCH: There exist a valid transition (s_1, a_1, o_1, s_1') and a list of values vl_1' such that $\mathsf{consume}\ (s, a, o, s')\ vl_1\ vl_1'$ and $\varDelta\ s_1\ vl'\ s_1'\ vl_1'$ hold
- 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 $\mathsf{consume}\ a_1\ o_1\ vl_1\ vl_1'$ and $\varDelta\ s\ vl\ s_1'\ vl_1'$ hold

The intent is that BD security should hold if there exists an unwinding $\varDelta$ 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 lists. 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 $vl \neq [] \lor vl_1 = []$.

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:

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

The predicates iaction and reaction formalize INDEPENDENT ACTION (with its aforementioned strengthening) and REACTION, the latter being a disjunction of predicates formalizing IGNORE and MATCH. The predicate $\mathsf{exit}\ s\ v$ is defined as $\forall\ tr\ trn.\ (tr \cdot [trn]) \in \mathsf{Valid}_s \land \varphi\ trn \to f\ trn \neq v$. It expresses a safety property, and therefore can be verified in a trace-free manner. We can now prove that indeed any unwinding relation constructs an "alternative" trace tr_1 from any trace *tr* starting in a *P*-reachable state:

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

Unwinding Theorem. If $\mathsf{unwind}\,\Delta$ and $\forall vl\ vl_1.\ \mathsf{B}\ vl\ vl_1 \to \Delta\ \mathsf{istate}\ vl\ \mathsf{istate}\ vl_1$, then the system is BD-secure.

Proof ideas. The lemma follows by induction on $\mathsf{length}\ tr + \mathsf{length}\ vl_1$ (as discussed above about progress). The theorem follows from the lemma 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.

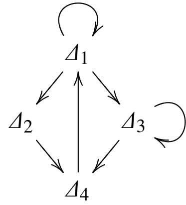


Fig. 3: A network of unwinding components

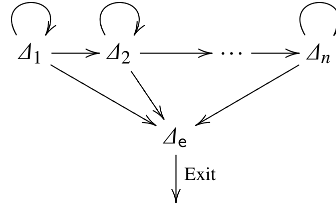


Fig. 4: A linear network with exit

5.2 Compositional Reasoning

To keep each reasoning step manageable, it is convenient to allow decomposing the single unwinding relation Δ into relations $\Delta_1, \ldots, \Delta_n$. Unlike Δ, a component Δ_i may unwind not only to itself but to any combination of Δ_j's. Technically, we define the predicate unwind_to just like unwind but taking two arguments instead of one: a first relation and a second relation to which the first one unwinds. We replace the single requirement unwind Δ with a set of requirements unwind_to Δ_i $(\mathsf{disj}\ (\mathsf{next}\ \Delta_i))$, where next Δ_i is a chosen subset of $\{\Delta_1, \ldots, \Delta_n\}$ and disj takes the disjunction of a set of predicates. This enables a form of sound compositional reasoning: if we verify a condition as above for each component Δ_i, we obtain an overall unwinding relation disj $\{\Delta_1, \ldots, \Delta_n\}$.

The network of components can form any directed graph —Fig. 3 shows an example. However, our unwinding proofs will be phase-directed, and hence the following linear network will suffice (Fig. 4): 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. For the first component, Δ_1, we need to verify that it extends B on the initial state.

5.3 Verification of Concrete Instances

We have verified all the BD security instances listed in §4.5. For each of them we defined a suitable chain of unwinding components Δ_i as in Fig. 4.

Recall from the definition of BD security that one needs to construct an alternative trace tr_1 (which produces the value sequence vl_1) from the original trace tr (which produces the value sequence vl). A chain of Δ_i's witnesses the strategy for such a construction, although it does not record the whole traces tr_1 and tr but only the states they have reached so far, s and s_1. The separation between Δ_i's is guided by milestones in the journey of tr and tr_1, such as: a paper's registration to a conference, conference phases, the registration of a relevant agent like a chair, a non-conflicted PC member, or a reviewer. E.g., Fig. 5 shows the unwinding components in the proof of PAP$_2$, where B $vl\ vl_1$ is the declassification bound ($vl \neq [] \neq vl_1\ \wedge$ last $vl =$ last vl_1) and the changes from Δ_i to Δ_{i+1} are emphasized.

Each property has one or more critical phases, the only phases when vl and vl_1 can be produced. E.g., for PAP$_2$, paper uploading is only available in Submission (while for REV, there is an update action in Reviewing, and an u-update one in Discussion). Until those phases, tr_1 proceeds synchronously to tr taking the same actions—consequently, the states s and s_1 are equal in Δ_1. In the critical phases, the traces tr and tr_1 will diverge, due to the need of producing 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*

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

Fig. 5: The unwinding components for the proof of PAP_2

except on certain components of the state, e.g., the content of a given paper (written $=_{\mathsf{PID}}$ for PAP_2), or of a given review, or of the previous versions of a given review, etc.

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. 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. For PAP_2, such a strategy for transitioning from Δ_2 to Δ_3 (with emptying vl and vl_1 at the same time) is the following: 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 $\mathsf{last}\ vl = \mathsf{last}\ vl_1$. The exit component Δ_{e} witnesses situations (s, vl) not producible 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 $\mathsf{Submission}$ with values vl still to be produced. Δ_{e} is reached from Δ_2 when a change-phase action occurs.

Several safety properties are needed in the unwinding proofs. For PAP_2, we use that there is at most one conference to which a paper can be registered—this ensures that no value can be produced (i.e., φ ($\mathsf{head}\ vl$) does not hold) from within Δ_1 or Δ_2, since no paper upload is possible without prior registration.

The verification took us two person months, during which we also developed reusable proof infrastructure and automation. Eventually, we could prove the auxiliary safety properties automatically. The unwinding proofs still required some interaction for indicating the INDEPENDENT-ACTION/REACTION strategy—we are currently exploring the prospect of fully automating the strategy part too, based on a suitable security-preserving abstraction in conjunction with an external model checker.

Conclusion Most of the information-flow security models proposed by theoreticians have not been confronted with the complexity of a realistic application, and therefore fail to address, or abstract away from, important aspects of the conditions for information release or restraint. In our verification case study, we approached the problem bottom-up: we faithfully formalized a realistic system, on which we identified, formulated and verified confidentiality properties. This experience led to the design of a flexible verification infrastructure for restricted information flow in IO automata.

Acknowledgement. Tobias Nipkow encouraged us to pursue this work. Several people made helpful comments and/or indicated related work: the CAV reviewers, Jasmin Blanchette, Manuel Eberl, Lars Hupel, Fabian Immler, Steffen Lortz, Giuliano Losa, Tobias Nipkow, Benedikt Nordhoff, Martin Ochoa, Markus Rabe, and Dmitriy Traytel. The research was supported by the DFG project Security Type Systems and Deduction (grant Ni 491/13-2), part of Reliably Secure Software Systems (RS^3). The authors are listed in alphabetical order.

References

1. Jif: Java + information flow, 2014. http://www.cs.cornell.edu/jif.
2. The Scala Programming Language, 2014. http://www.scala-lang.org.
3. M. Arapinis, S. Bursuc, and M. Ryan. Privacy supporting cloud computing: Confichair, a case study. In *POST*, pp. 89–108, 2012.
4. E. D. Bell and J. L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, Bedford, MA.
5. B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *LICS*, pp. 331–340, 2005.
6. 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.
7. E. S. Cohen. Information transmission in computational systems. In *SOSP*, pp. 133–139, 1977.
8. A. A. de Amorim, N. Collins, A. DeHon, D. Demange, C. Hritcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach. A verified information-flow architecture. In *POPL*, pp. 165–178, 2014.
9. 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.
10. The EasyChair conference system, 2014. http://easychair.org.
11. The HotCRP conference management system, 2014. http://read.seas.harvard.edu/~kohler/hotcrp.
12. R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In *FOSAD*, pp. 331–396, 2000.
13. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.
14. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pp. 75–87, 1984.
15. D. Gollmann. *Computer Security*. Wiley, 2nd ed., 2005.
16. F. Haftmann. *Code Generation from Specifications in Higher-Order Logic*. Ph.D. thesis, Technische Universität München, 2009.
17. F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *FLOPS 2010*, pp. 103–117, 2010.
18. J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Trans. Inf. Syst. Secur.*, 12(1), 2008.
19. IEEE Symposium on Security and Privacy. Email notification, 2012.
20. S. Kanav, P. Lammich, and A. Popescu. The CoCon website. http://www21.in.tum.de/~popescua/rs3/GNE.html.
21. B. W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, 1974.
22. H. Mantel. Information flow control and applications - bridging a gap. In *FME*, pp. 153–172, 2001.
23. H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, University of Saarbrücken, 2003.
24. H. Mantel. Information flow and noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 605–607. 2011.
25. D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symposium on Security and Privacy*, 1987.
26. 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.

27. J. McLean. Security models. In *Encyclopedia of Software Engineering*, 1994.
28. T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein. Noninterference for operating system kernels. In *CPP*, pp. 126–142, 2012.
29. T. Nipkow and G. Klein. *Concrete Semantics. A Proof Assistant Approach*. forthcoming. 310 pp. `http://www.in.tum.de/~nipkow/Concrete-Semantics`.
30. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
31. C. O'Halloran. A calculus of information flow. In *ESORICS*, pp. 147–159, 1990.
32. G. J. Popek and D. A. Farber. A model for verification of data security in operating systems. *Commun. ACM*, 21(9):737–749, 1978.
33. Y. M. Ronald Fagin, Joseph Y. Halpern and M. Vardi. *Reasoning about knowledge*. MIT Press, 2003.
34. J. Rushby. Noninterference, transitivity, and channel-control security policies. Tech. report, dec 1992.
35. P. Y. A. Ryan. Mathematical models of computer security. In *FOSAD*, pp. 1–62, 2000.
36. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
37. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
38. D. Sutherland. A model of information. In *9th National Security Conference*, pp. 175–183, 1986.