Analyzing security requirements formally and flexibly based on suspicion

Nuno Amário

Abstract—Increasingly, engineers need to approach security and software engineering in a unified way. This paper presents an approach to the formal analysis of security requirements that is based on model-checking and uses the concept of suspicion to guide the search for threats and security vulnerabilities in requirements. It proposes an approach to security analysis that favours exploration of a system’s state space based on what is abnormal or suspicious to find threats and vulnerabilities, instead of ironclad security proofs that try to demonstrate that a system is secure; as this paper shows, such security proofs can often be misleading. The approach is tested and illustrated by conducting two experiments: one focussing on a system with a confidentiality security property, and another with an integrity security property enforced through the separation of duty principle. One of the advantages of the approach presented here is that threats are derived directly from a model of requirements and no prior knowledge about possible attacks is necessary to perform the analysis. The paper shows that suspicion is an effective search criteria for finding vulnerabilities and security threats in requirements, and that the feedback generated by the analysis helps in elaborating security requirements.

Index Terms—Security, requirements, formal analysis, Event-Calculus, planning, confidentiality, separation of duty.

I. INTRODUCTION

Traditional approaches to software engineering and current practice tend to treat security concerns as an after-thought [1]. Security requirements are handled as non-functional requirements and are kept separate from their functional counter-parts until design or implementation-time. This raises problems for the whole software development process because (as demonstrated in this paper) functionality has an impact on security. If security aspects are not treated properly at the requirements phase, then the resolution of the problem will inevitably be deferred but at a much higher cost, which is a well known software engineering problem [2]. This issue can be resolved by integrating security into the requirements engineering phase of the software life-cycle [1], [3]. However, the best way to capture, model and analyse security and system requirements in a unified way is still an open problem.

Security requirements have proved tricky to formulate and reason about [4]. There are several methods for reasoning about properties that a system must satisfy. Traditionally, properties are classified as either safety or liveness [5], [6]. Safety properties say that something bad must not happen, and liveness properties say that something good must eventually happen. We check safety to ensure that bad states are not reachable, and liveness to ensure that good states are eventually reachable. This is used to check that invariants are preserved, that operations are applicable when certain conditions are met (pre-conditions) and that operations have the desired effect taking the system into a valid state. However, important classes of security requirements are either difficult to express using traditional safety or liveness properties, or they are just not possible to express at all [7], [4], [8].

This paper claims that, from a practical point of view, not only it is important to verify safety (that some unsecure state is not reached) and liveness (that the security measures do what is expected from them), but also in finding security vulnerabilities and possible security threats that give attack opportunities to malicious users: we need to look for what can happen under certain suspicious conditions. Traditionally, such possibilistic properties [7] are hard to formulate and reason about. This paper proposes a practical approach for dealing with such properties.

This paper presents a practical approach to the formal analysis of security requirements based on model-checking, where the search for threats and vulnerabilities in requirements is based on what is suspicious from a security point of view. This is inspired by anomaly-based approaches to intrusion detection [9], where the search for intrusions at run-time is driven by abnormal (or suspicious) behaviour of system use. The approach presented here takes a formal model of requirements and an analysis goal (a description of suspicious states from the analysis point of view), which are used by the model checker to generate traces of events describing how the analysis goals is reached. Each trace gives a scenario illustrating a possible threat or security vulnerability. The generation of plans is done automatically with tool support. The approach is illustrated with the Event-Calculus temporal logic [10] and the analysis is conducted with tool support using the discrete event calculus reasoner1 (decreasoner).

The remainder of this paper starts by giving a brief introduction to event calculus (section II). Then it presents the approach to the formal analysis of security requirements that is proposed here (section III). After this, the analysis approach is used to conduct two experiments: analysis of a simple health-care system with a confidentiality security requirement (section IV), and analysis of a business system with an integrity requirement enforcing separation of duty (section V). Then, the paper summarises the experimental results that we obtained (section VI), discusses the paper’s results (section VII), presents some related work (section VIII), and takes the conclusions.

N. Amário is with the University of Luxembourg.

1http://decreasoner.sourceforge.net/
Fig. 1. Planning-based formal analysis. EC model of requirements, and EC description of analysis goal are fed into decreasoner tool to obtain a set of plans (scenarios) that achieve the goal.

II. THE EVENT CALCULUS

Event Calculus (EC) [10], [11], a temporal logic based on first-order predicate calculus designed for common-sense reasoning, enables representation and reasoning about action and change. Its basic ontology comprises events, fluents and timepoints. An event is an action that may occur in the world. A fluent is a time varying property of the world. A timepoint is an instance of time. EC includes a set of basic predicates to describe happening of events, their effects and state of fluents. An EC model is built by describing two types of facts: the fact that an event occurs at a timepoint, and the fact that a property holds at a timepoint [11]. These facts are either true or false.

The basic predicates of EC are as follows:

- HoldsAt \((f, t)\) says that fluent \(f\) is true at timepoint \(t\).
- Happens \((e, t)\) says that event \(e\) may occur at timepoint \(t\).
- Initiates \((e, f, t)\) says that if event \(e\) occurs at timepoint \(t\), then fluent \(f\) is true after \(t\).
- Terminates \((e, f, t)\) says that if event \(e\) occurs at timepoint \(t\), then fluent \(f\) is false after \(t\).
- Initially \((f)\) says that fluent \(f\) holds at timepoint 0.

III. FORMAL THREAT ANALYSIS BY STUDYING REACHABILITY

The analysis proposed here is essentially a study of reachability: it checks whether certain states are reachable from a model of the requirements. The actual generation of threat scenarios is based on AI planning [12] and uses the decreasoner tool, which is based on a SAT approach to EC reasoning [13]. This is related to what is known in software engineering as model-checking [14]: the exploration of all possible states and transitions of a model to determine if a certain property holds or not.

Figure 1 depicts the analysis approach followed here. The EC model and EC description of analysis goals are given as inputs to decreasoner, which generates a set of plans (or traces) that satisfy the goal. Each plan describes a scenario comprising a sequence of events (a trace) that takes the system from the initial state to one of the states described by the goal.

The goal (a predicate) describes a set of states that are interesting from the analysis point of view. Planning generates plans that reach such a state. If there are plans, then the goal is satisfiable: a state as described by the goal can be reached in the model. If no plans can be found, then no state as described by the goal is reachable. If the goal state describes something that should not happen, then the resulting plans (scenarios)

A. EC Model

To satisfy the requirements of SMIS, EC model presented here introduces a protection mechanism based on credentials: doctors need to request a credential prior to accessing the data; credentials have a validity period.

The building blocks of an EC model are sorts, events and fluents. EC model of SMIS comprises sort Delay, representing
time delays used in credential mechanism, and domain sorts User, representing a user of SMIS, Doctor, a sub-sort of User that represents doctors using SMIS, and Patient, which represents the patients recorded in the system.

EC model of SMIS comprises the following events:

- **AuthoriseAccess** \((d, p)\): occurs when doctors \((d)\) request a credential for accessing some patient’s data \((p)\).
- **GetMD** \((d, p)\): occurs when doctors \((d)\) actually access patient’s medical data \((p)\).
- **SetSubstituteDoctor** \((u, d_1, d_2)\): occurs when a user \((u)\) sets some doctor \((d_1)\) as substitute of another \((d_2)\).
- **SetDoctorOnLeave** \((u, d)\): occurs when a user \((u)\) informs system that some doctor \((d)\) is on leave.
- **DoctorNo LongerOnLeave** \((u, d)\): occurs when a user \((u)\) informs system that a doctor \((d)\) is no longer on leave.

Next EC equation says that a doctor \((d)\) as substitute of another \((d_2)\).

Next EC equation describes how doctors get credentials to access patient’s data by describing effect of event **AuthoriseAccess**:

\[
\forall d : \text{Doctor}; p : \text{Patient}; t : \text{Time} | \\
\text{HoldsAt (CanAccessMD \((d, p)\), t)} \\
\rightarrow \text{Initiates (AuthoriseAccess \((d, p)\), CredentialMD \((d, p, t)\))}
\] (5)

This says that some doctor gets a credential to access a medical file \((\text{fluent CredentialMD} \text{ is initiated})\) upon event **AuthoriseAccess**, provided doctor can access patient’s data \((\text{predicate CanAccessMD defined above})\). This formalises the scheme of credential-based protection.

Model objects of Delay sort have a duration, as defined by Duration predicate defined above. Next EC equation defines delay credentialValidity representing validity period of a credential:

\[
\text{credentialValidity} : \text{Delay}
\] (6)

Next EC equation defines validity conditions of credentials, which are captured by predicate **HasValidCredential**:

\[
\forall d : \text{Doctor}; p : \text{Patient}; t : \text{Time} | \\
\text{HoldsAt (HasValidCredential \((d, p)\), t)} \\
\leftrightarrow (\exists t_2, t_3 : \text{Time}) \\
\text{HoldsAt (CredentialMD \((d, p, t_2)\), t)} \\
\land \text{Duration (credentialValidity, t_3)} \\
\land (t_2 + t_3) \geq t
\] (7)

This says that a credential is valid for the duration period defined by credentialValidity.

Next EC equation defines precondition of event **GetMD**, which may happen provided requesting doctor has a valid credential to access requested patient’s data:

\[
\forall d : \text{Doctor}; p : \text{Patient}; t : \text{Time} | \\
\text{Happens (GetMD \((d, p, t)\))} \\
\rightarrow \text{HoldsAt (HasValidCredential \((d, p, t)\))}
\] (8)

Next EC equation describes how event **GetMD** initiates \((\text{sets to true})\) fluent **ExposedToAt** that records exposure to patient’s data has been accessed:

\[
\forall d : \text{Doctor}; p : \text{Patient}; t : \text{Time} | \\
\text{Initiates (GetMD \((d, p, t)\), ExposedToAt \((d, p, t)\))}
\] (9)

Next equation constrains fluent **IsSubstituteDoctor** to be non-reflexive; that is, a doctor may not be set as a substitute of himself:

\[
\forall d : \text{Doctor}; t : \text{Time} | \\
\neg \text{HoldsAt (IsSubstituteDoctor \((d, d, t)\))}
\] (10)

Next equation describes effect how event **SetSubstituteDoctor** initiates fluent **IsSubstituteDoctor** to record that some doctor is substitute of another:

\[
\forall u : \text{User}; d_1, d_2 : \text{Doctor}; t : \text{Time} | \\
\text{Initiates (SetSubstituteDoctor \((u, d_1, d_2)\), IsSubstituteDoctor \((d_2, d_1, t)\))}
\] (11)

Next equation describes how event **SetDoctorOnLeave** initiates \((\text{sets to true})\) fluent **OnLeave**; some user \((u)\) informs system that some doctor \((d)\) is on-leave:

\[
\forall u : \text{User}; d : \text{Doctor}; t : \text{Time} | \\
\text{Initiates (SetDoctorOnLeave \((u, d)\), OnLeave \((d, t)\))}
\] (12)
Next equation describes how event DoctorNo Longer On Leave terminates (sets to false) fluent DoctorNo Longer On Leave; this used so that users \( u \) inform system that a doctor \( (d) \) is no longer on leave:
\[
\forall u: \text{User}, d: \text{Doctor}; t: \text{Time} \mid \text{DoctorNo Longer On Leave} (u, d), \text{On Leave} (d, t) \]

Next equations describe the initial condition of SMIS:
\[
\forall d: \text{Doctor}; p: \text{Patient}; t: \text{Time} \mid \\
\text{Initially} (\neg \text{ExposedToAt} (d, p, t)) \\
\forall d_1, d_2: \text{Doctor} \mid \\
\text{Initially} (\neg \text{IsSubstituteDoctor} (d_1, d_2)) \\
\forall d: \text{Doctor} \mid \text{Initially} (\neg \text{On Leave} (d))
\]

Above, initially no one has been exposed to medical data (fluent ExposedToAt), no doctors are set as substitutes, and there are no doctors on leave.

This completes EC model of SMIS’s requirements (table I). Next section, formally analyses this model.

### B. Model Analysis

Analysis uses configuration depicted in Fig. 2. There are two doctors, Jones and Smith, and a patient of Jones, Anderson. This is formulated in EC as:
\[
jones, smith: \text{Doctor} \\
anderson: \text{Patient} \\
\text{Initially} (\text{IsDoctorOf} (jones, anderson))
\]

Configuration for model analysis also needs to define duration of credentialValidity delay, which is defined as taking three time-points:
\[
\text{Duration} (\text{credentialValidity}, 3)
\]

Analysis starts by formulating a security violation goal: it asks whether it is possible to reach a state where patient confidentiality is compromised. In the context of SMIS, this happens when some doctor accesses some patient’s data without a valid security credential; the goal expressing this is formulated as:
\[
\exists d: \text{Doctor}; p: \text{Patient}; t_1, t_2: \text{Time} \mid \\
\text{HoldsAt} (\text{ExposedToAt} (d, p, t_2), t_1) \\
\wedge \neg \text{HoldsAt} (\text{HasValidCredential} (d, p), t_2)
\]

For this goal, decreasoner does not find any traces (a fragment of decreasoner’s output for this goal is given appendix A1). This means that the modelled system cannot reach one of the goal’s states. At this point, one could argue that the modelled system is secure because it is not possible to reach an insecure state, but it isn’t so.

Analysis proceeds by applying suspicion. The substitute doctor rule (Requirement R2) enables access to patient’s data by doctors other than the main patient’s doctor. This should occur, but not very often; the situations under which this occurs are suspicious and deserve investigation. Analysis investigates states where those other than the main doctor access the patient’s data. This is formulated as the goal:
\[
\exists d: \text{Doctor}; p: \text{Patient}; t_1, t_2: \text{Time} \mid \\
\text{HoldsAt} (\text{ExposedToAt} (d, p, t_2), t_1) \\
\wedge \neg \text{HoldsAt} (\text{IsDoctorOf} (d, p), t_2)
\]

For this goal, decreasoner generates traces exposing a security vulnerability, which enables doctors to access patient’s data in non-legal ways (see appendix A2 for output generated by decreasoner). In some scenarios, the system behaves as intended: Jones sets Smith as his substitute, at a later time Jones informs system that he is on leave, and so Smith is able to access the medical data (model 5 in appendix A2). Other scenarios are more unusual. In some of them, it is possible that Smith himself requests to be the substitute of Jones (model 1 in appendix A2), and that it is Smith who informs the system that Jones is on leave (model 1 in appendix A2). Obviously this is strange and could be explored by a malicious doctor determined to get some patient’s medical data: (a) he sets himself as substitute of another doctor, then (b) he informs the system that main doctor is on-leave and (c) finally he is able to access the patient’s data.

Such scenarios are possible because events IsSubstituteDoctor and SetDoctorOnLeave (equations 10 and 11) are unconstrained: any user may execute them, which introduces a loophole providing an opportunity for access to patient’s data in ways that are not intended.

We can confirm the vulnerability by posing an analysis question. We want to know if it is possible that some user can nominate himself as some other doctor’s substitute and then to be able to access the data of a patient that is not his own. This is formulated as:
\[
\exists d_1, d_2: \text{Doctor}; p: \text{Patient}; t_1, t_2, t_3: \text{Time} \mid \\
\text{Happens} (\text{SetSubstituteDoctor} (d_1, d_2, d_3), t_2) \\
\text{HoldsAt} (\text{ExposedToAt} (d_2, p, t_3), t_1) \\
\wedge \neg \text{HoldsAt} (\text{IsDoctorOf} (d_1, p, t_3)) \\
\wedge t_2 \leq t_3
\]

For this goal, decreasoner is able to identify many scenarios, thus confirming the identified vulnerability.

### C. Fixing the Model

The analysis’ findings are used to elaborate the requirements. We try to remove the vulnerability that has been identified. The requirements that emerge as a result of this elaboration are given in table II; here, \( R1 \) and \( R2 \) of table I still hold, and there is new requirement \( R3 \), which introduces users of type administrators that execute administration tasks on behalf of doctors.

This new requirement needs to be reflected in the EC model. New version of EC model introduces sort \( \text{Admin} \), subsort of \( \text{User} \) sort, and which is disjoint from \( \text{Doctor} \) sort. It also introduces a new fluent:

- \( \text{HasAdmin} (d, a) \) indicates administrator user \( a \) doing administrative tasks on behalf of some doctor \( d \).

This new sort and fluent are used to describe the new requirement. Next EC equation constrains \( \text{HasAdmin} \) to be a total relation; each doctor must have at least one administrator:
There are two types of users: clerks and managers. Managers can delegate the authority on approval of refunds to clerks who are responsible for starting the refund procedure. Clerks should not both prepare and issue or cancel a refund.

Next EC equation constrains \( \text{HasAdmin} \) to be a surjective relation; each administrator must be associated with a doctor:

\[
\forall d : \text{Doctor}; t : \text{Time} \mid \\
(\exists a : \text{Admin}) \text{HoldsAt} (\text{HasAdmin} (d, a), t) \tag{21}
\]

Next EC equation defines predicate \( \text{CanDoAdmin} \), which indicates users (\( u \)) that can do administrative tasks on behalf of some doctor (\( d \)):

\[
\forall u : \text{User}; d : \text{Doctor}; t : \text{Time} \mid \\
\text{HoldsAt} (\text{CanDoAdmin} (u, d), t) \\
\Leftrightarrow u = d \lor ((\exists ad : \text{Admin}) u = ad \\
\land \text{HoldsAt} (\text{HasAdmin} (d, ad), t)) \tag{23}
\]

Next EC equations use predicate \( \text{CanDoAdmin} \) to define a pre-condition for events \( \text{SetSubstituteDoctor} \), \( \text{DoctorOnLeave} \) and \( \text{DoctorNo LongerOnLeave} \). These events may occur provided \( \text{CanDoAdmin} \) is true; that is, user executing them can do administrative tasks on behalf of affected doctor:

\[
\forall u : \text{User}; d_1, d_2 : \text{Doctor}; t : \text{Time} \mid \\
\text{Happens} (\text{SetSubstituteDoctor} (u, d_1, d_2), t) \\
\Rightarrow \text{HoldsAt} (\text{CanDoAdmin} (u, d_1), t) \tag{24}
\]

\[
\forall u : \text{User}; d : \text{Doctor}; t : \text{Time} \mid \\
\text{Happens} (\text{SetDoctorOnLeave} (u, d), t) \\
\Rightarrow \text{HoldsAt} (\text{CanDoAdmin} (u, d), t) \tag{25}
\]

\[
\forall u : \text{User}; d : \text{Doctor}; t : \text{Time} \mid \\
\text{Happens} (\text{DoctorNo LongerOnLeave} (u, d), t) \\
\Rightarrow \text{HoldsAt} (\text{CanDoAdmin} (u, d), t) \tag{26}
\]

D. Re-analysing the Model

The analysis configuration is refined by introducing two administrators; one for each doctor:

\[
\begin{align*}
\text{alice, sue} : \text{Admin} \\
\text{Initially} (\text{HasAdmin} (\text{Jones, alice})) \\
\text{Initially} (\text{HasAdmin} (\text{Smith, sue}))
\end{align*} \tag{27, 28, 29}
\]

Under the revised EC model, we re-submit the model to the security violation and suspicion goals:

- For the security violation goal (equation \( AG1 \) above), we still get no plans (not possible to break confidentiality in an obvious way).
- For the refined suspicious goal (equation \( AG3 \) above), we no longer get any plans. Meaning that the loophole has been eliminated.
- For the more abstract suspicious goal (equation \( AG2 \) above) we no longer get obvious security threats, but the results still prompt interesting requirements questions, such as: the system allows doctors to operate the system while they are recorded as being on leave, should this be allowed?

V. Separation of Duties

Separation of Duties (SoD) [16], [17] is a security mechanism used to prevent fraud and errors. It aims to prevent a single individual from executing business-critical tasks of some transactions or business processes. SoD requires such tasks to be performed by different users acting in cooperation (e.g. by requiring two persons to sign a cheque). Here, SoD is studied using a classical case study: a workflow of payment processing; SoD is used to enable payment authorisations to be performed by different users.

The requirements of this workflow system are given in Table III; Fig. 3 depicts underlying workflow. The two tasks involving approval of payments to be carried out by managers, and the tasks \( \text{prepare payment} \) and \( \text{issue/void payment} \), to be carried out by clerks, are subject to SoD.

A. EC Model

The EC model presented here models a \textit{workflow} as a set of activities. Each activity is made of several alternative tasks; one of the tasks must be carried out to complete the activity. In workflow of Fig. 3, activity \( \text{Approve/Refuse Payment} \) comprises tasks \( \text{Approve payment} \) and \( \text{Refuse payment} \), and activity \( \text{Issue/ Void Payment} \) comprises tasks \( \text{Issue payment} \) and \( \text{Void payment} \). There is always some active activity in some running workflow session.

Workflow tasks are executed by users who have different task execution permissions. Task permissions are defined at the level or roles; a user is assigned one or more user roles. In Fig. 3, users that have role \textit{clerk} may execute tasks \textit{Prepare Payment}, \textit{Issue Payment} and \textit{Void Payment}; users of role \textit{manager} may execute tasks \textit{Approve Payment} and \textit{Refuse Payment} and all other tasks that clerks do. The model also enables delegation; administrators of managers may also execute tasks on managers’ behalf.

| R1  | There are two types of users clerks and managers. Managers can perform the tasks that usually the clerks do, but clerks should not usually perform manager’s tasks (exception is delegation, below). |
| R2  | Clerks are responsible for starting the refund procedure, and for issuing or cancelling the refund. |
| R3  | The refund shall be issued by a clerk if approved by the managers, or cancelled otherwise. |
| R4  | A refund must be approved by two different managers. |
| R5  | A clerk shall not both prepare and issue or cancel a refund. |
| R6  | Managers can delegate the authority on approval of refunds to one of their administrators. |

![Diagram](image-url)
The following describes the following elements of EC model: sorts; activities; tasks, rôles and delegation; task execution; and payment processing workflow.

1) Sorts: EC model introduces sorts Activ, Task, Session and User. Activ represents activities of a workflow; Task represents a workflow task that is executed by users; Session represents a running session of a workflow; and User represents users that can execute tasks. Rôles are modelled as sub-sorts of sort User; rôles of payment processing workflow are defined below (section V-A5).

2) Activities: Fluent CurrActiv (a, s) of EC model records current activity (a) of some workflow session (s). Next EC equations constrain fluent CurrActiv to say that there is at most one current activity per workflow session:

\[
\forall a_1, a_2 : \text{Activ}; s : \text{Session}; t : \text{Time} | \\
\text{HoldsAt (CurrActiv (a_1, s), t)} \\
\text{HoldsAt (CurrActiv (a_2, s), t)} \\
\Rightarrow a_1 = a_2
\] (30)

Next EC equation gives initial state of fluent CurrActiv, saying that initially there are no current activities:

\[
\forall a : \text{Activ}; s : \text{Session} | \\
\text{Initially (∼CurrActiv (a, s))}
\] (31)

To represent workflow configurations in terms of its constituent activities, EC model uses predicates OccursBefore. This indicates the ordering of activities in a workflow. Predicate OccursBefore is defined below for payment processing workflow (section V-A5, equation 50).

Predicate IsStartActiv indicates start activity of a workflow; it is defined from predicate OccursBefore as:

\[
\forall a_1 : \text{Activ} | \\
\text{IsStartActiv (a_1)} \\
\Leftrightarrow (∀ a_2 : \text{Activ}) \text{OccursBefore (a_2, a_1)}
\] (32)

In model proposed here, separation of duties is enforced on activities. In workflow of Fig. 3, we have a SoD constraint between activities Approve/Refuse Pay 1 and Approve/Refuse Pay 2 and Prepare Pay and Issue/Void Pay. In EC model, such constraints are represented using predicate SoD, which is defined for each workflow that is to be described and analysed (described below for payment processing workflow in equation 51).

To define a precondition for event StartWrkf, EC model introduces predicate Started, which indicates whether some session has been started or not:

\[
\forall s : \text{Session}; t : \text{Time} | \\
\text{HoldsAt (Started (s), t)} \\
\Leftrightarrow (∃ t_2 : \text{Time}) t_2 ≥ t \\
\text{HoldsAt (CurrActiv (a, s), t_2)}
\] (33)

Event StartWrkf starts a workflow session. Next EC equation defines pre-condition of event StartWrkf; a workflow session may start if it has not already been started:

\[
\forall s : \text{Session}; t : \text{Time} | \\
\text{Happens (StartWrkf (s), t)} \\
\Rightarrow ∼\text{HoldsAt (Started (s), t)}
\] (34)

Next EC equation says how event StartWrkf initiates fluent CurrActiv; when a workflow starts, current activity becomes workflow’s start activity (predicate IsStartActiv):

\[
\forall s : \text{Session}; a : \text{Activ}; t : \text{Time} | \\
\text{IsStartActiv (a)} \\
\Rightarrow \text{Initiates (StartWrkf (s), CurrActiv (a, s), t)}
\] (35)

3) Tasks, rôles and delegation: As said above, tasks are associated with activities. This association is defined through predicate IsTaskOfActiv; this predicate is defined for each workflow being described and analysed (it is defined in equation 49, below, for payment processing workflow).

To know whether some task can be executed in a workflow, EC model introduces predicate IsTaskOfCurrActiv, which indicates whether some task belongs to the current activity of some workflow session. Next EC equation defines this predicate; it says that some task belongs to current activity of some session if it is a task of session’s current activity:

\[
\forall t : \text{Task}; s : \text{Session}; t : \text{Time} | \\
\text{HoldsAt (IsTaskOfCurrActiv (t, s), t)} \\
\Leftrightarrow (∃ a : \text{Activ}) \text{HoldsAt (CurrActiv (a, s), t)} \\
\text{∧ IsTaskOfActiv (t, a)}
\] (36)

Predicate CanDo (u, t) indicates the rôles (r) that are allowed to execute workflow tasks (t). Certain users may delegate their rôles; predicate MayDelegTo (u1, u2) says that some user (u1) is allowed to delegate his rôles to another user (u2). Both CanDo and MayDelegTo are defined for each workflow being described and analysed; it is described below for payment processing workflow in equations 52 and 53.

Event DelegsTo occurs whenever a rôle delegation takes place. Next EC equation defines this event’s pre-condition; users delegate to others provided they are allowed to do so:

\[
\forall u_1, u_2 : \text{User}; t : \text{Time} | \\
\text{Happens (DelegsTo (u_1, u_2), t)} \\
\Rightarrow \text{MayDelegTo (u_1, u_2)}
\] (37)

Fluent Delegated (u1, u2) says that some user (u1) has delegated to another. Next EC equation says that event DelegsTo initiates (sets to true) fluent Delegated to enable system to keep track of rôle delegations:

\[
\forall u_1, u_2 : \text{User}; t : \text{Time} | \\
\text{Initiates (DelegsTo (u_1, u_2), Delegated (u_1, u_2), t)}
\] (38)

Next EC equation defines initial condition of fluent Delegated; initially, no rôle delegations have taken place:

\[
\forall u_1, u_2 : \text{User} | \\
\text{Initially (∼Delegated (u_1, u_2))}
\] (39)

To capture permissions related with delegations, next EC equation introduces predicate CanExecAsDelegate, which indicates whether some user can execute a task as delegate:

\[
\forall u : \text{User}; t : \text{Task}; t : \text{Time} | \\
\text{HoldsAt (CanExecAsDelegate (u, t), t)} \\
\Leftrightarrow (∃ u_2 : \text{User}) \text{HoldsAt (Delegated (u_2, u), t)} \\
\text{∧ CanDo (u_2, ta)}
\] (40)

This predicate is used to define predicate HasPerm, which indicates whether some user has the required permissions to
execute some workflow task; this is true if user has required permissions to execute task with his rôle or if he has been delegated the rôle of someone else with required permissions:

\[
\forall u : User; ta : Task; t : Time \mid
\text{HoldsAt (HasPerm (u, ta), t)} \\
\Leftrightarrow \text{CanDo (u, ta)} \\
\lor \text{HoldsAt (CanExecAsDelegate (u, ta), t)} \tag{41}
\]

4) Task execution: Event ExecTask happens whenever some workflow task is executed. Next EC equation defines predicate ExecTaskOfActiv, which indicates whether some user executed a task of some activity in some workflow session:

\[
\forall u : User; a : Activ; s : Session; t : Time \mid
\text{HoldsAt (ExecTaskOfActiv (u, a, s, t))} \\
\Leftrightarrow (\exists ta : Task; t_2 : Time) t_2 < t \\
\land \text{Happens (ExecTask (ta, u, s), t_2)} \\
\land \text{HoldsAt (CurrActiv (a, s), t_2)} \tag{42}
\]

Predicate ExecTaskOfActiv is used to define predicate BreachesSoD, which indicates whether some user can breach SoD in some workflow session. BreachesSoD is true whenever some user executed some task of some activity for which there is a SoD constraint with current activity in some workflow session; it is defined as:

\[
\forall u : User; s : Session; t : Time \mid
\text{HoldsAt (BreachesSoD (u, s), t)} \\
\Leftrightarrow (\exists a_1, a_2 : Activ) \text{HoldsAt (CurrActiv (a_1, s, t))} \\
\land (\text{SoD (a_1, a_2)} \lor \text{SoD (a_2, a_1)}) \\
\land \text{HoldsAt (ExecTaskOfActiv (u, a_2, s, t))} \tag{43}
\]

When event ExecTask happens, current activity changes to next activity in the workflow; this goes on until the workflow session finishes. There are several restrictions associated with execution of tasks in a workflow; these are modelled as event pre-conditions of event ExecTask: (a) the activity belongs to the current activity (predicate IsTaskOfCurrActiv, equation 36), that (b) the user has the required permissions to execute the task (predicate HasPerm, equation 41), and that (c) the execution of the task by the user does not break separation of duties (predicate BreachesSoD, equation 43). This is defined in EC by the equation:

\[
\forall u : User; ta : Task; s : Session; t : Time \mid
\text{Happens (ExecTask (ta, u, s), t)} \\
\Rightarrow \text{HoldsAt (IsTaskOfCurrActiv (ta, s), t)} \\
\land \text{HoldsAt (HasPerm (u, ta), t)} \\
\land \neg \text{HoldsAt (BreachesSoD (u, s), t)} \tag{44}
\]

As said above, when a task is executed the current activity must change. This requires an EC equation defining a terminates predicate to set the current activity to false if there is a current activity. It also requires an initiates predicate to set the current activity to the next activity of the workflow. These are defined as:

\[
\forall u : User; a : Activ; ta : Task; \\
s : Session; t : Time \mid \\
\text{HoldsAt (CurrActiv (a, s), t)} \\
\Rightarrow \text{Terminates (ExecTask (ta, u, s), CurrActiv(a, s), t)} \tag{45}
\]

\[
\forall u : User; a_1, a_2 : Activ; ta : Task; \\
s : Session; t : Time \mid \\
\text{HoldsAt (CurrActiv (a_1, s), t)} \\
\land \text{OccursBefore (a_1, a_2)} \\
\Rightarrow \text{Initiates (ExecTask (ta, u, s), CurrActiv (a_2, s), t)} \tag{46}
\]

5) Payment processing workflow: EC equations above define infrastructure necessary to describe workflows with SoD constraints. The following EC equations actually define the payment processing system workflow of Fig. 3.

We start by defining the rôles of the workflow. Rôles are modelled as sub-sorts of sort User; and so we have User sub-sorts Clerk and Manager. Rôle administrator is modelled as the predicate IsAdminOf.

Next EC equation defines the tasks and activities of the workflow:

\[
\text{prepPay, approvePay1, approvePay2, FinPay : Activ} \tag{47}
\]

\[
\text{tPrepPay, tApprovePay, tRefusePay, tIssuePay, tVoidPay : Task} \tag{48}
\]

This says that the workflow activities are those identified in Fig. 3, prepare payment (prepPay), approve payment (approvePay1 and approvePay2), and finalise payment (FinPay), and that the tasks are also those of Fig. 3, prepare pay (tPrepPay), approve payment (tApprovePay), refuse payment (tRefusePay), issue payment (tIssuePay) and void payment (tVoidPay).

Next equation defines the relation that exists between tasks and activities by defining predicate IsTaskOfActiv:

\[
\forall ta : Task; a : Activ \mid \text{IsTaskOfActiv (ta, a)} \\
\Leftrightarrow (a = \text{PrepPay} \land ta = tPrepPay) \\
\lor ((a = \text{ApprovePay1} \lor a = \text{ApprovePay2}) \\
\land (ta = tApprovePay \lor ta = tRefusePay)) \\
\lor (a = \text{IssueOrVoidPay} \\
\land (task = tIssuePay \lor task = tVoidPay)) \tag{49}
\]

This says that prepare payment activity is made of prepare payment task, approve payment activities (ApprovePay1 and ApprovePay2) are composed of tasks approve pay and refuse pay, and that FinPay activity is composed of tasks issue pay and void pay.

Next equation defines the OccursBefore predicate. It says that Prep must occur before ApprovePay1, which must occur before ApprovePay2, and that ApprovePay2 must occur before IssueOrVoidPay:

\[
\forall a_1, a_2 : Activ \mid \text{OccursBefore (a_1, a_2)} \\
\Leftrightarrow (a_1 = \text{PrepPay} \land a_2 = \text{ApprovePay1}) \\
\lor (a_1 = \text{ApprovePay1} \land a_2 = \text{ApprovePay2}) \\
\lor (a_1 = \text{ApprovePay2} \land a_2 = \text{IssueOrVoidPay}) \tag{50}
\]

Next equation defines the SoD constraints of the payment processing workflow by defining predicate SoD. It says that there is a SoD constraint between activities PrepPay and
IssueOrVoidPay, and between activities ApprovePay1 and ApprovePay2:

\[ \forall a_1, a_2 : \text{Activ} \mid \text{SoD} (a_1, a_2) \]
\[ \iff (a_1 = \text{PrepPay} \land a_2 = \text{IssueOrVoidPay}) \]
\[ \lor (a_1 = \text{ApprovePay1} \land a_2 = \text{ApprovePay2}) \]  

(51)

Next equation defines the permissions of workflow tasks by defining predicate CanDo. Equation says that managers can execute any task and that clerks can execute tasks prepared, issue and void payments:

\[ \forall u : \text{User}; ta : \text{Task} \mid \text{CanDo} (u, ta) \]
\[ \iff ((\exists ma : \text{Manager}) u = ma) \]
\[ \lor ((\exists cl : \text{Clerk}) (ta = t\text{PrepPay}) \]
\[ \lor ta = t\text{IssuePay} \lor ta = t\text{VoidPay}) \]  

(52)

Next EC equation defines the delegation rule of the payment processing workflow. It says that managers may delegate tasks to their administrators:

\[ \forall u_1, u_2 : \text{User} \mid \text{MayDelegTo} (u_1, u_2) \]
\[ \iff ((\exists ma : \text{Manager}) u_1 = ma \]
\[ \land \text{IsAdminOf} (u_2, u_1)) \]  

(53)

A payment is issued provided both managers approve it. Next EC equation defines predicate PayApproved, which defines what it means for a payment to be approved:

\[ \forall s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{PayApproved} (s), t) \]
\[ \iff ((\exists u_1, u_2 : \text{User}; t_1, t_2, t_3 : \text{Time}; ta : \text{Task}) \]
\[ u_1 \neq u_2 \land t_2 < t \land t_3 < t \]
\[ \land ta = t\text{ApprovePay} \]
\[ \land \text{Happens} (\text{ExecTask} (ta, u_1, s), t_2) \]
\[ \land \text{Happens} (\text{ExecTask} (ta, u_2, s), t_3) \]  

(54)

This says that a payment is approved provided task tApprovePay has been executed at two different time-points by two different users in context of a workflow session.

Next two EC equations define the constraints associated with tasks tIssuePay and tVoidPay. They say that task tIssuePay may be executed provided the payment has been approved, and that task tVoidPay may be executed provided it has not been approved:

\[ \forall u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{Happens} (\text{ExecTask} (\text{IssuePay}, u, s), t) \]
\[ \Rightarrow \text{HoldsAt} (\text{PayApproved} (s), t) \]  

(55)

\[ \forall u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{Happens} (\text{ExecTask} (\text{VoidPay}, u, s), t) \]
\[ \Rightarrow \neg \text{HoldsAt} (\text{PayApproved} (s), t) \]  

(56)

This completes EC model of payment processing workflow requirements (table III). Next section analyses this model.

B. Model Analysis

Analysis is conducted in a configuration made of three managers, Bob, John and Martin, and three clerks Sam, Alice and Sue; Sue also works as an administrator for John. This is defined in EC as:

boh, john, martin : Manager  
alice, sam, sue : Clerk  
\[ \forall u_1, u_2 : \text{User} \mid \text{IsAdminOf} (u_1, u_2) \]
\[ \iff u_1 = sue \land u_2 = john \]  

(59)

Analysis starts with a security violation goal to know if it is possible to reach a state where SoD is breached. Next EC equation defines what it means to breach SoD; that is, a user executed tasks belonging to activities constrained under SoD:

\[ \forall u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{BreachedSoD} (u, s), t) \]
\[ \iff ((\exists a_1, a_2) a_1 \neq a_2 \]
\[ \land \text{HoldsAt} (\text{ExecutedTaskOfActiv} (u, a_1, s), t) \]
\[ \land \text{HoldsAt} (\text{ExecutedTaskOfActiv} (u, a_2, s), t) \]
\[ \land (\text{SoD} (a_1, a_2) \lor \text{SoD} (a_2, a_1)) \]  

(60)

This predicate is used to formulate the goal:

\[ \exists u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{BreachedSoD} (u, s), t) \]  

(AG4)

For this goal, decreasaner does not find any plans. This means that it is not possible to reach a state where SoD is breached. Again, one could argue that SoD is preserved and the system is secure, but it isn’t so.

Analysis proceeds by investigating the suspicious space. Although a user is allowed to execute more than one task in some workflow session, this should not happen very often and is suspicious. The idea is to explore this somehow suspicious or abnormal situation in order to find clues that help in finding security vulnerabilities. First, we define a predicate describing the suspicious system condition of having a user executing two tasks in a workflow session:

\[ \forall u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{ExecutedTwoTasks} (u, s), t) \]
\[ \iff \exists ta_1, ta_2 : \text{Task}; t_1, t_2, t_3 : \text{Time} \]
\[ \land \text{Happens} (\text{ExecTask} (ta_1, u, s), t_1) \]
\[ \land \text{Happens} (\text{ExecTask} (ta_2, u, s), t_3) \]
\[ \land ta_1 \neq ta_2 \land t_2 \leq t_1 \land t_3 \leq t \]  

(61)

Since we are interested in scenarios involving complete workflow runs, next EC equation defines predicate IsWrkfComplete, which says whether some workflow session is complete or not:

\[ \forall s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{IsWrkfComplete} (s), t) \]
\[ \iff \text{HoldsAt} (\text{Started} (s), t) \]
\[ \land \neg ((\exists a : \text{Activ}) \text{HoldsAt} (\text{CurrActiv}(a, s), t)) \]  

(62)

From these two predicates, we define the suspicious goal by describing states where some user executes two different tasks in some workflow run:

\[ \exists u : \text{User}; s : \text{Session}; t : \text{Time} \mid \text{HoldsAt} (\text{IsWrkfComplete} (s), t) \]
\[ \land \text{HoldsAt} (\text{ExecutedTwoTasks} (u, s), t) \]  

(AG5)

For this goal, decreasaner generates interesting plans. We have scenarios where a manager prepares the payment and then approves it, or that he approves and then issues the payment. This happens because managers may act as clerks and there is no SoD constraint between tasks that managers do and clerks do. As this may give a fraud opportunity, it is important to clarify the requirements regarding this issue.
Managers can delegate authority on approval of refunds to one of their administrators, but when administrators executes such tasks system should consider that they have been executed on behalf of manager and are is manager had executed them.

The same person may perform tasks as either manager or clerk, but not both, in any workflow session.

### TABLE IV

REQUIREMENTS RULING THE PROCESSING OF TAX REFUNDS THAT EMERGED AFTER ANALYSIS.

C. Clarifying the requirements

Clarification of the issue exposed by the analysis results in new requirement R7 (table IV), which says that a person can execute tasks under at most one role in any workflow session. To take this new requirement into account, EC model introduces predicate `RolesRequiredDiffer`, which says which workflow tasks require different roles to execute them. This predicate is defined for workflow of payment processing as:

\[
\forall t_1, t_2 : Task \mid
\begin{align*}
\text{RolesRequiredDiffer}(t_1, t_2) & \Leftrightarrow (t_1 = t\text{PrepPay} \lor t_1 = t\text{IssuePay} \\
& \lor t_1 = t\text{VoidPay}) \\
& \land (t_2 = t\text{ApprovePay} \lor t_2 = t\text{RefusePay})
\end{align*}
\]

This predicate says that the roles required for tasks `tPrepPay`, `tIssuePay` and `tVoidPay` is different for those of tasks `tApprovePay` and `tRefusePay`.

Predicate `RolesRequiredDiffer` is used to state the required requirement by constraining event `ExecTask`. Next EC equation describes this constraint by saying that if some user executes two different tasks then roles required to execute them must not differ:

\[
\forall u : User; s : Session; t_1, t_2 : Task; \\
\lnot \exists t_1 \neq t_2 : \text{ExecTaskAt}(u, t_1, s_1) \\
\land \exists t_2 \land \text{ExecTaskAt}(u, t_2, s_2) \\
\Rightarrow \lnot \text{RolesRequiredDiffer}(t_1, t_2)
\]

D. Re-Analysing the model

After the fix, the vulnerability identified above that could give a fraud opportunity is no longer allowed. We resubmit the analysis goal above and decrementor no longer generates scenarios with those possible fraudulent behaviours.

Analysis turns to delegation, which is known to generate security vulnerabilities. Someone executing a task on behalf of someone is legal but suspicious and deserves investigation. Again, the idea is too look in behaviours involving delegation for clues on possible system vulnerabilities. We introduce a predicate to say whether some user executed some task as delegate; next two EC equations define this predicate:

\[
\forall u : User; ta : Task; s : Session; t : Time \\
\text{HoldsAt}(\text{DelegExecutedFor}(u, ta, s, t)) \\
\Leftrightarrow (\exists t_2 : \text{Time} t_2 < t \\
\land \text{ExecTaskAt}(u, ta, s, t_2))
\]

\[
\forall u : User; ta : Task; s : Session; t : Time \\
\text{HoldsAt}(\text{DelegExecutedForAt}(u, ta, s)) \\
\Leftrightarrow (\exists u_2 : User) \\
\text{Happens}(\text{ExecTaskAt}(u_2, s, t)) \\
\land \text{HoldsAt}(\text{Delegated}(u_2, s, t))
\]

Above, predicate `DelegExecutedFor` says whether some task was executed by delegate for some user. This is defined from predicate `DelegExecutedForAt`, which says whether some user executed the task as delegate.

Goal is defined from `DelegateExecutedFor` by describing states of complete workflow runs where someone executes a task on behalf of someone else. This results in the goal:

\[
\exists u : User; ta : Task; s : Session; t : Time \\
\text{HoldsAt}(\text{IsWrkFlt}(s, t)) \\
\land \text{HoldsAt}(\text{DelegExecutedFor}(u, ta, s, t))
\]

Plans generated by decrementor result in what is normally expected under delegation (someone executes a task on behalf of someone else), but they also result in plans that may be possible frauds: a delegate approves a payment on behalf of the manager and the same manager also approves the same payment.

E. Clarifying and elaborating the requirements

From this, we elaborate the requirements, and we get R6’ (table IV) an elaboration of requirement R6. This says that system must consider tasks executed by administrators acting as delegates as if they had been executed by the managers themselves.

To accommodate this new requirement, we introduce the predicate `ExecutedTask`, which indicates whether some user executed some task, either directly or indirectly through a delegate. This is defined as:

\[
\forall u : User; ta : Task; s : Session; t : Time \\
\text{HoldsAt}(\text{ExecutedTask}(u, ta, s, t)) \\
\Leftrightarrow (\exists t_2 : \text{Time} t_2 < t \\
\land \text{ExecTaskAt}(u, ta, s, t_2))
\]

\[
\forall u : User; ta : Task; s : Session; t : Time \\
\text{HoldsAt}(\text{ExecTaskAt}(u, ta, s, t)) \\
\Leftrightarrow \text{Happens}(\text{ExecTaskAt}(u, ta, s, t)) \\
\lor \text{DelegExecutedForAt}(u, ta, s)
\]

Predicate `ExecutedTask` defined above is used to redefine predicate 'ExecutedTaskOfActiv' equation 42). New formulation of this predicate is defined by EC equation:

\[
\forall u : User; a : Activ; s : Session; t : Time \\
\text{HoldsAt}(\text{ExecutedTaskOfActiv}(u, a, s, t)) \\
\Leftrightarrow (\exists t_2 : \text{Task} t_2 < t \\
\land \text{ExecTaskAt}(u, ta, s, t_2))
\]

In this revised EC model, the possible fraudulent behaviour identified above is no longer allowed.

VI. EXPERIMENTAL RESULTS

In both experiments presented above, formal analysis verified a straightforward safety security property, which could mislead analysts in concluding that an unsecure state would not be reached in the modelled system. However, suspicion-based analysis demonstrated that the modelled systems were in fact not secure.

Section IV analyses a simple medical information system that includes a confidentiality requirement. Following the
Anaysis Goal | Time
---|---
SMIS | Security violation (AG1) | 3.6s
SMIS | Suspicion goal 1 (AG2) | 4.9s
SMIS | Suspicion goal 2 (AG3) | 12.6s
SMIS | Security violation (AG1), after fix | 14.8s
SMIS | Suspicion goal 1 (AG2), after fix | 15.8s
SMIS | Suspicion goal 2 (AG3), after fix | 21.1s
Workflow | Security violation (AG4) | 235.9s (3.9m)
Workflow | Suspicion goal 1 (AG5) | 600.7s (10.0m)
Workflow | Suspicion goal 1, after fix (AG5) | 476.9s (7.9m)
Workflow | Suspicion goal 2 (AG6) | 619.2s (10.3m)
Workflow | Suspicion goal 2 (AG6), after fix | 611.11s (10.2m)

**TABLE V**
**Running times for analysis of EC models with decreasoner.** Table indicates case study, analysis goal and time taken to generate plans.

In traditional route of safety analysis, it was not possible to find ways in which confidentiality would be compromised: without a valid credential it would not be possible to obtain the patient’s medical data. Analysis based on suspicion then uncovered a security vulnerability (or loophole) that enabled a malicious user to obtain the required credentials in a non-legal way.

Section V analyses a business process whose security requirements included two integrity requirements enforced through SoD. Again, SoD could be breached, but not in an obvious way. Following the traditional safety analysis route, we checked that it was not possible that the same user would be able to execute two different tasks protected by SoD. Analysis-based on suspicion, however, uncovered several problems: the same user could execute different tasks in a workflow session under different roles, and delegation introduced a loophole that would enable users to indirectly breach SoD.

Table V presents the running times of the formal analysis based on planning with decreasoner. For each case study, it shows how much time it took to carry out the analysis for each analysis goal. We can see that the analysis of the workflow model of section V takes substantially longer than SMIS model because it is more complex.

**VII. DISCUSSION**

This paper proposes suspicion as a concept driving the analysis of security requirements. Through experiments, it argues that, from a practical point of view, in security the interesting question is not only to verify the in-existence of a state compromising some security property (safety), but also to look for what is suspicious in order to find security vulnerabilities and threats. The experiments conducted in the context of the EC temporal logic, planning and the decreasoner tool. They demonstrate the usefulness of suspicion. The traditional safety analysis route, which checks whether some security property is violated, would not expose any security issues; this can mislead analysts in concluding that the system being analysed is secure. Analysis based on suspicion uncovered security vulnerabilities and threats; such findings drive elaboration of the requirements.

One of the advantages of the approach presented here is that security threats can be derived directly from a model of requirements. The analysis that does not need prior knowledge about possible attacks to the modelled system, and so no need to enrich the model with attacker or intruder models. Instead, using the suspicion-based approach proposed here, it is possible to derive threats from a requirements model by posing the model questions based on what is suspicious.

All the vulnerabilities exposed by suspicion-based analysis are related with delegation or passing of capabilities, which are known in security as non-interference properties [18]. The formulation and verification of such properties have proved to be far from trivial [4]. The analysis conducted in this paper confirms that delegation can be trick and hard to get right. Suspicion-based analysis helped in identifying security problems with delegation, and in elaborating the security requirements in order to eliminate such problems. The paper also shows that proof of a straightforward safety property related with security does not deem a system secure. Often, as shown in this paper, the secure question is more involved and requires more in-depth knowledge of the requirements. As this paper shows, it can be more revealing to analyse the system in order to explore the consequences of the requirements, which leads to a better understanding of the security needs and issues of the modelled system, rather than trying to prove that a system is secure. For the delegation-related issues explored in these two experiments, such proofs are far from trivial.

The approach presented here generates automatically possible scenarios of misuse (threats) from a statement describing some security violation or suspicious condition (the goal). This provides a flexible and illuminating scheme to the analysis of security requirements. Rather then finding themselves possible threats, analysts describe instead what would constitute a violation of security or a suspicious system condition. Analysis goals require an understanding of the requirements domain, and should be described with some security asset in mind.

The security vulnerabilities exposed by the analysis illustrate the sort of vulnerabilities that attackers exploit to intrude into today’s software systems. The vulnerabilities identified in the health-care system give insiders the opportunity to perpetrates attacks on the system; the insider threat has been identified as one of the main sources of attacks in the medical domain [15]. Once a source of threats is identified in the requirements, two decisions can be made: (a) introduce further constraints by elaborating the requirements so that the source of threats is eliminated, or (b) do nothing in terms of requirements, but take the problem into account in terms of run-time intrusion and threat detection which then has to judge whether some uses of the system are malicious or not. The latter must be considered because it is not possible to eliminate all possible security threats; doing so could result in a system design that is rigid and over-constrained. The approach presented here enables the detection of threats or vulnerabilities in the system, which also constitutes valuable information for run-time intrusion and threat detection.

The experiments conducted here confirm the importance of...
of modelling and analysing security together with system requirements. Both case studies show how a functionality of the system, delegation, have a serious impact on security and how it was necessary to further elicit and elaborate the requirements in order to eliminate threats.

Formal security analysis with tool support is capable of exposing many unexpected situations, providing a level of assurance not guaranteed by semi-formal approaches. The drawback of the analysis with dereasoner lies in the efficiency of the tool: as models get more complex, the solution to analysis problems take more time to the point that the analysis becomes unpractical. The workflow model of section V is a simplification of an earlier model to enable practical analysis. As usual, the secret is in getting the right abstraction in order to analyse the property of interest.

It is interesting to comment on the usability of the approach presented here. The process of defining analysis goals may require domain knowledge and skill in building and analysing models. However, the process can be partially or fully automated by following the pattern-based approach proposed of [19], which uses the Formal Template Language [20], [21] to represent patterns of EC models together with their associated security monitoring goals. [19] defines templates security violation goals, but patterns of suspicious goals can also be defined if we know in advance what can arouse suspicion. In our experiments, delegation was the focus of our suspicious goals; this is something that can be known in advance and captured using patterns. Following [19], we can have goals that capture what is known to arouse suspicion; actual suspicious goals would then be automatically generated from templates. [19] also uses UML models to enable intuitive requirements modelling; the same approach can also be used to enhance the usability of the approach proposed here.

VIII. RELATED WORK

This paper is a revised and extended version of the work presented in [22]. It shows in detail the EC models that are used to illustrate the analysis based on suspicion with EC, and provides a more in-depth discussion on the verification of security properties, such as the one explored in the paper.

The results of this paper argue against ironclad proofs of security and how one needs to be careful in interpreting formal demonstrations of security properties. This theme is not new; in [23], McLean refutes what used to be a widely held belief: that the security model of Bell and LaPadula [24] and its basic security theorem would capture the essence of security and that implementations following it would be secure. This refutation was done by stating a similar theorem for a model that is clearly not secure. Both experiments of this paper demonstrated that the modelled system would not breach a straightforward safety property of security (confidentiality and separation of duty), and how that could mislead analysts in concluding that the system was secure. However, more flexible means of analysis exposed vulnerabilities showing that the systems being analysed were in fact not secure; the paper suggests analysis lead by what is suspicious in order to find security vulnerabilities in models of requirements.

The case studies used in this paper illustrate behaviours that are usually tricky to be verified using safety or liveness arguments. Most vulnerabilities identified in sections IV and V are related to delegation, which has traditionally proved to be tricky; [18] introduces non-interference, a confidentiality policy that deals with delegation-based functionality. The verification of non-interference is far from trivial, and requires a simplified model of a system that is difficult to obtain when modelling requirements. Such behaviours or properties have also been termed possible [7] and it is known that certain security policies cannot be expressed using safety or liveness properties represented as sets of traces [7], [8]. It is also known that, in general, non-interference policies cannot be expressed as safety or liveness properties [8]. In [8], the authors propose hyperproperties, which are defined as sets of properties (sets of sets of traces), to represent what is not normally captured with traditional properties. This paper provides a pragmatic approach to formally find security vulnerabilities involving delegation in models of security requirements.

There has been substantial interest on security requirements threat analysis [25], [3], [26], [27]. In [26], [27], specifiers need to explicitly identify scenarios or use-cases of abuse and misuse; here, such scenarios are generated automatically from a description of suspicious states (the goal).

[25] proposes a method based on the more flexible abuse frames, specifying undesirable phenomena that the system should prevent from happening. The approach presented here enables the specification of such undesirable phenomena as goals (here called security violation goals). However, it does not only consider what should not happen, but also considers specification of flexible suspicious conditions that (as shown here) have the potential of exposing unknown threats.

[3] proposes a goal-based method that is similar to the approach presented here. Security goals, such as confidentiality, integrity and availability, are negated to obtain goals that specify what should not happen (our security violation goals). Then, these negations are refined to obtain more flexible goals. The approach presented here is more flexible in that it does not only allow specification of goals that come from negation and refinement security goals, but also leaves the specifier the flexibility of defining what constitutes a suspicious condition. Since it is based on tool support, the specifier can use the feedback coming from the tool to either refine existing analysis goals or specify entirely new ones. Essentially, the work presented here is complementary to the body of work on security requirements threat analysis. It explores automated formal analysis (missing in the works above) and provides experimental evidence to the usefulness and effectiveness of security threat analysis.

Suspicion is ubiquitous in intrusion detection [9]. Anomaly-based approaches to intrusion detection [9] are so called because the search for intrusions is driven by abnormal (or suspicious) behaviour patterns of system use. [28] proposes the inclusion of suspicion as a concept driving the models of misuse-based intrusion detection; it proposes models based on suspicious activities that may lead to an attack, as opposed

---

Footnote: This was not claimed by the authors of [24], but others that interpreted their work believed that that was the case.
to models based on actual attacks. Instead, the approach presented here detects abuse by identifying suspicious states in a model of normal system behaviour.

The approach presented here emerges from its preceding work on threat detection [19]. [19] uses an EC model of requirements and planning to find threats at run-time. In [19], however, the goals used to detect threats are more rigid than the ones used here; they say with absolute certainty whether there is an attack or not when the goal is satisfied. In the approach presented here, there is no certainty of attack if a suspicious goal is satisfied, the plans that reach the goal just give us threats (possible attacks). It would be possible to incorporate our approach based on suspicion as a strategy in looking for threats at run-time. Then, the probabilistic component of such a system (like the one of [19]) would try to assign a probability to the computed threats. Here we use suspicion to look for threats in requirements to avoid systems with security vulnerabilities.

The approach presented here analyses requirements automatically using a tool based on SAT-solving. The advantage of this method with respect to other approaches based on theorem-proving [29], [21] is that the reasoning is automatic, avoiding the need for user-intervention as it is usually the case with theorem proving. The disadvantages are that only a portion of the state space is analysed, and that the models that can be handled need to be small; this problem can be mitigated by using abstraction to produce smaller models enabling analysis of property of interest. Another disadvantage to the work in [29], [21] is that there is no visual description of requirements and properties to check; the user needs to be an expert in the formal language (here EC).

IX. CONCLUSIONS

This paper proposes a practical approach to the formal analysis of security requirements based on planning guided by the concept of suspicion. One of the advantages of the approach presented here is that threats can be detected directly from a requirements model, where no prior knowledge about possible attacks is needed to perform the analysis. Instead, the analysis derives threats automatically by posing the model questions based on what is suspicious. The approach was illustrated using the EC and the de reasoner tool by performing two experiments: one involving a simple health-care system with a confidentiality requirement and another a business system with an integrity requirement enforced through SoD. It showed that the more obvious way of analysing security, by doing the traditional safety verification would not give any useful results: following this path analysts could be mislead in concluding that an insecure state could not be reached. However, it was through more flexible analysis based on suspicion that we could obtain useful results exposing subtle security vulnerabilities.

The main contributions of this paper are: (a) the proposal of suspicion as a driving concept in the analysis of security requirements, and (b) the experimental confirmation that, from a practical point of view, it is important to use flexible criteria for the security analysis in order to find vulnerabilities in system requirements (suspicion was proposed as basis for such criteria). The paper also provides experimental evidence to certain claims made in the security requirements literature: (a) it confirmed that it is important to model security requirements together with other functional requirements because functionality impacts on security; (c) it confirmed the existence of security relevant phenomena that is hard or impossible to capture as safety or liveness properties; and (d) demonstrated the importance of formality and tool support and usefulness of automated reachability analysis of requirements.

REFERENCES

A. Sample outputs of decreasoner

This appendix presents sample outputs generated by the decreasoner tool, while carrying out the model analysis presented in this paper.

1) Security Violation Goal (AG1): The output generated by decreasoner for the security violation goal (AG1) of section IV-B is:

no models found

This means that decreasoner could not find any solutions for the analysis goal.

2) Suspicion Goal 1 (AG2): The following output of decreasoner shows one sample solution for the security violation goal (AG1) of section IV-B is:

The remaining 4 sample solutions generated by decreasoner for analysis goal AG2 are as follows:

model 2:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Smith, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Smith, Jones, Smith), 1).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 2).
4
+CredentialMD(Jones, Anderson, 3).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 3).
5
+CredentialMD(Jones, Anderson, 4).
+HasValidCredential(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 4).
6
+CredentialMD(Jones, Anderson, 5).
+HasValidCredential(Jones, Anderson).
Happens(GetMD(Smith, Anderson), 6).
7
+HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 7).
+GoalSatisfied().

model 3:
0
CanAccessMD(Jones, Anderson).
IsDoctorOf(Jones, Anderson).
Happens(AuthoriseAccess(Jones, Anderson), 0).
1
+CredentialMD(Jones, Anderson, 0).
+HasValidCredential(Jones, Anderson).
Happens(SetDoctorOnLeave(Smith, Jones), 1).
2
+OnLeave(Jones).
Happens(SetSubstituteDoctor(Jones, Jones, Smith), 1).
3
+CanAccessMD(Smith, Anderson).
+IsSubstituteDoctor(Smith, Jones).
Happens(AuthoriseAccess(Smith, Anderson), 3).
4
+CredentialMD(Smith, Anderson, 3).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Smith, Anderson), 4).
5
+CredentialMD(Smith, Anderson, 4).
+HasValidCredential(Smith, Anderson).
Happens(AuthoriseAccess(Smith, Anderson), 5).
6
+CredentialMD(Smith, Anderson, 5).
+HasValidCredential(Smith, Anderson).
Happens(GetMD(Smith, Anderson), 6).
7
+HasValidCredential(Smith, Anderson).
+ExposedToAt(Smith, Anderson, 7).
+GoalSatisfied().

The sample above says that event AuthoriseAccess(Jones, Anderson) happens at timepoint 0, and that SetDoctorOnLeave(Smith, Jones) happens at timepoint 1. The analysis goal is satisfied by event GetMD(Smith, Anderson) that happens at timepoint 6.
\[\text{HasValidCredential (Smith, Anderson).}\]
\[\text{ExposedToAt (Smith, Anderson, 6).}\]
\[\text{GoalSatisfied ().}\]

\[\text{model 4:}\]
\[0\]
\[\text{CanAccessMD (Jones, Anderson).}\]
\[\text{IsDoctorOf (Jones, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 0).}\]
\[1\]
\[\text{CredentialMD (Jones, Anderson, 0).}\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{Happens (SetDoctorOnLeave (Jones, Jones), 1).}\]
\[2\]
\[\text{OnLeave (Jones).}\]
\[\text{Happens (SetSubstituteDoctor (Smith, Jones, Smith), 2).}\]
\[3\]
\[\text{CanAccessMD (Smith, Anderson).}\]
\[\text{IsSubstituteDoctor (Smith, Jones).}\]
\[\text{Happens (AuthoriseAccess (Smith, Anderson), 3).}\]
\[4\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{CredentialMD (Smith, Anderson, 3).}\]
\[\text{HasValidCredential (Smith, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 4).}\]
\[5\]
\[\text{CredentialMD (Jones, Anderson, 4).}\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 5).}\]
\[6\]
\[\text{CredentialMD (Jones, Anderson, 5).}\]
\[\text{Happens (GetMD (Smith, Anderson), 6).}\]
\[7\]
\[\text{HasValidCredential (Smith, Anderson).}\]
\[\text{ExposedToAt (Smith, Anderson, 6).}\]
\[\text{GoalSatisfied ().}\]

\[\text{model 5:}\]
\[0\]
\[\text{CanAccessMD (Jones, Anderson).}\]
\[\text{IsDoctorOf (Jones, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 0).}\]
\[1\]
\[\text{CredentialMD (Jones, Anderson, 0).}\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{Happens (SetSubstituteDoctor (Jones, Jones, Smith), 1).}\]
\[2\]
\[\text{IsSubstituteDoctor (Smith, Jones).}\]
\[\text{Happens (SetDoctorOnLeave (Jones, Jones), 2).}\]
\[3\]
\[\text{CanAccessMD (Smith, Anderson).}\]
\[\text{OnLeave (Jones).}\]
\[\text{Happens (AuthoriseAccess (Smith, Anderson), 3).}\]
\[4\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{CredentialMD (Smith, Anderson, 3).}\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 4).}\]
\[5\]
\[\text{CredentialMD (Jones, Anderson, 4).}\]
\[\text{HasValidCredential (Jones, Anderson).}\]
\[\text{Happens (AuthoriseAccess (Jones, Anderson), 5).}\]
\[6\]
\[\text{CredentialMD (Jones, Anderson, 5).}\]
\[\text{Happens (GetMD (Smith, Anderson), 6).}\]
\[7\]
\[\text{HasValidCredential (Smith, Anderson).}\]