Modeling Security Requirements Through Ownership, Permission and Delegation

Paolo Giorgini
University of Trento
paolo.giorgini@unitn.it

Fabio Massacci
University of Trento
fabio.massacci@unitn.it

John Mylopoulos
University of Toronto
jm@cs.toronto.edu

Nicola Zannone
University of Trento
zannone@dit.unitn.it

Abstract

Security Requirements Engineering is emerging as a branch of Software Engineering, spurred by the realization that security must be dealt with early on during the requirements phase. Methodologies in this field are challenging as they must take into account subtle notions such as trust (or lack thereof), delegation, and permission; they must also model entire organizations and not only systems-to-be.

In our previous work we introduced Secure Tropos, a formal framework for modeling and analyzing security requirements. Secure Tropos is founded on three main notions: ownership, trust, and delegation. In this paper we refine Secure Tropos introducing the notions of delegation and trust of execution (at-least delegation and trust) and delegation and trust of permission (at-most delegation and trust). We also propose the use of monitoring as security pattern that can be a design solution to overcome the problem of lack of trust between actors. The paper presents a semantics for these notions, and describes an implemented formal reasoning tool based on Datalog.

1 Introduction

The last years have seen many proposals that incorporate security in the software engineering process. At one end of the spectrum, such proposals ensure good coding practices [26]. At the other extreme, the emphasis is on securing the organization within which a software system functions [2]. A consequence of this is that, modeling and analysis of security requirements has become a key challenge for Software Engineering [8, 6], and is the subject of this paper.

Proposals for Security Requirements Engineering can be roughly classified into two main streams, approaches such as [3, 18, 24] use an off-the-shelve framework, such as UML, KAOS, or i*/Tropos, and model in that framework security requirements. The features of the framework are then used to formally analyze the model or guide the implementation. The second class of approaches [9, 23, 15, 19, 21, 25] takes an RE framework and enhances it with novel constructs specific to security. For such approaches, formal analysis techniques and implementation guidelines need to be revised and/or extended to accommodate the new concepts.

Most proposals focus on protection aspects of security and explicitly deal with a series of security services (integrity, availability etc.) and related protection mechanisms (such as passwords, or crypto). A shift from this perspective towards early requirements was proposed by Giorgini et al. [14, 13] who extended the i*/Tropos modeling framework [5] to define Secure Tropos. The proposal introduces concepts such as ownership, trust, and delegation *within* a normal functional requirements model and shows as security and trust requirements be derived from that.

The baseline for the contributions of this paper is the work of Giorgini et al [14]. After a large case study on the compliance with the Italian privacy legislation of an ISO-19977-like security policy [20], it emerged that the concepts offered in SecureTropos are the right ones but are too coarse-grained to capture important security facets. Specifically, unlike what the framework provides for, found that we are often forced for pragmatic reasons to delegate services and permissions to people we do not trust. Still we consider the overall system secure if we have a way to hold such delegations as accountable, by monitoring their (wrong)doings.

The second observation is that (mis)trust in people may be due to different factors: we may (not) trust them to actually deliver the services we required, or to not abuse the permissions we gave them. In trust management and authorization settings (e.g. [4, 16, 7]) one only finds delegations of permission only (through authorization). Requirements of availability are equally important and can only be captured by modeling delegation of execution (where one actor delegates to another the responsibility to execute a service.)

Thus, the key contributions of this paper is a refined framework for modeling and analyzing security requirements over what has been proposed in [14, 13]. The refinement includes a distinction of the notions of delegation of execution (at-least delegation) and delegation of permission (at-most delegation), the distinction of the notions of trust of execution (at-least trust) and trust of permission (at-most trust), and the use of monitoring as a security pattern, a design solution to overcome the problem of lack of trust between actors. These constructs have been formalized and can be formally analyzed through a tool-supported process.

In the remainder of the paper, we introduce a running example (§2) discuss the overall framework (§3, §4, and §5), its formal semantics (§6) and some formal properties for verification (§7). We conclude with a brief discussion of related work and a summary of our contributions (§8).

2 A Running Example

The running example will be a view of a substantial case study: the compliance to the Italian security and privacy legislation of public administrations such as universities, local governments and health care authorities.

In summary, the law requires administrations to set up a sophisticated security and privacy policies that, for what security is concerned, is fairly close to the complexity of the ISO-17999 standard for security management. Dealing with privacy introduces additional complications such as data ownership, trust and consent. More details on the requirements analysis for an university can be found in [20].

For readability we introduce here a dramatis personae[1]:

Alice is an administrative officer, for example of the teaching evaluation office.

Bob, Bert, and Bill are students;

Sam is (the manager of) the student IT systems,

Paul and Peter are professors

3 Tropos and Secure Tropos

Secure Tropos has been proposed in [14, 13] as a formal framework for modeling and analyzing security has been proposed. It enhances the agent-oriented software development methodology Tropos [5]. Tropos uses the concepts of actor, goal, task, resource and social dependency for defining the obligations of actors (dependees) to other actors (dependers). A goal represents the strategic interests of an actor. A task specifies a particular course of action that produces a desired effect, and can be executed in order to satisfy a goal. A resource represents a physical or an informational entity. Finally, a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource. Tropos is well suited to to describe both an organization and an IT system. However, in [12] we have argued that it lacks the ability to capture at the same time the functional and security features of the organization, and hence the new proposal.

[1]This impersonation is closer to reality than one may think: the law requires to assign the responsibility of each IT sub-system to human beings.

For sake of simplicity, in this new framework, the notion of *service* is used to refer to a goal, task, or resource, and three new relationships have been introduced.

- *Ownership* (between an actor and a service) if an agent is the legitimate owner of a service.
- *Trust* (among two actors and a service), so that an actor A trust another actor B on a certain goal G.
- *Delegation* (among two actors and a service), if an actor A explicitly delegates to an actor B a goal, or the execution of a task or the access to a resource.

Example 1 *By law, Bob is the owner of his personal data. Yet, the data is stored on servers that are managed by Sam who give access to Alice and Paul. So, Sam should seek the consent of Bob for data processing.*

Another feature is the distinction between trust and delegation. Delegation marks a formal passage in the domain that is currently modeled by the requirements engineers.

Example 2 *The letter of the CEO of the University that assign to the CIO the responsibility for enacting privacy protection measures is an example of delegation.*

In digital trust management systems, this would be matched by the issuance of a delegation certificate. In contrast, trust marks simply a social relationship that is not formalized by a "contract" (such as digital credential or a letter). There might be cases (e.g. because it is impractical or too costly), where we might be happy with a "social" protection, and cases in which formal delegation is essential. Such decisions are taken by the designer and the formal model just offers support to spot inconsistencies. The basic consequence of delegation is having more permission holders.

4 Refining Delegation and Trust

Now we introduce a conceptual refinement of the delegation and trust relationships, that will allow us to capture and model important security facets

Example 3 *Alice is interested in gathering data on students' performance, for which she depends on Sam. Bob owns his sensitive personal information, such as his student careers. Bob delegates permission to provide information about his career to Sam on condition that his privacy is protected (i.e., his identity is not revealed).*

In this scenario (Fig. 1(a)), there is a difference of relationship between Alice–Sam and Bob–Sam. This difference is due to a difference in the type of delegation.

Example 4 *Bob* delegates permission *to Sam to provide only the relevant information and nothing else. On the other hand, Alice, who wants student data,* delegates *the* execution *of her goal to Sam. According to Alice, Sam should at least fulfill the goal she requires. She is not interested in what Sam does with Bob's trust, apart from getting her information. The major worry of Alice is availability whereas Bob cares about authorization. In other words, Alice's major concerns would be that tasks are delegated to people that can actually do them, whereas Bob would be concerned that subtasks are given to trusted people who will not misuse the permissions they have acquired.*

If we want to check that requirements are consistent and that security requirements of each actor are met, it is essential to distinguish between these two notions of delegation. We use **at-most delegation** when the delegater wants the delegatee to fulfill at most a service. This is *delegation of permission*, where the delegatee thinks "I have the permission to fulfill the service (but I do not need to)", whereas **at-least delegation** means that the delegater wants the delegatee to perform at least the service. This is the *delegation of execution*. The delegatee thinks, "Now, I have to get the service fulfilled (let's start working)". In the pictorial representation of Fig. 1 we represent these relationship as edges respectively labeled **P** and **E**.

Further, we want to separate the concepts of trust and delegation, as we might need to model systems where some actors must delegate permission or execution to other actors they don't trust. Also in this case it is convenient to have a suitable distinction for trust in managing permission and trust in managing execution. The meaning of **at-most trust** is that an actor (truster) trusts that another actor (trustee) at most fulfills the goal but will not overstep it. The meaning of **at-least trust** is that an actor (truster) trusts that another actor (trustee) at least fulfills the goal.

Example 5 *At-most trust is good for permissions: Bob trusts Sam to remain within certain bounds. He may delegate Sam more permissions than actually needed because Sam will not abuse them. At-least trust fits execution. Alice believe Sam can accomplish her tasks and possibly more.*

Tropos original dependency is just at-least delegation plus implicit at-least trust. The delegation proposed by Giorgini et al. [14] blurs the distinction between at-least and at-most delegation and at-least and at-most trust.

In the development of a system, a designer should be able to guarantee and implement the trust and delegation relationships captured in the social setting during the requirements analysis phase. This analysis can be done with the methodology advocated in [14].

However, specific situations may impose that some services have to be delegated to some actors even when there is no trust relationship with those actors. In our example [20], this is the outsourcing of services to outside providers for which a trust relationship must still be built. These providers might range from cleaning ladies to security guards, from ERP clock cycles to network backbones. In these cases we can adopt a monitor as a design pattern able to provide a solution to the problem of lack of trust between actors. In the following section we present the idea of monitoring.

5 Monitoring

When work needs to be delegated even when there is no trust, then monitoring can offer a surrogate for trust. Accordingly to Gans's et al. [11], the existence of distrust can be tolerated with an additional overhead of monitoring the untrustworthy delegatee. Here we refine Gans's et al. intuition integrating it in our framework.

The goal of an actor playing the role of *monitor* is to check for the violation of trust[2]. The act of monitoring can be done by the delegater himself[3], or he can delegate it to some other actors to get it done. Depending on the type of delegation, we have two different kinds of monitors: **at-most monitor** and **at-least monitor**. Consider the situation presented in Fig. 1(a).

Example 6 *Suppose that there is no trust between Bob and Sam for the goal "maintain privacy", but the student must delegate permission nonetheless. In this case, he depends (***D***) on the ombudsman (O) for monitoring if Sam transgresses her permissions. This is shown in Fig. 1(b)) with an at-most monitor (monitor for permission –* **Mp***) relationship between the ombudsman and Sam.*

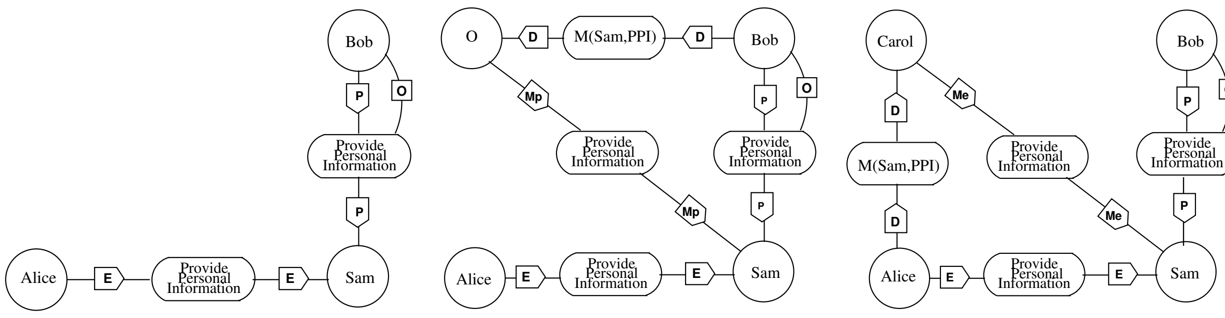
Example 7 *If Alice is not confident that Sam will provide updated information, she may delegate to her secretary Carol the task of confirming with, or nagging Sam to insert new data as soon as it becomes available. This is shown in Fig. 1(c)) with an at-least monitor (monitor for execution –* **Me***) relationship between Carol and Sam.*

Another important distinction that emerges when we use a monitor is related to type of service (goal, task or resource) for which the monitoring is required for.

Let us assume that the service S in Fig. 1 is a task (i.e., a specific sequence of actions). So, the Monitor has to check if Sam executes the actions of the task. What happens if Sam delegates the task or some of its subtasks to other actors?

[2]Indeed, monitoring could also be used for the evaluation of the fulfillment of a service assigned to a trusted actor.

[3]Intuitively, this is like saying that fellow is unreliable, I'll give him the job but keep an eye on him myself".


Bob
P
O
Provide Personal Information
P
Alice
E
Provide Personal Information
E
Sam
O
D
M(Sam,PPI)
D
Bob
Mp
P
O
Provide Personal Information
Provide Personal Information
Mp
P
Alice
E
Provide Personal Information
E
Sam
Carol
Bob
D
Me
P
O
M(Sam,PPI)
Provide Personal Information
Provide Personal Information
Me
P
D
Alice
E
Provide Personal Information
E
Sam

(a) Delegation (b) At-most Monitor (c) At-least Monitor

Figure 1. Delegation and Monitoring

Example 8 *To achieve the goal delegated to him in Example 7, Sam will issue a letter to the head of each student secretariat office so that student marks are entered into the system within 30 days from the date that exams have taken place.*

A solution to this problem is to extend the monitoring to all sublevels of delegation until the level where the actual execution takes place. So, there will be a monitor relationship between the Monitor and all the actor involved in the execution of at least a part of the task.

Example 9 *To reach the objective of 30 days requires that professors return to the office assigned marks. This is a further step of delegation of execution. Then, the actor responsible at the office, beside actually monitoring his employees, may also assign the task of reminding professors that they must return on time their mark sheets.*

Notice that monitoring as such is not a primitive construct. It can be captured by other constructs within our modeling framework. Specifically, every service will either be delegated during the design process to a trusted actor, or it will be delegated to an untrusted one, in which case the delegatee will be monitored by a trusted actor.

On the formal model this corresponds to a design pattern formalized in terms of additional axioms that allow us to conclude that an actor is confident that a service will be executed or a permission will not be abused even if existing trust relations suggest otherwise.

Once we see monitoring as a simple design solution (essentially a security pattern) we can treat monitoring goals just as any other goal. So they can be further subject to refinement, delegation of execution and delegation of permission. Trust relationships linked to monitoring can then be captured with standard constructs. For example, monitoring often requires having permission to access monitored data or personnel. This itself may create problems of permission and authorization that can be model in the framework.

General predicates
delegate($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s$)
delegateChain($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s$)
trust($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s$)
trustChain($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s$)
monitoring($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Actor} : b, \texttt{Service} : s$)
confident($\texttt{Type} : t, \texttt{Actor} : a, \texttt{Service} : s$)
Specific for execution
requests($\texttt{Actor} : a, \texttt{Service} : s$)
provides($\texttt{Actor} : a, \texttt{Service} : s$)
should_do($\texttt{Actor} : a, \texttt{Service} : s$)
can_satisfy($\texttt{Actor} : a, \texttt{Service} : s$)
Specific for Permission
owns($\texttt{Actor} : a, \texttt{Service} : s$)
has_per($\texttt{Actor} : a, \texttt{Service} : s$)
Goal refinement
goal($\texttt{Service} : s$)
subgoal($\texttt{Service} : s_1, \texttt{Service} : s_2$)
OR_subgoal($\texttt{Service} : s_1, \texttt{Service} : s_2$)
AND_subgoal($\texttt{Service} : s_1, \texttt{Service} : s_2$)
AND_decomp($\texttt{Service} : s_1, \texttt{Service} : s_2, \texttt{Service} : s_3$)

Table 1. Predicates

6 Formalization

As done in [14, 13], we to use Datalog [1] as the underlying semantic framework, also to be close to the semantics of other frameworks for trust or security (e.g. [7, 17, 22]).

A Datalog program is a set of rules of the form L:- $L_1 \wedge ... \wedge L_n$ where L, called head, is a positive literal and $L_1, ..., L_n$ are literals and they are called body. Intuitively, if $L_1, ..., L_n$ are true in the model then L must be true in the model. In Datalog, negation is treated as negation as failure: if there is no evidence that an atom is true, it is considered to be false. Hence if an atom is not true in some model, then its negation should be considered to be true in that model.

We start by presenting the predicates for our framework. Table 1 extends the predicates already presented in [14, 13] introducing specific predicates for execution, permission and monitoring[4]. For compactness' sake we use the first argument of the predicates to indicate the type of actions. Thus, delegate, delegateChain, and monitoring have a type $t \in \{exec, perm\}$; trust, trustChain have a type $t \in \{exec, perm, mon\}$; and confident has a type $t \in \{satisfy, exec, owner\}$. Once again, we specify predicates for generic "services" because differentiating them into goals, tasks and resources is immediate[5].

6.1 Formal Model for Execution

The predicates that we introduced corresponds to the relations that the requirements engineer can actually draw during her analysis. The predicate requests(a, s) holds if actor a wants service s fulfilled, while provides(a, s) holds if actor a has the capability to fulfill service s. The predicate delegate$(exec, a, b, s)$ holds if actor a delegates[6] the execution of service s to actor b. The actor a is called the *delegater*; the actor b is called the *delegatee*. The predicate trust$(exec, a, b, s)$ holds is actor a trusts that actor b at least fulfills service s. The actor a is called the *truster*; the actor b is called the *trustee*. The predicate trust(mon, a, b, s) holds if actor a trusts that actor b monitors whether service s will be satisfied. The predicate monitoring$(exec, a, b, s)$ holds if actor a monitors if actor b at least can satisfy service s.

Other predicates are used to define properties that will be used during formal analysis. The predicates delegateChain$(exec, a, b, s)$ and trustChain$(exec, a, b, s)$ hold if there is a delegation and a trust chain respectively, between actor a and actor b. The predicate should_do(a, s) identify actors who should directly fulfill the service. The basic idea of the predicate can_satisfy is that "for every goal I have assigned responsibilities so that it can be fulfilled". In other words, if an actor has the objective of fulfilling a service, he can satisfy it. Thus it locates the common leaves of the delegation trees of execution and permission. Thus, the predicate can_satisfy(a, s) holds if actor a can satisfy service s. The predicate confident$(satisfy, a, s)$ holds if actor a is confident that service s can be satisfied. Finally, we have the predicates for goal refinement. Their semantics and axiomatization are straight-forward.

The axiomatization is more complex for delegation of execution as shown in Table 2. The first batch of axioms deals with actors and trust diagrams: E1 and E2 build a delegation chain of execution; E3 and E4 (M1 and M2) build a trust chain for execution (monitoring); E5 builds chains over monitoring steps.

E6 and M4 have chains propagate to subgoals. According to E6 execution-trust flows top-down with respect to goal refinements. The axiom for monitoring M4 states that trustChain flows top-down with respect to goal refinements. M5 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation.

The remaining axioms describe how global properties of the model are defined. E7 and E8 state that an actor has to execute the service if he provides a service and if either some actor delegates the service to him, or he himself aims for the service. E9 and E10 state an actor, who requests for a service, can satisfy the service if either he provides it or he has delegated it to someone who can satisfy it. Goal refinements are taken care of by using the axioms E11 and E12. If an actor can satisfy at least one of the or-subgoals of a service, then he can satisfy the main service. Also, if he can satisfy all and-subgoals, then he can satisfy the main service.

The notion of confidence is captured by axioms E13-E16. A, the aimer, is confident that S will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. Goal refinements are taken care of by using axioms E15 and E16: if an actor is confident that at least one of the or-subgoals of a service will be fulfilled, then he can be confident that the service will be fulfilled. And-subgoals are analogous.

6.2 Formal Model for Permission

In Table 1 we also have predicates for modeling delegation of permission. The first set of predicates corresponds to the relations drawn by the requirements engineer. The predicate owns(a, s) holds if actor a owns service s: the owner of a service has full authority concerning access and usage of his services, and he can also delegate this authority to other actors. The intuition is that delegate$(perm, a, b, s)$ holds if actor a at most delegates the permission to fulfill service s to actor b. As for execution, the actor a is called the *delegater*; the actor b is called the *delegatee*. The predicate trust$(perm, a, b, s)$ holds is actor a trusts that actor b at most has the permission to fulfill service s. The predicate monitoring$(perm, a, b, s)$ is the dual of the execution counterpart.

Also in this case other predicates are used to define interesting properties for the formal analysis by the requirement engineer. The predicates delegateChain$(perm, a, b, s)$ and trustChain$(perm, a, b, s)$ hold if there is a delegation, resp. a trust chain of permission among actor a and actor b. The basic idea of has_per sums up the possible ways in which an

[4]Monitoring is not a primitive predicate and is rather a defined predicate. However, for sake of readability, it is convenient to use the pattern's name rather than expanding the pattern's definition.

[5]For resources we must replace the subgoal relation with the part-of relation.

[6]For the sake of simplicity we do not deal with the question of depth here. See Li et al. [16] for an account of delegation with depth. What has emerged from several case studies is that depth is less important than qualifications such as "only to members of the same office".

Delegation	
E1	$\mathsf{delegateChain}(exec, A, B, S) \leftarrow \mathsf{delegate}(exec, A, B, S)$
E2	$\mathsf{delegateChain}(exec, A, C, S) \leftarrow \mathsf{delegate}(exec, A, B, S) \wedge \mathsf{delegateChain}(exec, B, C, S)$
Trust	
E3	$\mathsf{trustChain}(exec, A, B, S) \leftarrow \mathsf{trust}(exec, A, B, S)$
E4	$\mathsf{trustChain}(exec, A, C, S) \leftarrow \mathsf{trust}(exec, A, B, S) \wedge \mathsf{trustChain}(exec, B, C, S)$
E5	$\mathsf{trustChain}(exec, A, C, S) \leftarrow \mathsf{trustChain}(mon, A, B, S) \wedge \mathsf{monitoring}(exec, M, C, S)$
E6	$\mathsf{trustChain}(exec, A, B, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(exec, A, B, S)$
M1	$\mathsf{trustChain}(mon, A, B, S) \leftarrow \mathsf{trust}(mon, A, B, S)$
M2	$\mathsf{trustChain}(mon, A, C, S) \leftarrow \mathsf{trust}(mon, A, B, S) \wedge \mathsf{trustChain}(mon, B, C, S)$
M3	$\mathsf{trustChain}(mon, A, C, S) \leftarrow \mathsf{trustChain}(exec, A, B, S) \wedge \mathsf{trustChain}(mon, B, C, S)$
M4	$\mathsf{trustChain}(mon, A, B, S_1) \leftarrow \mathsf{subgoal}(S, S_1) \wedge \mathsf{trustChain}(mon, A, B, S)$
Monitoring	
M5	$\mathsf{monitoring}(exec, M, B, S_1) \leftarrow \mathsf{delegateChain}(exec, A, B, S) \wedge \mathsf{monitoring}(exec, M, A, S) \wedge \mathsf{subgoal}(S_1, S)$
Should do	
E7	$\mathsf{should\_do}(A, S) \leftarrow \mathsf{delegateChain}(exec, B, A, S) \wedge \mathsf{provides}(A, S)$
E8	$\mathsf{should\_do}(A, S) \leftarrow \mathsf{requests}(A, S) \wedge \mathsf{provides}(A, S)$
Can satisfy	
E9	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{should\_do}(A, S)$
E10	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{delegate}(exec, A, B, S) \wedge \mathsf{can\_satisfy}(B, S)$
E11	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{can\_satisfy}(A, S_1)$
E12	$\mathsf{can\_satisfy}(A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{can\_satisfy}(A, S_1) \wedge \mathsf{can\_satisfy}(A, S_2)$
Confident to can satisfy	
E13	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{should\_do}(A, S)$
E14	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{delegateChain}(exec, A, B, S) \wedge \mathsf{trustChain}(exec, A, B, S) \wedge \mathsf{confident}(satisfy, B, S)$
E15	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{OR\_subgoal}(S_1, S) \wedge \mathsf{confident}(satisfy, A, S_1)$
E16	$\mathsf{confident}(satisfy, A, S) \leftarrow \mathsf{AND\_decomp}(S, S_1, S_2) \wedge \mathsf{confident}(satisfy, A, S_1) \wedge \mathsf{confident}(satisfy, A, S_2)$

Table 2. Axioms for execution

actor can grab the permission on a service: either directly or by delegation. From the point of view of the owner, confidence means that the owner is confident that the permission that he has delegated will not be misused. Alternatively, the owner is confident that he has delegated permission only to trusted or monitored agents. This means that even if there is one untrusted or unmonitored delegation, then the owner could be uneasy about the likely misuse of his permissions. So, an owner is confident, if there is no likely misuse of his permission. It can be seen that there is an intrinsic double negation in the statement. So we try to model it using a predicate $\mathsf{diffident}(a, s)$. At any point of delegation of permission, the delegating agent is diffident, if the delegation is being done to an agent who is neither trusted not monitored or if the delegatee could be diffident himself. In this way, $\mathsf{confident}(owner, a, s)$ holds if the owner a is confident to give the permission on service s only to trusted actors.

Table 3 presents the axioms for delegation of permission. The first batch deals with actors and trust diagrams: P1 and P2 build a delegation chain of permission; P3 and P4 build a trust chain for permission; P5 builds chains over monitoring steps. P6 has the chain propagate through subgoals. If an actor trusts that another will not overstep the set of actions required to fulfill a subgoal of a service, then the first can trust the last not to overstep the set of actions required to fulfill S. The permission trust, with respect to goal refinements, flows bottom-up. M6 is used to build a trust chain for monitor. M7 states that if an actor under monitoring delegates a service to another, then the monitor have to watch for the delegatee, that is, the monitor follows the delegation. The owner of a service has full authority concerning access and disposition of it. Thus, P7 states that if an actor owns a service, he has it. P8 states that the delegatee has the service. The notion of confidence and diffidence that we have sketched above is captured by the axioms P10-P12.

6.3 Combining Execution and Permission

More sophisticated properties require reasoning with both execution and permission. To this end, we introduce some notions that put together these two notions. In Table 4 we present the notions from both the point of view of the requester and the point of view of the owner. The predicate $\mathsf{can\_execute}(a, s)$ holds if actor a can see service s fulfilled. The predicate $\mathsf{confident}(exec, a, s)$ holds if actor a is confident to see service s fulfilled. Actor a, who aims for service s, is confident that s will be fulfilled, if he knows that all delegations have been done to trusted or monitored agents and that the agents who will ultimately execute the service, have the permission to do so. This is done using the axioms Ax5-Ax6. Goal refinements are taken care of by using the axioms Ax7-Ax8. If a is confident that at least one of the

Delegation	
P1	$\textsf{delegateChain}(perm, A, B, S) \leftarrow \textsf{delegate}(perm, A, B, S)$
P2	$\textsf{delegateChain}(perm, A, C, S) \leftarrow \textsf{delegate}(perm, A, B, S) \wedge \textsf{delegateChain}(perm, B, C, S)$
Trust	
P3	$\textsf{trustChain}(perm, A, B, S) \leftarrow \textsf{trust}(perm, A, B, S)$
P4	$\textsf{trustChain}(perm, A, C, S) \leftarrow \textsf{trust}(perm, A, B, S) \wedge \textsf{trustChain}(perm, B, C, S)$
P5	$\textsf{trustChain}(perm, A, C, S) \leftarrow \textsf{trustChain}(mon, A, B, S) \wedge \textsf{monitoring}(perm, B, C, S)$
P6	$\textsf{trustChain}(perm, A, B, S) \leftarrow \textsf{subgoal}(S, S_1) \wedge \textsf{trustChain}(perm, A, B, S_1)$
M6	$\textsf{trustChain}(mon, A, C, S) \leftarrow \textsf{trustChain}(perm, A, B, S) \wedge \textsf{trustChain}(mon, B, C, S)$
Monitoring	
M7	$\textsf{monitoring}(perm, M, B, S_1) \leftarrow \textsf{delegateChain}(perm, A, B, S) \wedge \textsf{monitoring}(perm, M, A, S) \wedge \textsf{subgoal}(S_1, S)$
Has permission	
P7	$\textsf{has\_per}(A, S) \leftarrow \textsf{owns}(A, S)$
P8	$\textsf{has\_per}(A, S) \leftarrow \textsf{delegateChain}(perm, B, A, S) \wedge \textsf{has\_per}(B, S)$
P9	$\textsf{has\_per}(A, S_1) \leftarrow \textsf{subgoal}(S_1, S) \wedge \textsf{has\_per}(A, S)$
Owner is confident to give the service to trusted actors	
P10	$\textsf{confident}(owner, A, S) \leftarrow \textsf{owns}(A, S) \wedge \textsf{not diffident}(A, S)$
P11	$\textsf{diffident}(A, S) \leftarrow \textsf{delegateChain}(perm, A, B, S) \wedge \textsf{diffident}(B, S)$
P12	$\textsf{diffident}(A, S) \leftarrow \textsf{delegateChain}(perm, A, B, S) \wedge \textsf{not trustChain}(perm, A, B, S)$
P13	$\textsf{diffident}(A, S) \leftarrow \textsf{subgoal}(S_1, S) \wedge \textsf{diffident}(A, S_1)$

Table 3. Axioms for permission

Can see the service fulfilled (can execute)	
Ax1	$\textsf{can\_execute}(A, S) \leftarrow \textsf{should\_do}(A, S) \wedge \textsf{has\_per}(A, S)$
Ax2	$\textsf{can\_execute}(A, S) \leftarrow \textsf{delegateChain}(exec, A, B, S) \wedge \textsf{can\_execute}(B, S)$
Ax3	$\textsf{can\_execute}(A, S) \leftarrow \textsf{OR\_subgoal}(S_1, S) \wedge \textsf{can\_execute}(A, S_1)$
Ax4	$\textsf{can\_execute}(A, S) \leftarrow \textsf{AND\_decomp}(S, S_1, S_2) \wedge \textsf{can\_execute}(A, S_1) \wedge\ \textsf{can\_execute}(A, S_2)$
Confident to see the service fulfilled (confident to execute)	
Ax5	$\textsf{confident}(exec, A, S) \leftarrow \textsf{should\_do}(A, S) \wedge \textsf{has\_per}(A, S)$
Ax6	$\textsf{confident}(exec, A, S) \leftarrow \textsf{delegateChain}(exec, A, B, S) \wedge \textsf{trustChain}(exec, A, B, S) \wedge \textsf{confident}(exec, B, S)$
Ax7	$\textsf{confident}(exec, A, S) \leftarrow \textsf{OR\_subgoal}(S_1, S) \wedge \textsf{confident}(exec, A, S_1)$
Ax8	$\textsf{confident}(exec, A, S) \leftarrow \textsf{AND\_decomp}(S, S_1, S_2) \wedge \textsf{confident}(exec, A, S_1) \wedge\ \textsf{confident}(exec, A, S_2)$
Need to know	
Ax9	$\textsf{need\_to\_have\_perm}(A, S) \leftarrow \textsf{should\_do}(A, S)$
Ax10	$\textsf{need\_to\_have\_perm}(A, S) \leftarrow \textsf{delegate}(perm, A, B, S) \wedge \textsf{not other\_delegater}(A, B, S) \wedge \textsf{need\_to\_have\_perm}(B, S)$
Ax11	$\textsf{other\_delegater}(A, B, S) \leftarrow \textsf{delegate}(perm, C, B, S) \wedge \textsf{need\_to\_have\_perm}(C, S) \wedge A \neq C$

Table 4. Axioms Involving both permission and execution

or-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled. Also, if a is confident that all and-subgoals of s will be fulfilled, then a can be confident that s will be fulfilled.

Owners may wish to delegate permissions to providers only if the latter actually do need the permission. The last part of Table 4 defines the predicates that are necessary to analyze *need-to-know* properties. As a result of absence of diffidence, the owner can be confident that his permission will not be misused. But has this permission reached the agents who actually *need* it? The owner might also want to ensure that there has been not unwanted delegation of permission. This can be achieved by identifying the agents who actually *need-to-know* (or rather *need-to-have*) the permission. This set of axioms captures also the possibility of having alternate paths of permission delegations. In this case the formal analysis will not yield one model but multiple models in which only one path of delegation is labeled by the need-to-have property and the others are not.

Example 10 (Figure 2) *Alice and Carol (7 and 8) have both received the consent (permission) by Bob (1) for using his personal data, and both delegate it to the faculty secretariat (3), which must have the permission to provide the data to Paul (6), the university tutor who should provide personal counseling to Bob. In this case only one of either Alice or Carol needs to have the permission.*

7 Analysis and Verification

Design *properties* are not enforced with axioms for two reasons. At first the actual system drawn by the requirement

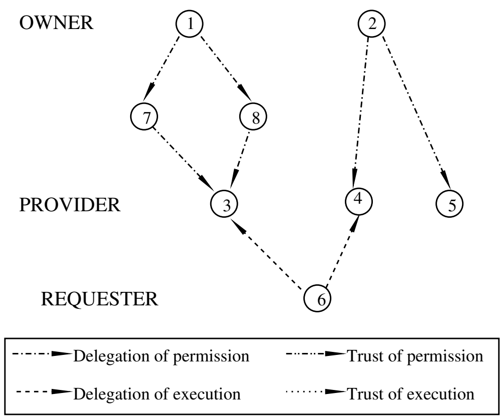
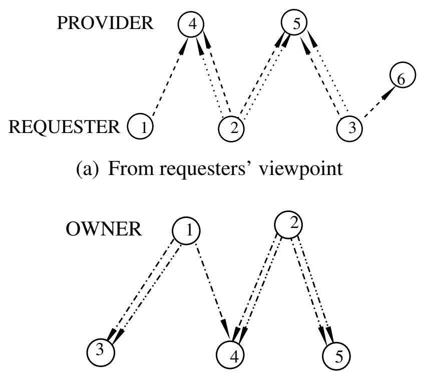


Figure 2. Need-to-Know and Multiple Permissions Paths



(a) From requesters' viewpoint

(b) From owners' viewpoint

Figure 3. Design for delegation of execution and permission

Execution	
Pro1	$\mathsf{delegateChain}(exec, A, B, S) \Rightarrow? \mathsf{trustChain}(exec, A, B, S)$
Pro2	$\mathsf{requests}(A, S) \Rightarrow? \mathsf{can\_satisfy}(A, S)$
Pro3	$\mathsf{requests}(A, S) \Rightarrow? \mathsf{confident}(satisfy, A, S)$
Pro4	$\mathsf{should\_do}(A, S) \Rightarrow? \mathsf{not}\ \mathsf{delegateChain}(exec, A, B, S)$
Permission	
Pro5	$\mathsf{delegateChain}(perm, A, B, S) \Rightarrow? \mathsf{trustChain}(perm, A, B, S)$
Pro6	$\mathsf{owns}(A, S) \Rightarrow? \mathsf{confident}(owner, A, S)$
Pro7	$\mathsf{owns}(A, S) \Rightarrow? \mathsf{not}\ \mathsf{delegateChain}(perm, B, A, S) \wedge A \neq B$
Execution & Permission	
Pro8	$\mathsf{requests}(A, S) \Rightarrow? \mathsf{can\_execute}(A, S)$
Pro9	$\mathsf{requests}(A, S) \Rightarrow? \mathsf{confident}(exec, A, S)$
Pro10	$\mathsf{owns}(A, S) \Rightarrow? \mathsf{need\_to\_have\_perm}(A, S)$
Pro11	$\mathsf{owns}(A, S) \Rightarrow? \begin{cases} \mathsf{need\_to\_have\_perm}(A, S) \wedge \\ \mathsf{confident}(owner, A, S) \end{cases}$

Table 5. Desirable Properties of a Design

engineer may not satisfy them, and therefore the missing link may be actually a bug. Second, there might be many ways in which a requirement engineer may wish to fulfill desired properties. We use the DLV system[7] to verify security properties with respect to a Secure Tropos model.

In Table 5 we use the $A \Rightarrow? B$ to mean that one must check that each time A holds it is desirable that B also holds. In Datalog this can be represented as the constraint :- A, not B. If the set of features is not consistent, i.e., they cannot all be simultaneously satisfied, the system is inconsistent, and hence it is not secure. This also guarantee us that our proposed axioms are consistent if we check for consistency of the model without trying to enforce any property.

Pro1 states that if there is a delegation chain either the delegater trusts the delegatee or there is the monitor and the delegater trust the monitor. Pro2 states that a requester wants to can satisfy his goals, and Pro3 states that a requester wants to be confident to satisfy the service.

Example 11 (Figure 3(a)) *Bob and Bert (1 and 2) need counseling. They can receive it (formal relation can satisfy) because they delegate the execution to Paul and Peter (4 and 5), while Bill (3) cannot receives all necessary advices because he requested some of them only to Alice (6) which is not able to provide counseling on faculty matters.*

Bob is also confident to receive all counseling he needs since he delegates the execution to Paul and Peter (4 and 5) whom he trusts, while Bert is not confident since he delegates to Paul (4) that he does not trust.

Pro4 states that if an actor provides a service, then, if either some actor delegates the service to him, or if he himself requests the service, then he has to execute the service without further delegation. Pro5 states that if there is a delegation chain, either the delegater trusts the delegatee or there is the monitor. Pro6 states that the owner of the service has to be confident to give the service to trusted actors, and Pro7 states that a service cannot come back to the owner.

Example 12 (Figure 3(b)) *Bob and Bert (1 and 2) need to provide their personal data for receiving accurate counseling. Bob is confident on his personal data since he delegates the permission on it to two Paul and Peter (4 and 5) who he trusts to use the data at most for counseling. On the other hand, Bert is not confident on her data since she delegates it to Paul (4) whom she does not trust to keep her information confidential.*

This example is very close to the example that we have previously seen on misplaced delegation (Example 11). What changes is what can be obtained by poor Bert. In the former case he is afraid to receive a bad advice (delegation of

[7]http://www.dbai.tuwien.ac.at/proj/dlv

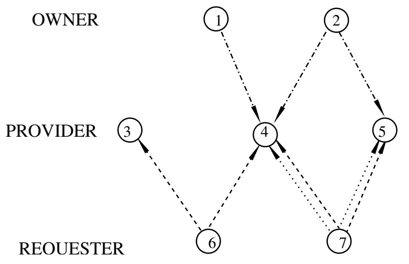


Figure 4. Owner vs Requester

execution), in the latter that her information can be used for other things than providing counseling.

The last part of Table 5 shows properties to verify at-most model and at-least model at the same time. Pro8 states that the requester has to can see the service fulfilled. Pro9 states that the requester has to be confident to see the service fulfilled.

Example 13 (Figure 4) *Bob and Bert (1 and 2) delegate the permission to use their personal information to Sam (4). Further, Bob delegates the permission on his personal information to his parents (5). Paul (7), needs to get student' information to provide accurate counseling, but he cannot directly ask them. Thus, he delegates the execution to get this sensitive data to Sam. Paul could delegate the execution to provide Bob's data to his parents. So Paul can suppose that someone provides the personal information of his students. On the other hand, Peter (6), delegates the execution to provide the personal information of his students to Carol (3), but the latter does not have the permission to manage it. Thus, Carol cannot forward the information to Peter. Further, Paul delegates the execution to provide the personal information of his students to the student information system, he does not trust it for this goal, and so he is not confident to get the personal information of his students.*

8 Related Work and Conclusions

The work by Liu et al. [18] uses the goal-oriented i*/Tropos RE methodology to introduce goals such as "Security" or "Privacy", and proposes dependency analysis to check if the system is secure. In [3], general taxonomies for privacy are proposed for a standard goal oriented analysis. Another early RE example is [24], which presents a requirements process model, based upon reuse and templates, for security policies in a organization.

On the side of approaches explicitly intended for security, we find the Jürjens proposal for UMLsec [15], an extension of the UML modeling low-level security mechanisms, and the CORAS methodology for modeling risk and vulnerability [9]. Lodderstedt et al. [19] present a UML-based modeling language (SecureUML). Their approach is

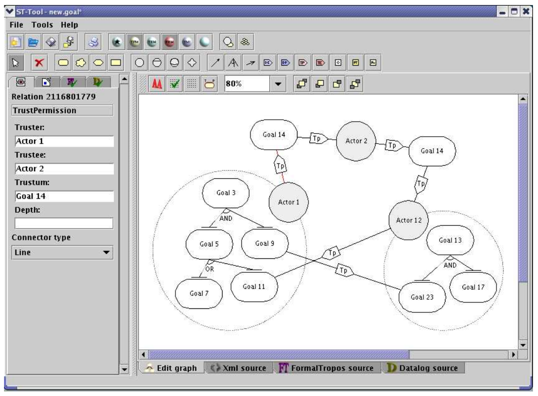


Figure 5. ST-Tool

focused on modeling access control policies and integrating them into a model-driven software development process. McDermott and Fox adapt use cases [21] to capture and analyze security requirements, and they call the adaption an abuse case model. An abuse case is an interaction between a system and one or more actors, where the results of the interaction are harmful to the system, or one of the stakeholders of the system. Guttorm and Opdahl [23] define misuse cases, the inverse of UML use cases, which describe functions that the system should not allow. Moving towards early requirements, the role of abuse-cases is played by Anti-Goals proposed by van Lamsweerde et al. [25].

In this paper we have extended our previous work [14, 13] providing a comprehensive modeling framework for security requirements. In particular, the framework offers

- the notions of delegation and trust of execution and delegation and trust of permission, respectively in the form of at-least and at-most delegation and trust;
- the use of monitoring as design solution (pattern) to overcome the problem of lack of trust between actors;
- a comprehensive semantic model based on Datalog to ease translations from requirements into security policies and trust management systems using the same semantics (as already stated in [13]).

Our framework with all new features presented in this paper is supported by ST-Tool[8] (Figure 5). ST-Tool is a graphical tool (implemented in java) to support the design of (Secure) Tropos models. ST-Tool allows system designers to draw (Secure) Tropos diagrams by selecting from the menu the desired (Secure) Tropos elements and to verify the correctness of the specification of the corresponding element. The tool allows an automatic transformation from Secure Tropos graphical models into formal specifications, especially into both Datalog specification and Formal Tropos specification [10]. For every Tropos elements it is possible to specify properties (i.e. creation-properties,

[8]On the web at `http://sesa.dit.unitn.it/sttool/`.

invar-properties, fulfill-properties) with respect to the syntax of Formal Tropos by selecting the corresponding panel in the menu. The resulting specification in Formal Tropos is automatically displayed by selecting the corresponding panel. Similarly, a Datalog specification is generated and displayed. This tool provides also a user friendly interface within the DLV system and permits a designer to select properties of each model and to specify additional security policies. The resulting Datalog specification is automatically verified by the DLV system with respect to the properties that the system designer want to check.

Future work aims at adding late refinement steps until we can automatically derive security services and mechanisms á la ISO-7498-2 and ISO-17799.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Computer Publishing, 2001.

[3] A. I. Antòn, J. B. Earp, and A. Reese. Analyzing Website privacy requirements using a privacy goal taxonomy. In *Proc. of RE'02*, pages 23– 31. IEEE Press, 2002.

[4] T. Aura. On the Structure of Delegation Networks. In *Proc. of the 1998 CSFW*, pages 14–26. IEEE Press, 1998.

[5] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An Agent-Oriented Software Development Methodology. *JAAMAS*, 8(3):203–236, 2004.

[6] R. Crook, D. Ince, L. Lin, and B. Nuseibeh. Security Requirements Engineering: When Anti-requirements Hit the Fan. In *Proc. of RE'02*, pages 203–205. IEEE Press, 2002.

[7] J. DeTreville. Binder, a logic-based security language. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 95–103. IEEE Press, 2002.

[8] P. T. Devanbu and S. G. Stubblebine. Software engineering for security: a roadmap. In *Proc. of ICSE'00 - Future of Software Eng. Track*, pages 227–239, 2000.

[9] R. Fredriksen, M. Kristiansenand, B. A. G. K. Stølen, T. A. Opperud, and T. Dimitrakos. The CORAS framework for a model-based risk management process. In *Proc. of SAFECOMP'02*, *LNCS* 2434, pages 94–105, 2002.

[10] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in tropos. In *Proc. of RE'01*, pages 174–181. IEEE Press, 2001.

[11] G. Gans, M. Jarke, S. Kethers, and G. Lakemeyer. Modeling the Impact of Trust and Distrust in Agent Networks. In *Proc. of AOIS'01*, pages 45–58, 2001.

[12] P. Giorgini, F. Massacci, and J. Mylopoulous. Requirement Engineering meets Security: A Case Study on Modelling Secure Electronic Transactions by VISA and Mastercard. In *Proc. of ER'03*, *LNCS* 2813, pages 263–276. Springer-Verlag, 2003.

[13] P. Giorgini, F. Massacci, J. Mylopoulous, and N. Zannone. Filling the gap between Requirements Engineering and Public Key/Trust Management Infrastructures. In *Proc. of EuroPKI'04*, *LNCS* 3093, pages 98–111. Springer-Verlag, 2004.

[14] P. Giorgini, F. Massacci, J. Mylopoulous, and N. Zannone. Requirements Engineering meets Trust Management: Model, Methodology, and Reasoning. In *Proc. of iTrust'04*, *LNCS* 2995, pages 176–190. Springer-Verlag, 2004.

[15] J. Jürjens. Towards Secure Systems Development with UMLsec. In *Proc. of FASE'01*, pages 187–200. Springer-Verlag, 2001.

[16] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *TISSEC*, 6(1):128–171, 2003.

[17] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of A Role-based Trust-management Framework. In *Proc. of 2002 IEEE Symp. on Sec. and Privacy*, pages 114–130. IEEE Press, 2002.

[18] L. Liu, E. S. K. Yu, and J. Mylopoulos. Security and Privacy Requirements Analysis within a Social Setting. In *Proc. of RE'03*, pages 151–161. IEEE Press, 2003.

[19] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *Proc. of UML'02*, LNCS 2460, pages 426–441. Springer-Verlag, 2002.

[20] F. Massacci, M. Prest, and N. Zannone. Using a Security Requirements Engineering Methodology in Practice: The compliance with the Italian Data Protection Legislation. *Comp. Standards & Interfaces*, 2005. To Appear. An extended version is available as Technical report DIT-04-103 at `eprints.biblio.unitn.it`.

[21] J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. In *Proc. of ACSAC'99*, pages 55–66. IEEE Press, 1999.

[22] P. Samarati and S. D. C. di Vimercati. Access Control: Policies, Models, and Mechanisms. In *Proc. of the 2nd FOSAD*, LNCS, pages 137–196. Springer-Verlag, 2001.

[23] G. Sindre and A. L. Opdahl. Eliciting Security Requirements by Misuse Cases. In *Proc. of TOOLS Pacific 2000*, pages 120–131. IEEE Press, 2000.

[24] A. Toval, A. Olmos, and M. Piattini. Legal requirements reuse: a critical success factor for requirements quality and personal data protection. In *Proc. of RE'02*, pages 95 –103. IEEE Press, 2002.

[25] A. van Lamsweerde, S. Brohez, R. De Landtsheer, and D. Janssens. From System Goals to Intruder Anti-Goals: Attack Generation and Resolution for Security Requirements Engineering. In *Proc. of RHAS'03*, pages 49–56, 2003.

[26] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.