Compositional Verification of Information Flow Security for Distributed Web Applications

Thomas Bauereiß∗, Armando Pesenti Gritti†, Andrei Popescu†‡, Franco Raimondi‡
∗German Research Center for Artificial Intelligence (DFKI) Bremen, Germany
†School of Science and Technology, Middlesex University, UK
‡Institute of Mathematics Simion Stoilow of the Romanian Academy

Abstract—We present the design, implementation and information flow verification of a distributed social media platform. To support this verification effort, we have formalized a framework for composing information flow security guarantees in a distributed system, applicable to arbitrary input/output automata. The framework is more flexible than previous approaches from the literature in that it allows fine control of information declassification between the components.

I. INTRODUCTION

Recent years have seen an explosion of web-based systems aimed at sharing information between multiple users in a convenient, but controlled fashion. Examples include enterprise systems, social networks, e-commerce sites and cloud services. These systems often deal with confidential information such as credit card details, medical data, location information and sensitive documents. Unfortunately, most of these systems offer no guarantees concerning the prevention of unintended flow of information. Programming errors or ambiguous policy specifications leading to information leakage can have different degrees of severity and can affect many users [3,4]. These kinds of errors are likely to occur in practice, since web applications engage in intensive exchange of information with the environment. The Open Web Application Security Project (OWASP) includes sensitive data exposure (with typical severe impact) and the related missing function level access control (with moderate impact) in their influential top ten of web application security risks.

Such errors are difficult to prevent, or even understand, because information flow security is a complex, global property of a program. Consider a system that stores information on medical patients and offers selective web access to individuals and companies. A life insurance company agent should not be given direct access to sensitive information such as a patient suffering from cancer. Yet, this protection may not prevent propagation of information: If it is known that certain discounts are only offered to cancer suffering patients, then the agent can infer that sensitive information if they can access a patient’s available discounts.

The above situation is aggravated by the heavy interconnectivity of today’s web applications. For example, sensitive user information in cloud storage systems such as Amazon may be shared with social media platforms such as Twitter or Facebook and web-based email systems such as Gmail. While often the information sharing occurs via intuitively secure protocols, it is far more difficult to foresee the confidentiality issues arising synergistically in a network of web applications. An attacker can piece together information from different sources to a devastating effect [5]. What is needed is a solid, mathematically grounded understanding of how local confidentiality guarantees can be combined to deliver guarantees for the entire distributed system.

This paper makes progress towards this goal. It starts from the challenges of verifying confidentiality for a particular distributed system, and generalizes the findings into a mathematical theory of compositionality.

We employ a notion of information flow security introduced by Kanav et al. [38], called Bounded Deducibility (BD) Security. The appeal of BD Security for dealing with web applications rests in its versatile mechanism for specifying rich security policies, inspired by epistemic logic. This notion has been formalized in the proof assistant Isabelle/HOL [47] and then utilized in confidentiality guarantees for two running web applications:

- CoCon [2] (introduced as a case study by the BD Security authors), is a conference management system that gained some momentum in the academic community, having already been deployed for two verification-friendly conferences, TABLEAUX 2015 and ITP 2016.
- CoSMed [7] is a prototype secure social media platform. It was separately developed by Bauereiß et al. [13] to serve the functionality and security needs of a charity organization [6].

The attacker models in [13,38] consider an arbitrary but fixed set of users, called observers, who interact with the system and from whom secret information should be kept confidential. It is verified that the systems indeed do not leak more information about the secret than specified by the respective policy.

However, so far BD Security has been lacking a theory of compositionality. The need, and indeed the opportunity, to develop such a theory on top of BD Security arose after a survey with potential users, which lead to the intention of CoSMed’s stakeholders to extend this monolithic system into a distributed system. Like in the diaspora∗ scheme [8], different communities should be able to clone the system and deploy it as nodes on different Internet locations. In addition, a user of one node should be able to establish friendship and share data with selected users of other nodes.

In this paper, we proceed to such an extension, in partnership with the main beneficiary. Concretely, we design and implement a distributed version of CoSMed, which we call CoSMedis, and extend its formal security guarantees. The extension consists of a communication infrastructure (Section
IV) which introduces new security-critical aspects:

- the ability to share secrets, namely, restricted-access post contents, between different nodes
- the ability to dynamically assign secret-sharing roles, by the designation of friends from remote nodes

The attacker model is lifted to sets of observing users at each node of the distributed system. Moreover, we assume that network traffic between the nodes is also observable. The only exception is confidential content of communication, which we assume to be inaccessible to attackers (implemented by encrypting communication). This is in line with assuming a Dolev-Yao-style network attacker, who can inspect and control communication without breaking encryption.

After a detailed analysis of how to cope compositionally with the security of the new features of CoSMeDis, we formulate a solution in general terms: a reusable framework for composing information flow security guarantees of input/output automata (Section V). At its heart lies a formal theorem for composing locally verified BD Security instances to a global BD Security property of the entire distributed system (Sections V-C and V-E). The theorem is policy-agnostic, in that it composes policies without any restriction on their content. This flexibility allows the seamless instantiation of the theorem in a variety of contexts, extending to CoSMeDis all the previously established CoSMed guarantees (Section VI). The compositionality theorem is complemented by a transfer theorem, used to customize the global guarantees into more intuitive statements.

In summary, we make two main contributions:

- a new framework for compositional information flow security able to cope with flexible declassification policies
- the use of the framework to provide formal confidentiality certification to a practical distributed web-based system

Because everything is mechanically verified in Isabelle, in this paper we will focus on informal explanations for the theorems, to help develop the reader’s intuition. We start with concrete challenges raised by our case study and move towards general formulations that apply well beyond the case study. The formal proofs and CoSMeDis’s source code and documentation are available online [1]. For illustration, we have deployed two sample nodes of the running system [9].

II. RELATED WORK

We split the discussion of related work according to our two main contributions: compositional information flow security and formal engineering and verification.

A. Compositional Information Flow Security

Research in information flow security has a rich history, beginning with early notions such as noninterference [28] and nondeducibility [54] (of which BD Security is a generalization). While our work, as well as the previous work on BD Security, are latecomers in the area, they fulfill a real need: modeling and verifying the rich information exchange the web applications engage in with their environment—consisting of their users and other web applications.

First, since web applications are all about information flow, policies for them should be sufficiently fine-grained: Together with undesired flows, they should not also prevent desired flows. Using a well-known taxonomy of controlled information release (typically called declassification) due to Sabelfeld and Sands [52], they should be able to describe precisely what information can be released (e.g., the content of a document), by whom (e.g., the owners or their designated “friends”), as well as where and/or when (e.g., only after the document is marked by the owner as public). Second, the policies should be stated as locally as possible (e.g., as self-contained properties of each node in a network), and then composed into policies for larger systems, such as distributed networks of web applications.

Our case study is a distributed social media platform. There are several approaches in the literature specifically targeting systems involving online social networks, e.g. using distributed architectures [16,21,36] or end-to-end encryption [10,11,31]. The approach underlying CoSMed does not employ such mechanisms (although it could be strengthened, e.g., with end-to-end encryption), but it is implemented as a client-server web application and verified to securely control information flow. We extend this approach to support a network of communicating servers, each with its own set of clients.

In practice, security requirements in web applications are typically implemented using access control. In particular, for web applications involving social networks, relationship-based access control [27,48] supports policies where access control decisions depend on connections in the social graph, such as friendship links. However, as explained above, we aim for more than access control: We want to protect confidential information not only from direct illegitimate access, but also from leaking to unintended recipients who draw inferences based on their observations of the system behavior. Such inferences can easily evade access control. For example, consider a social media application that allows users to re-post content submitted by other users under their own name, but annotated with the name of the original poster. Assume an observer sees such a post p by user u, originally submitted by user u'. If the original post is not publicly visible, then it can be inferred that it is restricted to friends, hence u must have a friendship relation with u'. If friendship information is considered to be confidential, then this constitutes an information leak, even though the observer never directly tried to access the friend list of u or u'. In short, information flow control aims to detect such implicit leaks as well. However, there is a large variety of approaches in the literature regarding the concrete features and security goals that are targeted.

Language-based approaches [51] focus on a specific programming language. Since they operate on the level of language syntax, they often achieve an impressive degree of automation. For example, Jif [46] extends Java with security labels for data values, and enforces security via a combination of static and run-time checking. It supports control over who may declassify information, but not what is declassified. Joana [34] checks noninterference of Java programs via static program analysis. Control of declassification is limited to where in the program declassification may occur. Jeeves [55] extends a core language for functionality with a language for flexible security policies. RF* [12] uses a relational Hoare logic to reason
about 2-safety properties of probabilistic programs, including language-based notions of information flow security. Secure multi-execution [25,50] is a run-time enforcement technique where multiple copies of a program are executed, one for each security level, controlling the information flows between the levels. Secure program partitioning [17,56] produces a distribution of the code and data in a program onto different hosts according to their different trust levels (e.g. trusted web server and untrusted client-side browser). While each of the aforementioned approaches supports certain forms of declassification, they do not consider the setting and the kind of compositional reasoning that we aim for: given a network of communicating, nondeterministic systems, derive a global, complex information flow security property from the local security properties of the components. DStar [57] and Fabric [40] do consider distributed systems, but only support control over which processes or principles may declassify information, not what is declassified. The WHAT&WHERE security property [41] does allow control over what is declassified by concurrent programs, but only in a non-interactive setting, not suitable for web applications.

By contrast, system-based approaches work with security properties expressed directly on the semantics, on variants of event systems or input/output-automata. Early work following this approach [44] has observed that even seemingly strong security properties are not preserved under composition in general. Consequently, comprehensive frameworks have been developed for the composition of security properties in various settings, e.g. event systems [43], reactive systems [49] or process calculi [15,26]. However, these frameworks do not consider declassification, except for very specific notions such as intransitive noninterference [42]. Chong and van der Meyden [18] discuss information flow policies (called architectures), where filter functions are used to restrict what information may flow between domains, together with an interpretation of the resulting security properties in terms of an epistemic logic. However, they do not consider compositional reasoning in our sense, i.e. composing the security properties of multiple systems. The same applies to the work on temporal logics and model checking approaches for hyperproperties [20], of which information flow security properties are an instance. Greiner and Grahl [29] present a compositionality result that supports what-declassification control, specified in terms of equivalence relations on communication events, but it cannot express dynamically changing confidentiality requirements—as needed for web applications in general and for CoSMeDis in particular: for example, whether a given post \( p \) by user \( u \) is confidential for an observer depends on the visibility setting of \( p \) and/or the friendship status between \( u \) and the observer.

The most general compositionality result we found in the literature is a recent one by Guttman and Rowe [32], formulated on top of blur operators, which are similar to the declassification bounds used for BD Security. However, their result does not allow the sharing of secrets: It identifies a subnetwork of nodes that handle secret information, and shows that security of the whole network amounts to security of this subnetwork. This is insufficient for our purposes, as in CoSMeDis, all nodes of the network potentially handle secret information.

In summary, (to the best of our knowledge) all the previous approaches, be they language-based or system-based, do not consider the combination of features we need for distributed web applications such as our case study CoSMeDis. In this paper, we show that BD Security (itself a system-based approach), is properly equipped for this task: its customizable bound and trigger operating at the trace level can fine-tune the dimensions of declassification for expressing precisely what one wants, and nothing more. Notwithstanding their versatility, we show that local BD Security policies can be composed towards holistic guarantees. As one might expect, however, compositionality is not an entirely free lunch. The local properties have to be stated in a sufficiently strong and communication-sensitive way—we develop a methodology to achieve this.

B. Formal Engineering and Verification of CoSMeDis

For CoSMeDis, we use a proof assistant and a code generator to obtain precise and faithful guarantees of confidentiality—they refer to a detailed model of the actual running code, and not to an abstraction. Unlike in most model-checking approaches, we prove properties about (a network of) infinite-state systems.

CoSMeDis belongs to a small, but expanding club of realistic systems proved to be secure using proof assistants, which includes an aircraft microprocessor [35] (in ACL2), a hardware architecture with information flow primitives [23] (in Coq), a separation kernel [22] (in HOL2), a noninterferent operating system kernel [45] (in Isabelle/HOL), a secure browser [37] (in Coq), and an e-voting system [39] (using the KeY theorem prover jointly with the Joana information flow analyzer).

III. THE ORIGINAL CoSMED

This section recalls the original CoSMed system and its confidentiality verification [13] based on BD Security [38]. It also introduces a specific instance of a BD Security property that will be used as a running example throughout the paper.

A. System

CoSMed is a social media platform loosely inspired by online social networks such as Facebook. It allows users to register and post information, and to restrict access to this information based on friendship relationships established between users. For example, the user Alice can log into the web site, and can browse through posts made by other users. She can create new posts herself, e.g. a comment on a sports event. By default, this post is visible to her friends only. She can add new friends by looking up their profile, e.g. the profile of Bob, and requesting friendship by optionally entering a greeting text and clicking the submit button. When Bob approves the request, they become friends, and Bob can now see Alice’s sports comment. Alice can also edit her posts and set their visibility level—either friends-only or public—at any time. The system has one user with special powers, called the administrator (or admin, for short), who is responsible for approving the creation of users.

State. CoSMed has a mutable state, which stores information about users, posts and the relationships between them: user and post IDs, along with maps associating content to these IDs. For example: a user ID has an associated name,
email address and info; a post ID has an associated title, text, image, owner ID and visibility; a friendship request is identified by two user IDs (the sender and the recipient) and has an associated greeting text created by the sender; the friendship status (“friend” or “not friend”) is stored as a symmetric association between user IDs.

Actions. Users can interact with the system via actions of different types: for creating, deleting, updating and reading items in the system, where an item can be a user, a post, a friendship request or a friendship status. There are also actions for listing items according to various filters or criteria—e.g., a user can list all posts, or all posts of a given friend, or all his friends, or all the friends of a given friend, etc. Since the system is multi-user, each action also contains the user ID of its issuer, denoted by userOf(a).

Outputs. When a user requests an action, the system first checks if the action is enabled (i.e., allowed), in which case the action is applied and the output is returned to the user. If it is not enabled, an error message is emitted and the state remains unchanged.

The above behavior is formalized as an I/O automaton Aut = (State, Act, Out, σ₀, →), where σ₀ ∈ State is the (empty) initial state and → : State × Act × Out × State →Bool is the transition relation. The inputs of the automaton are called actions. For example, an action that updates the content of a post has the form (updatePost, uid, pid, pst). Here, updatePost is a label indicating the particular type of action, uid is the ID of the acting user, pid is the ID of the post, and pst is the new content. (The action parameters also include user passwords, but we omit them to simplify the notation.) σ → a, o, σ′ has the following reading: if a user takes action a while the system is in state σ, the system responds producing output o and changing the state to σ′.

A system transition is a tuple trn = (σ, a, o, σ′) such that σ → a, o, σ′ holds. The states σ and σ′ are called the transition’s source and target. The transition’s action a is also denoted by actOf(trn). A system trace is a sequence tr = (trn₁, . . ., trnₙ) of transitions such that the source of the first transition (if any) is σ₀ and the target of each transition is the source of its successor in the sequence. The end state of a trace tr, written endState(tr), is the target state of the tr’s last transition if tr is non-empty, and the initial state σ₀ otherwise. Note that a system trace interleaves transitions containing actions from, and outputs to, different users. We let Trans denote the set of transitions and Trace the set of traces.

B. Security Model

Bauereiß et al. proved for CoSMed the following type of confidentiality properties: Only under the circumstances specified by the policy may users learn information about the documents of the system. Our running example in this paper will be the following:

(P₁) A group of users can learn nothing about the updates to a post content beyond the existence of an update unless one of them is the admin or the post’s owner, or becomes friends with the owner, or the post is marked as public.

Let us analyze what (P₁) expresses, and how this is formalized. It focuses on the confidentiality of the content of a given post by a CoSMed user. It does not describe a policy for access control to this data, but rather a policy for information flow control. Indeed, it does not state that the post content can not be accessed, but something stronger: that information about it can not flow. Hence we write “learn” instead of “access.”

The (partial) secret that (P₁) refers to is the content of a particular post, say, stored in the system under the ID PID. Since the content can be updated by the owner several times, we need to speak about a sequence of secrets: all the content updates, which are the same as all the versions of this content held in the state during a run of the system. Formally, one defines a domain Sec of secrets, a filter isSec : Trans → Bool and a secret-producing function getSec : Trans → Sec. This yields a function getSecσ : Trace → Sec* that extracts a sequence of secrets from a trace transition-wise, by filtering with isSec and then applying getSec to each of the trace’s transitions. For example, consider the trace tr = (trn₁, trn₂, trn₃) where isSec is only true for trn₁ and trn₃, then getSecσ(tr) = (getSec(trn₁), getSec(trn₃)). For (P₁), Sec is obviously taken to consist of post contents. Moreover, isSec(σ, a, o, σ′) holds if a is a post-update action for PID, i.e., has the form (updatePost, uid, PID, pst) for some uid, pst, and the transition is successful, as signaled by an “OK” output. The function getSec for (P₁) extracts the post content from a: getSec(σ, a, o, σ′) = pst.

The observers (possible attackers) are here a group of users, say, with their IDs in a fixed (but arbitrary) set UID. Formally, observations are managed similarly to the secrets. One defines the observation domain Obs, the filter isObs : Trans → Bool, and the production function getObs : Trans → Obs, and then defines getObs : Trace → Obs* by filtering with isObs and applying getObs transition-wise. For (P₁), Obs is taken to be the set of all action-output pairs. isObs(σ, a, o, σ′) is true just in case the action is issued by one of the designated observers (userOf(a) ∈ UID), and getObs retrieves the action and output form the transition (getObs(σ, a, o, σ′) = (a, o)).

So what can the observers learn about the secrets? According to (P₁), nothing beyond the (non-)existence of an update. This is formalized as a bound, i.e., a binary relation B between sequences of secrets. We require sequences of secrets related by B to be exchangeable, without interfering with the observations. Intuitively, given the sequence of secrets sl produced by a system trace, the set {sl′ | B(sl, sl′)} represents the amount of uncertainty of the observers: for what they know, sl could be any sl′ in this set. Hence, B describes a lower bound on the uncertainty, or in other words, an upper bound on the allowed declassification from the designated secrets to the designated observers. For (P₁), B(sl, sl′) is simply defined as: sl empty implies sl′ empty. Thus, the only piece of information that can flow to the observers is a harmless one: that no update has been performed yet (possibly because the post has not been created yet).

Finally, (P₁) prescribes a legitimate way out of the declassification bound: if one of the observers has or acquires a role (system admin, post owner or owner’s friend) or an intended declassification happens (the post is made public). This is formalized as a trigger, i.e., a unary predicate T on system transitions. The bound B is only imposed unless the trigger T occurs. In this case, T (σ, a, o, σ′) is defined as a property
of the transition’s target state \( \sigma' \): that the system’s admin is in UIDs \( \text{admin}(\sigma') \in \text{UIDs} \), the registered owner of PID is in UIDs \( \text{owner}(\sigma', \text{PID}) \in \text{UIDs} \) or has a user in UIDs as a registered friend \( \text{UIDs} \cap \text{friendUIDs}(\sigma', \text{owner}(\sigma', \text{PID}) \neq \emptyset) \) or the visibility of PID is public \( \text{visPost}(\sigma', \text{PID}) = \text{public} \). This formalizes the \textit{unless} part of \((P_1)\).

In summary, \((P_1)\) is specified w.r.t. an I/O automaton Aut (here, CoSMed’s specification) and is placed within an attacker model, consisting of

- a secrecy infrastructure \((\text{Secret}, \text{isSec}, \text{getSec})\)
- and an observation infrastructure \((\text{Obs}, \text{isObs}, \text{getObs})\)

Its security policy is specified by

- a declassification trigger \( T \)
- and a declassification bound \( B \)

In this context, \((P_1)\) states that \( O_{\text{isObs}}^{\text{getSec}} \) can not learn anything about \( S_{\text{isSec}}^{\text{getSec}} \) beyond \( B \) unless \( T \) occurs. Formally: For all system traces \( tr \) in which \( T \) never holds for any transition, and for all sequences of secrets \( st' \) such that \( B(S_{\text{isSec}}^{\text{getSec}}(tr), st') \) holds, there exists a system trace \( tr' \) such that \( S_{\text{isSec}}(tr') = st' \) and \( O_{\text{isObs}}^{\text{getSec}}(tr') = O_{\text{isObs}}^{\text{getSec}}(tr) \).

The above statement of confidentiality takes an epistemic logic perspective: Given any actual trace of events \( tr \), there exists an \textit{alternative} trace \( tr' \) that offers an equally valid explanation of the observations within the specified uncertainty/declassification bound.

\((P_1)\) is an instance of \textit{BD Security}, a framework introduced by Kanav et al. [38]. The framework allows to state and prove desired confidentiality properties by instantiating all of the above parameters: the I/O automaton, the attacker model and the security policy.

C. Formalization and Implementation

Bauereiß et al. implement the CoSMed I/O automaton in the proof assistant Isabelle/HOL. This is also where they verify the system for confidentiality, based on Kanav et al.’s BD Security framework.

The Isabelle/HOL code is automatically extracted into Scala code using Isabelle’s code generator [33]. Around this code, there is a thin layer of web-specific code written in the Scalatra framework [53]—this effectively turns the aforementioned I/O automaton into an API, able to communicate data in JSON format in response to HTTP requests. Finally, a web application (offering the standard user-interface conveniences) is built around this API kernel.

IV. THE DISTRIBUTED SYSTEM

In what follows, we present our novel contribution. We start with the specification and implementation of CoSMeDis, an extension of CoSMed with inter-node communication.

A. Communication Infrastructure

We extend the original system with mechanisms for communicating and sharing data with other nodes located at different sites across the Internet. All nodes will have identical behavior, i.e., will be CoSMeDis clones (but their internal states will of course be different due to their different interactions with users and among themselves).

Any number of nodes can be created. A node is designated by a unique ID, its URL. We implement an asymmetric communication model. Any two nodes with IDs \( nid_1 \) and \( nid_2 \) can agree on a client-server relationship: The client \( nid_1 \) makes a request and the server \( nid_2 \) approves it (both actions being triggered by the admin users of the corresponding nodes). After that, \( nid_2 \) can share its posts with \( nid_1 \). In addition, users of \( nid_2 \) are allowed to mark as friends selected users of \( nid_1 \). Hence, the admins are now responsible for setting up inter-node communication as well as approving local user creation. From a user perspective, the system looks just like CoSMed, but users can see posts from other nodes if the owner has granted access, and can add remote friends by selecting the node and entering the username.

To achieve the above, we extend the state of the system with communication infrastructure (the IDs and passwords of the registered client and server nodes) and shared data (inter-node friendship and shared posts). We also add new types of actions to support the desired communication: \( \text{sendServerReq} \) and \( \text{receiveClientReq} \), \( \text{sendPost} \) and \( \text{receivePost} \), and \( \text{sendUpdateRFriend} \) and \( \text{receiveUpdateRFriend} \). They come in pairs: there is an action on the receiving side to match that on the sending side. For successful communication, the parameters of these actions (consisting of user, node and post IDs, post content, etc.) must also match, in that what is being received must coincide with what is being sent. Here is the intended workflow and the matching patterns for these actions:

*Request server connection.* The admin \( uid_1 \) of a node \( nid_1 \) can issue a server request to another node \( nid_2 \) with the intention of establishing a client-server relationship. The corresponding action is \( \text{sendServerReq}(uid_1, nid_2, \text{request}) \), where \( \text{request} \) is the content of the request message. When the request reaches \( nid_2 \), the action \( \text{receiveClientReq}(uid_1, \text{request}) \) is triggered on \( nid_2 \), to the effect of recording in \( nid_2 \)’s state that \( nid_1 \) wishes to become a client.

*Connect client with server.* At a later time, the admin \( uid_2 \) of \( nid_2 \) can inspect and approve the request. This is done through the communication action \( \text{connectClient}(uid_2, nid_1, \text{id}) \), which registers, in \( nid_2 \)’s state, the node \( nid_1 \) as a client. The matching action on the \( nid_1 \)’s side is \( \text{connectServer}(uid_2) \), which registers, in \( nid_1 \)’s state, the node \( nid_2 \) as a server.\(^1\)

*Share posts.* After \( nid_1 \) and \( nid_2 \) have recognized each other as a client-server pair, other communication actions are possible. The admin \( uid_2 \) of the server \( nid_2 \) (or a program running on behalf of the admin) can send a local post \( pid \) at any time to the client \( nid_1 \), via \( \text{sendPost}(uid_2, nid_1, pid) \). This action will produce the output \( \text{pid}, \text{post}, \text{uid}_2 \), consisting of the post ID \( \text{pid} \), the content \( \text{post} \) of the post, the ID \( \text{uid}_2 \) of the post’s

\(^1\)Just like the interaction with the users involves user passwords, the communication between clients and servers involves authenticating keys. For example, the full syntax of the actions for connecting clients with servers is \( \text{connectClient}(uid_2, pid, uid_1, k) \) and \( \text{connectServer}(uid_2, k) \), where \( pid \) is the password of the user \( uid_1 \) (assumed to be \( uid_2 \)’s admin) and \( k \) is a key issued by the server \( nid_2 \) and sent to the client \( nid_1 \)—to be used for authenticating this server in later communication. To simplify notation, all the password and key parameters of the actions will be omitted.
owner, and information on the post’s visibility, \( v \). In this output, \( pid \) is copied from the action’s parameter, whereas all the other components are retrieved from \( nid_2 \)’s state. The matching action on the \( nid_1 \) side is \((\text{receivePost}, nid_2, pid, pst, uid_2, v)\). Sending an updated version of a previously shared post is possible, and will have the effect of updating the remote version. A flag is stored in the server node’s state for each shared post with each client, indicating whether the remote version is up to date.

**Assign remote friends.** Sharing a local post \( pid \) between the server \( nid_2 \) and the client \( nid_1 \) is at the discretion of \( nid_2 \)’s admin \( uid_2 \), which would typically send several posts in batch mode. However, the post owner \( uid_2 \) retains control on the remote access rights for his friends-only posts. Namely, the remote version of \( pid \) will only be accessible to users \( uid'_1 \) of \( nid_1 \) which \( uid_2 \) designates as remote friends. Remote friend designation is achieved through the action \((\text{sendUpdateRFriend}, uid_2, nid_1, uid'_1, st)\), which sends an update to the friendship-like permission user \( uid'_2 \) of \( nid_2 \) to user \( uid'_1 \) of \( nid_1 \)—the Boolean flag \( st \) indicates whether the friendship status is to be granted or revoked. The matching action on the \( nid_1 \) side is \((\text{receiveUpdateRFriend}, nid_2, uid'_2, uid'_1, st)\), which updates the indicated permission.

**B. Modeling the Distributed System**

We will eventually model a network of an arbitrary number of nodes. But to keep the discussion simple, we initially assume only two nodes, represented by two I/O automata \( Aut_1 \) and \( Aut_2 \). In Section V-E, we describe the n-ary case.

In our case study, \( Aut_1 \) and \( Aut_2 \) are identical—as CoSMeDis clones. However, this assumption will not be needed in our discussion. We shall use the indexes 1 and 2 to indicate the components of these automata, e.g., \( \text{State}_1 \), \( \text{Act}_1 \), \( \text{State}_2 \), \( \text{Act}_2 \), etc. An exception will be their transition relations, where we write \( \rightarrow \) for both rather than \( \rightarrow_1 \) or \( \rightarrow_2 \).

As seen in Section IV-A, communication proceeds by matching certain transitions of the two components: each sending transition with a corresponding receiving transition. We model matching as a relation match between the transitions of \( Aut_1 \) and those of \( Aut_2 \), taking match\((\sigma_1, a_1, o_1, o'_1), (\sigma_2, a_2, o_2, o'_2)\) to mean that either \((a_1, o_1)\) is a sending action-output pair and \((a_2, o_2)\) is the corresponding receiving action-output pair or vice versa. Thus, two transitions are matched if their actions are dual to each other and the parameters occurring in the sending action or in its output correspond to the input parameters of the receiving action. For example, the input parameters of a receivePost action have to match the output of the sendPost action.

Formally, \( \text{match}(\sigma_1, a_1, o_1, o'_1), (\sigma_2, a_2, o_2, o'_2) \) requires that the actions and outputs have one of the forms in Figure 1’s table, where \( \text{Nil}_1 \) and \( \text{Nil}_2 \) are the IDs of \( Aut_1 \) and \( Aut_2 \) or the symmetric forms, with \( \text{Nil}_1 \) and \( \text{Nil}_2 \) swapped. (It might be unclear why match should also depend on the transition’s states. Indeed, for CoSMeDis’s communication, it does not: we instantiate it independently of the states, only considering the actions and outputs. However, state-dependent matching would make sense if the components had a shared part of the state, so we don’t forbid it in our emerging framework.)

In our discussion, we distinguished separate component actions from communication actions. We write isCom\((a)\) to state that \( a \) is in the latter category. This predicate can be derived from the communication interface: isCom\((a)\) holds whenever there exist \( trn_1 \) and \( trn_2 \) such that match\((trn_1, trn_2)\) holds and \( a \) is the action of either \( trn_1 \) or \( trn_2 \).

With these preparations, we can define the distributed system as an I/O automaton, \( Aut_1 \times_{\text{match}} Aut_2 \), representing the communicating product of the components:

- Its set of states is \( \text{State}_1 \times \text{State}_2 \), with the initial state being the pair of initial states, \((\sigma^0_1, \sigma^0_2)\).
- Its set of actions is \( \text{Act}_1 + \text{Act}_2 + \text{Act}_1 \times \text{Act}_2 \), i.e., a disjoint union of \( \text{Act}_1 \) (representing separate actions of the first component), \( \text{Act}_2 \) (for separate actions of the second component), and \( \text{Act}_1 \times \text{Act}_2 \) (for joint communicating actions). We shall write \((1, a_1), (2, a_2), (a_1, a_2)\) for actions of the first, second, and third kind, respectively.
- Similarly, its set of outputs is \( \text{Out}_1 + \text{Out}_2 + \text{Out}_1 \times \text{Out}_2 \), and we use similar notations: \((1, o_1), (2, o_2), (o_1, o_2)\).
- Its transition system is defined in Figure 2. As can be seen, the SEP rules allow each component to proceed separately, whereas the COM rule allows matching communication transitions of any two components.

Note that a transition \( trn \) of \( Aut_1 \times_{\text{match}} Aut_2 \) has one of the following three forms:

\[
(1) \quad ((\sigma_1, \sigma_2), (1, a_1), (1, o_1), (\sigma'_1, \sigma_2)) \\
(2) \quad ((\sigma_1, \sigma_2), (2, a_2), (2, o_2), (\sigma_1, \sigma'_2)) \\
(3) \quad ((\sigma_1, \sigma_2), (a_1, a_2), (o_1, o_2), (\sigma'_1, \sigma'_2))
\]

In the first case, \( trn \) is completely determined by the \( Aut_1 \)-transition \( trn_1 = (\sigma_1, a_1, a_1, \sigma'_1) \) and by the \( Aut_2 \)-state \( \sigma_2 \)—we write \( trn = \text{sep}_1(trn_1, \sigma_2) \), marking that \( trn \) is given by the separate transition \( trn_1 \). Similarly, in the second case we write \( trn = \text{sep}_2(\sigma_1, trn_2) \), where \( trn_2 = (\sigma_2, a_2, o_2, \sigma'_2) \). In the third case, we write \( trn = \text{com}(trn_1, trn_2) \), marking that \( trn \) proceeds as a communication transition. Thus, any transition of \( Aut_1 \times_{\text{match}} Aut_2 \) has either the form \( \text{sep}_1(trn_1, \sigma_2) \), or \( \text{sep}_2(\sigma_1, trn_2) \), or \( \text{com}(trn_1, trn_2) \).

Given an \( Aut_1 \)-trace \( tr_1 \) and an \( Aut_2 \)-trace \( tr_2 \), we define \( tr_1 \parallel \text{match} \ tr_2 \), the communicating shuffle of \( tr_1 \) and \( tr_2 \), to be the set of all \(( Aut_1 \times_{\text{match}} Aut_2 )\)-traces obtained from shuffling (i.e., interleaving) \( tr_1 \) and \( tr_2 \)—the inductive definition is shown in Figure 3, where \( \cdot \) denotes concatenation and \( [] \) the empty trace.

**C. Implementing Communication**

The implementation of CoSMeDis follows a similar route to that of CoSMed: We extract Scala code from the Isabelle specification, embed it in a Scataltra API, and build a web application around it.

To make sure that sending actions are correctly matched with remote receiving actions, the implementation follows a transactional policy. For example, when \( Aut_1 \)'s admin issues a sendPost request through the user interface indicating the target node as \( Aut_2 \), the following happens:

- The sendPost action is run locally, producing the new state \( \sigma'_1 \) and the output \( o_1 \); but the new state is not yet committed.
If $o_1 = \text{outOK}$ (meaning sendPost was successful), a receivePost request is made remotely to Aut$_2$.

If Aut$_2$ responds with output outOK, the new state $\sigma'_1$ is committed at Aut$_1$.

CoSMeDis is delivered as a bundle, which can be installed at any location on the web to form a new node.

V. Composing Security Guarantees

Each CoSMeDis node is an extension of CoSMed. So when trying to prove confidentiality for CoSMeDis we look into how to extend the CoSMed confidentiality properties to properties of a single node, and then into how to compose node confidentiality to obtain guarantees for the entire system.

As we shall see, these two steps are not independent, but we have to proceed in a feedback loop.

A. Security Models for the Components

Our running example, $(P_1)$, limits the amount of information flowing from the content of a given secret: a post PID. In relation to this secret, we can distinguish two types of components: the node where it originates, say, Aut$_1$, and the other nodes which are possible receivers of the post.

For the originator, it is intuitive that we should be able to prove the same property $(P_1)$, regardless of the fact that now the system is communicating on more channels. Indeed, as far as Aut$_1$’s users are concerned, the notions of secret and observation are the same: the secrets are the updates to PID’s content, while the observations are the actions and outputs of a given set of Aut$_1$ users, say, UID$_{S_1}$. The communication actions do not interfere with this local security model: it is irrelevant for the observation power of an Aut$_1$ user if the post is being sent to another node. Therefore, the proofs done for the original CoSMed work without essential modifications for the communication-updated version—with the same trigger and the same bound. In summary, we can easily (re)prove:

$(P_1)$ A group of users UID$_{S_1}$ of Aut$_1$ can learn nothing about the updates to the content of Aut$_1$’s post PID beyond the existence of an update unless one of them is the admin or PID’s owner, or becomes friends with the owner, or PID is marked as public.

Now, consider a node Aut$_2$ that may potentially receive the content of the post PID from Aut$_1$. We can also prove a version of $(P_1)$ for the receiving end, *mutatis mutandis*:

$(P_2)$ A group of users UID$_{S_2}$ of Aut$_2$ can learn nothing about the updates to the content of Aut$_1$’s post PID beyond the existence of an update unless PID is being shared between Aut$_1$ and Aut$_2$ and [one of the users is the admin or becomes a remote friend of PID’s owner, or PID is marked as public].
Formally, (P₁) and (P₂) are instances of BD Security, specified by the attacker models and security policies shown in Figure 4, where NID₁ and NID₂ are the IDs of Aut₁ and Aut₂. The formalization of (P₁) is essentially the one sketched in Section III-B. In particular, the secrets are the updates to PID’s content as produced by updatePost actions, and the trigger refers to one of the users in UID₁ being the admin, or PID’s owner, or a friend. For (P₂), the secrets are also updates to PID’s content, but they are produced differently: by receivePost actions having Aut₁ as sender. Any update of PID’s content is received along with the owner’s ID uid and with any possible update v of the visibility status, which is recorded as the “remote visibility” stored for (NID₁, PID). The trigger first requires that PID has been shared, which is stored in Aut₂’s state as a remote post ID coming from Aut₁; then it makes requirements similar to (P₁)’s trigger, but referring to remote versions friendship and visibility. We write S₁ and O₁ instead of Sᵣ getSec and Oᵣ getObs, and similarly for S₂ and O₂.

### B. The Compositionality Challenge

Let us analyze how two properties such as (P₁) and (P₂) can be composed into a property for Aut = Aut₁ × Aut₂. The compound attacker model should be a form of communication-aware “sum”, or “union”, of those for (P₁) and (P₂). Since the Aut-traces are obtained by the communicating shuffle of Aut₁- and Aut₂-traces, the observations produced by Aut are themselves shufflings of those of the components. So it is natural to take Obs, the compound observation domain, to be Obs₁ + Obs₂ + Obs₁ × Obs₂—meaning, as usual, that an element of Obs will have either the form (1, o₁) or (2, o₂) or (o₁, o₂), where oᵢ ∈ Obsᵢ. For an Aut-transition trn, we define isObs(trn) and getObs(trn) as follows. isObs(trn) is true iff :

- trn has the form sep₁((1, o₁), (2, o₂)) and isObs₁((1, o₁)) holds, in which case we define getObs(trn) = (1, getObs₁((1, o₁)))
- or trn has the form sep₂(o₁, o₂) and isObs₂((o₁, o₂)) holds, in which case we define getObs(trn) = (2, getObs₂((o₁, o₂)))
- or trn has the form com((1, o₁), (2, o₂)) and isObs₁((1, o₁)) or isObs₂((2, o₂)) hold, in which case we define getObs(trn) = (getObs₁((1, o₁)), getObs₂((2, o₂)))

Similar constructions are performed for secrets. The compound domain, Sec, is taken to be Sec₁ + Sec₂ + Sec₁ × Sec₂, and isSec and getSec are defined correspondingly. This concludes the attacker model definition—we again write S and O instead of Sᵣ isSec and Oᵣ getObs.

Before moving to the definition of the compound security policy, let us first contemplate some existing or missing symmetries in the matching of observations and secrets of the two components. The notion of matching of transitions induces a notion of matching observations. In fact, the latter can be regarded as a stand-alone predicate matchO : Obs₁ × Obs₂ → Bool. For our case study, it is defined in the same way as match—since match does not actually depend on the states, but only on the action-output pairs, which also make up the observations. The corresponding communicating shuffle for observations, |matchO| : Obs₁ × Obs₂ → Pow(Obs⁺) (where Pow is the powerset operator), is obtained from matchO similarly to how |match| is obtained from match.

When trying to define a similar notion of matching for secrets, matchS : Sec₁ × Sec₂ → Bool, we encounter an anomaly: There are secrets on the receiving end (where (P₂) holds), produced by receivePost actions, that are not matched at the sending end (where (P₁) holds). Indeed, (P₁)’s only secret-producing actions are updatePost actions, which are non-communicating. The anomaly is easy to repair by amending (P₁)’s attacker model to factor in sendPost actions as well:

**Amendment 1.** (P₁)’s secrecy infrastructure (from Figure 4) is extended as highlighted below:

- Sec₁ = upd Post + snd Post. We use this notation to mean that secrets are now post contents annotated with the labels upd or snd, in order to distinguish between posts produced by an update action, (upd, pst), and posts produced by a sending action, (snd, pst). (A more standard way to write this set would be ( upd × Post ) ∪ ( snd × Post ).)
- isSec₁(σ, a, o, o′) iff
  
  o = outOK ∧ (∃uid, pst. a = (updatePost, uid, PID, pst) ∨
While restoring symmetry with respect to communication, this amendment creates disturbance in \((P_1)\)'s security policy, namely, in its bound: it is no longer the case that the observers can learn nothing about a sequence of produced secrets beyond possible emptiness. Now, it is also known that the contents of the updatePost and sendPost actions are correlated, in that what is being sent is precisely what was last updated. For example, (unless the trigger fires) from the sequence what is being sent is precisely what was last updated. For example, (unless the trigger fires) from the sequence
\[
\left( (\text{updatePost}, \ldots, \text{sendPost}, \ldots) \right) \Rightarrow (\text{snd}, \text{pst})
\]

The notion of shuffling secrets in a communication-aware fashion is essential for defining a compound bound \(B\):
\[
B(s, s') \iff \forall s_1, s_2, s'_1, s'_2.
\]
\[
s_1 \in s_1 \land s_2 \land s'_1 \in s'_1 \land s'_2 \rightarrow B_1(s_1, s'_1) \land B_2(s_2, s'_2)
\]
This is the strongest bound we can hope for the composite. It performs a "shuffling intersection" of \(B_1\) and \(B_2\): \(B(s, s')\) states that no matter how we decompose the secrets as a communicating shuffle of component secrets, both component bounds hold. Thus, \(B\) specifies an intersection of the amounts of uncertainty about the secret enforced by the components, or, in other words, a union of the amounts of information that is being declassified: if each component decclassifies one particular aspect of the overall secret, then the composite decclassifies both.

Finally, the natural composed trigger \(T\) is "\(T_1\) or \(T_2\)"—meaning that if the trigger of either component is fired (during either separate transitions or communication), then the secrets are legitimately accessible to observers. Formally, \(T\) \((trn)\) is
\[
\left( \exists m_1, \sigma_2, \text{send} = \text{match}_1(m_1, \sigma_2) \land T_1(trn) \right) \lor \left( \exists \alpha_1, \text{trn}_2 = \text{send}_2(\alpha_1, \text{trn}_2) \land T_2(trn) \right)
\]

So far, so good: we have defined an attacker model and a security policy for the compound system—i.e., an instance of BD Security, which we denote by \((P_1) \parallel (P_2)\). But can we prove that it indeed holds for \(A_{\text{Aut}} = A_{\text{Aut}_1 \times \text{match}_{\text{Aut}_2}}\), assuming \((P_1)\) and \((P_2)\) hold for the components?

The challenge of proving \((P_1) \parallel (P_2)\) is depicted in Figure 5. Let \(tr\) be a trace of \(Aut\). Suppose it produces the secrets \(s\) and the observations \(ol\), and let \(s'\) be an alternative sequence of secrets such that \(B(s, s')\) holds. We know that \(tr\) is given by the communicating shuffle of some traces \(tr_1\) of \(Aut_1\) and \(tr_2\) of \(Aut_2\), i.e., \(tr = tr_1 \parallel tr_2\). Say \(tr_1\) and \(tr_2\) produce the secrets \(s_1\) and \(s_2\) and the observations \(ol_1\) and \(ol_2\). In order to make a connection to the bounds of the components, and therefore take advantage of \((P_1)\) and \((P_2)\), we invoke Corollary 1, which gives us \(s \equiv s_1 \land s_2 \land \text{match}_1(\alpha_1, \text{trn}_2) \land \text{match}_2(\alpha_1, \text{trn}_2)\). Then, by the definition of \(B\), we have \(B_1(s_1, s'_1) \land B_2(s_2, s'_2)\). By the security properties of the components, we obtain the alternative component traces \(tr'_1\) and \(tr'_2\) such that each \(tr'_i\) yields the secrets \(s'_i\) and is observationally equivalent to \(tr_i\).

So the original problem is reduced to the following problem: can we use \(tr'_1\) and \(tr'_2\) to produce a suitable alternative trace for \(Aut\), namely, a trace \(tr'\) that produces \(s'\) and is observationally equivalent to \(tr\)? The right side of Figure 5 shows the problem in isolation, where for readability we rename \(tr'\) to \(tr\), \(tr'_1\) to \(tr_1\) and \(s'_1\) to \(s_1\):

**Problem.** We are given \(s, \text{ol}, s_1, \text{ol}_1\) and \(tr\) such that \(S_i(tr) = s_i, O_i(tr) = \text{ol}_i, s_i \in s_i \land s_2 \land \text{ol} \in \text{ol}_1 \land \text{ol}_2\), and need to find \(tr\) such that \(S(tr) = s\) and \(O(tr) = \text{ol}\).

In other words, we know how to shuffle the observations and secrets and we are required to shuffle the entire transitions in a compatible way.

To this end, we clearly need the observation and secret matching to provide enough information for transition matching. For example, assume that each \(tr_i\) consists of a single transition, \text{trn}_i, which has a communicating action.

\[\exists \text{uid}, \text{nid}, \text{pid}, \text{uid}'v. \text{ a } = (\text{pid}, \text{ps}, \text{uid}', v) \land \text{ a } = (\text{sendPost}, \text{uid}, \text{nid}, \text{pid})\]
way to shuffle these into a solution \( tr \) is if they actually match, i.e., \( \text{match}(trn_1, trn_2) \) holds. And for proving this, what we have at our disposal is that observations and secrets of these transitions can be shuffled, hence they do match. Formally, what we would need is the following:

**Definition 2.** A communication infrastructure \((\text{matchO}, \text{matchS})\) is called strong if, for all \( trn_1 \) and \( trn_2 \), assuming:

- \( \text{isCom}_1(\text{actOf}_1(trn_1)) \land \text{isCom}_2(\text{actOf}_2(trn_2)) \)
- \( \text{isObs}_1(trn_1) \land \text{isObs}_2(trn_2) \to \text{matchO}(\text{getObs}_1(trn_1), \text{getObs}_2(trn_2)) \)
- \( \text{isSec}_1(trn_1) \land \text{isSec}_2(trn_2) \to \text{matchS}(\text{getSec}_1(trn_1), \text{getSec}_2(trn_2)) \)

then \( \text{match}(trn_1, trn_2) \) holds.

Intuitively, this property requires that the observations and secrets of communication transitions fully capture their matching behavior. Unfortunately, the communication infrastructure from \((P_1)\) and \((P_2)\) in our case study is not strong. Consider a sending and a receiving action for different posts (other than the secret post PID), not involving any of the designated observer users (UIDS\(_1\) or UIDS\(_2\)). These actions are neither observable nor secret. Hence, the preconditions of the above property are trivially satisfied, but still the actions fail to match, because they refer to different posts. In order to remedy this, we must strengthen the attackers. For our case study, a reasonable strategy would be to extend the observation power to communicating actions and their outputs, provided they do not compromise the secret, as follows:

**Amendment 3.** \((P_1)\)'s observations are extended as follows:

- \( \text{isObs}_1(\sigma_1, a_1, o_1, \sigma'_1) \iff \text{userOf}(a_1) \in \text{UIDS}_1 \lor \text{isCom}_1(a_1) \)
- \( \text{getObs}_1(\sigma_1, a_1, o_1, \sigma'_1) = (\text{purgeA}_{\text{PID}}(a_1), \text{purgeO}_{\text{PID}}(o_1)) \)

where \( \text{purgeA}_{\text{PID}} : \text{Act} \to \text{Act} \) and \( \text{purgeO}_{\text{PID}} : \text{Out} \to \text{Out} \) purge away from communicating actions and their outputs the content of PID's post (which constitutes the secret). The observations of \((P_2)\) are extended analogously.

The parameters of actions and outputs that do not pertain to communication or do not manipulate PID's content are not affected by purging—this is the case for sendServerReq and receiveClientReq. By contrast, PID's content \( pst \) is replaced by the non-informative \( \perp \) from everywhere it appears, e.g., \( \text{purgeA}_{\text{PID}}(\text{receivePost}, \text{NID}_1, \text{PID}, \text{pst}, \text{uid}_1, v) = (\text{receivePost}, \text{NID}_1, \text{PID}, \perp, \text{uid}_1, v) \) and \( \text{purgeO}_{\text{PID}}((\text{PID}, \text{pst}, \text{uid}_1, v)) = (\text{PID}, \perp, \text{uid}_1, v) \).

Since it was done in a secrecy-sensitive way, this increase in observation power keeps the properties \((P_1)\) and \((P_2)\) true. At the same time, it makes the communication infrastructure strong, as desired. The reason is that, for each pair of communication transitions \( trn_1 \) and \( trn_2 \):

- If they don't produce secrets, then purging does not affect their observations, so \( \text{match}(trn_1, trn_2) \) is equivalent to \( \text{matchO}(\text{getObs}_1(trn_1), \text{getObs}_2(trn_2)) \).
- If they produce secrets, then \( \text{matchO}(\text{getObs}_1(trn_1), \text{getObs}_2(trn_2)) \) caters for part of the condition required for \( \text{match}(trn_1, trn_2) \); the other part is ensured by \( \text{matchS}(\text{getSec}_1(trn_1), \text{getSec}_2(trn_2)) \). For example, say the action of \( trn_1 \) is \((\text{sendPost}, \text{uid}_1, \text{NID}_2, \text{PID})\) and its output is \((\text{PID}, \text{pst}_1, \text{uid}_1, v_1)\), and the action of \( trn_2 \) is \((\text{receivePost}, \text{NID}_2, \text{PID}, \text{pst}_2, \text{uid}_2, v_2)\). Then matching the observations yields \((\perp, \text{uid}_1, v_1) = (\perp, \text{uid}_2, v_2)\) and matching the secrets yields \( \text{pst}_1 = \text{pst}_2 \); together, they yield \((\text{pst}_1, \text{uid}_1, v_1) = (\text{pst}_2, \text{uid}_2, v_2)\), as required for \( \text{match}(trn_1, trn_2) \).

Note that Amendment 3 has made all communication actions observable (to various degrees)—in line with the common assumption of a network attacker with Dolev-Yao capabilities. We record this as a useful property in our emerging framework, since it has the potential of making the strong communication infrastructure assumption more effective.

**Definition 3.** Two attacker models have observable network traffic if, for all \( trn_1 \) and \( trn_2 \), \( \text{isCom}_1(\text{actOf}_1(trn_1)) \) implies \( \text{isObs}_1(trn_1) \) and \( \text{isCom}_2(\text{actOf}_2(trn_2)) \) implies \( \text{isObs}_2(trn_2) \).

In general, having strong communication infrastructure
and observable network traffic still does not entirely solve our problem. Part of it concerns not communication alone, but also the order of the individual component transitions. To illustrate this, consider the traces \( tr_1 = \langle trn_1, trn'_1 \rangle \) and \( tr_2 = \langle trn_2, trn'_2 \rangle \) where \( trn_1 \) and \( trn_2 \) match and produce local observations \( o_1 \) and \( o_2 \), respectively, and \( trn'_1 \) and \( trn'_2 \) are local transitions that produce local secrets \( s_1 \) and \( s_2 \), respectively. Then \( s_1 = \langle (1, s_1), (2, s_2) \rangle \) and \( o_1 = \langle (o_1, o_2) \rangle \) are valid shuffles of these secrets and observations. However, in \( tr_1 \), the secret comes after the (communicating) observation, while in \( tr_2 \) it comes before. Hence, it is not possible to shuffle \( tr_1 \) and \( tr_2 \) to a trace that produces \( s_1 \) (since the only possibility is \( s'_1 = \langle (2, s_2), (1, s_1) \rangle \)).

This difficulty resides at the heart of our security model: BD Security prescribes a relation, or rather a lack of (co)relation, between observations and secrets produced by a trace, but does not, in general, constrain the time ordering between the production of these observations and secrets. This loose coupling is useful for the local verification of individual systems: not having to worry about the time ordering between observations and secrets allows for flexibility in the proof strategy. However, when composing systems, it leads to the problem that, in general, the bound of the compound system cannot “foresee” which time orderings that arise from freely mixing the component secrets and observations are actually possible, potentially causing compositionality to fail.

Our case study suggests a way out of this conundrum. In its context, the above situation can not appear because the second component never produces secrets by an individual transition, but only as a result of communication:

**Definition 4.** Two attacker models are secret-polarized if isSec\(_2\)(trn\(_2\)) implies isCom\(_2\)(actOf\(_2\)(trn\(_2\))).

Let us reflect on the meaning of the above property. In a system composed of several nodes, such as CoSMEDis, it is natural to think of a source of a secret as a node that produces the secret not by communication with other nodes, but by communication with the outside world. Note that, for simplicity, \((P_1)\) considers the confidentiality of only one arbitrary but fixed post PID of Aut\(_1\). (For an extension to multiple posts in arbitrary nodes across the network, see Appendix C.) Hence, users can “upload” secrets in Aut\(_1\) via updatePost actions, but the only contact of Aut\(_2\) with secrets is via receivePost actions in pair with sendPost actions by Aut\(_1\). In this context, Aut\(_2\) is never the source of the secret, but can only receive it (as well as, possibly, send it back to the issuer or make it available to its users under specified conditions). Thus, we have secret polarization, with Aut\(_1\) being the issuing pole.

**C. Compositionality Theorem**

So far, we have identified some conditions that would help proving compositionality. In fact, it turns out that these conditions are also sufficient—and this is true in general, outside our case study. Indeed, although they have been motivated and discussed in the context of a particular example (post confidentiality in CoSMEDis), all the concepts appearing in the next theorem apply, generally, to any two (not necessarily identical) I/O automata satisfying any two BD-security properties such that the stated conditions hold.

**Theorem 1.** Given

- two I/O automata Aut\(_1\) and Aut\(_2\)
- two BD-security properties \((P_1)\) and \((P_2)\) for these automata

Assume the attacker models for \((P_1)\) and \((P_2)\):

- have a compatible and strong communication infrastructure,
- have observable network traffic, and
- are secret-polarized.

If \((P_1)\) holds for Aut\(_1\) and \((P_2)\) holds for Aut\(_2\), then \((P_1) \parallel (P_2)\) holds for Aut = Aut\(_1\) \(\times\) match Aut\(_2\). This theorem is the culmination of our theoretical development. (What will follow is a technique for customizing the result of applying this theorem, and a generalization to the \(n\)-component case.) So let us pause and reflect on the scope of this result, which shows how to compose the security guarantees of two communicating systems. The result is generic in that it applies to systems specified as I/O automata—arguably, covering any reactive system one can imagine. Moreover, the information flow security guarantees of these systems can be specified in a flexible and fine-grained manner: Any aspect of a system transition can be specified as a secret or an observation (forming the attacker model), and then any mathematical formulas can be used to describe bounds and triggers for the allowed secret classification (forming the security policy).

In order to compose these guarantees, the theorem makes three requirements about the component attacker models:

1. **Compatible Communication Infrastructure**, asking that not only the component transitions, but also their secrets and observations are composable. This is essential for being able to formulate, let alone prove, the composition of the security guarantees—so in a way it is a prerequisite to the very question about compositionality. As we illustrate with our examples in this paper, producing a compatible communication infrastructure seems to come naturally from inspecting the interaction between communication on the one hand and secrets and observations on the other.

2. **Strong Communication Infrastructure and Observable Network Traffic**, asking that communication between the components be substantially exposed to the attacker. This requirement is clearly not something that comes natural for individual components, but is an artifact for achieving compositionality. In fact, it may be argued that it is counter-intuitive to “allow” the attacker such power. However, the requirement needs to be regarded from the opposite angle: it is not about weakening the system by offering power to the attacker, but showing that, even if the attacker could observe most of the communication, he would still not learn more about the secrets. And indeed, in our case study we achieved communication strength by letting the attacker observe everything in a communication except for sensitive information.

3. **Secret Polarization**, asking that only one of the components can issue secrets. For multi-user systems, this means that, once we agree on what the secrets are, only users of one component can upload secrets. Note that this does not prevent us from considering another notion of secret, where the other component is the issuer. For example, in our case...
study the secrets are the post contents for a post ID, which can belong to either component—the requirement only prevents us from considering two sources at the same time, e.g., in order to speak of the concatenation of secrets from two sources. The requirement seems to be fulfilled by a wide category of secrets, namely, those produced and stored locally, on a single node (and possibly communicated to other nodes). However, it is easy to imagine situations when it breaks—if we allowed users of different nodes to upload versions of the same (shared) post, or to jointly edit documents. Consequently, secret localization is the major limitation of our result. In Appendix C, we discuss a workaround to combine multiple sources of secrets after composition, provided these sources are independent of each other in a certain sense (which applies to the secret sources in CoSMedDis).

The main strength of our result is its policy agnosticism: While the theorem requirements restrict the component attacker models, they say nothing about the security policies, i.e., their bounds and triggers. Hence, the theorem composes any given security policies, no questions asked. This “quantitative” flavor makes our theorem applicable in a variety of contexts, to seamlessly combine arbitrarily complex policies—as we illustrate in Section VI. These include classification during the process of inter-component communication. Indeed, as our examples abundantly illustrate, communication transitions can influence both the attacker models and the security policies.

D. Relaxing Security Properties

But policy agnosticism has an inconvenience: its general-purpose property composition may not be, in concrete cases, the most natural desired property for the compound system. For our running example, such a natural property would be:

(P') A coalition consisting of two groups of users, UID1 of Aut1 and UID2 of Aut2, can learn nothing about the updates to the content the Aut2’s post PID beyond the existence of an update unless one of the following holds:

1) one of UID1 is the admin or PID’s owner, or becomes friends with the owner, or PID is marked as public
2) PID is being shared between Aut1 and Aut2 and [one of UID2 is the admin or becomes a remote friend of PID’s owner, or PID is marked as public]

This reads almost like (P1 || P2). In particular, the trigger T is clearly that formalized by (P1 || P2): the disjunction of the component triggers. However, (P')’s bound, phrased as “beyond the existence of an update,” is not verbatim captured by (P1 || P2)’s bound B. Indeed, (P') suggests that the desired bound is the same as that of the first component, in particular, refers to the domain of secrets Sec1, whereas B operates on Sec1 + Sec2 + Sec1 x Sec2. More precisely, a faithful formalization of (P') would have the same components as (P1 || P2), except for:

- the secret domain and the secret filter, which become Sec1 and getSec1, and
- the bound, which becomes B1.

Fortunately, we can easily derive (P') from (P1 || P2) using a general-purpose theorem for transfer between two security policies, which we describe next.

Let Aut be an I/O automaton and (P) and (P') two security properties operating on it. (P) is said to have a stronger security model than (P') if there are two partial functions f : Sec → Sec' and g : Obs → Obs' from the secrets and observations of (P) to those of (P'), such that the following are true:

- ifSec'(trn) holds iff isSec(trn) and getSec(trn) ∈ dom(f), and in this case f(getSec(trn)) = getSec'(trn)
- isObs'(trn) holds iff isObs(trn) and getObs(trn) ∈ dom(g), and in this case g(getObs(trn)) = getObs'(trn)
- T(trn) implies T'(trn)
- B'(sl', tl') and map_f(sl) = sl' imply that there is a tl such that map_f(tl) = tl' and B(sl, tl)

Above, map_f : Sec → Sec' denotes the secret-wise extension of the partial function f : Sec → Sec to sequences, omitting any secrets s /∈ dom(f).

In case f is defined and injective on all secrets occurring in Aut, then it can be inverted and the last condition above simplifies to checking that B'(sl', tl') implies B(map_f−1(sl'), map_f−1(tl')).

In general, choosing partial functions f and g allows us to weaken the power of the observer and the notion of secrets by making the sets of observable and secret transitions smaller. In particular, observable transitions whose observations are not translated by g become unobservable in the new security model.

Theorem 2. Assume (P) has a stronger security model than (P'). If (P) holds for Aut, then so does (P').

To prove that (P1 || P2) implies (P') for our example system, we use the theorem where Obs' = Obs, g is the identity, Sec = Sec1, and f extracts the secret s1 out of (s1, s2) or (s1, s2). Note that Aut2 does not produce local secrets, hence (2, s2) never occurs. Moreover, f is injective on valid secrets, because the sets of secrets occurring locally in Aut1 and in communication with Aut2 are disjoint, and, in a communication, the secret received by Aut2 is uniquely determined by the secret sent by Aut1. The inverse of f is

\[ f^{-1}(s1) = \begin{cases} \{(1, (upd, pst))\} & \text{if } s1 = (upd, pst) \\ \{(snd, pst)\} & \text{if } s1 = (snd, pst) \end{cases} \]

The above assumptions are then immediate to check.

We rely on the same theorem to prove that the various amendments of the component properties required to achieve compositionality are indeed strengthenings of the previous versions—which is important for guaranteeing that our quest for compositionality did not weaken any bit of the component properties we started with. Thus, to prove that the version of property (P1) after amendments 1–3 implies the original, we define g as the identity on local observations and f as f(upd, pst)) = pst.

E. The N-ary Case

We now engage in the final stage of our compositionality quest: proving a security property for the entire distributed system, not just for two components. For our case study, we want to prove the following for every post PID belonging to a component Aut, in a network of n components Aut1, ..., Autn:
(P') A coalition of n groups of users, UID_{Sk} for each Aut_{tk}, can learn nothing about the updates to PID's content beyond the existence of an update unless one of the following holds:

1) one of UID_{Sk} is the admin, or is PID's owner, or becomes friends with the owner, or PID is marked as public
2) the post is being shared by Aut_{tk} with some Aut_{tk} for k ≠ i and [one of UID_{Sk} is the admin or becomes a remote friend of PID's owner, or PID is marked as public]

To this end, we generalize the communication product automaton construction from 2 to n mutually communicating components Aut_{tk}. We fix, for each k, k' with k ≠ k', a matching predicate match_{k,k'} : Trans_{k} × Trans_{k'} → Bool (between the transitions of Aut_{tk} and Aut_{tk'}). We write match for the family (match_{k,k'}) : Aut_{tk} → Bool for the corresponding notion of communication action (belonging to Aut_{tk} and pertaining to communication with Aut_{tk'}). We assume that the communication infrastructure is pairwise dedicated, in that the predicates isCom_{k,k'} and isCom_{k,k''} are disjoint for k' ≠ k''.

The product is denoted by \prod_{k\in\{1,...,n\}}^{\text{match}} Aut_{tk}. Its states are families (σ_k)_{k\in\{1,...,n\}} \in \text{States}_{tk}. The transition relation is shown in Figure 6, where, to avoid ambiguity, we use labeling to indicate the components not only for separate actions or outputs, but also for communicating actions or outputs, e.g. (i, a_i), (j, a_j)). We write (σ_k)[i := σ'_i] for the family of states that is the same as (σ_k), except for the index i where it is updated from σ_i to σ'_i; and similarly for (σ_k)[i := σ'_i, j := σ'_j].

Given BD Security properties (P_k) for each Aut_{tk}, we define their composition || k\in\{1,...,n\} (P_k) by a straightforward generalization of the binary case. For example, the observation domain is

\[ \text{Obs} = \sum_{k\in\{1,...,n\}} \text{Obs}_{Sk} + \sum_{k,k'\in\{1,...,n\}} \text{Obs}_{k} \times \text{Obs}_{k'} \]

so that it contains either separate observations (k, o_k) or joint observations (k, o_k, o_{k'}). Assuming that each pair of attacker models have compatible communication infrastructure consisting of match_{S_{k,k'}} and matchO_{k,k'}, n-ary shuffle operators are defined:

- for traces, || k\in\{1,...,n\} Trace_k → Pow(Trace)
- for secrets, || k\in\{1,...,n\} Sec_k → Pow(Sec*)
- for observations, || k\in\{1,...,n\} Obs_k → Pow(Obs*)

Similarly to the binary case, the composite bound B(sl, sl') is defined from the component bounds:

\[ \forall (s_lk)_k, (s'_l)_{k'} \in \|| (s_lk)_k \land s_l' \in \|| (s'_l)_{k'} \rightarrow \forall k. B_k(sl_k, s'_l) \]

and the composite trigger is defined as the disjunction of the component triggers.

With these definitions, a generalization of Theorem 1 can be formulated. Most of the assumptions will be those of Theorem 1 applied to all pairs Aut_{tk} and Aut_{tk'}. An exception is secret polarization, where we need something stronger. We call the n component attacker models uniquely secret-polarized if there is a unique secret issuer, say, Aut_{tk}, in the whole network. Formally: for all k ≠ i and Aut_{tk}-transitions trn, isSec_{tk}(trn) implies isCom_{tk}(actO_{tk}(trn)).

**Theorem 3.** Given

- I/O automata Aut_1, ..., Aut_n
- BD-security properties (P_1), ..., (P_n) for these automata

Assume that the following properties hold:

- the communication is pairwise dedicated
- any two of the attacker models have compatible and strong communication infrastructure and observable network traffic
- the attacker models are uniquely secret-polarized

If each (P_k) holds for Aut_{tk}, then || k\in\{1,...,n\} (P_k) holds for Aut = \prod_{k\in\{1,...,n\}}^{\text{match}} Aut_{tk}.

Thus, the generalization to the n-ary case proceeds fairly smoothly, with the nuance that a single source of secrets is allowed in the whole network.

Back to CoSMed, to capture our concrete (P'), match_{k,k'} is defined just like in the binary case (shown in Figure 1), but using the identifiers NID_k and NID_{k'} for Aut_{tk} and Aut_{tk'} instead of NID_1 and NID_2. Now (P') follows from this theorem along with the transfer theorem—the latter being used to customize the bound, similarly to the binary case.

**F. Heuristic for Achieving Compositionality**

In addition to the compositionality theorems, another end product of our analysis is a heuristic for making the components fit these theorems, which can be summarized as follows:

**Context.** We have a monolithic system modeled as an I/O automaton Aut, delivered with a confidentiality guarantee (P) modeled as BD Security. We want to extend the system to a distributed system, consisting of several nodes able to communicate to each other. Each node is an extension Aut' of Aut, with new actions for inter-node communication. The question is what confidentiality property we can prove for the distributed system.

**Step 1.** By its nature, (P) states something about a notion of secret and the way it is protected during interaction with the outside world, which takes place through actions and outputs.
of the original automaton $\text{Aut}$. We analyze what happens with the secret during inter-node communication in $\text{Aut}'$, identifying the roles of secret issuer and secret receiver for the nodes. This leads to a split of $(P)$ in two variants, $(P_1)$ and $(P_2)$.

Step 2. If necessary, we modify the secrecy infrastructure of $(P_1)$ and $(P_2)$ so that the communication of secret-correlated items is acknowledged by both as a secret-producing action. This may require a modification of the bound as well, to account for the correlation.

Step 3. We strengthen the observation power for $(P_1)$ and $(P_2)$ by allowing the observer to access any communication information that does not compromise the secrets.

(The purpose of Step 3 is to achieve an attacker with strong communication infrastructure and observable network traffic. But what about communication transitions whose observation would compromise the secret, even though they are not marked as secrets? The answer is that such transitions must have already been incorporated as actual secrets in Step 2.)

Step 4. If we can identify a unique source for the secrets, we apply Theorem 3 for a distributed system consisting of one component satisfying $(P_1)$ and $n - 1$ components satisfying $(P_2)$. Then we apply Theorem 2 to customize the compound bound into a property that uses the notion of secret from the secret-issuer component only.

In the next section, we employ this heuristic for lifting to CoSMeDis all of CoSMeD’s confidentiality properties.

VI. VERIFYING CoSMeDis’S CONFIDENTIALITY

Our running example of a confidentiality property had the form: Nothing is inferable about a given secret (a post content) unless a trigger is being fired. The properties Bauereiß et al. had proved for the original CoSMed actually made stronger claims: Nothing is inferable about a given secret beyond the trace portions during which a trigger is active, i.e., when the observers’ access to the secret is legitimate. This makes it possible to consider dynamic triggers, which can be repeatedly fired and canceled. For example, a user can become a friend of the post’s owner, but later the friendship can be canceled by either user “unfriending” the other—only the post updates performed outside the times of friendship should be protected from that user.

Technically, in the stronger properties the trigger is “swallowed” by the bound. So the price for the gained strength in confidentiality is a more complex bound, which operates on an enriched domain of secrets that include trigger information. For post confidentiality, the domain of secrets now consist not only of post contents, $(\text{psec}, \text{pst})$, but also of openness indicators, $(\text{osec}, b)$, where $b$ is a Boolean flag indicating whether the legitimate access window is open. The secret-producing function returns $\text{osec}$ $b$ only if the openness status changes, with $b$ indicating the new status—e.g., $b$ becomes True when an observer is marked as friend, and becomes False if the last observer is unfriended (and the other legitimate access conditions fail as well, e.g., the admin is not an observer). These indicators are used to formulate the bound in an access-window sensitive way. For example, if a trace produces the secrets:

$$\langle (\text{osec, True}), (\text{psec, pst}_1), (\text{osec, False}), (\text{psec, pst}_2) \rangle$$

then the bound protects $\text{pst}_2$, but not $\text{pst}_1$. This is because the update $\text{pst}_1$ occurred in a phase of the system execution where the observers had legitimate access to the post content, resulting in a modification of the secrecy infrastructure. The update $\text{pst}_2$ occurred outside of that time frame, thus not being protected by the bound.

---

**Figure 7:** Confidentiality properties for the original CoSMeD

<table>
<thead>
<tr>
<th>Observers</th>
<th>Secrets</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group of users, UIDs</td>
<td>Content of a given post, say, PID</td>
<td>Updates performed while or last before one of the following holds: Some user in UIDs is the admin, is the post owner or is friend with the post owner PID is marked as public</td>
</tr>
<tr>
<td>Friendship status between two given users, say, UID$_1$ and UID$_2$</td>
<td>Status changes performed while or last before the following holds: Some user in UIDs is the admin or is friend with UID$_1$ or UID$_2$</td>
<td></td>
</tr>
<tr>
<td>Friendship requests between two given users, say, UID$_1$ and UID$_2$</td>
<td>Existence of accepted requests while or last before the following holds: Some user in UIDs is the admin or is friend with UID$_1$ or UID$_2$</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8:** Confidentiality properties for CoSMeDis, lifted from CoSMeD

<table>
<thead>
<tr>
<th>Observers</th>
<th>Secrets</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$ group of users, UIDs$_1$, ..., UIDs$_n$, one for each node</td>
<td>Content of a given post, say, PID of node NID$_i$</td>
<td>Updates performed while or last before one of the following holds: Some user in UIDs is NID$_i$’s admin, is PID’s owner or is friend with PID’s owner PID is marked as public Some user in UIDs$_j$ for $j \neq i$ is remote friend with PID’s owner</td>
</tr>
<tr>
<td>Friendship status between two given users, say, UID$_1$ and UID$_2$ of NID$_i$</td>
<td>Status changes performed while or last before the following holds: Some user in UIDs is NID$_i$’s admin or is friend with UID$_1$ or UID$_2$</td>
<td></td>
</tr>
<tr>
<td>Friendship requests between two given users, say, UID$_1$ and UID$_2$ of NID$_i$</td>
<td>Existence of accepted requests while or last before the following holds: Some user in UIDs is NID$_i$’s admin or is friend with UID$_1$ or UID$_2$</td>
<td></td>
</tr>
</tbody>
</table>
whereas the access window was closed before $p_{st_2}$ occurred. (Section 3.3 in [13] gives more details.)

Fortunately, our policy-agnostic theorems can smoothly accommodate such complex bounds as well. In fact, the whole heuristic from Section V-F works in this case essentially the same as for the simpler, static-trigger property. In particular, the context and steps 1, 3 and 4 are precisely the same. For step 2, we work with a notion of secret that has already been enriched with openness indicators. Our own amendment with the context and steps 1, 3 and 4 are precisely the same. For step 2, we work with a notion of secret that has already been enriched with openness indicators. Our own amendment with step 2, we work with a notion of secret that has already been enriched with openness indicators. Our own amendment with step 2, we work with a notion of secret that has already been enriched with openness indicators.

We have used the heuristic to extend to CoSMeDis all the properties proved for the original CoSMed, summarized in Figure 7—where the triggers are always vacuously false (since they are “swallowed” by the bounds) and the observers are always a given set of users. In Figure 8, we summarize the end product after lifting these to CoSMeDis via our theorems.

Besides the already discussed post confidentiality, there are confidentiality properties regarding friendship status, i.e., the information on whether two users are friends, and friendship requests, i.e., the information on whether a user has issued a friendship request to another user. In both cases, the legitimate access windows are defined to mean that an observer is the admin or is currently friends with either of the involved users.

The application of our compositionality theorem to the confidentiality of friendship status and requests is easier than for post confidentiality. Unlike the latter, the former does not involve sharing secrets between nodes. Consequently, as seen in the corresponding entries in Figures 7 and 8, the bound of the distributed system is the same as that of a single node.

So far, we only discussed secret sources inherited from CoSMed, of which we knew they were protected in CoSMed and needed to prove that they are not compromised in CoSMeDis. However, there is one source that is specific to CoSMeDis, namely remote friendship, whose confidentiality needs to be proved from scratch. Here, the secret of interest is the set of remote friends of a user $UID_i$, if they exist. It turns out that this secret is protected similarly to local friendships: only $UID_i$, the (remote) friends of $UID_i$, and their friends can learn anything about this information. To prove this, we first formulate the receiver and sender properties, $(P_1)$ and $(P_2)$:

$(P_1)$ A group of users $UID_{i,j}$ of $Aut_i$ can learn nothing about the remote friends of $UID_i$ unless one of $UID_{i,j}$ is $UID_i$ or is friends with $UID_i$.

$(P_2)$ A group of users $UID_{i,j}$ of $Aut_j$ can learn nothing about the remote friends of $UID_j$ unless one of $UID_{i,j}$ is a remote friend of $UID_j$ or is friends with one of them.

We apply Theorem 3 to one component satisfying $(P_1)$ and $n-1$ components satisfying $(P_2)$. This, together with the usual application of Theorem 2, yields the following property for an $n$-component CoSMeDis:

A coalition of $n$ groups of users, $UID_{i,k}$ for each $Aut_{i,k}$, can learn nothing about the remote friends of the user $UID_i$ of $Aut_i$ unless one of the following holds:

1) one of $UID_{i,k}$ is $UID_i$ or is friends with $UID_i$, or
2) one of $UID_{i,k}$ is a remote friend of $UID_i$ or is friends with one of them.

VII. CONCLUSION

We integrated programming with formal verification in a security-by-design approach, producing: (1) a reusable formal framework for composing security guarantees and, employing this framework, (2) machine-checked confidentiality guarantees for a distributed social media platform.

We aimed to cover exhaustively the confidentiality aspects of CoSMeDis’s application layer logic, by classifying the information sources of interest and proving upper bounds for the flows. However, our security guarantees should be understood in the proper context. By themselves, they do not achieve end-to-end security, since they ignore several aspects:

- Identity theft: Our security models idealistically assume that all users interact with the system using their own credentials. So our guarantees do not protect against weak or stolen passwords.
- Network-level attacks: We implicitly count on communication being perfectly encrypted (even though, for the purpose of compositionality, we occasionally empower the attackers beyond the Dolev-Yao model).
- Outer code layers: The I/O automata forming the kernel of CoSMeDis are programmed and verified in the proof assistant, but the outer layers, including the API-specific code and the UI, are currently being trusted.

Our approach can (and should!) be combined with approaches that handle the above aspects, which are largely orthogonal. For example, a step towards end-to-end security would be the combination of our declassification policies with the noninterference policies required by the client side, for which powerful monitoring tools are available [14,19,30].

Another area of future work is the integration of our verification framework with fully automated frameworks that target simpler properties, e.g., safety and liveness for database-driven web applications [24]—with the goal of employing the expressive power of a proof assistant on a need basis, only for cases where automation fails.

REFERENCES

“Two deployed CoSMedia nodes,” https://cap1.globalnoticeboard.com and https://cap2.globalnoticeboard.com, 2016, Anonymous access for both is possible with username “demo” and password “demo”, and administrator access (for managing user accounts and inter-node communication) is possible with username “admin” and password “sifoeZ6b”.


R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin, “Personas: an online social network with user-defined privacy,” in SIGCOMM. ACM, 2009, pp. 135–146.


APPENDIX

A. More Details on Post Confidentiality

The post confidentiality property of the original CoSMed has a “dynamic” bound, incorporating changes in the confidentiality of the post due to visibility updates and friending or unfriending of observers. We summarize the formal attacker
models (AM) and security properties (SP) in Table I. For a detailed explanation, see [13]. The corresponding property of the issuer node of CoSMed extends that of the original CoSMed with the sending of posts: the additions are highlighted. On the receiver side, nothing is leaked to local observers beyond the number of updates (because communication traffic is observable). Only when an observer is added as a remote friend of the post owner, or the post is marked as public, the trigger fires and the post is declassified. The property for the entire distributed system has the observations built from all the component observations (as described in the main paper’s n-ary compositionality theorem). However, the secrets, as well as the bound, are those of the secret issuer component—this is because we show the end product property, obtained after applying both the compositionality and the transfer theorems.

B. More Details on Friendship Confidentiality

For the confidentiality of (local) friendship information, we have proved that the security properties of CoSMed, as discussed in [13], still hold in CoSMedDis as they are. This is to be expected/desired, since local friendship does not involve communication between nodes. However, remote friendship does. Hence, we prove an additional property for CoSMedDis, summarized in Table II. We consider the remote friends of an arbitrary, but fixed user UID, who is not an observer. The secrets are the contents of remote friendship actions performed by UID: the ID of the remote node, the remote user, and Boolean a flag whether the remote user is added or deleted as a friend. Since we assume communication traffic to be observable, we can’t keep secret that a remote friendship action occurred, but we keep secret who was added or deleted as a friend. Consequently, the bound states that the remote friends of the given user UID can be replaced arbitrarily with other users (that are not observers themselves). Moreover, the bound includes the static knowledge that friendship addition and deletion can only occur alternatingly (first addition, then deletion, and so on). The trigger states that the friend list of UID is declassified to friends of UID.

C. Combining Independent Secret Sources

For simplicity, in this paper we have always considered the confidentiality of one secret source at a time, e.g. one given (arbitrary but fixed) post, or the friendship information between two given users. A legitimate question is therefore how to deal with multiple sources simultaneously.

Consider, for example, the confidentiality of two different posts, PID_i in Aut_i and PID_j in Aut_j. We can instantiate the results of the paper for each post separately, and obtain two security properties of the distributed system. It turns out that we can easily combine these two properties. This relies on two key assumptions:

1) The secrets are independent of each other. Indeed, updates to different posts are completely orthogonal in the system; there is no interference between different posts.
2) We assume that the scheduling of the different secrets is not confidential; i.e., the contents of PID_i and PID_j are considered confidential, for example, but the relative timing of uploads is not.

The first assumption guarantees the soundness of our approach to first consider the secrets in isolation, not having to worry about possible inter-dependencies. The second assumption is important, because it allows us to ignore the scheduling of secrets—after composition. Before composing the system, this would not be possible, because scheduling information is still needed for the composition of traces.

We formalize these assumptions as follows. Let (P_i) and (P_j) be two security properties of the same system, where (P_i) comprises the observation producing function O_i, the secret producing function S_i, the trigger T_i, and the bound B_i, and analogously for (P_j) (we omit the indices getSec, etc. for brevity).

We capture the first assumption by assuming that the secret of one security property is observable in the other security property. Formally, the observation function of one property is assumed to fully determine the secret and the trigger information of the other, i.e. O_i(tr) = O_j(tr') implies

- S_j(tr) = S_j(tr') and
- the trigger T_j holds in tr iff it holds in tr'.

and the same has to hold symmetrically for O_j, S_i, and T_i. This means that the variation of the secret S_i as required by (P_i) is possible without interfering with the secret information of (P_j); the latter stays fixed.

The second assumption is formalized by a combined secret producing function S that does not have the familiar shape of producing an interleaving of secrets, but it produces a pair of secret sequences:

S(t) = (S_i(t), S_j(t))

This combination captures the content of both secrets, but not their scheduling. Consequently, the combined bound is

B((sl_i, sl_j), (sl'_i, sl'_j)) = B_i(sl_i, sl'_i) ∧ B_j(sl_j, sl'_j)

The combined observation function O is assumed to correspond to an intersection of the observations of (P_i) and (P_j), i.e. either O_i(t) = O_j(t') or O_j(t) = O_i(t') implies O(t) = O(t).

The proof of the combined security property follows easily from the assumptions: Given a trace tr with S(tr) = (sl_i, sl_j) and an alternative secret pair (sl'_i, sl'_j) within B_i we first invoke (P_i) to obtain tr' with S_i(tr') = sl'_i, keeping sl_i and T_i constant, and then invoke (P_j) to obtain tr'' with S_j(tr'') = sl'_j, keeping sl_j constant such that S(tr'') = (sl'_i, sl'_j). The combined observation O(tr) remains unchanged in every step.

This proof technique is applicable to arbitrary security properties, as long as the above assumptions are satisfied (and it is straightforward to lift it from pairs to tuples of multiple security properties). We have instantiated it in our Isabelle formalization for the above example in CoSMedDis: two posts PID_i and PID_j in arbitrary network nodes Aut_i and Aut_j. In order to satisfy the assumptions, we first had to strengthen the observation power of the security property discussed in the paper. In addition to the actions of observing users, we declare all actions that potentially contribute to other secret posts to be observable. This includes updating actions of other posts, but also trigger-relevant actions such as the creation of friends of observers. The proof in Isabelle was automatic:
Table I: Post confidentiality

For CoSMed

<table>
<thead>
<tr>
<th>AM</th>
<th>Sec = psec Post + osec Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isSec(σ, a, a, o, o') if (1) (o = outOK ∧ ∃uid, pst, a = (updatePost, uid, PID, pst)) ∨</td>
</tr>
<tr>
<td></td>
<td>(2) open(o') ≠ open(o) where open(o) if PID ∈ postIDs(a) ∧ admin(σ) ∈ UIDs ∨ owner(σ, PID) ∈ UIDs ∨</td>
</tr>
<tr>
<td></td>
<td>UIDs ∩ friendIDs(σ, owner(σ, PID)) ≠ ∅ ∧ vis(σ, PID) = public</td>
</tr>
<tr>
<td></td>
<td>getSec(σ, a, a, o, o') = (psec, pst) in case (1) and (osec, open o') in case (2)</td>
</tr>
<tr>
<td></td>
<td>Obs = Act × Out isObs(σ, a, a, o, o') iff userOf(σ) ∈ UIDs getObs(σ, a, a, o, o') = (a, o)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th>B is defined as follows, mutually inductively with another predicate BO:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>post ≠ [] → post' ≠ [] B(sl, sf)</td>
</tr>
<tr>
<td></td>
<td>BO(map_psec(post), map_psec(post')) BO(map_psec(post), map_psec(post')) BO(sl, sf) post ≠ [] ® post' ≠ [] post ≠ [] ® last pst = last pst</td>
</tr>
<tr>
<td></td>
<td>B(map_psec(post), map_psec(post')) BO(sl, sf) (osec, False) · sl, map_psec(post) (osec, False) · sf</td>
</tr>
</tbody>
</table>

T is vacuously False

For CoSMedDis

<table>
<thead>
<tr>
<th>AM_i</th>
<th>Sec_i = upd(psec Post + osec Bool) + sendPost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isSec_i(σ, a, a, o, o') if (1) (o = outOK ∧ ∃uid, pst, a = (updatePost, uid, PID, pst)) ∨</td>
</tr>
<tr>
<td></td>
<td>(2) open(o') ≠ open(o) ∨</td>
</tr>
<tr>
<td></td>
<td>(3) (uid, pst, uid', v, a = (pst, uid', i) ∧ a = (sendPost, uid, nid, PID))</td>
</tr>
<tr>
<td></td>
<td>getSec_i(σ, a, a, o, o') = (psec, pst) in cases (1) and (3) and (osec, open o') in case (2)</td>
</tr>
<tr>
<td></td>
<td>Obs_i = Act × Out isObs_i(σ, a, a, o, o') iff userOf(σ) ∈ UIDs ∨ ∃k isCom_i(a)</td>
</tr>
<tr>
<td></td>
<td>getObs_i(σ, a, a, o, o') = (purgeA_i[(a, purgeO_i)])</td>
</tr>
</tbody>
</table>

| SP_i        | B_i(sl, sf) iff (filter_i(sf), filter_i(sf')) ∧ cor_i(sf') T_i is vacuously False |

For CoSMedDis

<table>
<thead>
<tr>
<th>AM_j</th>
<th>Sec_j = Post</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isSec_j(σ, a, a, o, o') iff (1) (o = outOK ∧ ∃uid, pst, a = (receivePost, NID, PID, pst, uid, v) ∨</td>
</tr>
<tr>
<td></td>
<td>(2) open(o') ≠ open(o) ∨</td>
</tr>
<tr>
<td></td>
<td>getSec_j(σ, a, a, o, o') = pst</td>
</tr>
<tr>
<td></td>
<td>Obs_j = Act × Out isObs_j(σ, a, a, o, o') iff userOf(σ) ∈ UIDs ∨ ∃k isCom_j(a)</td>
</tr>
<tr>
<td></td>
<td>getObs_j(σ, a, a, o, o') = (purgeA_j[(a, purgeO_j)])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP_j</th>
<th>B_j(sl, sf) iff length sf = length sf' T_j(sl, sf) iff (remoteVis_i(sl, sf), remoteOwner(sl, sf')) ∨</th>
</tr>
</thead>
</table>

For CoSMedDis

<table>
<thead>
<tr>
<th>AM</th>
<th>Sec = Sec_i + Sec_j + getSec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Obs = Obs_i + ∑k∈S Obs_j + ∑k′∈S′ Obs_j, Obs _ Obs_j</td>
</tr>
<tr>
<td></td>
<td>isObs(trn) = isObs_i(trn) ∧ isObs_j(trn) if trn = (k, trn_k, k', trn_k')</td>
</tr>
<tr>
<td></td>
<td>getObs(trn) = (k, getObs_i(trn_k), k', getObs_j(trn_k)) if trn = (k, trn_k, k', trn_k')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP</th>
<th>B = B_i T(trn) = T_i(trn) if trn = (k, trn_k, k', trn_k')</th>
</tr>
</thead>
</table>

Table II: Friendship confidentiality

For CoSMed

<table>
<thead>
<tr>
<th>AM_i</th>
<th>Sec_i = NodeID × UserID × Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isSec_i(σ, a, o, o') = (o = outOK ∧ ∃uid, nid, st, a = (sendUpdateRFriend, UID, nid, uid, st) ∧ uid ∉ UIDs_mwi)</td>
</tr>
<tr>
<td></td>
<td>getSec_i(σ, a, sendUpdateRFriend, UID, nid, uid, st, o, o') = (nid, uid, st)</td>
</tr>
<tr>
<td></td>
<td>Obs_i = Act × Out isObs_i(σ, a, o, o') iff userOf(σ) ∈ UIDs ∨ ∃k isCom_i(a)</td>
</tr>
<tr>
<td></td>
<td>getObs_i(σ, a, o, o') = (purgeA_i[(a, purgeO_i)])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SP_i</th>
<th>B_i(sl, sf) iff BC(sl, sf) ∧ alter(sl') where:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• alter(sl') states that friendship creation and deletion occurs alternately in sf for each remote user, and</td>
</tr>
<tr>
<td></td>
<td>• BC is defined inductively by (1) BC(</td>
</tr>
</tbody>
</table>

| T_i(σ, a, o, o') = (uid ∈ UIDs_mwi ∧ friendIDs(σ, UID)) |

For CoSMedDis

| AM_j        | same as for the issuer, only replacing sendUpdateRFriend actions by receiveUpdateRFriend actions coming from the issuer |

| SP_j        | B_j(sl, sf) T_j is vacuously False |

For CoSMedDis

| AM          | analogously to the attacker model for the network case of post confidentiality in Table I |

| SP          | B = B_i T = T_i |


after extending the observation function, there were hardly any changes necessary to the existing proof scripts. The original proof strategy still worked. In order to instantiate the above combination technique, it was necessary to add the (generic) infrastructure for the technique itself and a few helper lemmas for the concrete system, but the proofs were straightforward.