Applying the Isabelle Insider Framework to Airplane Security

Florian Kammüller

Middlesex University London and Technische Universität Berlin

Manfred Kerber

University of Birmingham, UK

Abstract

Avionics is one of the fields in which verification methods have been pioneered and brought about a new level of reliability to systems used in safety-critical environments. Tragedies, like the 2015 insider attack on a German airplane, in which all 150 people on board died, show that safety and security crucially depend not only on the well-functioning of systems but also on the way humans interact with the systems. Policies are a way to describe how humans should behave in their interactions with technical systems. Formal reasoning about such policies requires integrating the human factor into the verification process. In this paper, we report on our work on using logical modelling and analysis of infrastructure models and policies with actors to scrutinize security policies in the presence of insiders. An insider is a user of a system who behaves like an attacker abusing privileges thereby bypassing security controls. We model insider attacks on airplanes in the Isabelle Insider framework. This application motivates the use of an extension of the framework with Kripke structures and the temporal logic CTL to enable reasoning on dynamic system states. Furthermore, we illustrate that Isabelle modelling and invariant reasoning reveal subtle security assumptions. This results in a methodology for the development of policies that satisfy stated properties. To contrast our approach to model checking, we provide an additional comparative analysis.

Keywords: Airplane safety and security, Insider threats, Interactive theorem proving, Security policies, Verification

1. Introduction

Airplanes offer a very safe way of travelling. Accidents and terror attacks are extremely rare. We believe that one reason for this is that scrutiny and rigorous verification including formal methods are routinely applied in most technical developments as well as for organizational measures of airplanes. After the 2001-09-11 attacks stringent measures were taken and have been to the day of writing successful. The most recent major incident was an

Email addresses: f.kammueller@mdx.ac.uk (Florian Kammüller), M.Kerber@cs.bham.uk (Manfred Kerber)

insider attack in which the copilot of Germanwings Flight 9525 on 2015-03-24 hijacked the aircraft by locking out the captain, who had left the cockpit, and subsequently brought the aircraft to a crash in which all 150 persons on board died. As a consequence, airlines introduced a two-person rule that a pilot must never be on their own in the cockpit. The two-person rule has been rescinded in 2017 only two years after it was introduced. The 2015-03-24 incident shows that insider attacks are an important issue. Despite their common use in other parts of avionics, formal methods are not commonly applied to airplane policies including human actors which is necessary to analyze insider threats. Therefore, we investigate whether recent advancements in formal modeling and analysis of insider threats may produce advancements. The Isabelle Insider framework [1] is an instantiation of the interactive theorem prover Isabelle using its expressive Higher Order Logic (HOL) to provide a theory for formalizing infrastructures with actors and policies including insiders to prove security properties fully formally with computer support. Experimenting with this Insider framework on real case studies, motivated earlier work [2] of applying the existing Isabelle Insider framework to verify airplane policies in the presence of insider attacks. This earlier work has revealed some major challenges for the Isabelle Insider framework that we want to address in the current work:

- Since the policies are dealing with actors and their possibilities of moving within the infrastructure, for example an airplane, a fixed association of actors with locations, roles, and credentials in the model must be extended to enable representing dynamic change.

- We need to integrate dedicated logics into the framework enabling the expression of security and safety guarantees over the dynamically changing infrastructure state. We need to express global validity of logical properties of policies over all reachable states; for example, we want to express "for all states reachable from an acceptable initial state, a suicidal copilot cannot crash the plane."

In the current paper, we provide solutions to these challenges and demonstrate them on the airplane case study by the following two contributions.

- State transitions as well as rules for expressing changes to the state of infrastructures including locations, actors, their roles, credentials and behaviours are provided by Kripke structures. This allows modelling state change and state transition.

- Temporal logic CTL is provided within the framework to formalize and prove logical properties. This enables (a) detecting attack paths through the graph of infrastructure state evolution and (b) from there identifying additional security assumptions that when met guarantee that the attack is not possible any more on any path.

In the process of realizing these extensions to the Isabelle Insider framework and testing them on the airplane application other contributions emerged:

- We identify an improved methodology for policy invalidation and model refinement. It consists of attempting to prove global security properties in the Isabelle Insider

framework showing potential attacks and moreover revealing missing assumptions that may then be added as additional locale assumptions in the model refinement step.

- Moreover, in order to show the relation to other approaches to verification, most notably model checking, we present a comparative model and verification [3] in the NuSMV model checker [4]. We formalize various different implementations of policies.
- To show the surplus gained by using the Isabelle Insider framework rather than NuSMV we then proceed by generalizing our main result to arbitrary policies which can be done only in a powerful system such as Isabelle.

After discussing related work in Section 2, we present in Section 3 a retrospective of the development of safety and security regulations for airplanes. We then present the existing Isabelle Insider framework in Section 4. Next, we use this framework to model an airplane scenario including an insider attacker. We first present our methodology for applying the Isabelle Insider framework as an opening to Section 5, which also provides an overview of the following technical sections. We integrate Kripke structures into the model and express and interactively prove central security properties using the branching time temporal logic CTL (Section 5). Section 6 presents the analysis of those properties on the airplane scenario showing how the framework can be used to scrutinize the security policies and thereby reveal existing loopholes within their formal specifications. Section 7 introduces an alternative verification of the airplane scenario using model checking with NuSMV. It also shows how the main result can be generalized in Isabelle to arbitrary policies to illustrate what can be done here that cannot be done using model checking. Section 8 concludes.

The full Isabelle sources [5] as well as the NuSMV code [3] are available online. In order to give an impression of the kind of formalization the most important definitions and theorems can be found in Appendix A.

2. Related Work

In this section, we present some related work from the field of insider threats and work in which reasoning approaches similar to the one applied in our work are applied. Furthermore we discuss work related to the verification in avionics.

Malicious insiders are defined by Glasser and Lindauer as follows: "[...] insiders are current or former employees or trusted partners of an organization who abuse their authorized access to an organization's networks, systems, and/or data." [6]. Insider threats refers to "the malicious acts carried out by these trusted insiders" [6]. Matt Bishop et al. [7] add "[t]he insider problem is one of the most difficult problems in computer security, and indeed in all aspects of real-world security. The ancient Romans even had an aphorism about it: 'Who will guard the guards themselves?'"

More recently, there has been a distinction between malicious and unintentional insiders [8]. For the scope of our work, we focus on malicious insiders and define insider following the spirit of Glasser and Lindauer [6]:

Definition 2.1 (Insider). *An insider is a trusted user of a system who behaves like an attacker abusing privileges thereby bypassing security controls.*

The insider threat patterns provided by CERT [9] use the System Dynamics model, which can express dependencies between variables. The System Dynamics approach is also successfully being applied in other approaches to insider threats, for example, in the modelling of unintentional insider threats [10]. Axelrad et al. [11] have used Bayesian networks for modelling insider threats in particular the human disposition. In comparison, the model we rely on for modelling the human disposition in the Isabelle Insider framework is a simplified classification following the taxonomy provided in [12]. In contrast to all these approaches, our work provides an additional model of infrastructures and policies allowing reasoning at the individual and organizational level.

A major field of application of formal methods is avionics. Companies (such as Airbus and Boeing) and organizations (such as NASA) use formal methods to prove formal properties of aircrafts and spacecrafts. There is a large body of work, including work based on model checking and theorem proving, which we cannot give justice in this paper. We will mention only a few. Moy et al. [13] explore mainly the relationship between software testing and formal verification. They argue that in many application areas formal verification outperforms testing, firstly in that the proofs show the correctness on all inputs and not just the ones tested, but secondly also in the person power required. O'Halloran [14] shows how a Z-based toolset is used to prove the correctness of embedded real time safety critical software for Eurofighter Typhoon. Khan et al. [15] argue that complexity of avionics has increased to a level that verification and validation of the systems need computer based approaches. They use model abstraction to simulate hardware and software interactions.

In the domain of rigorous analysis of airplane systems, work often follows for practical and economic reasons a philosophy of using a mix of formal and systematic informal methods. An example from airplane maintenance procedures by Oheimb et al. [16] uses a security evaluation methodology following the Common Criteria and a formal model and verification with the model checker AVISPA. In comparison, we use a more expressive logical model in the Isabelle Insider framework than the AVISPA specification. To our knowledge, the focus of work on formal methods in avionics is directed towards the correct functioning of the hardware and the software. However, it is very important to consider the human factor.[1]

[1]Quote by Chesley B. Sullenberger [http://www.sullysullenberger.com/my-testimony-today-before-the-house-subcommittee-on-aviation/]:
Pilots must be able to handle an unexpected emergency and still keep their passengers and crew safe, but we should first design aircraft for them to fly that do not have inadvertent traps set for them.

We must also consider the human factors of these accidents.

From my 52 years of flying experience, and my many decades of safety work – I know that nothing happens in a vacuum, and we must find out how design issues, training, policies, procedures, safety culture, pilot experience and other factors affected the pilots' ability to handle these sudden emergencies, especially in this global aviation industry.

Dr. Nancy Leveson, of the Massachusetts Institute of Technology, has a quote that succinctly encapsulates much of what I have learned over many years: 'Human error is a symptom of a system that needs to be redesigned.'

Human error in avionics has been addressed in works using formal methods including model checking and theorem proving. For example, mode confusion as an important origin for airplane accidents has been addressed by Luettgen and Carreno [17]. In this work, the model checkers murϕ, SPIN, and SMV are used to model Flight Guidance Systems (FGS)[2] and analyze their *mode logic*. While such a machine component reflects some part of human behaviour (the interface input) it is very different from our insider model that integrates human actors, their psychological disposition and behaviour into the state.

Another example of application of formal methods in avionics by Munoz et al. [18, 19] analyzes the operational concept of NASA's Small Aircraft Transportation System, Higher Volume Operation (SATS HVO), a system allowing up to four small aircrafts to enter and leave airports simultaneously. The authors model the airport's surrounding area (self-controlled area, SCA) and an airport management module (AMM). The SATS HVO "is [modelled as] a set of rules and procedures using finite state machines" by Munoz et al. [19]. The authors provide an abstract specification of the system written in the theorem prover PVS with "fixes" (latitude/longitude points in space) representing the positions of planes in the SCA and "landing sequences" (lists of integers) to model the queuing management of the AMM. The rules of the AMM are encoded as rules of a state transition system. The verification is by state exploration in PVS. The PVS model of the real system is explored by an "algorithm written and proved correct in PVS" by Munoz et al. [18] that shows that only a finite number (2811) of a potentially infinite number of states is reachable [18]. For each of these states the safety properties are then manually verified in PVS. There is some resemblance between this approach and ours in that they also use an interactive theorem prover (PVS) to emulate state exploration. In contrast to their approach, we model the foundations of CTL model checking based on lattice and fixpoint theory in a conservative extension of Isabelle thus fundamentally guaranteeing correctness of our approach of state exploration. They first prove a result about reduction of exploration to reachability and then check all remaining states individually thus implicitly mimicking proper model checking. Besides, they do not model the human actor nor do they address security nor insider attacks. We assume that our work is the first to consider insider threats within airplane safety and security in a formal way.

Logical modelling and analysis of insider threats has started off by investigating insider threats with invalidation of security policies in connection with model checking by one of us in [20, 21]. This early approach also uses infrastructure models of organizations, actors and policies but was more restricted than the Isabelle Insider framework discussed in Section 4. The use of sociological explanation has been pioneered in [22] by one of us and others already with first formal experiments in Isabelle. Finally, one of us has established the Isabelle Insider framework in [1]. It has been validated on two of the main three insider patterns the Entitled Independent and Ambitious Leader. Relevant in the context of this application are other applications of the Isabelle Insider framework. In particular, the Isabelle Insider framework has been applied to IoT Insiders [23, 24]. In this work, the framework was

[2]An FGS is a component of the flight control system that monitors the difference between the actual state of the aircraft and its desired state as inputted by the crew.

extended by attack trees. Attack trees provide the possibility to refine attacks once they have been identified. This refinement is formalized together with the notion of attack trees as first introduced for insider models in general in [25]. In other work, we applied the insider framework to auction protocols [26]. In the CHIST-ERA project SUCCESS [27] we use the framework in combination with attack trees and the Behaviour Interaction Priority (BIP) component architecture model to develop security and privacy enhanced IoT solutions.

In [28] Kamali et al. present reasoning that integrates deduction based reasoning and model checking for the formal verification of vehicle platooning. The idea is that vehicles move in platoons and can join and leave them under certain safety conditions. In order to model the hybrid aspects of the real-time system a hybrid system is used that makes use of discrete decision making (such as, initiating joining a platoon) and continuous control (of actually driving the vehicle). The formal discrete reasoning is translated to a timed automaton which can then be used to produce actual running code (in a simulator). The right level of abstraction is important in order to deal with complexity issues.

3. Development of Airplane Safety and Security

On 2001-09-11, four terrorist attacks took place in the USA, two on the two towers of the World Trade Center, one on the Pentagon, and in a fourth attack the airplane crashed when passengers tried to overcome the hijackers.[3] Before these attacks, aircraft hijacking typically meant that the hijackers had some negotiable demands. Because of the risk to life for the people on board the aircraft, the standard approach was to enter negotiations and to avoid a resolution by force while the aircraft was in the air.

In particular, also there was no secured door between the passenger compartment and the cockpit in airplanes; actually the door was occasionally open, even allowing passengers to get a glimpse of the cockpit during the flight. In Western countries there were no airplane hijackings with major loss of life between the 1970s and the 2001-09-11 attacks. This may have created in the USA and other countries a false sense of security. In the wake of the attacks a serious rethink of the security provision has happened. In particular, the cockpit doors were reinforced and made bullet-proof, making it nearly impossible to open by intruders [32].

These (and other) changes seem to have had the wanted effect, since in between the time of the introduction of secured cockpit doors and the time of writing this article there were only 18 airplane hijackings or attempted airplane hijackings[4] (as listed on [31]), all but two of them could be prevented from causing fatalities. In one the hijacker was shot and killed and the other one was an insider attack. One nearly successful airplane hijacking has been caused by the copilot who forced Ethiopian Airlines Flight 702 to land at Zurich airport in

[3]For a description of the events, see [29], including more than 300 further pointers. A detailed account of the events of 9/11 and recommendations can be found in a 585 page report by the 9/11 commission [30]. A list of aircraft hijackings can be found as [31].

[4]Note however that there were other attacks on flights which did not originate from passengers, such as the Malaysia Airline Flight MH17 which was brought down by a missile over Ukraine on 2014-07-17.

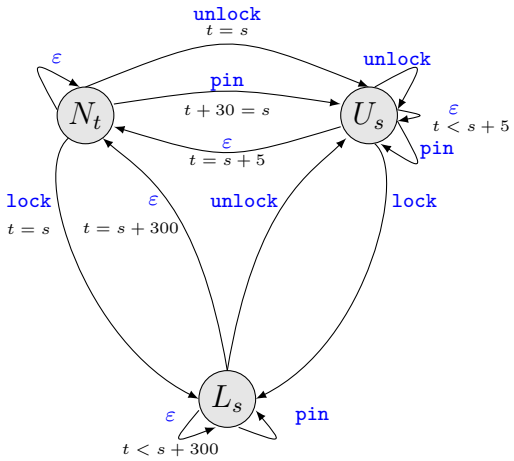

Figure 1: A finite timed automaton to describe the lock mechanism of the door. In the three states, N_t, U_s, and L_s (for normal, unlocked, locked at times t or s, respectively), the pilots can `lock` the door, or `unlock` the door at any time with immediate effect, or do nothing with respect to the door – indicated by ε. Cabin crew can enter the `pin` of the door; entering an incorrect PIN corresponds to the empty action ε. Entering the correct PIN has an effect only in the state N_t after 30 seconds in a time window of five seconds unless the pilots take the `lock` action. After no action for 300 seconds the L_s state is transformed to the N_t state.

an attempt to blackmail asylum for himself in Switzerland. Also this airplane hijacking can be characterized as an insider attack since the attacker was part of the crew.

The one major exception to the rule was Germanwings Flight 9525 on 2015-03-24, which was on the way from Barcelona to Düsseldorf. The aircraft was hijacked by the copilot who locked out the captain who had left the cabin. The pilot tried to regain access to the cockpit but did not succeed. Subsequently, the copilot brought the aircraft to a crash in which all 150 people on board died.

Let us now look more closely into the door and its release mechanism.[5] The door is operated by a switch from inside the cockpit (with three positions: "unlock", "norm", "lock") and a keypad outside the cockpit. In order to gain access to the cockpit normally a crew member would use the inter-phone to contact a pilot in the cockpit to request access, then presses the hash key on the keypad, which triggers a buzzer in the cockpit, and the pilot releases the door using the switch to open the door (by keeping it in the "unlock" position). In case the pilot(s) is/are incapacitated a crew member outside the cockpit can enter an emergency code to open the door. After 30 seconds (during which the buzzer sounds in the cockpit) of no reaction by the pilots the crew member can open the door for five seconds.

Since this access method could be used by a hijacker to force a crew member to open the door from outside the cockpit, the pilots can, within the 30 seconds between entering the emergency code and the release of the door, lock the cockpit door by putting the toggle button into the "lock" mode. In that case the keypad is disabled for five minutes and the door can be opened during this time only from inside the cockpit by putting the button in the position "unlock".

[5]The information is extracted from a 5:32 film by Airbus [33].

The mechanism can be described on different levels and each level requires certain assumptions (for instance, that the door itself will withstand any physical force that may be exerted by an attacker). According to Occam's razor, we try to give a representation that is as easy as possible and still describes the situation in sufficient detail that the important aspects are modelled. A first approximation can be given by the timed finite state machine in Figure 1 with three states "N", "U", and "L" for "normal", "unlocked", and "locked", respectively. While time plays a role and it makes a difference for humans whether the door is locked for 300ms, 300s, or 300 minutes, we will abstract from this in the following formalization. During the fatal flight, the copilot used this locking mechanism to lock out the captain from the cockpit. While the mechanism has been successful so far from preventing any fatal attempt by an outsider to hijack an aircraft, the same mechanism prevented the captain from re-entering the cockpit and take action to rescue the aircraft in this case.

Note that there are more details that we could model. However, we try to be in the description above (according to Occam's razor) minimal. That is, in this representation, if the door is in the locked state an agent at the door *cannot* go from the cabin to the cockpit; however, if the door is in the unlocked state they *can indeed* (whether they do is up to the agent). In this model it can then be reasoned whether an (outside) agent is in principle able to take control of the airplane or not. Details such as 'the door needs to be opened before agents can walk through', 'they need a hand free to operate the handle', or 'they need to be able to walk' are not modelled, since they are not essential for the reasoning about policies. In other contexts such details may be crucial.

4. Isabelle Insider Framework

Before we formalize the airplane scenario in section 5, we give first a brief introduction to Isabelle in this section; describe the Isabelle Insider framework with infrastructures, policies, actors, and insiders; and describe how Kripke Structures and CTL are modelled.

4.1. Isabelle and Modular Reasoning

Isabelle/HOL is an interactive proof assistant based on Higher Order Logic (HOL). Application specific logics are formalized into new theories extending HOL. They are called object-logics. Although HOL is undecidable and therefore proving needs human interaction, the reasoning capabilities are very sophisticated supporting "simple", i.e., repetitive, tedious proof tasks to a level of complete automation. The use of HOL has the advantage that it enables expressing even the most complex application scenarios, conditions, and logical requirements and HOL simultaneously enables the analysis of the meta-theory. That is, repeating patterns specific to an application can be abstracted and proved once and for all. As an example, we will see how general preservation theorems of the state transition relation over the system graph and over policies can be proved as part of the insider framework and applied in concrete applications like the airplane scenario (see Section 4.5).

An object-logic contains new types, constants, and definitions. These items reside in a theory file. For instance, the file `Insider.thy` contains the object-logic for insider threats described in the following paragraphs. This Isabelle Insider framework is a *conservative*

extension of HOL. This means that our object logic does not introduce new axioms and hence guarantees consistency. Conceptually, new types are defined as subsets of existing types and properties are proved using a one-to-one relationship to the new type from properties of the existing type. This process of conservative extension has been greatly facilitated by the datatype package that offers a restricted sort of simple recursive type definitions. Inductive definitions are a similar tool to define new predicates by a set of rules. Both extension features offer the specification of model elements with a theory of induction and exhaustion properties necessary for the proof of theorems over the model.

Besides datatypes and inductive definitions, we make also use of local assumptions within *locales*. This is the reasoning process we propose as part of our methodology: the insider condition in Section 5.4 is not an axiom but is locally assumed to analyze the infrastructure's policies.

This process has been conceived as *Modular Reasoning* in Isabelle [34] and implemented in the locales mechanism. Locales have been motivated by case studies from abstract algebra where proofs about algebraic structures – like groups, rings, or fields – frequently use assumptions – like $\forall x.x \circ 1 = x$ – that are valid within these algebraic structures but not outside. Rather than repeating those local assumptions continuously in large numbers of property statements and proofs, locales realize contexts in which those assumption can be used. Insider threat modelling and analysis using logics shows the same needs, since assumptions about actors are specific to a certain application's infrastructure. Moreover, the definition and the assumption of a locale are accessible later on, whenever the locale is invoked. But since they are local assumptions and definitions they do not endanger HOL's principle of conservative extension.

We are going to use Isabelle syntax and concepts in this paper and will explain them when they are used.

4.2. Structure of the Framework

Figure 2 gives an overview of the Isabelle Insider framework with its layers of object-logics – each level below embeds the one above. The blue levels on the top represent the parts that are the novel contributions specific to this paper. The lower levels of the framework, that is, Kripke structures, CTL and Attack Trees are completely generic meaning that they need no changes for different applications. They use sophisticated concepts of Isabelle, like axiomatic type classes, polymorphism and locales to guarantee this genericity. The Insider theory is a bit of a mix: in its main parts it is generic (formalization of actors, local policies, actors' behaviour, and insiderness which will be discussed in detail in the next section). However, the generic state transition relation that is provided by the fully generic Kripke structure must be defined in an inductive definition that defines the overall system behaviour for the various actions and policies. Also, despite the genericity of the notions of actors, policies, and insiderness, even the main actions and the graph structure used to assemble the state may differ slightly from one application to the next. Because of these slight variations, it is not possible to have one generic Insider theory as the topmost part of the framework. However, we still consider the Insider theory as part of a general framework since it is quite evident from one application to the next how to reuse it.

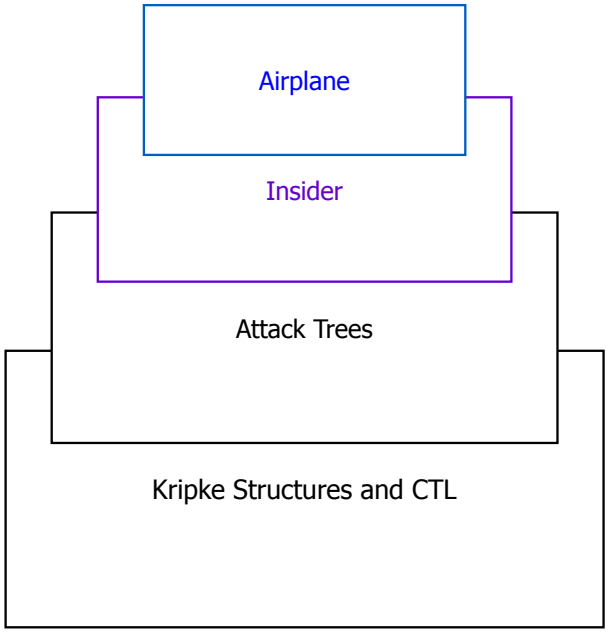

Figure 2: Generic Isabelle Insider framework applied to Airplane case study.

Various different publications have documented the development of the Isabelle Insider framework as already described in Section 2 but we pick it up again to clarify the relation to the framework. The publication [1] by the first author and Probst introduces the major notions of insiderness, policies, and behaviour showing how these notions suffice to serve the known insider threat patterns identified by CERT [9]. Only when applying this early version of the Isabelle Insider framework to the current Airplane case study [2] by the authors, the necessity to add a mutable state has been identified. Model checking, that is, Kripke structures and CTL have been emulated by the first author in [35] and used to provide a semantics for attack trees [24] that is not used in the current application. Attack trees together with a formal notion of refinement constitute also another branch of the development of the Isabelle Insider framework, the Isabelle Infrastructure framework, aiming to support a dedicated security refinement process [36].

4.3. Representing human factors and insiders

Before we delve into the formal definition of our Isabelle model of infrastructures, policies, actors, and insiders, we need to clarify the intuitions of these modeling elements to avoid confusion, misunderstanding and raising wrong expectations.

In line with Definition 2.1, and thus the common notion in the research community, we

use the intuition that an insider is a trusted user that behaves like an attacker. To represent the attacker, we use an actor `Eve`, outside the system who has malicious ("evil") intentions. Insiderness can be variably explored by explicitly identifying to what extent the "evil" actor `Eve` can bypass security controls by impersonating privileged users. Typically a malicious insider is such a user that acts like an actor who has a split personality, that is, the actor acts with the privileges of a trusted user but with the intentions of an outside attacker.

Technically, we model this explicit yet flexible impersonation of privileged users inside the system by a function `Actor` that maps identities to roles. In places where an impersonation is deemed feasible the function may map the identity of the "evil" actor `Eve` to the same role as that of a privileged user inside the system. For all other identities that are not compromised the function actor maps these identities exclusively to roles in the system, that is, for these identities `Actor` is injective: $id_0 \neq id_1 \Rightarrow$ `Actor` $id_0 \neq$ `Actor` id_1 . Note, that in this model it is not necessary that `Eve` is a member of the system, for example, the crew of the airplane. For the analysis, we can choose for which user role(s) we want to investigate the system policy to guarantee safety against insiderness by specifying the actor(s) that is (are) being impersonated. Safety against insider threats is then expressed by defining a global policy that specifies that `Eve` cannot achieve the security goal whose security we want to investigate.

In our Isabelle model, we represent actors and their dispositions using a taxonomy from the insider threat literature based on psychological studies (see, e.g. Nurse et al. [12]). We do not model the human disposition (or mood) within the Isabelle framework. Nevertheless, we represent the values of their description, for example "disgruntled", which may be used to make policies dependent on this. For example, a legislation could appear that excludes pilots with certain medical conditions from controlling airplanes. However, to avoid raising wrong expectations in the readers: we do not model the actual estimation of these values for human actors but rather treat them as some kind of parameters to our model. It should be noted that Isabelle's expressiveness does not exclude such modeling for future work. We just focus on the representation of these human aspects assuming that their values be provided externally.

4.4. Infrastructures, Policies, Actors, and Insiders

In the Isabelle/HOL theory for insiders, one expresses policies over actions `get`, `move`, `eval`, and `put`. An actor may be enabled to

- `get` data or physical items, like keys,
- `move` to a location,
- `eval` a program,
- `put` data at locations or physical items – like airplanes – "to the ground".

The precise semantics of these actions is refined in the state transition rules for the concrete infrastructure. The framework abstracts from concrete data – actions have no parameters:

```
datatype action = get | move | eval | put
```

The human component is the *Actor* which is represented by an abstract type `actor` and a function `Actor` that creates elements of that type from identities (of type `string`):

```
typedecl actor
type_synonym identity = string
consts Actor :: string ⇒ actor
```

Note that it would seem more natural and simpler to just define `actor` as a datatype over identities with a constructor `Actor` instead of a simple constant together with a type declaration like, for example, in the Isabelle inductive package by Paulson [37]. This would, however, make the constructor `Actor` an injective function by the underlying foundation of datatypes therefore excluding the fine grained modelling that is at the core of the insider definition: in fact, the core insider property `UasI` (see below) defines the function `Actor` to be injective for all except insiders and explicitly enables insiders to have different roles by identifying `Actor` images.

Atomic policies of type `apolicy` describe prerequisites for actions to be granted to actors given by pairs of predicates (conditions) and sets of (enabled) actions:

```
type_synonym apolicy = ((actor ⇒ bool) × action set)
```

For example, the `apolicy` pair[6] `(λx. has (x, ''PIN''), {move})` specifies that all actors who know the `PIN` are enabled to perform action `move`. To represent the macro level view seeing the actor within an infrastructure, we define a graph datatype `igraph` (see below) for infrastructures. This datatype has generic input parameters that are going to be supplied as concrete parts of an application infrastructure on instantiation of an `igraph`. These parameters are specified by the list of types behind the constructor `Lgraph` as: a set of location pairs – the actual "map" of an application infrastructure and a list of actor identities associated with each node (location) in that graph. Locations can be used to represent physical or logical positions in an application, for example, cockpit and door in an airplane or parts of a software, respectively. Moreover, an `igraph` contains a function associating actors with a pair of string lists: the first list describes the credentials an actor has while the second list defines the roles that an actor can take. Finally, an `igraph` has a component assigning locations to a string list describing the state of the component. Slightly adapting the original insider framework, we needed to integrate the credentials, roles, and location state into the infrastructure graph to enable the dynamic view of state transition and Kripke structures (see Section 4.5). For each of the components there exist corresponding projection functions and predicates `has` and `role` to express that actors have credentials or that they can perform in specified roles, respectively, and `isin` to express that locations are in a specified state (see Appendix A).

[6]Note that λ is the usual lambda-operator of higher order logic that describes functions. For instance, the square function can be defined – without giving it a name – as $\lambda x.x * x$.

```
datatype igraph = Lgraph (location × location)set
                         location ⇒ identity list
                         actor ⇒ (string list × string list)
                         location ⇒ string list
```

Infrastructures combine an infrastructure graph of type `igraph` with a policy function that assigns local policies over a graph to each location of the graph, that is, it is a function mapping an `igraph` to a function from `location` to `apolicy set`. The Isabelle type `[igraph, location] ⇒ apolicy set` abbreviates `igraph ⇒ (location ⇒ apolicy set)` hence the stepwise application to `igraph` to return a function is possible.

```
datatype infrastructure = Infrastructure igraph
                                         [igraph, location] ⇒ apolicy set
```

Elements of the datatype infrastructure can thus be constructed using the constructor `Infrastructure`, which is a higher order function, because it takes as (second) input a policy valued function. This higher order parameter represents local policies, that is, maps from graph locations to policies for that location. In the following section, we will see how this higher order function enables proof of general preservation properties.

Policies specify the expected behaviour of actors of an infrastructure. We define the behaviour of actors using a predicate `enables`: within infrastructure `I`, at location `l`, an actor `h` is enabled to perform an action `a` if there is a pair `(p,e)` in the local policy of `l` – `delta I l` projects to the local policy – such that action `a` is in the action set `e` and the policy predicate `p` holds for actor `h`.

```
enables I l h a ≡ ∃ (p,e) ∈ delta I l. a ∈ e ∧ p h
```

For example, the statement `enables I l (Actor''Bob'') move` is true if the atomic policy `(λx. True, {move})` is in the set of atomic policies `delta I l` at location `l` in infrastructure `I`. Double quotes as in `''Bob''` create a string in Isabelle/HOL.

The human actor's level is modelled in the Isabelle Insider framework by assigning the individual actor's psychological disposition[7] `actor_state` to each actor's identity.

```
datatype actor_state = State psy_state motivations
```

The values used for the definition of the types `motivations` and `psy_state` (see Appendix A) are based on a taxonomy from psychological insider research by Nurse et al. [12]. The transition to become an insider is represented by a *Catalyst* that tips the insider over the edge so he acts as an insider formalized as a "tipping point" predicate. To embed the fact that the attacker is an insider, the actor can then impersonate other actors. In the Isabelle Insider framework, the predicate `Insider` must be used as a *locale* assumption to enable

[7]Note that the determination of the psychological state of an actor is of course not done using the formal system. It is up to a psychologist to determine this. However, if for instance, an actor is classified as `disgruntled` then this may have an influence on what they are allowed to do according to a company policy and this can be formally described and reasoned about in Isabelle.

impersonation for the insider: this assumption entails that an insider `Actor ''Eve''` can act like their alter ego, say `Actor ''Charlie''` within the context of the locale. This is realized by the predicate `UasI`:

```
UasI a b ≡ (Actor a = Actor b) ∧
          ∀ x y. x ≠ a ∧ y ≠ a ∧ Actor x = Actor y ⟶ x = y
```

Note that this predicate also stipulates that the function `Actor` is injective for any other than the identities `a` and `b`. This completion of the Actor function to an "almost everywhere injective function" is needed in some proofs (for an example see Section 6.4). We generalize here from other approaches on formal security analysis used in particular in security protocol verification known as the Dolev-Yao attacker model [38, 39]. The Dolev-Yao model allows an attacker to eavesdrop, forge, replay, delay and rush, reorder, and delete any message and also the attacker may impersonate any given role of a protocol participant. Our approach is more flexible because it addresses not just one specific attacker with a set range of abilities (for example eavesdrop or forge in the Dolev-Yao model). We define a general insider not only for security protocols but any system. An insider can impersonate other actors and can attain any ability or access rights that exist in the system. This flexibility also allows talking about arbitrary properties of actors, policies and systems, for example, the fact that always two actors need to be in a location, as seen in this paper – something that goes beyond the specifics of the usual assumptions of Dolev-Yao for security protocols.

4.5. Kripke Structures and CTL

The expressiveness of Higher Order Logic allows formalizing the notion of Kripke structures as sets of states and a transition relation over those in Isabelle. Moreover, temporal logic can be directly encoded using Isabelle's fixpoint definitions for each of the CTL operators by the first author [35]. Combining the two, we can then apply them as generic tools to analyze dynamically changing infrastructures with insiders: we consider snapshots of infrastructures as states, use the actors and their action based behaviour definition to define a state transition, to then use temporal logic to express safety and security properties over dynamically changing infrastructures. This application will be demonstrated on our case study in Section 5.2. We briefly introduce here the necessary facts of Kripke structures and CTL showing how they are instantiated for insiders.

The transition relation on system states is defined as an inductive predicate called `state_transition_in`. It introduces the syntactic infix notation `I` $\rightarrow_n$ `I'` to denote that system state `I` and `I'` are in this relation.

inductive `state_transition_in :: [state, state] ⇒ bool  ("_  →n _")`

The specification of the behaviour of actors in the Insider framework allows defining the rules for the state transition relation of the Kripke structure for infrastructures for each of the actions. Here is the rule for put. The expression `h` $@_G$ `l` says that `h` is at location `l` in the graph `G`. The next state construction `I'` uses the projections `gra`, `agra`, `cgra`, `lgra` to select the graph itself, the actors-location association, the credentials and roles, and the

location state map, respectively. The rule expresses that an actor – who is at location `l` and is "put"-enabled in the infrastructure `I` by its policy at location `l` – can "put" the location into a state `z`[8] in the successor state `I'` of the state transition for infrastructures. The double brackets enclose the preconditions of the meta-implication $\Longrightarrow$ in Isabelle. A proposition $[\![A; B]\!] \Longrightarrow C$ simply abbreviates $A \Longrightarrow (B \Longrightarrow C)$.

```
put: ⟦ G = graphI I; a @_G l;
       enables I l (Actor a) put;
       I' = Infrastructure
              (Lgraph (gra G)(agra G)(cgra G)((lgra G)(l := [z])))
              (delta I)
     ⟧ ⟹ I →_n I'
```

We illustrate this particular rule here because we use it in the case study to express that an actor can put the airplane to the ground (see Section 5.3).

We can already develop some very useful theorems for the state transition relation and Kripke structures. For example, the following lemma motivates why we define infrastructures as higher order functions where the local policies map the graph to a function over its locations: precisely because of that generality of the infrastructure constructor we can prove that state transitions do not change the policy `delta` – as one would expect.

```
lemma init_state_policy: I →_n^* I' ⟹  delta I = delta I'
```

The relation $\rightarrow_n$`^*` is the reflexive transitive closure – an operator supplied by the Isabelle theory library – applied to the relation $\rightarrow_n$.

The proof of this invariant illustrates why for policy verification as we show here a deductive framework like Isabelle is well suited. To deduce the above theorem, we first prove that single step state transitions preserve the policy.

```
∀ I I'. I →_n I' ⟶  delta I = delta I'
```

Then we use this lemma within an application of the induction for reflexive transitive closure of relations that is provided in the Isabelle theory library to infer the above lemma `init_state_policy`. Note that it is the specification in HOL of the state transition relation that provides the case analysis rule and the induction scheme as sound rules automatically generated from the definition.

Branching time temporal logic CTL over Kripke structures has been integrated as part of the Isabelle Insider framework by the first author [35]. A generic type `state` including a transition $\rightarrow_i$ is defined there using the concept of type classes in Isabelle. This type class `state` is then instantiated to the type of `infrastructures` thereby instantiating the state transition relation to $\rightarrow_n$ defined in the insider theory presented above (see Appendix A).

[8]The variable `z` has no constraints in the rule and can thus be instantiated to any value when applying the Isabelle Infrastructure framework and thus instantiating this inductive definition of the state transition.

Thereby, the theory constructed and proved for this state transition $\rightarrow_i$ over a generic type `state` are transferred automatically to infrastructures and their transition relation $\rightarrow_n$.

Summarizing, the CTL-operators `EX` and `AX` express that property f holds in some or all next states, respectively.

$$\texttt{EX}\ f \equiv \{\ \texttt{s.}\ \exists\ \texttt{f0} \in f\texttt{.}\ \texttt{s} \rightarrow_i \texttt{f0}\ \}$$
$$\texttt{AX}\ f \equiv \{\ \texttt{s.}\ \{\texttt{f0.}\ \texttt{s} \rightarrow_i \texttt{f0}\ \} \subseteq f\ \}$$

The CTL formula `AG` f means that on all paths branching from a state s the formula f is always true (`G` stands for 'globally'). It can be defined using the Tarski fixpoint theory by applying the greatest fixpoint operator.

$$\texttt{AG}\ f \equiv \texttt{gfp}(\lambda\ \texttt{Z.}\ f \cap \texttt{AX Z})$$

In a similar way, the other CTL operators are defined. The formal Isabelle definition of what it means that formula f holds in a Kripke structure `M` for insiders can be stated as: the initial states of the Kripke structure `init M` need to be contained in the set of all states `states M` that imply f (see Clarke et al. [40]).

$$\texttt{M} \vdash f \equiv\ \ \texttt{init M} \subseteq \{\ \texttt{s} \in \texttt{states M.}\ \texttt{s} \in f\ \ \}$$

In an application, the set of states of the Kripke structure will be defined as the set of states reachable by the infrastructure state transition from some initial state, say `example_scenario`.

$$\texttt{example\_states} \equiv \{\ \texttt{I.}\ \texttt{example\_scenario} \rightarrow_i\texttt{\^{}*}\ \ \texttt{I}\ \}$$

The `Kripke` constructor combines the constituents state set, set of initial states, and state transition relation $\rightarrow_i$ by application of the dataype constructor `Kripke`.[9]

$$\texttt{example\_Kripke} \equiv \texttt{Kripke example\_states \{example\_scenario\}} \rightarrow_i$$

Given some `example_policy` – a predicate over an infrastructure using actors, actions, and their behaviours – we can then for example try to prove that this property holds generally by attempting the following proof in Isabelle.

$$\texttt{example\_Kripke} \vdash \texttt{AG example\_policy}$$

If the proof fails, the failed attempt will reveal conditions describing a state in the Kripke structure as well as actions leading to this state that identify an attack possibility. In the example in Section 5.3, this will be illustrated. Additionally, the failed attempts to prove the global validity also lead to identifying invariants of the system helping to establish decisive side conditions as well as identifying loopholes. The loopholes lead to a deeper insight into problems with the policy. By defining new locale assumptions and re-proving global properties, the newly found assumptions can be refined until the proof succeeds. This procedure will be illustrated on the airplane case study as well in Section 6.4.

[9] To this end, the state `example_scenario` is inserted into a singleton set using the Isabelle set constructor { _ }.

Note that all the definitions in the locale `airplane` that we use in Section 5 have been implemented as *locale* definitions using the locale keywords **fixes** and **defines** [41]. Thus they are accessible whenever the locale `airplane` is invoked. But since definitions are essentially abbreviations, they adhere to the principle of conservative extension of HOL not endangering consistency.

5. Formalizing the Airplane Scenario

In this section, we present a methodology for the modeling and analysis of insider threats illustrating it in detail on the airplane case study. Before we embark on the case study we first present the methodology which serves as a section overview simultaneously.

5.1. Our Methodology for Insider Threat Analysis

We propose an informal methodology by summarizing the steps for the development of secure policies in the presence of insiders using the Isabelle Insider framework. In each step we provide forward references to the relevant sections where the case study illustrates it.

1. Build a model of the infrastructure, its actors, and local policies with roles and credentials and define the security property of interest as global policy (Sections 5.2 and 5.3).
2. Identify initial state(s) and define Kripke structure (Section 6.1).
3. Use the tipping point and insider assumptions to specify the potential insider(s) (Section 5.4).
4. Invalidate the global policy, that is, negate the property to specify an infrastructure state in which the insider can violate it (Section 5.4, Property `ex_inv3`).
5. Explore the state transition function to find a path from the initial state(s) to this state in which the global policy is violated. For the invalidation and exploration, CTL can be used: first attempt to prove `AG {x. global_policy x ''Eve''}`; failure produces potentially a candidate for an attack; next prove `EF ¬{x. global_policy x ''Eve''}` to establish the attack path (Section 6.1, Property `aid_attack`).
6. Repeat the previous two steps to improve the policy (Section 6.2), until the proof of `AG {x. global_policy x ''Eve''}` succeeds (Section 6.4).
7. If after repeated cycles in the previous 3 steps the proof of the `AG` property is still not successful, try to identify a missing global assumption (Section 6.3). Going back to step 4, add the assumption as a locale assumption and re-iterate.

Final step: When a model is found that successfully passes the above loop, generalize over its local policies to isolate their essential defining properties. This step will be illustrated by showing an advantage of the HOL approach over model checking in Section 7.2.

5.2. Formalization of Airplane Infrastructure and Properties

We restrict the Airplane scenario to four identities: Bob, Charlie, Alice, and Eve. Bob acts as the pilot, Charlie as the copilot, and Alice as the flight attendant. Eve is an identity

representing the malicious agent that can act as the copilot although not officially acting as an airplane actor. The identities that act legally inside the airplane infrastructure are listed in the set of airplane actors.

```
fixes airplane_actors :: identity set
defines airplane_actors_def: airplane_actors ≡ {''Bob'', ''Charlie'', ''Alice''}
```

In the above locale definition we use the **fixes** keyword to introduce a locale constant with its type which is then specified by **defines**. In the following, we drop all these elements but the actual definition to make the exposition shorter and clearer.

To represent the layout of the airplane, a simple architecture is best suited for the purpose of security policy verification. The locations we consider for the graph are `cabin`, `door`, and `cockpit`. They are defined as locale definitions and assembled in a set `airplane_locations`.

```
cabin ≡ Location 0
door ≡ Location 1
cockpit ≡ Location 2
airplane_locations ≡ { cabin, door, cockpit }
```

The actual layout and the initial distribution of the actors in the airplane infrastructure is defined by the following graph `ex_graph` ("`ex`" stands for "example") in which the actors Bob and Charlie are in the cockpit and Alice is in the cabin.

```
ex_graph ≡ Lgraph
          {(cockpit, door),(door,cabin)}
           (λ x. if x = cockpit then [''Bob'', ''Charlie'']
                 else (if x = door then []
                       else (if x = cabin then [''Alice''] else [])))
          ex_creds ex_locs
```

The two additional inputs `ex_creds` and `ex_locs` for the constructor `Lgraph` are the credential and role assignment to actors and the state function for locations (introduced in Section 4.4), respectively. For the airplane scenario, we use the function `ex_creds` to assign the roles and credentials to actors. For example, for `Actor ''Bob''` the following function returns the pair of lists `([''PIN''], [''pilot''])` assigning the credential `PIN` to this actor and designating the role `pilot` to him.

```
ex_creds ≡ (λ x.
  (if x = Actor ''Bob'' then ([''PIN''], [''pilot''])
   else (if x = Actor ''Charlie'' then ([''PIN''],[''copilot''])
         else (if x = Actor ''Alice'' then ([''PIN''],[''flightattendant''])
               else ([],[])))))"
```

The final parameter `ex_locs` describes different states of the airplane. Concretely, we have modelled them as locations, distinguishing three locations: `cabin`, `door`, and `cockpit`. The door can be in three different states (`''norm''`, `''locked''`, and `''unlocked`); the cockpit in two (`''air''` and `''ground''`), where the cockpit stands here for the whole airplane

(that is, all locations are either in the air or on the ground). Different ways to model this are possible, important is only that there are these 3×2 different states. Similar to the previous function `ex_creds`, the function `ex_locs` assigns these states to the locations of the infrastructure graph. For instance,

```
ex_locs ≡ λ x. if x = door then [''norm'']
else (if x = cockpit then [''air''] else [])
```

means that the door is in the normal state and the airplane is in the air.

5.3. Initial Global and Local Policies

In the Isabelle Insider framework, we define a global policy reflecting the global safety and security goal and then break that down into local policies on the infrastructure. The verification will then analyze whether the infrastructure's local policies yield the global policy.

Globally, we want to exclude attackers to ground the plane. In the formal model, landing the airplane results from an actor performing a `put` action (see Section 4.5) in the cockpit and thereby changing the state from `air` to `ground`.

Therefore, we specify the global policy as "no one except airplane actors can perform `put` actions at location cockpit" by the following predicate over infrastructures `I` and actor identities `a`.

```
global_policy I a ≡  a ∉ airplane_actors ⟶ ¬(enables I cockpit (Actor a) put)
```

We next attempt to define the local policies for each location as a function mapping locations to sets of pairs: the first element of each pair for a location `l` is a predicate over actors specifying the conditions necessary for an actor to be able to perform the actions specified in the set of actions which is the second element of that pair. The local policy functions are additionally parameterized over an infrastructure graph `G` since this may dynamically change through the state transition.

```
local_policies G ≡
(λ y. if y = cockpit then
      {(λ x. (∃ n. (n @G cockpit) ∧ Actor n = x), {put}),
       (λ x. (∃ n. (n @G cabin) ∧ Actor n = x
             ∧ has (x, ''PIN'')∧ isin G door ''norm''), {move}) }
      else (if y = door then {(λ x. True, {move})}
            else (if y = cabin then {(λ x. True, {move})} else {})))
```

This policy expresses that any actor can move to door and cabin but places the following restrictions on cockpit.

`put:` to perform a `put` action, that is, put the plane into a new position or put the lock, an actor must be at position cockpit, i.e., in the cockpit, which is expressed using the special location operator `n` $@_G$`l` stating that identity `n` is at location `l` in graph `G`;

`move:` to perform a move action at location cockpit, that is, move into it, an actor must be at the position cabin, must be in possession of PIN (formalized with the operator `has`), and door must be in state norm expressed using `isin`.

Although this policy abstracts from the buzzer, the 30 sec delay, and a few other technical details, it captures the essential features of the cockpit door.

The graph, credentials, and features are plugged together with the policy into the infrastructure `Airplane_scenario`, which represents the initial state of the airplane.

```
Airplane_scenario ≡ Infrastructure ex_graph local_policies
```

5.4. Insider Attack, Safety, and Security

We now first stage the insider attack and introduce basic definitions of safety and security for the airplane scenario. To invoke the insider within an application of the Isabelle Insider framework, we assume in the locale `airplane` as a locale assumption with `assumes` that the tipping point has been reached for `Eve` which manifests itself in her `actor_state` assigned by the locale function `astate`: if the identity `x` input to `astate` is `''Eve''` then `x` is depressed and is in danger of tipping over to insider; in all other cases (captured by the wild card `"_"`) actor `x` is happy and has no "insider `motivations`" expressed by the empty `motivations` set {}.

```
astate x =  (case x of
           ''Eve'' ⇒ Actor_state depressed {revenge, peer_recognition}
          | _ ⇒ Actor_state happy {})
```

In addition, we state that she is an insider being able to impersonate `Charlie` by locally assuming the `Insider` predicate. This predicate allows an insider to impersonate a set of other actor identities; in this case the set is singleton.

```
assumes Eve_precipitating_event: tipping_point(astate ''Eve'')
assumes Insider_Eve : Insider ''Eve'' {''Charlie''}
```

Next, the process of analysis uses this assumption as well as the definitions of the previous section to prove security properties interactively as theorems in Isabelle. We use the strong insider assumption here up front to provide a first sanity check on the model by validating the infrastructure for the "normal" case. We prove that the global policy holds for the pilot Bob. To illustrate a proof in Isabelle, we show the statement of the theorem including the Isabelle proof script. The system replies of the interaction with Isabelle are omitted but can be simply recreated by running that script.

```
lemma ex_inv: global_policy Airplane_scenario ''Bob''
by (simp add: Airplane_scenario_def global_policy_def airplane_actors_def)
```

The proof is finished with one complex step: unfold the definitions of the scenario given by `Airplane_scenario_def` and two other definitions and then apply the simplifier, an automated technique that applies equational (including conditional) rewriting to solve a goal.

We can prove the same theorem for `Charlie` who is the copilot in the scenario (omitting the proof and accompanying Isabelle commands).

```
global_policy Airplane_scenario ''Charlie''
```

But `Eve` is an insider and is able to impersonate `Charlie`. She will ignore the global policy. This insider threat can now be formalized as an invalidation of the global company policy for `''Eve''` in the following "attack" theorem named `ex_inv3`:

```
theorem ex_inv3: ¬ global_policy Airplane_scenario ''Eve''
```

This theorem can be proved by first invoking the above insider assumption about Eve unfolding the corresponding underlying definitions provided in the Isabelle Insider framework but finally then again using the powerful simplification tactic `simp`. The attack theorem is proved in Isabelle: it says that `Eve` can get access to the cockpit and put the position to `ground`. In other words, Eve can crash the plane. The proof is very similar to proofs of comparable theorems in other applications of the Isabelle Insider framework, for instance, for the IoT [24] or for auctions [23], and can basically be copied from there just replacing local definition names. Summarizing, the insider assumption allows modelling that actors may be the same as other actors. Policies that are expressed according to roles thus apply to those insiders that – given that they are attackers – are harmful.

Safety and security are sometimes introduced in textbooks as complementary properties, see, e.g., [42]. Safety expresses that humans and goods should be protected from negative effects caused by machines while security is the inverse direction: machines (computers) should be protected from malicious humans. Similarly, the following descriptions of safety and security in the airplane scenario also illustrate this complementarity: one says that the door must stay closed to the outside; the other that there must be a possibility to open it from the outside.

Safety: If the actors in the cockpit are out of action, there must be a possibility to get into the cockpit from the cabin, and

Security: If the actors in the cockpit fear an attack from the cabin, they can lock the door.

In the formal translation of these properties into HOL, this complementarity manifests itself even more clearly: the conclusions of the two formalizations of the properties are negations of each other. Safety is quite concisely described by stating that airplane actors can move into the cockpit.

```
Safety I a ≡ a ∈ airplane_actors ⟶ (enables I cockpit (Actor a) move)
```

Security can also be defined in a simple manner as the property that no actor can move into the cockpit if the door is on lock.

```
Security I a ≡ isin (graphI I) door ''locked''
               ⟶ ¬(enables I cockpit (Actor a) move)
```

These two properties are defined for any infrastructure `I` so we can apply them to the initial airplane scenario we have defined in the previous section. For this `Airplane_scenario`, we can show safety, for example, for `Alice` because she is in the cabin.

lemma `Safety: Safety Airplane_scenario ''Alice''`

In general, we could prove safety for any airplane actor who is in the cabin for this state of the infrastructure.

In a slightly more complex proof, we can prove security for any other identity which can be simply instantiated to `''Bob''`.

lemma `Security: Security Airplane_scenario ''Bob''`

The simple formalizations of safety and security enable proofs only over a particular state of the airplane infrastructure at a time but this is not enough since the general airplane structure is subject to state changes. For a general verification, we need to prove that the properties of interest are preserved under potential changes. Since the airplane infrastructure permits, for example, that actors move about inside the airplane, we need to verify safety and security properties in a dynamic setting. After all, the insider attack on Germanwings Flight 9525 appeared when the pilot had moved out of the cockpit. Furthermore, we want to redefine the policy into the two-person policy and examine whether safety and security are improved. For these reasons, we next apply the general Kripke structure mechanism introduced in Section 4.5 to the airplane scenario.

6. Analysis of Safety and Security Properties

In this section we first introduce a Kripke structure to model state transitions in the airplane scenario. Then we formalize the two-person rule and look how this rule is related to the property that the airplane is not in danger with respect to an insider attack. We show that an additional assumption is necessary to prove this property.

6.1. Kripke Structure for Airplane Scenario

The state transition relation $\rightarrow_i$ introduced in Section 4.5 is generally defined for a type class `state`. Therefore, we can instantiate the state transition for the type `infrastructure` as `state_transition_in` written as infix operator $\rightarrow_n$. Consequently, we can define the set of all states that are in the reflexive transitive closure of the infrastructure transition relation when starting in the infrastructure `Airplane_scenario` as a locale definition `Air_states`.

`Air_states` $\equiv$ `{ I. Airplane_scenario` $\rightarrow_n$`^* I }`

From there, we can define a corresponding Kripke structure by applying the constructor `Kripke` to the above state set and the singleton set of `Airplane_scenario` as the (only) initial state.

`Air_Kripke` $\equiv$ `Kripke Air_states {Airplane_scenario}`

We now illustrate how we can use this Kripke structure to explore and potentially invalidate the policy. The state of the infrastructure that represents the fatal state is when the pilot has moved out and the door is locked. We introduce a locale definition `aid_graph` to represent the graph for this infrastructure using the acronym `aid` for "**a**irplane **i**n **d**anger" to signify the relation to the state in its graph component.

```
aid_graph ≡ Lgraph
            {(cockpit, door),(door,cabin)}
             (λ x. if x = cockpit then [''Charlie'']
                 else (if x = door then []
                       else (if x = cabin then [''Bob'', ''Alice''] else [])))
            ex_creds ex_locs'
```

The function `ex_locs'` encodes the state of the airplane where the door is now locked.

```
ex_locs' ≡ λ x. if x = door then [''locked'']
                 else (if x = cockpit then [''air''] else [])
```

We finally define a new infrastructure state that takes this graph and the same `local_policies` as `Airplane_scenario`.

```
Airplane_in_danger ≡ Infrastructure aid_graph local_policies
```

The airplane is potentially in danger in such a situation, since the copilot is on his own and may crash the airplane as a result if he executes an insider attack. For the analysis of security, we need to ask whether this new infrastructure state `Airplane_in_danger` is reachable via the state transition relation from the initial state. It is. We can prove the following as a theorem in the locale `airplane`.

theorem `step_allr: Airplane_scenario` $\rightarrow_n$`^* Airplane_in_danger`

As the name of this theorem suggests it is the result of lining up a sequence of steps that lead from the initial `Airplane_scenario` to that `Airplane_in_danger` state. In fact there are three steps via two intermediary infrastructure states `Airplane_getting_in_danger0` and `Airplane_getting_in_danger` (see Appendix A). The former encodes the state where `Bob` has moved to the cabin and the latter encodes the successor state in which additionally the door state has changed to `locked`. The definitions of these states are very similar to the above definition of `Airplane_in_danger` (see Appendix A). The proof of the theorem `step_allr` correspondingly lines up lemmas for each of the state transitions between the involved states. Once provided with these lemmas, the main proof is just one simplification with the underlying definition of the reflexive transitive closure of a relation. This is the advantage of using a richly equipped proof assistant: the theory library is well equipped with standard mathematics and the tactics work well on this basis. The only real work has to be done to prove the individual steps. However, although the proof scripts are a bit lengthy, this is just simple step by step unfolding of definitions and simplification. The only reason why it is not done in one step fully automatically is that some instantiations under existential quantifiers have to be inserted in the application of the state transition rules, like for example the rule `put` we have seen in Section 4.5.

Using the formalization of CTL over Kripke structures introduced in Section 4.5, we can now transform the attack sequence represented implicitly by the above theorem `step_allr` into a temporal logic statement. This attack theorem states that there is a path from the initial state of the Kripke structure `Air_Kripke` on which eventually the global policy is violated by the attacker.

theorem `aid_attack: Air_Kripke ⊢ EF ({x. ¬ global_policy x ''Eve''})`

The proof uses the underlying formalization of CTL and the lemmas that are provided to evaluate the `EF` statement on the Kripke structure. However, the attack sequence is already provided by the previous theorem. So the proof just consists in supplying the step lemmas for each step and finally proving that for the state at the end of the attack path, i.e., for `Airplane_in_danger`, the global policy is violated. This proof corresponds precisely to the proof of the attack theorem `ex_inv3`. It is not surprising that the security attack is possible in the reachable state `Airplane_in_danger` when it was already possible in the initial state. However, this statement is not satisfactory since the model does not take into account whether the copilot is on his own when he launches the attack. This is the purpose of the two-person rule which we want to investigate in more detail in this paper. Therefore, we next address how to add the two-person rule to the model.

6.2. Introduce Two-Person Rule

To express the rule that two authorized personnel must be present at all times in the cockpit, we define a second set of local policies called `local_policies_four_eyes` (two people have four eyes). The following function realizes this two-person constraint by providing prerequisites for when actions can be performed, by whom, and where. It requests that the number of actors at the location `cockpit` in the graph `G` given as input must be at least two to enable actors at the location to perform the action `put`. Formally, we can express this here as `2 ≤length(agra G cockpit)` since we have all of arithmetic available (remember `agra G y` is the list of actors at location `y` in `G` introduced in Section 4.5).

```
local_policies_four_eyes G ≡
(λ y. if y = cockpit then
       {(λ x. (∃ n. n @_G cockpit ∧ Actor n = x) ∧ 2 ≤ length(agra G y) ∧
             ∀ h ∈ set(agra G y). h ∈ airplane_actors), {put}),
       (λ x. (∃ n. n @_G cabin ∧ Actor n = x) ∧ has (x, ''PIN'')∧
                   isin G door ''norm''), {move})}
       else (if y = door then
             {(λ x. ((∃ n. n @_G cockpit ∧ Actor n = x)
                     ∧ 3 ≤ length(agra G cockpit)), {move})}
            else (if y = cabin then
                  {(λ x. ∃ n. n @_G door ∧ Actor n = x), {move})}
                 else {})))
```

Note that the two-person rule requires three people to be at the cockpit before one of them can leave. This is formalized as a condition on the `move` action of location `door`. A move of an actor `x` in the cockpit to `door` is allowed only if three people are in the cockpit. Practically, it enforces a person, say Alice, to first enter the cockpit before the pilot, Bob, can leave. However, this condition is necessary to guarantee that the two-person requirement for `cockpit` is sustained by the dynamic changes to the infrastructure state caused by actors' moves. A move to location `cabin` is allowed only from `door` so no additional condition is necessary here.

What is stated informally above seems intuitive and quite easy to believe. However, comparing to the earlier formalization of this two-person rule [2], it appears that the earlier version did not have the additional condition on the action `move` to `door`. One may argue that in the earlier version the authors did not consider this because they had neither state transitions, Kripke structures, nor CTL to consider dynamic changes. However, in the current paper this additional side condition occurred to us only when we tried to prove the following invariant `two_person_inv1` which is needed in a subsequent security proof.

lemma `two_person_inv1:`
`Airplane_not_in_danger_init` $\rightarrow_n$`^* I` $\Longrightarrow$ `2` $\leq$ `length (agra (graphI I) cockpit)`

This proof requires an induction over the state transition relation starting in the infrastructure state `Airplane_not_in_danger_init` with Charlie and Bob in the cockpit and the two-person policy `local_policies_four_eyes` in place.

`Airplane_not_in_danger_init` $\equiv$ `Infrastructure ex_graph local_policies_four_eyes`

The corresponding Kripke structure of all states originating in this infrastructure state is defined as `Air_tp_Kripke`. Within the induction for the proof of the above `two_person_inv1`, a preservation lemma is required that proves that if the condition `2` $\leq$ `length (agra (graphI I) cockpit)` holds for `I` and `I` $\rightarrow_n$ `I'` then it also holds for `I'`. The preservation lemma is actually trickier to prove. It uses a case analysis over all the transition rules for each action. The rules for `put` and `get` are easy to prove for the user as they are solved by the simplification tactic automatically. The case for action `move` is the difficult case. Here we actually need to use the precondition of the policy for location `door` in order to prove that the two-person invariant is preserved by an actor moving out of the cockpit. In this case, we need for example, invariants like the lemma `actors_unique_loc_aid_step` below that shows that in any infrastructure state originating from `Airplane_not_in_danger_init` actors ever appear in one location only and they do not appear more than once in a location – which is expressed in a predicate `nodup` (see Appendix A). The following lemma is an instantiation of a similar general lemma proved for all Kripke structures – similar to the lemma `init_state_policy` mentioned in Section 4.5.

lemma `actors_unique_loc_aid_step:`
`Airplane_not_in_danger_init` $\rightarrow_n$`^* I`
$\Longrightarrow$ $\forall$ `a. (`$\forall$ `l l'. a` $@_{\texttt{graphI I}}$`) l` $\wedge$ `a` $@_{\texttt{graphI I}}$ `l'` $\longrightarrow$ `l = l'`
$\wedge$ `(`$\forall$ `l. nodup a (agra (graphI I) l))`

6.3. Revealing Necessary Assumption by Proof Failure

So far we used CTL only to discover attacks using `EF` formulas. What we need for general security and what we will consider next is to prove a global property with the temporal operator `AG` that proves that from a given initial state the global policy holds in all (`A`) states globally (`G`).

As we have seen in the previous section when looking at the proof of `two_person_inv1`, it is not evident and trivial to prove that all state changes preserve security properties. However, even this invariant does not suffice. Even if the two-person rule is successfully enforced in a state, it is on its own still not sufficient. When we try to prove

```
Air_tp_Kripke ⊢ AG {x. global_policy x ''Eve''}
```

for the Kripke structure `Air_tp_Kripke` ("`tp`" stands for "two-persons") consisting of all states originating in state `Airplane_not_in_danger_init`, we cannot succeed. In fact, in that Kripke structure there are infrastructure states where the insider attack is possible. Despite the fact that we have stipulated the two-person rule as part of the new policy and despite the fact that we can prove that this policy is preserved by all state changes, the rule has no consequence on the insider. Since Eve can impersonate the copilot Charlie, whether two people are in the cockpit or not, the attack can happen.

What we realize through this failed attempt to prove a global property is that the policy formulation does not entail that the presence of two people in itself actually disables an attacker.

This insight reveals a hidden assumption. Formal reasoning systems have the advantage that hidden assumptions must be made explicit. In human reasoning they occur when people assume a common understanding, which may or may not be actually the case. In the case of the rule above, its purpose may lead to an assumption that humans accept but which is not warranted.

We use again a locale definition to encode this intentional understanding of the two-person rule. The formula `foe_control` encodes that for any location `l` and action `c` in the Kripke structure `K` holds that if in all states `I` in `K` there is an `Actor x` present that is not an insider, that is, is not impersonated by Eve, that then the insider is disabled for that action `c`. In other words, it is assumed that a potential insider is controlled by the presence of a non-insider. In particular, the insider will not be able to knock out the non-insider.

```
foe_control l c K ≡ (∀ I ∈ states K. (∃ x. x @_I l ∧ Actor x ≠ Actor ''Eve'')
                    ⟶ ¬(enables I l (Actor ''Eve'') c))
```

6.4. Proving Security in Refined Model

Having identified the missing formulation of the intentional effects of the two-person rule, we can now finally prove the general security property using the above locale definition. We assume in the locale `airplane` an instance of `foe_control` for the cockpit and the action `put` in the Kripke structure `Air_tp_Kripke`.

```
assumes cockpit_foe_control: foe_control cockpit put
```

With this assumption, we are now able to prove that for all infrastructure states of the system `airplane` originating in state `Airplane_not_in_danger_init` Eve cannot put the airplane to the ground (intuition: "Four eyes guarantee there is no danger").

```
theorem Four_eyes_no_danger: Air_tp_Kripke ⊢ AG {x. global_policy x ''Eve''}
```

The proof uses as a key lemma `tp_imp_control` (for "two persons implies control") stating that within Kripke structure `Air_tp_Kripke` there is always someone in the cockpit who is not the insider.

lemma `tp_imp_control: Airplane_not_in_danger_init` $\rightarrow_n$`^* I`
$\Longrightarrow$ $\exists$ `x. x` $@_I$ `cockpit` $\wedge$ `Actor x` $\neq$ `Actor ''Eve''`

This lemma can be proved by using the invariant that always two people are in the cockpit. However, the invariant `two_person_inv1` cannot be used directly since it is a lemma over lists rather than sets. Instead of re-formulating the model with sets, we use the simple fact about finite sets and lists that a list without duplications has a length that is equal to the cardinality of the corresponding set.

```
(∀ a. nodup a l) ⟶ card (set l) = length l
```

This general lemma enables together with the invariant `actors_unique_step_loc_aid_step` the proof of the more suitable invariant `two_person_set_inv`.

lemma `two_person_set_inv: Airplane_not_in_danger_init` $\rightarrow_n$`^* I`
$\Longrightarrow$ `2` $\leq$ `card (set (agra (graphI z) cockpit))`

Using the assumption `foe_control`, we can now – mainly by applying modus ponens – derive that Eve is not enabled in `cockpit` to perform `put` for any infrastructure state originating from `Airplane_not_in_danger_init`.

```
Airplane_not_in_danger_init →n^* I ⟹ ¬ enables I cockpit (Actor ''Eve'') put
```

Now, the proof of theorem `Four_eyes_no_danger` (see Appendix A) uses simplification on basic lemmas for Kripke structures and CTL to reduce to the above fact which finishes the proof.

7. Model checking and Generalizing over Policies

In this section, we consider an alternative approach to formalizing and verifying the airplane case study using a model checker. We then show how the formalization in the Isabelle Insider framework demonstrated in this paper can be generalized over policies. This additional work serves to illustrate the surplus gained by using the heavier Isabelle approach.

7.1. Model Checking the Airplane Case Study

We will in the following present a NuSMV (see Cimatti et al. [4]) representation of the airplane scenario and introduce three different policies. The full NuSMV sources of our case studies are available online [3]. The different policies each guarantee the two-person rule that means that at least two crew members must be in the cockpit at any moment in time. In the first policy, which is the policy explored in our Isabelle formalization, see section 6.2, this is achieved by the fact that a cabin crew member has to enter the cockpit before a pilot can leave; in the second policy the two pilots must not leave the cockpit; and in the third policy only one of three pilots may leave.

Common in the representations is to represent the location of the agents (numbered by 1, 2, 3, and 4 for Alice (cabin crew), Bob, Charlie, and Doris (pilots), as an array with the three values `cockpit`, `dr`, and `cockpit`.

Any member of staff knows the pin to open the door in case the pilots are all incapacitated. In order to express the two-person rule we define the number of non-incapacitated crew members (or pilots for the third policy) in the cockpit by the definition:

```
cocknum :=
  -- Three non-incapacitated crew members are in the cockpit.
  case (((airplane[1] = cockpit) & (airplane[2] = cockpit) & (airplane[3] = cockpit) &
          !incapacitated[1] & !incapacitated[2] & !incapacitated[3]))                    :3;
  -- Two non-incapacitated crew members are in the cockpit.
    (((airplane[1] = cockpit) & (airplane[2] = cockpit) & !incapacitated[1] & !incapacitated[2]) |
     ((airplane[2] = cockpit) & (airplane[3] = cockpit) & !incapacitated[2] & !incapacitated[3]) |
     ((airplane[1] = cockpit) & (airplane[3] = cockpit) & !incapacitated[1] & !incapacitated[3]))    :2;
   -- One non-incapacitated crew member is in the cockpit.
    (((airplane[1] = cockpit) & !incapacitated[1]) | ((airplane[2] = cockpit)  & !incapacitated[2]) |
                                                     ((airplane[3] = cockpit) & !incapacitated[3]))    :1;
  -- No non-incapacitated crew member is in the cockpit.
    TRUE                                                                                             :0;
   esac;
```

We make the assumption that the crew members know initially the pin for the door and never forget it, non-crew members do not know the pin initially but may get hold of it (e.g. by forcing cabin crew to disclose it). Initially the crew members are all not incapacitated and nobody wants to leave the cockpit.

We then express when a crew member may leave the cockpit. For the first policy, as introduced at the start of this subsection, it is represented for one pilot as:

```
next(s.airplane[2]) :=
  case
    s.airplane[2] = cockpit & s.threep   & s.leave = 2: {dr};      -- 2 leaves the cockpit.
    s.airplane[2] = cockpit & !s.threep  & s.leave = 2: {cockpit}; -- 2 may not leave the cockpit.
    s.airplane[2] = cockpit & !(s.leave = 2): {cockpit};           -- 2 does not want to leave the cockpit.
    s.airplane[2] = dr & s.door = UL : {dr, cockpit, cabin};       -- 2 may enter the cockpit.
    s.airplane[2] = dr & !(s.door = UL) : {dr, cabin};             -- 2 may not enter the cockpit.
    s.airplane[2] = cabin : {cabin, dr};                           -- 2 may move to the door.
  esac;
```

The main safety requirement is expressed by

```
AG(!s.incapacitated[1] & !s.incapacitated[2] & !s.incapacitated[3]  -> s.airplaneSafe)
```

It means as long as the crew members are not incapacitated the airplane is safe (under the first policy from above).

The same is the case under the second policy where no pilot may leave the cockpit:

```
next(s.airplane[2]) :=
  case
    s.airplane[2] = cockpit: {cockpit};                            -- 2 cannot leave the cockpit.
    s.airplane[2] = dr & s.door = UL : {dr, cockpit, cabin};       -- 2 may enter the cockpit.
    s.airplane[2] = dr & !(s.door = UL) : {dr, cabin};             -- 2 may not enter the cockpit.
    s.airplane[2] = cabin : {cabin, dr};                           -- 2 may move to the door.
  esac;
```

Under the third policy we have a third pilot in the cockpit and the main safety requirement is expressed by

```
AG ((!s.incapacitated[2] & !s.incapacitated[3] & !s.incapacitated[4]) -> s.airplaneSafe)
```

Finally, we present a formula in CTL that is not expressible in LTL. It states that along ALL paths there EXISTS a path such that if both pilots are incapacitated and the cabin crew member is not that then the cabin crew member is in the cockpit. That is, this property verifies that under the policy it is always possible for the flight attendant to open the door if the two pilots are incapacitated.

```
AG EF(!s.incapacitated[1] & s.incapacitated[2] & s.incapacitated[3] ->
      (s.airplane[1] = cockpit))
```

All four formulae are proved by NuSMV instantaneously. Note that the representation above makes use of translations of the policies into concrete code and does not allow for a generalization of the kind: "All policies that mean that always at least two crew members are in the cockpit and no non-crew member can enter the cockpit ensure that the airplane is safe."

7.2. Generalizing over policies

The main theorem `Four_eyes_no_danger` in Section 6.4 establishes safety from the insider Eve for the Kripke structure `Air_tp_Kripke`. In this Kripke structure, the local policy `local_policies_four_eyes` holds in the initial state and is preserved in all reachable states. It is a specific implementation that corresponds to one of the policies that have been illustrated in the previous section: the global security invariant guaranteeing that always two people are in the cockpit is achieved by enforcing that three actors need to be there before one can leave.

Compared to the model checking approach described and illustrated on the current case study, we seem to gain nothing by using the Isabelle Insider framework. Clearly, a benefit of using Isabelle is that we could reuse the model abstraction to help us build the NuSMV model but that hardly justifies the effort. Additionally, one may argue that in Isabelle, we may use general datatypes like natural numbers and quantify over them which is not possible in NuSMV due to finite state spaces. However, for the considered case study we may safely assume a finite number of states and use of quantification of functions in formulas could then simply be replaced by enumerating all cases (for example, in `cocknum` or `airplane` in Section 7.1).

Nevertheless, Isabelle's Higher Order Logic (HOL) has none of the restrictions that model checking has. Therefore, we can make a final important step in our developments that is not possible in model checking. We can generalize over any parameter since we are using HOL. Practically, this means that we can extract a parameter from any formula, quantify it and consider theorems on it. To illustrate this on a meaningful example, we now add a final step to the developments described in this paper so far and show how we can generalize over the policy in the Kripke structure and still gain the main result. That is, we can prove a generalized version of the theorem `Four_eyes_no_danger`.

```
theorem Gen_policy:
  foe_control cockpit put (Kripke { I. I0 →ᵢ^* I } {I0}) ⟹
   (∀ I. (I0, I) ∈ {(x::infrastructure, y::infrastructure). x →ₙ y}^*
              ⟶  2 ≤ card (set (agra (graphI I) cockpit))) ⟹
   (∀ z. (I0 ,z) ∈ {(x::infrastructure, y::infrastructure). x →ₙ y}^* ⟶
    (∀ h::identity ∈ set (agra (graphI z) cockpit). h ∈ airplane_actors)) ⟹
    Kripke  { I. I0 →ₙ^* I } {I0} ⊢ AG {x. global_policy x ''Eve''}
```

Compared to the theorem `Four_eyes_no_danger`, the theorem `Gen_policy` generalizes the Kripke structure `Airplane_tp_Kripke` to an arbitrary structure `Kripke { I. I0` $\rightarrow_n$`^* I } {I0}`, that is, any Kripke structure that starts from an arbitrary infrastructure `I0` and is closed under $\rightarrow_n$`^*` and may contain any security policy. Additional provisos guarantee that (1) `foe_control` holds for `cockpit` and `put` in this arbitrary Kripke structure, (2) the 2-person invariant holds in any reachable state, and (3) only airplane actors are in the cockpit in all reachable states. The proviso (1) corresponds to the hidden assumption `cockpit_foe_control` that we have assumed as a locale assumption for the specific case `Airplane_tp_Kripke` before. The proviso (2) is the abstract expression of the two person policy. Previously in the specific case `Airplane_tp_Kripke`, this property had been proved as the lemma `two_person_inv1`. Finally, the proviso (3) corresponds to the lemma `air_plane_actors_inv` (see Appendix A).

The proof of the theorem `Gen_policy` now follows quite closely that of the special case `Four_eyes_no_danger` simply replacing either the additional provisos (1–3) or the generalized lemmas for their specific counterparts.

As a validation of the generality of the newly derived theorem `Gen_policy`, we finally "reprove" the special theorem `Four_eyes_no_danger`.

```
theorem Four_eyes_no_danger': Air_tp_Kripke ⊢ AG (x. global_policy x ''Eve'')
unfolding Air_tp_Kripke_def Air_tp_states_def
by (rule Gen_policy, fold Air_tp_Kripke_def Air_tp_states_def,
    rule cockpit_foe_control, simp add: two_person_set_inv,
    simp add: airplane_actors_inv)
```

The theorem `Four_eyes_no_danger'` restates exactly the same proof goal as `Four_eyes_no_danger`. After unfolding the definitions of the specific Kripke structure `Air_tp_Kripke`, the Isar proof is contained in the brackets behind the `by`: applying the new generic theorem `Gen_policy` reduces to subgoals that can then be completely solved by plugging in the previously proved lemmas that fit to the provisos (1–3) of the theorem (after folding again the definition of `Air_tp_Kripke` to fit it back to the special form).

8. Discussion and Conclusions

In this section, we briefly discuss limitations and approaches to developing airplane policies, summarize the contributions of the paper, and present some concluding remarks.

8.1. Comparison of Isabelle and NuSMV

To provide a comparison between Isabelle and model checking, we give here some pros and cons of Isabelle and model checking.

Isabelle pros

The Isabelle Insider framework offers an explicit level of concepts, like policy, graph, actors and their roles and credentials, allowing reasoning at a meta-level due to Isabelle's expressive Higher Order Logic. This makes the formalization a framework. Generally, Isabelle is highly expressive, contains recursive datatypes and functions allowing representation of complex parts of applications. Generalization over higher order parameters is possible leading, as illustrated, on to the generalization of our main theorem. This leads to a "meta-theory" that can then be applied to more concrete cases as illustrated by applying the generalized theorem to re-prove the special case of one specific implementation of the policy.

Isabelle cons

Isabelle is not fully automated, thus expert level knowledge is necessary at least to do meta-level proofs. The application of the framework allows reuse and proofs that are simple (and thus highly automated). Isabelle's expressiveness leads to complex specifications that necessitate a deeper understanding for human users. An Isabelle application scenario is that a team of security experts and policy makers work with some security/formal methods engineer, experiment with policies and their implications, for example, for airplane security.

Model checking pros

The big advantage of model checking is that due to the restrictions on datatypes, the specifications are usually simpler. Also model checking offers fully automated verification (checking). Using a high level of abstraction, the same application scenarios as in Isabelle can be modelled. As far as only specific implementations in finite instantiations are concerned the same properties can also be verified as in Isabelle.

Model checking cons

Compared to Isabelle only restricted datatypes, only propositional logic in states, no general function types, and no general predicates are available. This makes a generalization over a higher order concept like a policy impossible as we illustrated. It also leads to a low level of abstraction when modelling applications. Consequently, the human needs to make sure that the abstraction does still adequately represent the real world. This is similar for Isabelle but there the much higher level of expressiveness makes that judgement easier and more natural. In model checking, almost the only obvious liaison are the chosen names for elements, "`airplaneSafe`" for example.

8.2. Aspects of Airplane Policies

In order to prove consequences of policies certain assumptions have to be made and it is important to analyze the assumptions, since any consequences hold only with respect to

the assumptions. An important assumption is that the airplane is initially not in danger, `Airplane_not_in_danger_init`. That is, if the assumption is violated initially (before the airplane leaves the ground) then we cannot conclude that the airplane will not be in danger later. Current policies do not assume that the cockpit door must be locked before passengers board the airplane. Actually, often it is still open and closed only later. This means that an attack by an outsider during this phase cannot be ruled out, or by an insider if only one pilot were in the cockpit and locked the door.

For airlines it is an important question whether they should follow a two-person rule and as a consequence of the events on 2015-03-24 with the Germanwings flight 9525 a number of countries recommended the rule and a number of airlines[10] introduced them – without public consideration of possible negative consequences. In a more recent development, some German airlines have rescinded the two-person rule,[11] since the introduction has also the disadvantage that it takes considerably longer for one person to leave and another to enter the cockpit than just for one person to leave. This means that with the two-person policy, each time a pilot/co-pilot leaves the cockpit the door is open for much longer than without the policy, hence increasing the risk of a hostile attack. Up to now no good improvement on the protocol for the door has been found, since any change seems to be paired with substantial disadvantage as well.

We have not formally modelled the situation and the reasoning behind this. We do this informally here. If we assume p_0, the probability that one pilot is an insider; p_1, the probability that a terrorist can use the time the door is open to enter the cockpit following the one-person rule and take over the plane; and p_2, the corresponding probability that a terrorist can enter the cockpit following the two-person rule.

Fortunately all these probabilities are very small. This means, however, that there is no reliable way to determine their values. It seems obvious that $p_2 > p_1$, it can be assumed that p_2 is considerably bigger than p_1.[12]

With these probabilities we get that an aircraft is in danger according to the one-person rule by $P(\textit{insider}\ \text{OR}\ \textit{terrorist}) = p_0 + p_1 - P(\textit{insider}\ \text{AND}\ \textit{terrorist}) \approx p_0 + p_1$. With the two-person rule we get $P(\textit{insider}\ \text{OR}\ \textit{terrorist}) = 0 + p_2 - 0 \cdot p_2 = p_2$.

The second equation of the first case assumes that the events that a pilot is an insider and that a terrorist can use the one-person rule to enter the cockpit are independent. The approximate equality follows since both p_0 and p_1 are very small, that is, the size of $p_0 \cdot p_1$ is negligible compared to either p_0 or p_1. In the second case it is assumed that the probability that an insider can harm the plane if not on their own is 0.

In order to follow a rational policy, an airline should look at the relationship of the probabilities in the two cases, that is, between p_2 and $p_0 + p_1$. It should go for the smaller probability. If the probability of a terrorist getting in following the two-person rule is greater

[10] This is reported, for instance, in an article of 2015-03-26 by Reuters, http://www.reuters.com/article/france-crash-cockpits-idUSL6N0WS6GR20150326.

[11] See https://phys.org/news/2017-04-german-airlines-scrap-two-person-cockpit.html and https://www.swiss.com/corporate/EN/media/newsroom/press-releases/media-release-20170428.

[12] See, https://www.easa.europa.eu/newsroom-and-events/news/minimum-cockpit-occupancy-easa-issues-revised-safety-information-bulletin

than that of getting in following the one-person rule plus the probability of an insider doing harm then follow the one-person rule, else the two-person rule.

However, as we have mentioned above it is very difficult to determine these probabilities. Hence, when it comes to defining policies, it looks much more fruitful to consider possibilistic specifications of systems, actors, and their possible behaviours in order to understand better the shortcomings and possible glitches when imposing policies as security rules than to apply probabilistic reasoning.

8.3. Summary of Contributions

We have presented an extended version of the Isabelle Insider framework demonstrating it on a case study of airplane policies in the presence of Insider threats. Isabelle is a tool that allows to build well-founded definitions that come with proof rules for model features. Datatypes and induction on predicates are derived from first principles like fixpoint induction and datatype isomorphism in HOL and thus are mathematically sound. This is known as the principle of conservative extension. It is this principle that adds a special quality of mathematical soundness to Isabelle formalizations. The Isabelle Insider framework is such a conservative extension, hence consistent in itself. It is also a framework since its theories may be applied to arbitrary applications of infrastructures including human actors with actions and policies. The applicability is achieved by fully exploiting Isabelle's genericity enabling a generic state type with transition relations, Kripke structures and temporal logic CTL. The airplane case study presented in this paper has served as a source for requirements and test case for the Isabelle Insider framework and has yielded the following extensions:

- A generic notion of state allows to embrace infrastructures with actors, actions, and policies in one type of state. The state transition relation over this state is now defined as an axiomatic type class allowing to instantiate the framework to the complex state type of an application like airplane.
- Kripke structures and CTL allow stating and reasoning over temporal properties of these dynamic states in a fully consistent way in Isabelle.

A further contribution given by the airplane application is the extension of the framework by a methodology. This framework enables experimenting with policies over infrastructures with humans and actions as has been demonstrated on the airplane case study. The experimentation may lead to revealing missing assumptions which allowed us to define an informal methodology. Further contributions are not of a technological but more of a scientific nature in comparing the expressivity and performance of the Isabelle Infrastructure framework with respect to a dedicated implementation of CTL in the model checker NuSMV.

- We present the airplane model using a suitable abstraction of details in NuSMV proving the policy in various implementations.
- By contrast, to illustrate the expressive power of Isabelle, we generalize over concrete policies reproving the global security theorem.

Concerning the human aspect, the Isabelle Insider framework offers now a representation including motivation and insiderness which allows policies to make dynamic state based reasoning dependent on human aspects. However – and this could be future work – the formalization does not accommodate the cognitive dynamics of the human mood although this could have been a possible extension.

8.4. Conclusions and Acknowledgments

The current work has picked up on the earlier application [2] on investigating airplane safety and security in the presence of insiders. We have successfully proved the major observation of that earlier paper: a thorough logical analysis of the airplane scenario requires the exploration of the state space for all possible changes to the state. Integrating the extensions to Kripke structures and CTL in our model we were now able to explore the airplane scenario thoroughly and completely. The analysis in the interactive theorem prover Isabelle has shown that earlier results were partly misleading because security results were only relating statically to one specific state at a time. This work has shown that it is possible to support formal modeling and analysis for insider threats by a rich structural state supporting representation of human aspects and general policies in the Isabelle Insider framework while simultaneously enabling dynamic state change and temporal specification. The point is also made that the effect of an Isabelle formalization provides us the possibility of systematically revealing hidden (implicit) assumptions. Since this can be done systematically, it leads to a methodology. Moreover, we have provided an alternative analysis with the model checker NuSMV and generalized the Isabelle theorems over arbitrary policies to show advantages and drawbacks of our approach. In section 8.2, we have also discussed how the policy makers' decisions should rationally follow from the relationships between the probabilities of attacks, which depend on the policies adopted.

We are very much indebted to the anonymous referees that have very constructively commented on various stages of this article. We are very grateful for their time and effort spent because it has greatly supported us in improving the technical contribution as well as the exposition.

References

[1] F. Kammüller, C. W. Probst, Modeling and verification of insider threats using logical analysis, IEEE Systems Journal, Special issue on Insider Threats to Information Security, Digital Espionage, and Counter Intelligence 11 (2) (2017) 534–545. doi:10.1109/JSYST.2015.2453215.
URL http://dx.doi.org/10.1109/JSYST.2015.2453215

[2] F. Kammüller, M. Kerber, Investigating airplane safety and security against insider threats using logical modeling, in: IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'16, IEEE, 2016.

[3] M. Kerber, F. Kammüller, NuSMV formalisation of airplane scenarios with two-person-in-cockpit policies (2020).
URL https://github.com/flokam/NuSMV_Airplane

[4] A. Cimatti, M. Roveri, R. Cavada, R. Sebastiani, S. Tonetta, A. Mariotti, A. Micheli, S. Mover, M. Dorigatti, NuSMV: a new symbolic model checker (2020).
URL http://nusmv.fbk.eu

[5] F. Kammüller, IsabelleInsider – insider framework based on Kripke structures and CTL with example of airplane attack, available from https://github.com/flokam/IsabelleInsider. (2020).
[6] J. Glasser, B. Lindauer, Bridging the gap: A pragmatic approach to generating insider threat data, in: WRIT'13, IEEE, 2013.
[7] M. Bishop, H. M. Conboy, H. Phan, B. I. Simidchieva, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, S. Peisert, Insider threat identification by process analysis, in: Proceedings of the third IEEE Workshop on Research in Insider Threats, WRIT'14, IEEE, 2014.
[8] M. Bishop, K. Nance, J. Clark, Inside the insider threat (introduction), in: Proceedings of the 50th Hawaii International Conference on System Sciences, 2017, p. 2637.
URL http://hdl.handle.net/10125/41474
[9] D. M. Cappelli, A. P. Moore, R. F. Trzeciak, The CERT Guide to Insider Threats: How to Prevent, Detect, and Respond to Information Technology Crimes (Theft, Sabotage, Fraud), 1st Edition, SEI Series in Software Engineering, Addison-Wesley Professional, 2012.
URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321812573
[10] F. L. Greitzer, J. R. Strozer, S. Cohen, A. P. Moore, D. Mundie, J. Cowley, Analysis of unintentional insider threats deriving from social engineering exploits, in: Proceedings of the third IEEE Workshop on Research in Insider Threats, WRIT'14, IEEE, 2014.
[11] E. T. Axelrad, P. J. Sticha, O. Brdiczka, J. Shen, A bayesian network model for predicting insider threats, in: 2013 IEEE Security and Privacy Workshops, IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 82–89. doi:http://doi.ieeecomputersociety.org/10.1109/SPW.2013.35.
[12] J. R. C. Nurse, O. Buckley, P. A. Legg, M. Goldsmith, S. Creese, G. R. T. Wright, M. Whitty, Understanding Insider Threat: A Framework for Characterising Attacks, in: IEEE Security and Privacy Workshops (SPW), IEEE, 2014.
[13] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, B. Monate, Testing or formal verification: Do-178c alternatives and industrial experience, IEEE Software 30 (3) (2013) 50–57. doi:10.1109/MS.2013.43.
[14] C. O'Halloran, Automated verification of code automatically generated from simulink, Automated Software Engineering 20 (2) (2013) 237–264. doi:10.1007/s10515-012-0116-5.
[15] M. O. Khan, M. Sievers, S. Standley, Model-based verification and validation of spacecraft avionics, NASA Jet Propulsion Laboratory.
URL http://hdl.handle.net/2014/44932
[16] D. v. Oheimb, M. Maidl, R. Robinson, Security architecture and formal analysis of an airplane software distribution system, in: AIAA (Ed.), 26th Congress of the International Council of the Aeronautical Sciences (ICAS), Proceedings on CD-ROM available from secr.exec@icas.org, 2008, pp. 1–12, http://ddvo.net/papers/ICAS08.html.
[17] G. Luettgen, V. Carreno, Analyzing mode confusion via model checking, in: International SPIN Workshop on Model Checking of Software, Vol. 1680 of LNCS, Springer, 1999, pp. 120–135.
URL https://eur02.safelinks.protection.outlook.com/?url=httpsapps.dtic.mildtictr
[18] C. Munoz, G. Dowek, V. Carreno, Modeling and verification of an air traffic concept of operations, in: ISSTA'04, ACM, 2004.
[19] C. Munoz, V. Carreno, G. Dowek, Formal analysis of the operational concept for the small aircraft transportation system, in: Rigorous Engineering of Fault-Tolerant Systems, Vol. 4157 of LNCS, 2006, p. 306325.
[20] F. Kammüller, C. W. Probst, Invalidating policies using structural information, in: IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'13, 2013.
[21] F. Kammüller, C. W. Probst, Combining generated data models with formal invalidation for insider threat analysis, in: IEEE Security and Privacy Workshops, Workshop on Research in Insider Threats, WRIT'14, 2014.
[22] J. Boender, M. G. Ivanova, F. Kammüller, G. Primiero, Modeling human behaviour with higher order logic: Insider threats, in: STAST'14, IEEE, 2014, co-located with CSF'14 in the Vienna Summer of Logic.

[23] F. Kammüller, J. R. C. Nurse, C. W. Probst, Attack tree analysis for insider threats on the IoT using Isabelle, in: Human Aspects of Information Security, Privacy, and Trust - Fourth International Conference, HAS 2015, Held as Part of HCI International 2016, Toronto, Lecture Notes in Computer Science, Springer, 2016, invited paper.
[24] F. Kammüller, Human centric security and privacy for the iot using formal techniques, in: 3d International Conference on Human Factors in Cybersecurity, Vol. 593 of Advances in Intelligent Systems and Computing, Springer, 2017, pp. 106–116, affiliated with AHFE'2017.
[25] M. G. Ivanova, C. W. Probst, R. R. Hansen, F. Kammüller, Transforming graphical system models into graphical attack models, in: Graphical Models for Security, GraMSec'15, LNCS, Springer, 2015, co-located with CSF'15.
[26] F. Kammüller, M. Kerber, C. Probst, Towards formal analysis of insider threats for auctions, in: 8th ACM CCS International Workshop on Managing Insider Security Threats, MIST'16, ACM, 2016.
[27] CHIST-ERA, Success: Secure accessibility for the internet of things, http://www.chistera.eu/projects/success (2016).
[28] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, S. M. Veres, Formal verification of autonomous vehicle platooning, Science of Computer Programming 148 (2017) 88–106. doi:10.1016/j.scico.2017.05.006.
[29] Wikipedia, September 11 attacks, accessed June 2019 (2019).
URL https://en.wikipedia.org/wiki/September_11_attacks
[30] T. H. Kean et al., Complete 9/11 commission report, http://govinfo.library.unt.edu/911/report/911Report.pdf (2004).
[31] Wikipedia, List of aircraft hijackings, accessed June 2019 (2019).
URL https://en.wikipedia.org/wiki/List_of_aircraft_hijackings
[32] The Star, Jet cockpit doors nearly impossible to open by intruders, accessed June 2019 (2018).
URL http://www.thestar.com/news/world/2015/03/26/jet-cockpit-doors-nearly-impossible-to-open-by-intruders.html
[33] Reinforced cockpit door – description & procedures, an Airbus film directed by Bertrand Sirven. Accessed June 2019 (September 2002).
URL https://www.youtube.com/watch?v=ixEHV7c3VXs
[34] F. Kammüller, Modular reasoning in isabelle, in: D. MacAllester (Ed.), 17th International Conference on Automated Deduction, CADE-17, Vol. 1831 of LNAI, Springer, 2000.
[35] F. Kammüller, Isabelle modelchecking for insider threats, in: Data Privacy Management, DPM'16, 11th Int. Workshop, Vol. 9963 of LNCS, Springer, 2016, co-located with ESORICS'16.
[36] F. Kammüller, A formal development cycle for security engineering in isabelle (2020). arXiv:2001.08983.
[37] L. C. Paulson, Proving properties of security protocols by induction, in: CSFW, IEEE Computer Society, 1997, pp. 70–83.
[38] D. Dolev, A. C. Yao, On the security of public key protocols, in: 22nd Annual Symposium on Foundations of Computer Science, SFCS '81, IEEE, 1981.
[39] D. Dolev, A. Yao, On the security of public key protocols, IEEE Transactions on Information Theory 29 (2) (1983) 198–208. doi:10.1109/TIT.1983.1056650.
[40] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, The MIT Press, 1999.
[41] F. Kammüller, M. Wenzel, L. C. Paulson, Locales – a sectioning concept for Isabelle, in: Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, L. Thery (Eds.), Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Vol. 1690 of LNCS, Springer, 1999.
[42] D. Gollmann, Computer Security, Wiley, 2008.

Appendix A. Isabelle Code Extracts

This section contains a subset of the Isabelle formalization of the Insider framework and the airplane case study showing all relevant definitions, most interesting lemmas and

theorems without proofs (proofs are replaced by the tag ⟨*proof*⟩), and some proof examples. The following code has been abridged from the latex generated from the Isabelle sources available online [5]. In this repository there is also a directory `latex` that contains the latex-generated pdf outputs of the formalization in full (`document.pdf`, 61 pages) as well as the outline (`outline.pdf`, 25 pages).

Appendix A.1. Kripke Structures and CTL

theory *MC*
imports *Main*
begin

definition *monotone* :: ('*a set* ⇒ '*a set*) ⇒ *bool*
where *monotone* $\tau \equiv (\forall\ p\ q.\ p \subseteq q \longrightarrow \tau\ p \subseteq \tau\ q\ )$

lemma *monotoneE*: *monotone* $\tau \Longrightarrow p \subseteq q \Longrightarrow \tau\ p \subseteq \tau\ q$
⟨*proof*⟩

lemma *lfp1*: *monotone* $\tau \longrightarrow (lfp\ \tau = \bigcap\ \{Z.\ \tau\ Z \subseteq Z\})$
⟨*proof*⟩

lemma *gfp1*: *monotone* $\tau \longrightarrow (gfp\ \tau = \bigcup\ \{Z.\ Z \subseteq \tau\ Z\})$
⟨*proof*⟩

primrec *power* :: ['*a* ⇒ '*a*, *nat*] ⇒ ('*a* ⇒ '*a*) ((- ^ -) *40*)
where
power-zero: $(f\ \hat{}\ 0) = (\lambda\ x.\ x)$ |
power-suc: $(f\ \hat{}\ (Suc\ n)) = (f\ o\ (f\ \hat{}\ n))$

lemma *predtrans-empty*:
  assumes *monotone* τ
  shows $\forall\ i.\ (\tau\ \hat{}\ i)\ (\{\}) \subseteq (\tau\ \hat{}(i + 1))(\{\})$
proof (*rule allI*, *induct-tac i*)
  show $(\tau\ \hat{}\ 0{::}nat)\ \{\} \subseteq (\tau\ \hat{}\ (0{::}nat) + (1{::}nat))\ \{\}$ **by** *simp*
next show $\bigwedge(i{::}nat)\ n{::}nat.\ (\tau\ \hat{}\ n)\ \{\} \subseteq (\tau\ \hat{}\ n + (1{::}nat))\ \{\}$
    $\Longrightarrow (\tau\ \hat{}\ Suc\ n)\ \{\} \subseteq (\tau\ \hat{}\ Suc\ n + (1{::}nat))\ \{\}$
  proof −
    fix *i n*
    assume *a* : $(\tau\ \hat{}\ n)\ \{\} \subseteq (\tau\ \hat{}\ n + (1{::}nat))\ \{\}$
    have $(\tau\ ((\tau\ \hat{}\ n)\ \{\})) \subseteq (\tau\ ((\tau\ \hat{}\ (n + (1\ ::\ nat)))\ \{\}))$ **using** *assms*
      apply (*rule monotoneE*)
      by (*rule a*)
    thus $(\tau\ \hat{}\ Suc\ n)\ \{\} \subseteq (\tau\ \hat{}\ Suc\ n + (1{::}nat))\ \{\}$ **by** *simp*
  qed
qed

lemma *infchain-outruns-all*:
assumes *finite* (*UNIV* :: *'a set*)
and $\forall i :: nat.\ (\tau\ \hat{}\ i)\ (\{\}::'a\ set) \subset (\tau\ \hat{}\ i + (1 :: nat))\ \{\}$
shows $\forall j :: nat.\ \exists i :: nat.\ j < card\ ((\tau\ \hat{}\ i)\ \{\})$
⟨*proof*⟩

lemma *no-infinite-subset-chain*:
assumes *finite* (*UNIV* :: *'a set*)
and *monotone* (τ :: (*'a set* ⇒ *'a set*))
and $\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i)\ \{\} \subset (\tau\ \hat{}\ i + (1 :: nat))\ (\{\} :: 'a\ set)$
shows *False*
⟨*proof*⟩

lemma *finite-fixp*:
assumes *finite*(*UNIV* :: *'a set*)
and *monotone* (τ :: (*'a set* ⇒ *'a set*))
shows $\exists\ i.\ (\tau\ \hat{}\ i)\ (\{\}) = (\tau\ \hat{}(i + 1))(\{\})$
⟨*proof*⟩

lemma *predtrans-UNIV*:
assumes *monotone* τ
shows $\forall\ i.\ (\tau\ \hat{}\ i)\ (UNIV) \supseteq (\tau\ \hat{}(i + 1))(UNIV)$
⟨*proof*⟩

lemma *down-chain-reaches-empty*:
assumes *finite* (*UNIV* :: *'a set*) **and** *monotone* (τ :: *'a set* ⇒ *'a set*)
and $(\forall i :: nat.\ ((\tau :: 'a\ set \Rightarrow 'a\ set)\ \hat{}\ i + (1 :: nat))\ UNIV \subset (\tau\ \hat{}\ i)\ UNIV)$
shows $\exists\ (j :: nat).\ (\tau\ \hat{}\ j)\ UNIV = \{\}$
⟨*proof*⟩

lemma *lfp-loop*:
assumes *finite* (*UNIV* :: *'b set*) **and** *monotone* (τ :: (*'b set* ⇒ *'b set*))
shows $\exists\ n\ .\ lfp\ \tau\ = (\tau\ \hat{}\ n)\ \{\}$
⟨*proof*⟩

lemma *gfp-loop*:
assumes *finite* (*UNIV* :: *'b set*)
and *monotone* (τ :: (*'b set* ⇒ *'b set*))
shows $\exists\ n\ .\ gfp\ \tau\ = (\tau\ \hat{}\ n)(UNIV :: 'b\ set)$
⟨*proof*⟩

class *state* =
fixes *state-transition* :: [*'a* :: *type*, *'a*] ⇒ *bool* ((- →$_i$ -) 50)

definition *AX* **where** $AX\ f \equiv \{s.\ \{f0.\ s \rightarrow_i f0\} \subseteq f\}$

definition EX' **where** $EX'\ f \equiv \{s\ .\ \exists\ f0 \in f.\ s \rightarrow_i f0\ \}$

definition AF **where** $AF\ f \equiv lfp\ (\lambda\ Z.\ f \cup AX\ Z)$
definition EF **where** $EF\ f \equiv lfp\ (\lambda\ Z.\ f \cup EX'\ Z)$
definition AG **where** $AG\ f \equiv gfp\ (\lambda\ Z.\ f \cap AX\ Z)$
definition EG **where** $EG\ f \equiv gfp\ (\lambda\ Z.\ f \cap EX'\ Z)$
definition AU **where** $AU\ f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap AX\ Z))$
definition EU **where** $EU\ f1\ f2 \equiv lfp(\lambda\ Z.\ f2 \cup (f1 \cap EX'\ Z))$
definition AR **where** $AR\ f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup AX\ Z))$
definition ER **where** $ER\ f1\ f2 \equiv gfp(\lambda\ Z.\ f2 \cap (f1 \cup EX'\ Z))$

datatype $'a\ kripke = \quad Kripke\ 'a\ set\ 'a\ set$

primrec *states* **where** $states\ (Kripke\ S\ I) = S$
primrec *init* **where** $init\ (Kripke\ S\ I) = I$

definition *check* $(- \vdash -\ 50)$
where $M \vdash f \equiv (init\ M) \subseteq \{s \in (states\ M).\ s \in f\ \}$

definition *state-transition-refl* $((- \rightarrow_i* -)\ 50)$
where $s \rightarrow_i* s' \equiv ((s,s') \in \{(x,y).\ state\text{-}transition\ x\ y\}^*)$

lemma *EX-step*: **assumes** $x\ \rightarrow_i y$ **and** $y \in f$ **shows** $x \in EX'\ f$
⟨*proof*⟩

lemma *EF-step*: **assumes** $x\ \rightarrow_i y$ **and** $y \in f$ **shows** $x \in EF\ f$
⟨*proof*⟩

lemma *EF-step-step*: **assumes** $x\ \rightarrow_i y$ **and** $y \in EF\ f$ **shows** $x \in EF\ f$
⟨*proof*⟩

lemma *EF-step-star*: $[\![\ x\ \rightarrow_i* y;\ y \in f\ ]\!] \Longrightarrow x \in EF\ f$
⟨*proof*⟩

lemma *EF-induct*: $(a::'a::state) \in EF\ (f :: 'a :: state\ set) \Longrightarrow$
$mono\ \ (\lambda\ Z.\ (f::'a::state\ set) \cup EX'\ Z) \Longrightarrow$
$(\bigwedge x::'a::state.$
$x \in ((\lambda\ Z.\ (f::'a::state\ set) \cup EX'\ Z)(EF\ f \cap \{x::'a::state.\ (P::'a::state \Rightarrow bool)\ x\})) \Longrightarrow$
$P\ x) \Longrightarrow$
$P\ a$
⟨*proof*⟩

lemma *EF-step-star-rev*[*rule-format*]: $x \in EF\ s \Longrightarrow\ (\exists\ y \in s.\ \ x\ \rightarrow_i* y)$
⟨*proof*⟩

lemma *EF-step-inv*: $(I \subseteq \{sa::'s :: state.\ (\exists\, i::'s \in I.\ i \rightarrow_i * sa) \wedge sa \in EF\ s\})$
$\Longrightarrow \forall\ x \in I.\ \exists\ y \in s.\ x \rightarrow_i * y$
⟨*proof*⟩

lemma *AG-in-lem*: $x \in AG\ s \Longrightarrow x \in s$
⟨*proof*⟩

lemma *AG-step*: $y \rightarrow_i z \Longrightarrow y \in AG\ s \Longrightarrow z \in AG\ s$
⟨*proof*⟩

lemma *AG-all-s*: $x \rightarrow_i * y \Longrightarrow x \in AG\ s \Longrightarrow y \in AG\ s$
⟨*proof*⟩

lemma *AG-imp-notnotEF*:
$I \neq \{\} \Longrightarrow ((Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i * s)\}\ (I :: ('s :: state)set)\ \vdash AG\ s)) \Longrightarrow$
$(\neg(Kripke\ \{s :: ('s :: state).\ \exists\ i \in I.\ (i \rightarrow_i * s)\}\ (I :: ('s :: state)set)\ \vdash EF\ (-\ s)))$
⟨*proof*⟩

end

Appendix A.2. Insider Framework

theory *AirInsider*
imports *MC*
begin
datatype *action* = *get* | *move* | *eval* |*put*

typedecl *actor*
type-synonym *identity* = *string*
consts *Actor* :: *identity* ⇒ *actor*

type-synonym *policy* = ((*actor* ⇒ *bool*) ∗ *action set*)

datatype *location* = *Location nat*

datatype *igraph* = *Lgraph* (*location* ∗ *location*)*set location* ⇒ *identity list*
actor ⇒ (*string list* ∗ *string list*) *location* ⇒ *string list*
datatype *infrastructure* =
Infrastructure igraph
[*igraph*, *location*] ⇒ *policy set*

primrec *loc* :: *location* ⇒ *nat*
where *loc*(*Location n*) = *n*
primrec *gra* :: *igraph* ⇒ (*location* ∗ *location*)*set*
where *gra*(*Lgraph g a c l*) = *g*
primrec *agra* :: *igraph* ⇒ (*location* ⇒ *identity list*)

where *agra*(*Lgraph g a c l*) = *a*
primrec *cgra* :: *igraph* ⇒ (*actor* ⇒ *string list* ∗ *string list*)
where *cgra*(*Lgraph g a c l*) = *c*
primrec *lgra* :: *igraph* ⇒ (*location* ⇒ *string list*)
where *lgra*(*Lgraph g a c l*) = *l*

definition *nodes* :: *igraph* ⇒ *location set*
where *nodes g* == { *x*. (*? y*. ((*x*,*y*): *gra g*) | ((*y*,*x*): *gra g*))}

definition *actors-graph* :: *igraph* ⇒ *identity set*
where *actors-graph g* == {*x*. *? y*. *y* : *nodes g* ∧ *x* ∈ *set*(*agra g y*)}

primrec *graphI* :: *infrastructure* ⇒ *igraph*
where *graphI* (*Infrastructure g d*) = *g*
primrec *delta* :: [*infrastructure*, *igraph*, *location*] ⇒ *policy set*
where *delta* (*Infrastructure g d*) = *d*
primrec *tspace* :: [*infrastructure*, *actor*] ⇒ *string list* ∗ *string list*
where *tspace* (*Infrastructure g d*) = *cgra g*
primrec *lspace* :: [*infrastructure*, *location*] ⇒ *string list*
where *lspace* (*Infrastructure g d*) = *lgra g*

definition *credentials* :: *string list* ∗ *string list* ⇒ *string set*
where *credentials lxl* ≡ *set* (*fst lxl*)
definition *has* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *has G ac* ≡ *snd ac* ∈ *credentials*(*cgra G* (*fst ac*))
definition *roles* :: *string list* ∗ *string list* ⇒ *string set*
where *roles lxl* ≡ *set* (*snd lxl*)
definition *role* :: [*igraph*, *actor* ∗ *string*] ⇒ *bool*
where *role G ac* ≡ *snd ac* ∈ *roles*(*cgra G* (*fst ac*))
definition *isin* :: [*igraph*,*location*, *string*] ⇒ *bool*
where *isin G l s* ≡ *s* ∈ *set*(*lgra G l*)

datatype *psy-states* = *happy* | *depressed* | *disgruntled* | *angry* | *stressed*
datatype *motivations* = *financial* | *political* | *revenge* | *curious* | *competitive-advantage* | *power* | *peer-recognition*

datatype *actor-state* = *Actor-state psy-states motivations set*
primrec *motivation* :: *actor-state* ⇒ *motivations set*
where *motivation* (*Actor-state p m*) = *m*
primrec *psy-state* :: *actor-state* ⇒ *psy-states*
where *psy-state* (*Actor-state p m*) = *p*

definition *tipping-point* :: *actor-state* ⇒ *bool* **where**
tipping-point a ≡ ((*motivation a* ≠ {}) ∧ (*happy* ≠ *psy-state a*))

definition *UasI* :: [*identity*, *identity*] ⇒ *bool*
where *UasI a b* ≡ (*Actor a* = *Actor b*) ∧ (∀ *x y*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *Actor x* = *Actor y* ⟶ *x* = *y*)

definition *Insider* :: [*identity*, *identity set*, *identity* ⇒ *actor-state*] ⇒ *bool*
where *Insider a C as* ≡ (*tipping-point* (*as a*) ⟶ (∀ *b*∈*C*. *UasI a b*))

definition *atI* :: [*identity*, *igraph*, *location*] ⇒ *bool* (- @$_{(-)}$ - 50)
where *a* @$_G$ *l* ≡ *a* ∈ *set*(*agra G l*)

definition *enables* :: [*infrastructure*, *location*, *actor*, *action*] ⇒ *bool*
where
enables I l a a′ ≡ (∃ (*p*,*e*) ∈ *delta I* (*graphI I*) *l*. *a′* ∈ *e* ∧ *p a*)

primrec *nodup* :: ['*a*, '*a list*] ⇒ *bool*
where
nodup-nil: *nodup a* [] = *True* |
nodup-step: *nodup a* (*x* # *ls*) = (*if x* = *a then* (*a* ∉ (*set ls*)) *else nodup a ls*)

definition *move-graph-a* :: [*identity*, *location*, *location*, *igraph*] ⇒ *igraph*
where *move-graph-a n l l′ g* ≡ *Lgraph* (*gra g*)
(*if n* ∈ *set* ((*agra g*) *l*) & *n* ∉ *set* ((*agra g*) *l′*) *then*
((*agra g*)(*l* := *del n* (*agra g l*)))(*l′* := (*n* # (*agra g l′*)))
else (*agra g*))(*cgra g*)(*lgra g*)

inductive *state-transition-in* :: [*infrastructure*, *infrastructure*] ⇒ *bool* ((- →$_n$ -) 50)
where
move: ⟦ *G* = *graphI I*; *a* @$_G$ *l*; *l* ∈ *nodes G*; *l′* ∈ *nodes G*;
(*a*) ∈ *actors-graph*(*graphI I*); *enables I l′* (*Actor a*) *move*;
I′ = *Infrastructure* (*move-graph-a a l l′* (*graphI I*))(*delta I*) ⟧ ⟹ *I* →$_n$ *I′*
| *get* : ⟦ *G* = *graphI I*; *a* @$_G$ *l*; *a′* @$_G$ *l*; *has G* (*Actor a*, *z*);
enables I l (*Actor a*) *get*;
I′ = *Infrastructure*
(*Lgraph* (*gra G*)(*agra G*)
((*cgra G*)(*Actor a′* :=
(*z* # (*fst*(*cgra G* (*Actor a′*))), *snd*(*cgra G* (*Actor a′*)))))
(*lgra G*))
(*delta I*)
⟧ ⟹ *I* →$_n$ *I′*
| *put* : ⟦ *G* = *graphI I*; *a* @$_G$ *l*; *enables I l* (*Actor a*) *put*;
I′ = *Infrastructure*
(*Lgraph* (*gra G*)(*agra G*)(*cgra G*)
((*lgra G*)(*l* := [*z*])))
(*delta I*) ⟧
⟹ *I* →$_n$ *I′*

```
| put-remote : ⟦ G = graphI I; enables I l (Actor a) put;
        I' = Infrastructure
                (Lgraph (gra G)(agra G)(cgra G)
                        ((lgra G)(l := [z])))
                (delta I) ⟧
        ⟹ I →n I'


instantiation infrastructure :: state
begin
definition
   state-transition-infra-def: (i →i i') =  (i →n (i' :: infrastructure))

instance
  by (rule MC.class.MC.state.of-class.intro)

definition state-transition-in-refl ((- →n* -) 50)
where s →n* s' ≡ ((s,s') ∈ {(x,y). state-transition-in x y}*)

lemma move-graph-eq: move-graph-a a l l g = g
  by (simp add: move-graph-a-def, case-tac g, force)

lemma delta-invariant: ∀ z z'. z →n z' ⟶  delta(z) = delta(z')
  by (clarify, erule state-transition-in.cases, simp+)

lemma init-state-policy: ⟦ (x,y) ∈ {(x::infrastructure, y::infrastructure). x →n y}* ⟧ ⟹
                         delta(x) = delta(y)
proof −
  have ind: (x,y) ∈ {(x::infrastructure, y::infrastructure). x →n y}*
            ⟶ delta(x) = delta(y)
  proof (insert assms, erule rtrancl.induct)
    show (⋀ a::infrastructure.
       (∀ (z::infrastructure)(z'::infrastructure). (z →n z') ⟶ (delta z = delta z')) ⟹
       (((a, a) ∈ {(x ::infrastructure, y :: infrastructure). x →n y}*) ⟶
       (delta a = delta a)))
    by (rule impI, rule refl)
next fix a b c
  assume a0: ∀ (z::infrastructure) z'::infrastructure. z →n z' ⟶ delta z = delta z'
     and a1: (a, b) ∈ {(x::infrastructure, y::infrastructure). x →n y}*
     and a2: (a, b) ∈ {(x::infrastructure, y::infrastructure). x →n y}* ⟶
         delta a = delta b
     and a3: (b, c) ∈ {(x::infrastructure, y::infrastructure). x →n y}
     show (a, c) ∈ {(x::infrastructure, y::infrastructure). x →n y}* ⟶
       delta a = delta c
  proof −
```

have *a4*: *delta b* = *delta c* **using** *a0 a1 a2 a3* **by** *simp*
show *?thesis* **using** *a0 a1 a2 a3* **by** *simp*
qed
qed
show *?thesis*
by (*insert ind*, *insert assms*(*2*), *simp*)
qed

lemma *same-nodes*: (*I*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
⟹ *nodes*(*graphI y*) = *nodes*(*graphI I*)
⟨*proof*⟩

lemma *same-actors*: (*I*, *y*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
⟹ *actors-graph*(*graphI I*) = *actors-graph*(*graphI y*)
⟨*proof*⟩

end
end

Appendix A.3. Airplane

theory *Airplane*
imports *AirInsider*
begin
datatype *doorstate* = *locked* | *norm* | *unlocked*
datatype *position* = *air* | *airport* | *ground*

locale *airplane* =
fixes *airplane-actors* :: *identity set*
defines *airplane-actors-def*: *airplane-actors* ≡ {*''Bob''*, *''Charly''*, *''Alice''*}
fixes *airplane-locations* :: *location set*
defines *airplane-locations-def*:
airplane-locations ≡ {*Location 0*, *Location 1*, *Location 2*}
fixes *cockpit* :: *location*
defines *cockpit-def*: *cockpit* ≡ *Location 2*
fixes *door* :: *location*
defines *door-def*: *door* ≡ *Location 1*
fixes *cabin* :: *location*
defines *cabin-def*: *cabin* ≡ *Location 0*

fixes *global-policy* :: [*infrastructure*, *identity*] ⇒ *bool*
defines *global-policy-def*: *global-policy I a* ≡ *a* ∉ *airplane-actors*
⟶ ¬(*enables I cockpit* (*Actor a*) *put*)

fixes *ex-creds* :: *actor* ⇒ (*string list* ∗ *string list*)
defines *ex-creds-def*: *ex-creds* ≡

$(\lambda\ x.(\mathit{if}\ x = \mathit{Actor}\ ''Bob''$
$\quad \mathit{then}\ ([''PIN''],\ [''pilot''])$
$\quad \mathit{else}\ (\mathit{if}\ x = \mathit{Actor}\ ''Charly''$
$\quad\quad \mathit{then}\ ([''PIN''],[''copilot''])$
$\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{Actor}\ ''Alice''$
$\quad\quad\quad \mathit{then}\ ([''PIN''],[''flightattendant''])$
$\quad\quad\quad\quad \mathit{else}\ ([],[])))))$

fixes *ex-locs* :: *location* ⇒ *string list*
defines *ex-locs-def*: *ex-locs* ≡ $(\lambda\ x.\ \mathit{if}\ x = \mathit{door}\ \mathit{then}\ [''norm'']\ \mathit{else}$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''air'']\ \mathit{else}\ []))$

fixes *ex-locs′* :: *location* ⇒ *string list*
defines *ex-locs′-def*: *ex-locs′* ≡ $(\lambda\ x.\ \mathit{if}\ x = \mathit{door}\ \mathit{then}\ [''locked'']\ \mathit{else}$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''air'']\ \mathit{else}\ []))$
fixes *ex-graph* :: *igraph*
defines *ex-graph-def*: *ex-graph* ≡ *Lgraph*
$\quad \{(\mathit{cockpit},\ \mathit{door}),(\mathit{door},\mathit{cabin})\}$
$\quad (\lambda\ x.\ \mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''Bob'',\ ''Charly'']$
$\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{door}\ \mathit{then}\ []$
$\quad\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{cabin}\ \mathit{then}\ [''Alice'']\ \mathit{else}\ [])))$
$\quad \mathit{ex\text{-}creds}\ \mathit{ex\text{-}locs}$

fixes *aid-graph* :: *igraph*
defines *aid-graph-def*: *aid-graph* ≡ *Lgraph*
$\quad \{(\mathit{cockpit},\ \mathit{door}),(\mathit{door},\mathit{cabin})\}$
$\quad (\lambda\ x.\ \mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''Charly'']$
$\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{door}\ \mathit{then}\ []$
$\quad\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{cabin}\ \mathit{then}\ [''Bob'',\ ''Alice'']\ \mathit{else}\ [])))$
$\quad \mathit{ex\text{-}creds}\ \mathit{ex\text{-}locs′}$

fixes *aid-graph0* :: *igraph*
defines *aid-graph0-def*: *aid-graph0* ≡ *Lgraph*
$\quad \{(\mathit{cockpit},\ \mathit{door}),(\mathit{door},\mathit{cabin})\}$
$\quad (\lambda\ x.\ \mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''Charly'']$
$\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{door}\ \mathit{then}\ [''Bob'']$
$\quad\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{cabin}\ \mathit{then}\ [''Alice'']\ \mathit{else}\ [])))$
$\quad\ \mathit{ex\text{-}creds}\ \mathit{ex\text{-}locs}$

fixes *agid-graph* :: *igraph*
defines *agid-graph-def*: *agid-graph* ≡ *Lgraph*
$\quad \{(\mathit{cockpit},\ \mathit{door}),(\mathit{door},\mathit{cabin})\}$
$\quad (\lambda\ x.\ \mathit{if}\ x = \mathit{cockpit}\ \mathit{then}\ [''Charly'']$
$\quad\quad \mathit{else}\ (\mathit{if}\ x = \mathit{door}\ \mathit{then}\ []$

$$\textit{else } (\textit{if } x = \textit{cabin then } [''Bob'', ''Alice''] \textit{ else } []))$$
$$\textit{ex-creds ex-locs}$$

fixes $\textit{local-policies} :: [\textit{igraph}, \ \textit{location}] \Rightarrow \textit{policy set}$
defines $\textit{local-policies-def}: \textit{local-policies } G \equiv$
$$(\lambda\ y.\ \textit{if } y = \textit{cockpit then}$$
$$\{(\lambda\ x.\ (?\ n.\ (n\ @_G\ \textit{cockpit}) \wedge \textit{Actor } n = x), \{\textit{put}\}),$$
$$(\lambda\ x.\ (?\ n.\ (n\ @_G\ \textit{cabin}) \wedge \textit{Actor } n = x \wedge \textit{has } G\ (x, ''PIN'')$$
$$\wedge\ \textit{isin } G\ \textit{door } ''norm''),\{\textit{move}\})$$
$$\}$$
$$\textit{else } (\textit{if } y = \textit{door then } \{(\lambda\ x.\ \textit{True}, \{\textit{move}\}),$$
$$(\lambda\ x.\ (?\ n.\ (n\ @_G\ \textit{cockpit}) \wedge \textit{Actor } n = x), \{\textit{put}\})\}$$
$$\textit{else } (\textit{if } y = \textit{cabin then } \{(\lambda\ x.\ \textit{True}, \{\textit{move}\})\}$$
$$\textit{else } \{\})))$$

fixes $\textit{local-policies-four-eyes} :: [\textit{igraph}, \textit{location}] \Rightarrow \textit{policy set}$
defines $\textit{local-policies-four-eyes-def}: \textit{local-policies-four-eyes } G \equiv$
$$(\lambda\ y.\ \textit{if } y = \textit{cockpit then}$$
$$\{(\lambda\ x.\ \ (?\ n.\ (n\ @_G\ \textit{cockpit}) \wedge \textit{Actor } n = x) \wedge$$
$$2 \leq \textit{length}(\textit{agra } G\ y) \wedge (\forall\ h \in \textit{set}(\textit{agra } G\ y).\ h \in \textit{airplane-actors}), \{\textit{put}\}),$$
$$(\lambda\ x.\ (?\ n.\ (n\ @_G\ \textit{cabin}) \wedge \textit{Actor } n = x \wedge \textit{has } G\ (x, ''PIN'') \wedge$$
$$\textit{isin } G\ \textit{door } ''norm''\ ),\{\textit{move}\})$$
$$\}$$
$$\textit{else } (\textit{if } y = \textit{door then}$$
$$\{(\lambda\ x.\ \ ((?\ n.\ (n\ @_G\ \textit{cockpit}) \wedge \textit{Actor } n = x) \wedge 3 \leq \textit{length}(\textit{agra } G\ \textit{cockpit})), \{\textit{move}\})\}$$
$$\textit{else } (\textit{if } y = \textit{cabin then}$$
$$\{(\lambda\ x.\ ((?\ n.\ (n\ @_G\ \textit{door}) \wedge \textit{Actor } n = x)), \{\textit{move}\})\}$$
$$\textit{else } \{\})))$$

fixes $\textit{Airplane-scenario} :: \textit{infrastructure}$ (**structure**)
defines $\textit{Airplane-scenario-def}:$
$$\textit{Airplane-scenario} \equiv \textit{Infrastructure ex-graph local-policies}$$

fixes $\textit{Airplane-in-danger} :: \textit{infrastructure}$
defines $\textit{Airplane-in-danger-def}:$
$$\textit{Airplane-in-danger} \equiv \textit{Infrastructure aid-graph local-policies}$$

fixes $\textit{Airplane-getting-in-danger0} :: \textit{infrastructure}$
defines $\textit{Airplane-getting-in-danger0-def}:$
$$\textit{Airplane-getting-in-danger0} \equiv \textit{Infrastructure aid-graph0 local-policies}$$

fixes $\textit{Airplane-getting-in-danger} :: \textit{infrastructure}$
defines $\textit{Airplane-getting-in-danger-def}:$
$$\textit{Airplane-getting-in-danger} \equiv \textit{Infrastructure agid-graph local-policies}$$

fixes *Air-states*
defines *Air-states-def*: *Air-states* $\equiv \{\ I.\ Airplane\text{-}scenario \rightarrow_n* I\ \}$

fixes *Air-Kripke*
defines *Air-Kripke* $\equiv$ *Kripke Air-states* $\{Airplane\text{-}scenario\}$

fixes *Airplane-not-in-danger* :: *infrastructure*
defines *Airplane-not-in-danger-def*:
Airplane-not-in-danger $\equiv$ *Infrastructure aid-graph local-policies-four-eyes*

fixes *Airplane-not-in-danger-init* :: *infrastructure*
defines *Airplane-not-in-danger-init-def*:
Airplane-not-in-danger-init $\equiv$ *Infrastructure ex-graph local-policies-four-eyes*

fixes *Air-tp-states*
defines *Air-tp-states-def*: *Air-tp-states* $\equiv \{\ I.\ Airplane\text{-}not\text{-}in\text{-}danger\text{-}init \rightarrow_n* I\ \}$

fixes *Air-tp-Kripke*
defines *Air-tp-Kripke* $\equiv$ *Kripke Air-tp-states* $\{Airplane\text{-}not\text{-}in\text{-}danger\text{-}init\}$

fixes *Safety* :: $[infrastructure, identity] \Rightarrow bool$
defines *Safety-def*: *Safety I a* $\equiv$ *a* $\in$ *airplane-actors*
$\longrightarrow$ (*enables I cockpit* (*Actor a*) *move*)

fixes *Security* :: $[infrastructure, identity] \Rightarrow bool$
defines *Security-def*: *Security I a* $\equiv$ (*isin* (*graphI I*) *door* ''*locked*'')
$\longrightarrow \neg$(*enables I cockpit* (*Actor a*) *move*)

fixes *foe-control* :: $[location, action, infrastructure\ kripke] \Rightarrow bool$
defines *foe-control-def*: *foe-control l c K* $\equiv$
($\forall$ *I*:: *infrastructure* $\in$ *states K*. ($\exists$ *x* :: *identity*.
(*x* $@_{graphI\ I}$ *l*) $\wedge$ *Actor x* $\neq$ *Actor* ''*Eve*'')
$\longrightarrow \neg$(*enables I l* (*Actor* ''*Eve*'') *c*))

fixes *astate*:: *identity* $\Rightarrow$ *actor-state*
defines *astate-def*: *astate x* $\equiv$ (*case x of*
''*Eve*'' $\Rightarrow$ *Actor-state depressed* $\{revenge, peer\text{-}recognition\}$
| - $\Rightarrow$ *Actor-state happy* $\{\}$)

assumes *Eve-precipitating-event*: *tipping-point* (*astate* ''*Eve*'')
assumes *Insider-Eve*: *Insider* ''*Eve*'' $\{$''*Charly*''$\}$ *astate*
assumes *cockpit-foe-control*: *foe-control cockpit put Air-tp-Kripke*

begin

lemma *Safety*: *Safety Airplane-scenario* (*''Alice''*)
⟨*proof*⟩

lemma *Security*: *Security Airplane-scenario s*
⟨*proof*⟩

lemma *step0r*: *Airplane-scenario* $\rightarrow_n*$ *Airplane-getting-in-danger0*
⟨*proof*⟩

lemma *step1r*: *Airplane-getting-in-danger0* $\rightarrow_n*$ *Airplane-getting-in-danger*
⟨*proof*⟩

lemma *step2r*: *Airplane-getting-in-danger* $\rightarrow_n*$ *Airplane-in-danger*
⟨*proof*⟩

theorem *step-allr*: *Airplane-scenario* $\rightarrow_n*$ *Airplane-in-danger*
⟨*proof*⟩

theorem *aid-attack*: *Air-Kripke* $\vdash$ *EF* ({x. $\neg$ *global-policy x ''Eve''*})
proof (*simp add*: *check-def Air-Kripke-def*, *rule conjI*)
show *Airplane-scenario* $\in$ *Air-states*
by (*simp add*: *Air-states-def state-transition-in-refl-def*)
next show *Airplane-scenario* $\in$ *EF* {*x*::*infrastructure*. $\neg$ *global-policy x ''Eve''*}
by (*rule EF-lem2b*, *subst EF-lem000*, *rule EX-lem0r*, *subst EF-lem000*, *rule EX-step*,
unfold state-transition-infra-def, *rule step0*, *rule EX-lem0r*,
rule-tac y = *Airplane-getting-in-danger* **in** *EX-step*,
unfold state-transition-infra-def, *rule step1*, *subst EF-lem000*, *rule EX-lem0l*,
rule-tac y = *Airplane-in-danger* **in** *EX-step*, *unfold state-transition-infra-def*,
rule step2, *rule CollectI*, *rule ex-inv4*)
qed

lemma *actors-unique-loc-base*:
assumes $I \rightarrow_n I'$
and ($\forall$ l l'. a @$_{graphI\ I}$ $l \wedge a$ @$_{graphI\ I}$ $l' \longrightarrow l = l') \wedge$
($\forall$ l. *nodup a* (*agra* (*graphI I*) l))
shows ($\forall$ l l'. a @$_{graphI\ I'}$ $l \wedge a$ @$_{graphI\ I'}$ $l' \longrightarrow l = l') \wedge$
($\forall$ l. *nodup a* (*agra* (*graphI I'*) l))
⟨*proof*⟩

lemma *actors-unique-loc-step*:
assumes (I, I') $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
and $\forall$ a. ($\forall$ l l'. a @$_{graphI\ I}$ $l \wedge a$ @$_{graphI\ I}$ $l' \longrightarrow l = l') \wedge$
($\forall$ l. *nodup a* (*agra* (*graphI I*) l))
shows $\forall$ a. ($\forall$ l l'. a @$_{graphI\ I'}$ $l \wedge a$ @$_{graphI\ I'}$ $l' \longrightarrow l = l') \wedge$
($\forall$ l. *nodup a* (*agra* (*graphI I'*) l))

⟨*proof*⟩

lemma *two-person-inv*:
$z \rightarrow_n z'$
$\Longrightarrow$ (*2*::*nat*) $\leq$ *length* (*agra* (*graphI z*) *cockpit*)
$\Longrightarrow$ *nodes*(*graphI z*) = *nodes*(*graphI Airplane-not-in-danger-init*)
$\Longrightarrow$ *delta*(*Airplane-not-in-danger-init*) = *delta z*
$\Longrightarrow$ (*Airplane-not-in-danger-init*,*z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
$\Longrightarrow$ (*2*::*nat*) $\leq$ *length* (*agra* (*graphI z'*) *cockpit*)
⟨*proof*⟩

lemma *airplane-actors-inv*:
assumes (*Airplane-not-in-danger-init*,*z*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
shows $\forall$ *h*::*char list*$\in$*set* (*agra* (*graphI z*) *cockpit*). *h* $\in$ *airplane-actors*
⟨*proof*⟩

lemma *Eve-not-in-cockpit*: (*Airplane-not-in-danger-init*, *I*)
$\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}* $\Longrightarrow$
x $\in$ *set* (*agra* (*graphI I*) *cockpit*) $\Longrightarrow$ $x \neq$ *''Eve''*
⟨*proof*⟩

lemma *tp-imp-control*:
assumes (*Airplane-not-in-danger-init*,*I*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}*
shows (*? x* :: *identity*. x @$_{graphI\ I}$ *cockpit* $\wedge$ *Actor x* $\neq$ *Actor ''Eve''*)
⟨*proof*⟩

lemma *Fend-2*: (*Airplane-not-in-danger-init*,*I*) $\in$ {(*x*::*infrastructure*, *y*::*infrastructure*). $x \rightarrow_n y$}* $\Longrightarrow$
$\neg$ *enables I cockpit* (*Actor ''Eve''*) *put*
by (*insert cockpit-foe-control*, *simp add*: *foe-control-def*, *drule-tac x* = *I* **in** *spec*,
erule mp, *erule tp-imp-control*)

theorem *Four-eyes-no-danger*: *Air-tp-Kripke* $\vdash$ *AG* ({*x*. *global-policy x ''Eve''*})
proof (*simp add*: *Air-tp-Kripke-def check-def*, *rule conjI*)
show *Airplane-not-in-danger-init* $\in$ *Air-tp-states*
by (*simp add*: *Airplane-not-in-danger-init-def Air-tp-states-def*
state-transition-in-refl-def)
next show *Airplane-not-in-danger-init* $\in$ *AG* {*x*::*infrastructure*. *global-policy x ''Eve''*}
proof (*unfold AG-def*, *simp add*: *gfp-def*,
rule-tac x = {(*x* :: *infrastructure*) $\in$ *states Air-tp-Kripke*. $\sim$(*''Eve''* @$_{graphI\ x}$ *cockpit*)} **in** *exI*,
rule conjI)
show {*x*::*infrastructure* $\in$ *states Air-tp-Kripke*. $\neg$ *''Eve''* @$_{graphI\ x}$ *cockpit*}
$\subseteq$ {*x*::*infrastructure*. *global-policy x ''Eve''*}
by (*unfold global-policy-def*, *simp add*: *airplane-actors-def*, *rule subsetI*,
drule CollectD, *rule CollectI*, *erule conjE*,

simp add: *Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def*,
erule Fend-2)
next show {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*}
⊆ *AX* {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*} ∧
Airplane-not-in-danger-init
∈ {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*}
proof
show *Airplane-not-in-danger-init*
∈ {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*}
by (*simp add*: *Airplane-not-in-danger-init-def Air-tp-Kripke-def Air-tp-states-def*
state-transition-refl-def ex-graph-def atI-def Air-tp-Kripke-def
state-transition-in-refl-def)
next show {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*}
⊆ *AX* {*x*::*infrastructure* ∈ *states Air-tp-Kripke*. ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*}
proof (*rule subsetI*, *simp add*: *AX-def*, *rule subsetI*, *rule CollectI*, *rule conjI*)
show ⋀(*x*::*infrastructure*) *xa*::*infrastructure*.
x ∈ *states Air-tp-Kripke* ∧ ¬ *''Eve''* @$_{graphI\ x}$ *cockpit* ⟹
xa ∈ *Collect* (*state-transition x*) ⟹ *xa* ∈ *states Air-tp-Kripke*
by (*simp add*: *Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def*,
simp add: *atI-def*, *erule conjE*,
unfold state-transition-infra-def state-transition-in-refl-def,
erule rtrancl-into-rtrancl, *rule CollectI*, *simp*)
next fix *x xa*
assume *a0*: *x* ∈ *states Air-tp-Kripke* ∧ ¬ *''Eve''* @$_{graphI\ x}$ *cockpit*
and *a1*: *xa* ∈ *Collect* (*state-transition x*)
show ¬ *''Eve''* @$_{graphI\ xa}$ *cockpit*
proof −
have *b*: (*Airplane-not-in-danger-init*, *xa*)
∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
proof (*insert a0 a1*, *rule rtrancl-trans*)
show *x* ∈ *states Air-tp-Kripke* ∧ ¬ *''Eve''* @$_{graphI\ x}$ *cockpit* ⟹
xa ∈ *Collect* (*state-transition x*) ⟹
(*x*, *xa*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
by (*unfold state-transition-infra-def*, *force*)
next show *x* ∈ *states Air-tp-Kripke* ∧ ¬ *''Eve''* @$_{graphI\ x}$ *cockpit* ⟹
xa ∈ *Collect* (*state-transition x*) ⟹
(*Airplane-not-in-danger-init*, *x*) ∈ {(*x*::*infrastructure*, *y*::*infrastructure*). *x* →$_n$ *y*}*
by (*erule conjE*, *simp add*: *Air-tp-Kripke-def Air-tp-states-def state-transition-in-refl-def*)+
qed
show *?thesis*
by (*insert a0 a1 b*, *rule-tac P* = *''Eve''* @$_{graphI\ xa}$ *cockpit* **in** *notI*,
simp add: *atI-def*, *drule Eve-not-in-cockpit*, *assumption*, *simp*)
qed
qed
qed

qed
qed

lemma *Gen-Eve-not-in-cockpit*: $(I0,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$(\forall\ z.\ (I0\ ,z) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \longrightarrow$
$(\forall\ h::identity \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors)) \Longrightarrow$
$x \in set\ (agra\ (graphI\ I)\ cockpit) \Longrightarrow x \neq\ ''Eve''$
⟨*proof*⟩

lemma *Gen-Fend*: *foe-control cockpit put* $(Kripke\ \{\ I.\ I0 \rightarrow_n* I\ \}\ \{I0\}) \Longrightarrow$
$(I0,\ z) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \Longrightarrow$
$(\forall\ I.\ (I0,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\longrightarrow\ 2 \leq card\ (set\ (agra\ (graphI\ I)\ cockpit))) \Longrightarrow$
$(\forall\ z.\ (I0\ ,z) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \longrightarrow$
$(\forall\ h::identity \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors)) \Longrightarrow$
$\neg\ enables\ z\ cockpit\ (Actor\ ''Eve'')\ put$
⟨*proof*⟩

theorem *Gen-policy*:
foe-control cockpit put $(Kripke\ \{\ I.\ I0 \rightarrow_n* I\ \}\ \{I0\}) \Longrightarrow$
$(\forall\ I.\ (I0,\ I) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^*$
$\longrightarrow\ 2 \leq card\ (set\ (agra\ (graphI\ I)\ cockpit))) \Longrightarrow$
$(\forall\ z.\ (I0\ ,z) \in \{(x::infrastructure,\ y::infrastructure).\ x \rightarrow_n y\}^* \longrightarrow$
$(\forall\ h::identity \in set\ (agra\ (graphI\ z)\ cockpit).\ h \in airplane\text{-}actors)) \Longrightarrow$
$Kripke\ \{\ I.\ I0 \rightarrow_n* I\ \}\ \{I0\} \vdash AG\ \{x.\ global\text{-}policy\ x\ ''Eve''\}$
⟨*proof*⟩

theorem *Four-eyes-no-danger′*: $Air\text{-}tp\text{-}Kripke \vdash AG\ (\{x.\ global\text{-}policy\ x\ ''Eve''\})$
⟨*proof*⟩

end

interpretation *airplane airplane-actors airplane-locations cockpit door cabin global-policy*
ex-creds ex-locs ex-locs′ ex-graph aid-graph aid-graph0 agid-graph
local-policies local-policies-four-eyes Airplane-scenario Airplane-in-danger
Airplane-getting-in-danger0 Airplane-getting-in-danger Air-states Air-Kripke
Airplane-not-in-danger Airplane-not-in-danger-init Air-tp-states
Air-tp-Kripke Safety Security foe-control astate
⟨*proof*⟩

end